Diplomarbeit im Studiengang Medieninformatik

Konzeption eines generativen und modellgetriebenen Ansatzes für komponentenbasierte Architekturen der Business-Tier

vorgelegt von
Markus Kopf

Matrikelnr. 0013879
Fachhochschule Stuttgart – Hochschule der Medien
15.01.2007

1. Prüfer: Herr Prof. Dr. Edmund Ihler, Hochschule der Medien
2. Prüfer: Herr Dipl.-Inf. (FH) Klaus Banzer, Softlab GmbH

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

München, den 15.01.2007

_____

Markus Kopf

Danksagung

Mein Dank gilt der Softlab GmbH die es für wichtig erachtet Diplomarbeiten an Studenten zu vergeben und somit neue und interessante Themen für die Bearbeitung ermöglicht.
Mein Dank gilt insbesondere Michele Siegler, der die Einstellung möglich gemacht hat und mich auch in meiner weiteren beruflichen Laufbahn unterstützt hat. Des Weiteren möchte ich mich bei meinem Betreuer Klaus Banzer bedanken, der mich in kompetenter Weise beraten und mit konstruktiver Kritik weitergebracht hat. Als weitere Kollegen bei Softlab möchte ich Christian Märkle, Joachim Hengge und Markus Reiter erwähnen, die gleich zu Beginn der Diplomarbeit für eine gute Atmosphäre gesorgt haben sowie bei Fragen jederzeit zur Verfügung gestanden haben.
An der Hochschule der Medien möchte ich den Dank an Prof. Dr. Edmund Ihler richten, der die Diplomarbeit von Seiten der Hochschule betreut und durch die Vorlesung Informatik 4 die nötigen Grundkenntnisse für die Diplomarbeit geliefert hat.

Think before you click.

That's the trick.

Abstract

Die vorliegende Diplomarbeit beschäftigt sich mit der Ablösung eines Generator Frameworks und der Generierung von Artefakten für die Businesslayer einer J2EE-Applikation. Der bestehende Generator sowie die Transformationsabbildungen sollen durch openArchitectureWare 4, einem metamodellbasiertem Generator Framework, ersetzt werden.

Mittels des openArchitectureWare Frameworks wird der viel diskutierte Model Driven Architecture (MDA) Ansatz der OMG vorgestellt. Es wird der Unterschied zwischen herkömmlicher Softwareentwicklung und der modellgetriebenen Entwicklung aufgezeigt, sowie alle damit verbundenen Vor- und Nachteile. Es soll gezeigt werden, dass Modelle nicht nur zur Dokumentation dienen, sondern als Code gesehen werden können und sollten. Im Bezug auf die technische Realisierung der MDA werden die verschiedenen Vorgehensweisen bei der Entwicklung von Modellen betrachtet und die eingesetzten Technologien wie z.B. UML, UML–Profile und Meta Object Facility vorgestellt. Außerdem wird ein Ausblick in neue Technologien im Kontext modellgetriebener Architekturen, wie zum Beispiel das Eclipse Modelling Framework (EMF) und das Graphical Modelling Framework (GMF) gegeben.

Da die Schwerpunkte der Diplomarbeit auf der von BMW verwendeten Architekur für verteilte Applikationen sowie der Generatormigration liegen, wird die Architektur und der Generator von BMW im Aufbau sowie der Funktionsweise näher betrachtet. Die von BMW eingesetzte Architektur „Component Architecture 2.0“ ist eine komponentenbasierte Architektur, welche bei BMW entwickelt wurde und projektübergreifend zum Einsatz kommt.
Des Weiteren liegt der Diplomarbeit das PEP-PDM Projekt, das bei BMW zurzeit durch Softlab realisiert wird, zugrunde. Durch PEP-PDM wird bei BMW eine zentrale Datenreferenz über den gesamten Entstehungsprozess eines Produktes erreicht. Das gesamte Wissen über ein Fahrzeug mit allen Daten des Entstehungsprozesses wird an einer zentralen Stelle zusammengeführt und dadurch, zu jedem Zeitpunkt, der aktuelle Stand im Entwickluingsprozess eines Fahrzeuges bekannt.

Durch die Migration auf openArchitectureWare, dem Open Source Generator Framework und dem dazugehörigen Vorgehensmodell der generativen Entwicklung, werden die eingesetzten Technologien des Frameworks und deren Möglichkeiten wie Templates, Extensions und Modellvalidierung erläutert. Die Technologien und Designentscheidungen werden an der Transformation eines bestehenden Platform Independent Model (PIM) des Projektes PEP-PDM näher betrachtet und erläutert. Die Transformationsabbildungen für das PEP-PDM PIM basiert

auf der von BMW eingestzten Component Architecture 2.0 und der Enterprise Technolgie J2EE.

Ein Benefit der Diplomarbeit ist die Evaluierung des Graphical Modeling Framework von Eclipse. Dies beinhaltet implizit den Einsatz des Eclipse Modeling Framework (EMF) als Design Sprache für die Anwendung und das Metamodell. Mit diesen Technologien könnte zukünftig die Möglichkeit bestehen vom Erstellen der domänenspezifischen Sprache (Metamodell), über das Anwendungsdesign (PIM), bis hin zur Transformation und Implementierung alles in einem Entwicklungstool (Eclipse) zu bearbeiten.

Abschließend werden die gewonnenen Erkenntnisse und Erfahrungen bezüglich der MDA unter Verwendung von openArchitectureWare zusammengetragen und bewertet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Einleitung

Geschäftsanwendungen sind in der Regel durch eine lange Lebenszeit geprägt. Die zugrunde liegenden Technologien unterliegen jedoch einem raschen Wechsel. In der heutigen Zeit müssen Firmen flexibel und schnell neue Technologien einsetzen können. Eine starre architektonische Struktur oder sogar propritäre Lösungen, die nicht kompatibel sind, hindern den Fortschritt und können Unternehmen ausbremsen. Aus diesem Grund müssen standardisierte und bewehrte Technologien für die Konzeption und Entwicklung neuer Softwaresysteme eingesetzt werden.

Ein weiterer Schritt zum Erfolg ist ein effizienter Softwareentwicklungsprozess der mit einem Wechsel der Technologie zurecht kommt. Hier setzt das Vorgehensmodell MDA[1] der OMG[2] an. Die modellgetriebene Softwareentwicklung ist in den letzten Jahren besonders durch die OMG an die Öffentlichkeit geraten. Mit dem MDA-Ansatz soll die Portabilität von Anwendungen entscheidend verbessert werden.
In der modellgetriebenen Entwicklung wird Software abstrakt mittels formalen Modellen beschrieben und analysiert. Dies erhöht die Abstraktion der Softwaresysteme und in den Modellen befinden sich, je nach Abstraktionsgrad, nur die wesentlichen Details. Technologische Inhalte werden erst bei der Konkretisierung der Modelle hinzugefügt. Durch dieses Vorgehen wird eine bessere Kommunikation mit dem Kunden möglich und die erstellten Modelle bleiben plattformunabhängig, so dass durch technologiebehafteten Abbildungen das Modell auf eine gegebene Technologie (z.B. J2EE) unter Verwendung von Generatoren, transformiert werden kann. Der Mehrwert und vor allem der Zeitvorteil können beim durchdachten Einsatz immens sein.

Ein Generator Framework, welches den modellbasierten Ansatz verfolgt, ist das openArchitectureWare Framework. Das Framework ist mittlerweile ein Open Source Projekt welches schon in einigen Projekten sowie Diplomarbeiten bei Softlab erfolgreich eingesetzt wurde. Das Framework soll in dieser Diplomarbeit für die Ablösung eines propritären Generators von BMW verwendet werden.
openArchitectureWare integriert das Framework seit der Version 4.0 sehr stark in Eclipse und will dadurch eine vollständige Integration des Softwareentwicklungsprozess von der Konzeption bis zur Implementierung ermöglichen. Dieses Vorgehen wird im Ausblick unter Verwendung neuere Technologien behandelt.

[1] MDA – Model Driven Architecture
[2] OMG – Object Management Group

Softlab

Die Softlab Group besteht zum Zeitpunkt der Erstellung dieser Diplomarbeit aus vier Tochter-Unternehmen:

- Axentiv - SAP Beratungshaus der Softlab Group
- Entory - IT-Consultant für die Finanzdienstleister
- Nexolab - Consulting Company mit Automotive Kompetenz
- Softlab - Projekt- und Beratungshaus

Zusätzlich zu den aufgeführten Tochterunternehmen ist Softlab an der, seit 2006 hundertprozentige, Tochtergesellschaft F.A.S.T. GmbH beteiligt.
Softlab selbst ist eine hundertprozentige BMW-Group Company mit Hauptsitz in München und weiteren Standorten in Deutschland. Zusätzliche Tochtergesellschaften international existieren an den Standorten in Österreich, Schweiz und England. Die Softlab Group zählt derzeit inklusive der Tochtergesellschaften 1700 Mitarbeiter.
Das Geschäftsfeld der Softlab Group liegt überwiegend in den Branchen Fertigung, Banken, Industrie, Versicherungen und Telekommunikation. Der Fokus ist auf die Entwicklung kundenspezifischer Individuallösungen auf Basis neuer Technologien gerichtet. Im Vordergrund steht die Betreuung der Kunden aus Industrie, Automotive, Finanzdienstleister und Handel während des gesamten Prozesses von der Konzeption, Implementierung, Betrieb und Wartung von IT-Lösungen.

<u>Softlab GmbH München:</u>

• ca. 800 Mitarbeiter
• 1971: Gründung
• 1992: BMW Group Company

Anschrift
Softlab GmbH
Zamdorfer Straße 120
81677 München
http://www.softlab.com

1 Diplomarbeit

1.1 Ausgangssituation

Basis der Diplomarbeit ist eine bestehende komponentenbasierte Architektur, die „Component Architecture 2.0“, welche konzernweit bei BMW eingesetzt wird. Die Component Architecture (CA) wird zur Erstellung der Geschäftslogik von Anwendungen verwendet. Die Designsprache der CA definiert Geschäftsobjekte die in Komponenten strukturiert sind und Daten, Logik und Regeln zusammenfassen. Die CA basiert auf dem generativen Entwicklungsweg der MDA und den damit verbundenen drei Architekturschichten Platform Independent Model (PIM), Platform Specific Model (PSM) und Implementierungsrahmen. Mit Hilfe der Designsprache wird das plattformunabhängige Modell, auf Basis der UML, komponentenbasierte Entwicklung und allgemeinen Architekturprinzipien formuliert. Weiterhin bringt die CA einen Modelltransformator, der das Modellierungsprofil auf das J2EE konforme plattformspezifische Modell und anschließend auf Code abbildet mit. Häufig benötigte Basisfunktionalitäten der J2EE Architektur sind in einem Framework gekapselt. Dieses *ca20_framework* wird mit der Auslieferung der Component Architecture mitgeliefert.

Als Grundlage während der Entwicklung der Diplomarbeit dient das Projekt PEP-PDM, welches auf der CA2.0 basiert. Die zwei Akronyme stehen für Produkt Entstehungsprozess und Produkt Datenmanagement. Die BMW-Group hat sich für die nächsten Jahre ehrgeizige Ziele gesetzt. BMW will bis 2008 zahlreiche neue Modelle entwickeln und auf den Markt bringen. Im Bereich des Produktentstehungsprozesses steht das Unternehmen damit vor signifikanten Herausforderungen. Mit dem Projekt PEP-PDM soll die zunehmenede Datenmenge und die dadurch steigende Komplexität beherrschbar bleiben. Das gesamte Wissen über ein Fahrzeug wird durch PEP-PDM an einer zentralen Stelkle gesammelt und kann von allen genutzt werden. Somit ist PEP-PDM die zentrale und verbindliche Kommunikationsplattform im Produktentstehungsprozess. Mit der Realisierung des Projektes soll die gestiegene Komplexität transparenter gestaltet werden. Dieses Ziel erfordert eine erhöhte Produktdatentransparenz zu jedem Zeitpunkt der Fahrzeugentwicklung. Neue Prozesse werden dazu entwickelt und implementiert, welche im gesamten Produktentstehungsprozess die Verbindlichkeit und Durchgängigkeit der Daten und Strukturen sicherstellt.

Die Zielsetzung die durch das PEP-PDM Projekt erreicht werden soll ist in der folgenden Abbildung noch einmal verdeutlicht. PEP-PDM ist der zentrale Baustein der die verschiedenen Systeme zusammenhält.

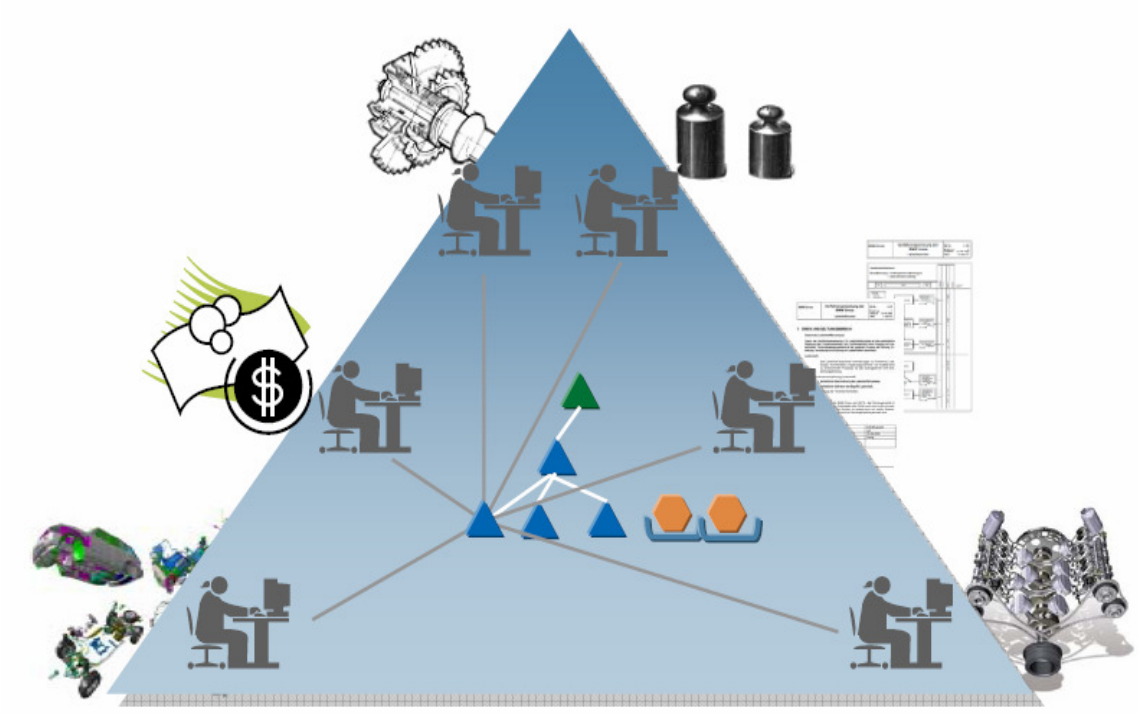


Abbildung 1. 1: Zentralisierung der Daten in PEP-PDM

Die für das PEP-PDM Projekt bestehende Systemlandschaft wurde in der Diplomarbeit weiterhin verwendet und nicht angepasst.

1.2 Aufgabenstellung und Ziele

In der Diplomarbeit muss die bestehende generative Architektur zuerst analysiert werden. Nach der Analyse soll dann der bestehende Transformator der Component Architecture 2.0 durch die aktuelle Version 4.0 des openArchitectureWare Frameworks ersetzt werden. Folgende Ziele sollen hierdurch erreicht werden:

- Lösen von den BMW spezifischen Vorgaben und dadurch freier im Entwicklungsprozess.

- Die feste Bindung an Together soll durch den Einsatz des openArchitectureWare Generators gelöst werden. Diese Toolbindung ist bei dem Einsatz des BMW Transformators implizit gegeben.

- Möglicher Wechsel des Modellierungstools soll gegeben sein. Das heißt, der verwendete Generator muss eine generische Schnittstelle für unterschiedliche Modellierungstools besitzen.

- Verringerung der Generierungsschritte bis zum Implementierungsrahmen. In der aktuellen Situation werden 3 Schritte bis zum Implementierungsrahmen benötigt. Der erste Schritt führt das Platform Independent Model (PIM) ins Platform Specific Model (PSM) über, danach wird ein XDoclet-Lauf gestartet der mittels im Code hinzugefügten Metadaten die Enterprise Java Bean Interfaces sowie die Deployment Deskriptoren generiert. Dies ist nach der OMG zwar ein sauberer Ansatz, allerdings soll durch den openArchitectureWare Einsatz eine passende Interpretation des MDA Konzeptes zum Einsatz kommen.

- Transformation des vollständigen PEP-PDM PIM durch die neue generative Architektur. Wenn möglich sollte der Implementierungsrahmen kein Delta zu der vorherigen Lösung des BMW Generators aufweisen.

- Einsatz der entwickelten generativen Architektur am externen Markt. Somit kann unter Einsatz der hier erstellten generativen Architektur am externen Markt Projekte realisiert werden ohne rechtliche Bedenken gegenüber BMW haben zu müssen.

- Durch den Besitz der Ressourcen und Quellen ist gegenüber des Supports der BMW Architektur ein wesentlich besserer Support möglich.

- Die zu generierende Artefakte müssen weiterhin CA2.0 konform sein damit Projekte auf BMW Seite mit der generativen Architektur realisiert werden können.

Nach der Analyse der bestehenden Architektur und der Abbildungsregeln, müssen die Templates für den oAW erstellt werden. Zur Verringerung der Generierungsschritte werden die Teile, die zuvor XDoclet in einem weiteren Schritt erzeugt hatte, in die Templates und in den Generierungsprozess mit aufgenommen.

In einem weiteren Abschnitt der Arbeit wird ein Ausblick über die Integration des Entwicklungsprozesses in Eclipse gegeben. Es wird evaluiert, ob es möglich ist von der Modellierung über die Transformation bis hin zur manuellen Codierung, alles in einem Tool zu bearbeiten. Für diesen Ansatz wird GMF/EMF im Zusammenspiel mit dem openArchitectureWare näher beleuchtet.

2 Struktureller Überblick

Die Diplomarbeit besteht aus einem theoretischen Teil, der die technologischen Grundlagen schafft und einem praktischen Teil, der die Realisierung aufführt.

Das Kapitel 3 erläutert die von der OMG aufgestellten Konzepte der Model Driven Architecture und zeigt die Differenzierung zur Model Driven Software Development. In Kapitel 4 wird die von BMW verwendete Component Architecture und in Kapitel 5 die für die Generierung der Component Architecture benötigte Artefakten erläutert. Im darauf folgenden Kapitel 6 wird das openArchitectureWare Framework und dessen Module für die Realisierung in Kapitel 7 vorgestellt.

In den letzten Kapiteln wird ein Ausblick in neue Technologien zur Integration eines kompletten Entwicklungsprozesses in Eclipse gegeben.

3 Model Driven Architecture (MDA)

3.1 Einführung und Differenzierung zu MDSD

MDA ist ein junger Standard der Object Management Group[3] (OMG), welche 1989 gegründet wurde und heute ein offenes Konsortium aus ca. 800 Firmen weltweit ist. Die OMG erstellt herstellerneutrale Spezifikationen zur Verbesserung der Interoperabilität und Portabilität von Softwaresystemen. Bekannte Ergebnisse sind CORBA, IDL, UML, XMI, MDA oder MOF.
Mit der MDA soll eine erhebliche Steigerung der Entwicklungsgeschwindigkeit möglich sein. Die Steigerung wird durch Formalisierung der Applikation in Modellen erreicht. Es wird aus formal eindeutigen Modellen unter Einsatz von Generatoren automatisch Code erzeugt.
Durch den Einsatz der Generatoren und der formal eindeutig definierten Modellierungssprachen wird darüber hinaus die Softwarequalität erheblich gesteigert, weil der Generator keine Flüchtigkeitsfehler begeht. Wenn trotzdem Fehler auftreten sind diese in der Transformation oder im Modell zu finden, also an einem zentralen Punkt. Dies erleichtert die Fehlersuche deutlich, da nicht im Sourcecode an vielen Stellen der Fehler gesucht werden muss. Die Qualität des generierten Sourcecodes ist gleich bleibend, was zu einem höheren Grad der Wiederverwendung führt.

Ein weiteres Ziel von MDA ist die bessere Handhabbarkeit von Komplexität durch Abstraktion. Mit den Modellierungssprachen wird Programmierung auf einer abstrakteren Ebene möglich. Durch diese abstraktere Ebene ist eine saubere Trennung von fachlichen und technischen Bereichen möglich, und zudem können die Modelle von Domänenexperten verstanden werden.

Abgrenzung MDA zu MDSD

MDA hat ähnliche Ansätze wie MDSD, unterscheidet sich aber in verschiedenen Details. Zum Beispiel hat MDA die Fokussierung auf UML-basierte Modellierungssprachen. MDSD im Allgemeinen macht diese Einschränkungen nicht. Das Ziel von MDA ist in erster Linie Interoperabilität zwischen Werkzeugen und auf längere Sicht die Standardisierung von Modellen für populäre Anwendungsbereiche. MDSD zielt hingegen auf die Bereitstellung von praktisch einsetzbaren Elementen für die Softwareentwicklung ab. Außerdem soll die Werkzeugauswahl frei möglich sein.
Generell kann man sagen, dass die MDA eine Standardisierung der OMG zum Thema MDSD ist.

[3] OMG - http://www.omg.org/ (Stand November 2006)

Die Abgrenzung zur Model Driven Software Development besteht hauptsächlich aus den unterschiedlichen Perspektiven der Entwickler. Bei der MDA wird ein direkter Bezug auf die Unified Modeling Language (UML) bezogen und dieser Ansatz auch restriktiv durchgezogen. MDSD sieht diesen Ansatz nicht ganz so streng. Jede formulierbare Designsprache kann hier verwendet werden und man ist nicht auf UML beschränkt.
Ein weiterer Unterschied besteht darin, dass bei der Transformation von PIM zu PSM bei MDSD der Zwischenschritt über das PSM nicht vorgenommen wird, sondern direkt aus dem Platform Independent Model in den Sourcecode transformiert wird.
Der Ansatz von MDSD sieht keine gravierenden Vorteile den Zwischenschritt über das PSM vorzunehmen. Im Gegenteil, die Versionierung des PSM ist ein weiterer erschwerender Punkt. Durch Änderungen, die am PSM vorgenommen werden können, entsteht der Aufwand der Versionierung der verschiedenen Änderungen und die Überprüfung, dass die jeweiligen Anpassungen auch konsistent sind. In großen Projekten kann man sich vorstellen, dass dies keine triviale Aufgabe ist und einen nicht zu unterschätzenden extra Aufwand darstellt.

Ein weiterer Punkt ist die Zielvorstellung von ausführbaren Modellen, die in der MDA-Spezifikation aufgeführt werden. Dies wird so auch nicht in MDSD adressiert. Es macht bei der MDSD keinen Sinn das Modell soweit zu verfeinern bis es am Ende ausführbar wird. Dieser Ansatz kann bei Verzicht der Modell zu Modell-Transformation nicht verfolgt werden, da es keine Verfeinerungen über mehrere Schritte bis hin zum Sourcecode gibt.

Zusammenfassend kann man wie in [Völ01] sagen, bei MDSD geht es darum, Softwareentwicklung durch größere Anlehnung an die Konzepte der jeweiligen Domänen effizienter zu gestalten. Die Lücke zwischen Modell und Zielplattform wird durch eine Plattform, die die wichtigsten Architektur-Konzepte bereits zur Verfügung stellt, verkleinert.
MDA geht zwar einen ähnlichen Weg, jedoch fehlt hier ein definierter Entwicklungsprozess und die Spezifikation ist generell zu abstrakt gehalten. Des Weiteren spezialisiert MDA die MDSD mit oben schon erwähnten Konzepten.

3.2 Grundprinzip

Ein zentraler Begriff bei einem modellgetriebenen Ansatz ist Abstraktion. Das Abstraktionsniveau ist auf der Programmiersprachenebene am niedrigsten und steigt dann über das technische und fachliche Modell an. Durch Abstraktion werden Informationen ignoriert da diese in einem bestimmten Kontext nicht relevant sind.

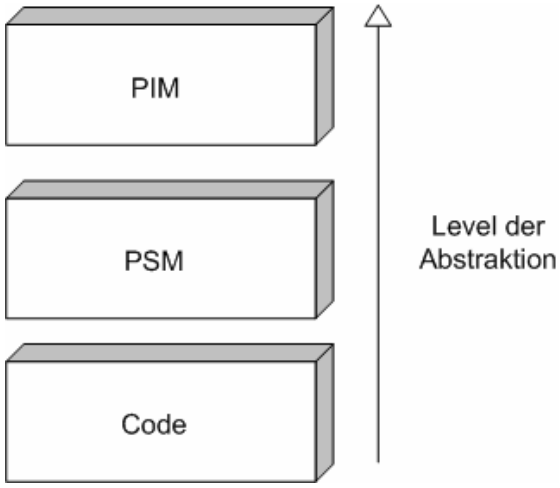


Abbildung 3. 1: Abstraktions-Level

Software soll mittels eines Modells so abstrakt dargestellt werden, dass dieses völlig unabhängig von der Zielplattform, auf welcher die Software später laufen soll ist. Somit wird es möglich, unterschiedliche Plattformen zu unterstützen. Solch ein PIM wird anschließend sukzessive konkretisiert bis im Idealfall die lauffähige Software entsteht.

Model Driven Architecture will mit folgendem Ansatz das oben aufgeführte Ziel erreichen. Fachliche Spezifikationen werden in plattformunabhängigen Modellen (PIM) definiert. Um ein PIM zu definieren wird eine formal eindeutige Modellierungssprache (Designsprache) verwendet. Diese Modellierungssprache ist bei dem Ansatz der OMG eine mittels Profilen erweiterte UML-Notation. Die damit erreichte Fachlichkeit ist vollständig unabhängig von der späteren Zielplattform. Wichtig hierbei ist die durch das Profil erreichte Eindeutigkeit, die somit ein UML-Modell erst zu einem MDA-Modell macht. Erst durch das Profiling kann man in einem späteren Generierungsschritt das Anwendungsdesign in einen Implementierungsrahmen überführen.

Durch eine mittels Werkzeuge automatisierte Modelltransformation werden zunächst aus der plattformunabhängigen Spezifikation, plattformabhängige Modelle (PSM) gewonnen. Diese Modelle enthalten spezifische Konzepte der Zielplattform. Zielplattformen können z.B. J2EE, .NET oder CORBA sein.

Durch eine weitere Transformation mittels Werkzeugen, wird dann aus dem Platform Specific Model der Implementierungsrahmen gewonnen.

In der folgenden Abbildung ist das eben beschriebene visuell dargestellt:

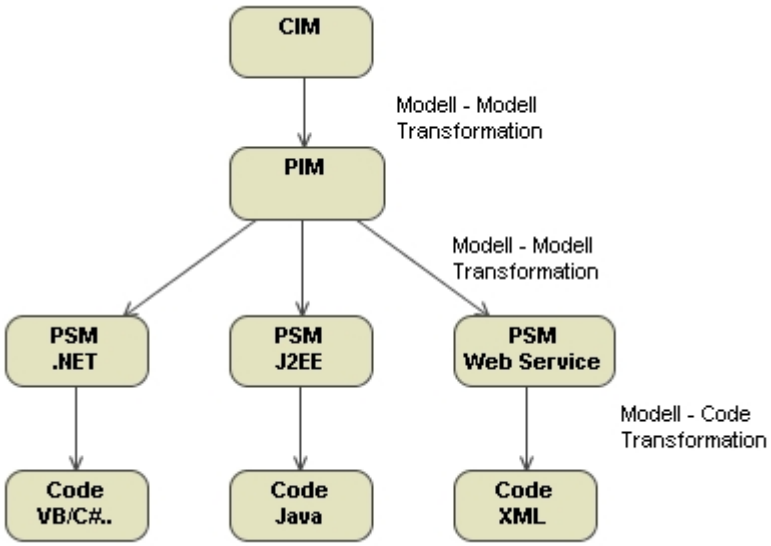


Abbildung 3. 2: Model Driven Architecture nach OMG

3.3 Basiskonzepte

3.3.1 Modell

Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems. MDA-Modelle werden in der Regel in UML definiert. Allerdings sind auch klassische Programmiersprachen MDA-Modellierungssprachen. Ihre Programme sind MDA-Modelle. Diese sind wiederum unabhängig von der darunter liegenden Plattform.
UML-Diagramme sind allerdings nicht generell MDA-Modelle. Der Unterschied zwischen MDA-Modellen und UML-Diagrammen liegt in der formal definierten Bedeutung (Semantik). Diese Bedeutung wird mittels einer formal eindeutigen Modellierungssprache sichergestellt, die in einem UML-Profil hinterlegt wird. Durch das Profiling wird erreicht, dass das erstellte Modell eindeutig transformiert werden kann. Die Transformation erfolgt durch einen Generator welcher das Eingabemodell auf ein spezielleres Modell oder Sourcecode überführt.

3.3.2 Plattform

Die Model Driven Architecture sagt nichts über den Abstraktionsgrad von Plattformen aus. Eine Plattform für Applikationen kann eine wohldefinierte Anwendungsarchitektur mit dazugehörigem Laufzeitsystem sein. Plattformen sind auch CORBA und J2EE für Business-Applikationen sowie ein PC eine Plattform für ein Betriebssystem darstellt.

3.3.3 UML-Profile

Ein UML-Profil ist der Standardmechanismus zur Erweiterung des Sprachumfangs der UML, um sie an spezifische Einsatzbedingungen (z.B. fachliche oder technische Domänen) anzupassen. UML-Profile bilden im Rahmen der MDA die Grundlage für die automatische Modelltransformation. Mit Hilfe von UML-Profilen (und entsprechenden Transformationsregeln) wird die Abbildung eines Modells auf eine gegebene Plattform eindeutig definiert, d.h. es kann z.B. ein PIM in ein für eine definierte Zielplattform entsprechendes PSM umgewandelt werden (Modell zu Modell Transformation).
Ein UML-Profil definiert die Syntax der Modellierungssprache (analog der Grammatik einer klassischen Programmiersprache) sowie die statische Semantik der Modellierungssprache durch Einschränkungen. UML-Profile werden mit Hilfe von Stereotypen, Tagged Values (Eigenschaftswerte) sowie Constraints (Einschränkungen) definiert. UML-Profile sind also Metamodelle und definieren formale Modellierungssprachen als Erweiterung der UML.

3.3.4 CIM

Mit Hilfe des CIM wird das Geschäftsmodell oder die Domänenansicht des zu entwickelnden Softwaresystems modelliert. Das CIM ist völlig frei von plattformspezifischen Details. Zur Modellierung des CIM werden Use Case Diagramme, Interaktionsdiagramme und Aktivitätsdiagramme eingesetzt. Somit kann ein Anwendungsdesigner mit den Domänenexperten, die in den jeweiligen Bereichen arbeiten, zusammen ein Konzept entwerfen. Die Verhaltensbeschreibungen und Implementationen die im PIM und PSM durchgeführt werden, müssen auf die Anforderungen im CIM rückführbar sein.

3.3.5 PIM und PSM

Die Trennung von Platform Independent Model (PIM) und Platform Specific Model (PSM) ist ein Schlüsselkonzept von MDA im Sinne der OMG. Die Trennung basiert darauf, dass Konzepte stabiler als Technologien sind und formale Modelle Potential für automatisierte Transformationen besitzen.
Das PIM abstrahiert von technologischen Details, diese sind im PSM wieder zu finden. Im PSM sind Konzepte einer spezifischen Plattform zu finden um diese zu beschreiben.

3.3.6 Transformationen

Transformationen bilden Modelle auf die jeweils nächste Ebene ab. Je nachdem wo man sich in der generativen Architektur befindet ist die nächste Ebene wieder ein Modell oder Sourcecode. Transformationen müssen nach MDA formal und flexibel beschrieben werden können. Somit ist es möglich mittels eines Generators die Abbildung auf Basis des gegebenen Profils automatisiert umzusetzen.
Abbildung 3.3 illustriert das MDA Pattern wie ein PIM in ein PSM transformiert wird.

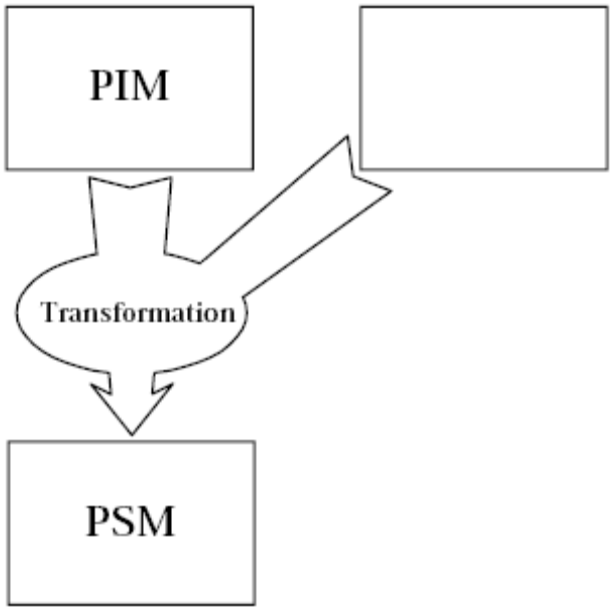


Abbildung 3. 3: Modell-Transformation

Wie aus der Abbildung ersichtlich, wird mit dem Platform Independent Model und der Kombination zusätzlicher Informationen über die Transformation zum Platform Specific Model transformiert.

3.4 Technologien

3.4.1 Meta Object Facility (MOF)

Die Modelling Object Facility[4] ist ein OMG Standard, welcher das Metametamodell beschreibt um Modellierungssprachen zu bilden. MOF definiert eine Reihe von Modellierungskonstrukte die ein Entwickler zum Definieren und Ändern von Metamodellen nutzen kann. Die MOF

[4] MOF - http://www.omg.org/mof/ (Stand November 2006)

befindet sich ihrerseits auf dem Level M3 der Vier-Schichten Metamodel Architektur, welche weiter unten erläutert ist, wieder. Sie beinhaltet die Struktur und Semantik beliebiger Metamodellen.

Ein vereinfachtes Modell von MOF ist in der folgenden Abbildung zu sehen:

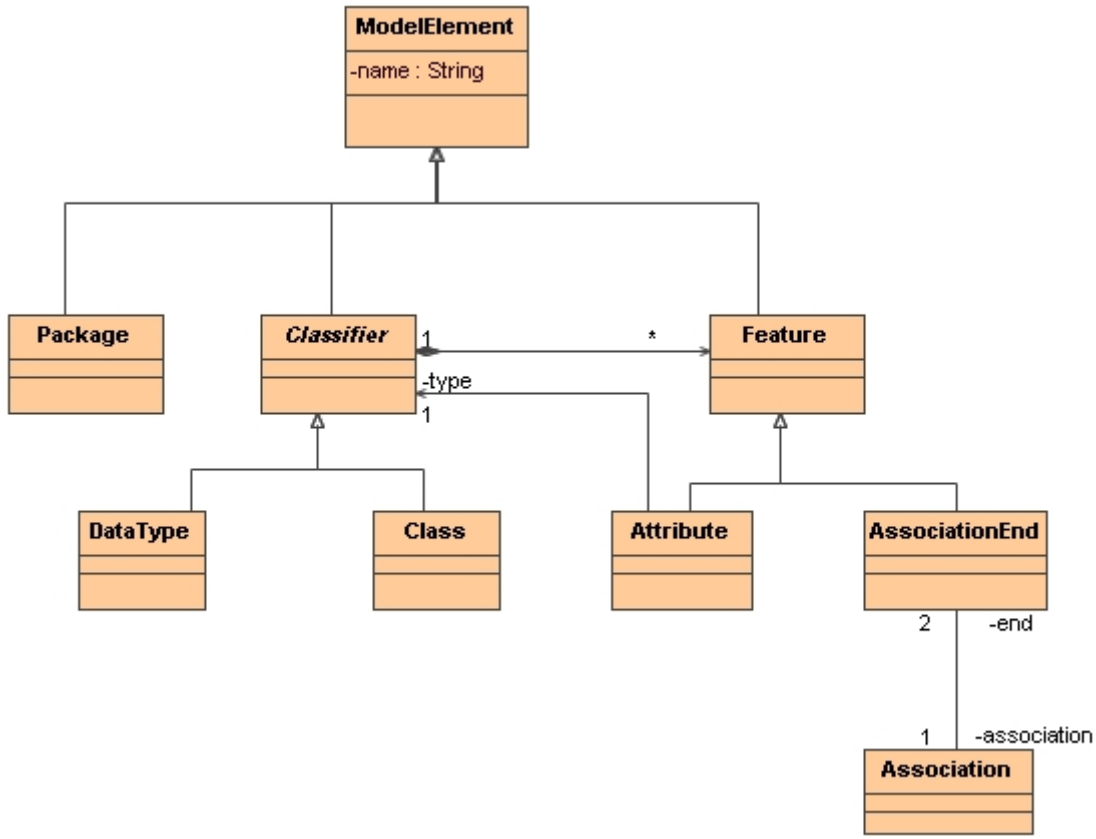


Abbildung 3. 4: MOF Metamodel

Die MOF ist allerdings nicht nur als Basis für Metamodelle wichtig, sondern hat essenzielle Bedeutung für MDA-Tools. Damit Portabilität zwischen MDA-Tools gewährleistet werden kann, müssen diese auf einen definierten Standard aufbauen. Genau solch einen Standard definiert die OMG mittels der MOF. Somit bauen alle Tools, die die MOF verwenden, auf dem gleichen Metametamodell auf und sind zueinander kompatibel.

4-Schichten Architektur

Die 4-Schichten Metamodellarchitektur definiert eine Modellhierarchie, die sukzessive die einzelnen Modellarten definiert. Sie ist zentral für das Verständnis der Unified Modelling Language und dient unter anderem dazu

- semantische Konstrukte, durch ihre rekursive Anwendung auf weitere Metalayern zu definieren und
- die architektonische Basis für zukünftige UML Erweiterungen festzulegen.

Die 4-Schichten Architektur besteht aus folgenden 4 Layern:

M3 Meta-Meta Modell:
Meta-Meta-Modelle (bzw. MOF-Ebene). Abstrakte Ebene, die zur Definition der M2-Ebene herangezogen wird.

M2 MetaModell:
Diese Ebene ist eine Instanz des Meta-MetaModells und definiert die Sprache zur Beschreibung der Modelle. Hier werden die Konstrukte festgelegt, die in der M1-Ebene Verwendung finden. Diese Ebene ist das zentrale Element der UML. Zudem werden auf dieser Ebene die Ergänzungen für die unterschiedlichen Plattformprofilen spezifiziert.

M1 Modell:
Ist eine Instanz des MetaModells und definiert die Sprache zur Beschreibung der Domäne. In dieser Ebene wird z.B. eine Klasse definiert.

M0: Objekte:
Sind Instanzen des Modells und beschreiben die Ausprägung einer bestimmten Domäne. Hier werden Instanzen einer Klasse aus dem Modell angelegt.

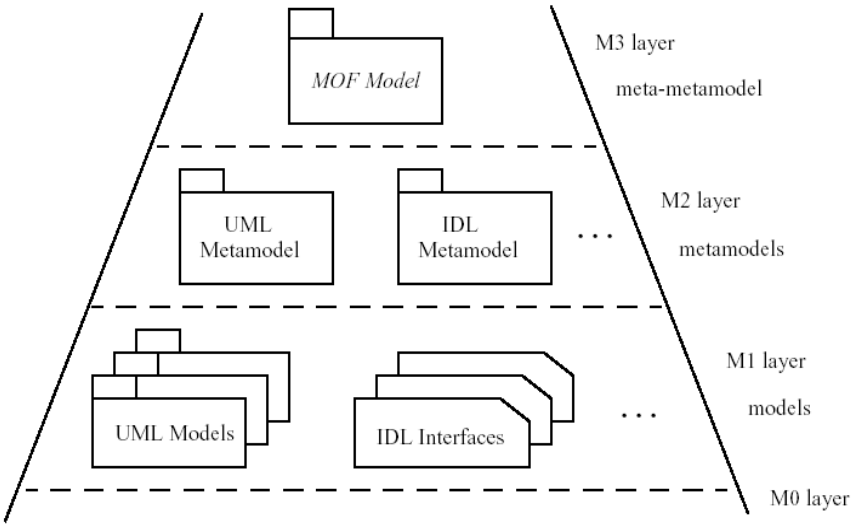


Abbildung 3. 5: MOF Metadatenarchitektur

Zur Veranschaulichung wird nachfolgend ein Beispiel aufgeführt. In der folgenden Abbildung sind die oben genannten Ebenen und zugehörige Artefakten aufgeführt.

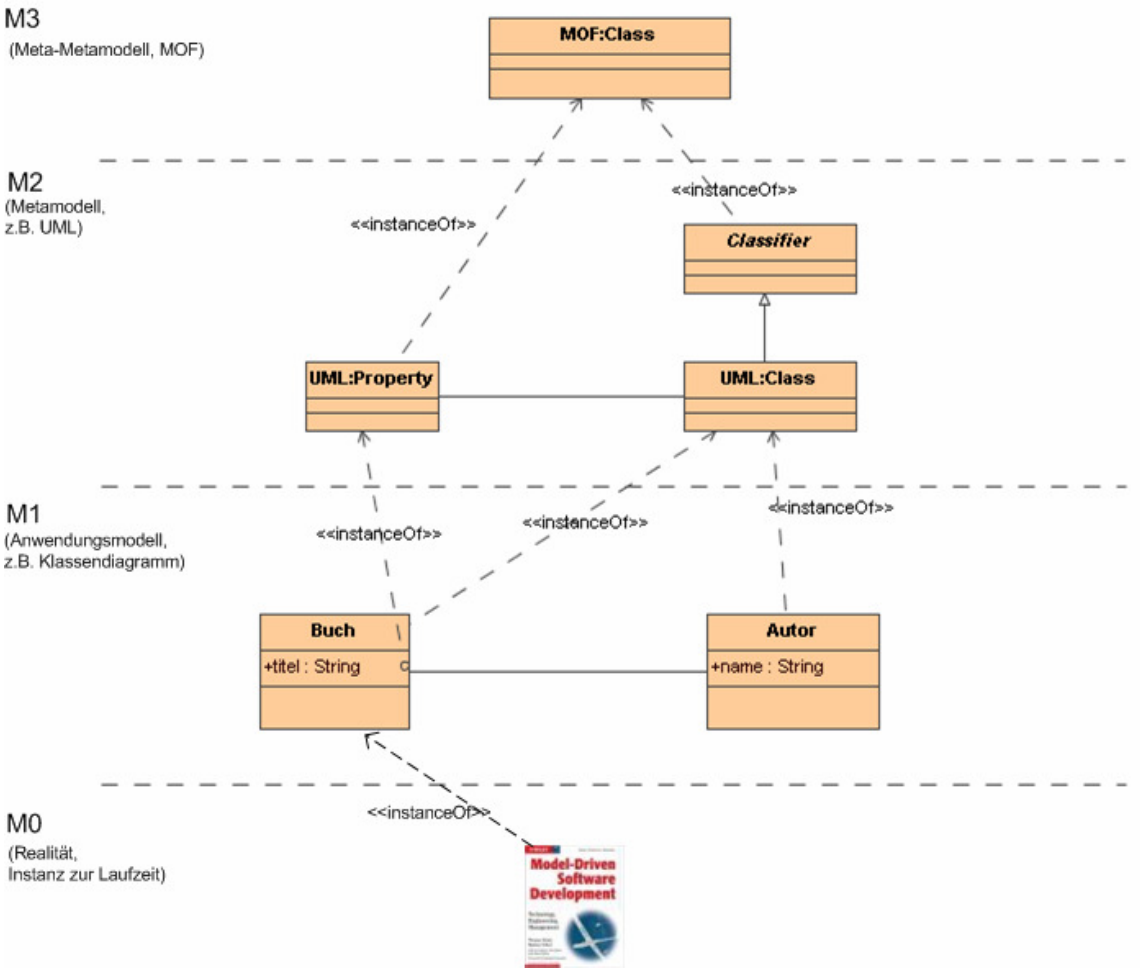


Abbildung 3. 6: Metamodellierung der Hierarchie

3.4.2 Unified Modelling Language

Die Unified Modelling Language ist eine von der OMG entwickelte und standardisierte Sprache für die grafische Beschreibung von Softwaresystemen. Die Unified Modelling Language ist die Standardmodellierungssprache, welche durch MOF beschrieben wird und findet sich in der 4-Schichten-Architektur auf dem M2 Level wieder. Der Grundgedanke bei der UML besteht darin, eine einheitliche Notation für verschiedene Einsatzgebiete zu haben. Mit UML ist es möglich, Datenbankanwendungen, Workflowanwendungen, Echtzeitsysteme, komponentenbasierte Anwendungen usw. zu beschreiben. Um diese Softwaresysteme zu beschreiben besteht die UML aus verschiedenen Diagrammen, die wiederum verschieden grafische Artefakte besitzen. Die Semantik der Artefakte ist in der UML eindeutig festgelegt.

3.4.3 UML-Profile

UML-Profile sind in der Model Driven Architecture ein zentraler Bestandteil. Erst durch das Profiling werden UML-Modelle zu MDA-Modelle und können automatisiert in die nächste Abstraktionsebene überführt werden. Mit dem Mechanismus Profile kann man den UML Sprachumfang erweitern und diesen auf eine gewünschte Domäne anpassen. In der Praxis werden Profile zur Unterscheidung des Verwendungskontextes der Klassen, Beziehungen oder Paketen verwendet.
Seit UML 2.0 ist allerdings der Begriff Profile mit Vorsicht zu verwenden. In UML 1.x gibt es kein formales Konstrukt Profile, sondern der Begriff wurde lediglich verbal für ein mittels Stereotypes angepasstes Metamodell verwendet. Im Folgenden wird das UML-Profile der unterschiedlichen UML-Versionen erläutert.

UML-Version 1.x

In der UML-Version 1.x wurden Stereotypen nur verbal und nicht konkret als Profile bezeichnet. Stereotypen erweitern ein schon existierendes UML-MetaElement zum Beispiel *UML::Class*. Somit werden die Struktur und die Semantik des erweiterten Elements nicht verändert.

Abbildung 3. 7: Definition Stereotype nach UML 1.x

Da es in der Version 1.x kein explizites Sprachkonstrukt gibt, wird das Metamodell-Element mittels Extend erweitert. Anzumerken ist, dass diese Generalisierung in demselben Abstraktions-Layer erfolgt. (Meta-Layer M2)
Nach der Definition des Stereotyps kann dann in der nächsten MetaEbene eine Instanz dieses Stereotyps angelegt werden. Um in dem oben aufgeführten Beispiel zu bleiben wird hier ein BF-Element (Business Facade) angelegt. Bei einer Transformation kann dann aus einem BF-Element über Abbildungsvorschriften andere Artefakte als aus einer definierten Class erstellt werden.

UML-Version 2.0

Mit der Definition von UML 2.0 wurde der Stereotype Mechanismus aus der Version 1.x erweitert und als eigenes UML Sprachkonstrukt definiert. Es wurde das Konzept der Extension eingeführt. Eine Extension ist ein völlig neues grafisches Symbol, welches nichts mit dem Extend-Mechanismus aus UML 1.x zu tun hat.

Abbildung 3. 8: Definition eines Stereotypen in UML 2.0

Ein Stereotype, definiert in UML 2.0, kann genau so wie ein Stereotype in der Version 1.x Attribute besitzen. Attribute werden als Tagged Values angezeigt. Allerdings können diese in der Version 2.0 auch einen Typ besitzen und sind nicht wie bisher einfach nur Strings.
Der Profile-Mechanismus in UML 2.0 geht noch weiter als hier beschrieben. Detaillierte Informationen kann man unter [UML01] finden.

Es gibt noch weitere Varianten um ein Profil zu definieren. Es werden hier diese aufgezählt jedoch nicht näher betrachtet da in der Praxis diese nur sehr selten Verwendung finden.

- Basic UML Konstrukte (z.B. Class, Association, Operation, u.v.m.)
- Tagged Values
- Constraints (durch OCL definierbar)

3.4.4 XML Metatdata Interchange (XMI)

Interoperabilität von Metamodellen über Domänen hinweg muss für die Integration von Tools und Applikationen gegeben sein. XMI ist der OMG-Standard zum werkzeugunabhängigen Transfer objektorientierter Modelle. XMI basiert auf der extensible Markup Language[5] (XML). Modellierungssprachen deren Metamodelle Instanzen von MOF sind, können mit XMI serialisiert werden. Da die MOF sich selbst definiert, kann XMI als Standardaustauschformat für Metamodelle benutzt werden.
Mit der Definition der Unified Modelling Language wurde für die Praxis eine akzeptierte grafische Beschreibungssprache geschaffen. Damit die Interoperabilität unter Designwerkzeugen gegeben ist, benötigt UML eine textuelle Notation. Mittels XMI ist eine textuelle Beschreibung möglich und die grafischen Notationen können serialisiert werden.

Weil XMI hauptsächlich zum Austausch von UML-Modellen benutzt wird, wird suggeriert, XMI sei das UML-Austauschformat. Dem ist aber nicht so, denn XMI wird noch in anderen Anwendungsdomänen, wie in der Abbildung 3.9 veranschaulicht, verwendet.

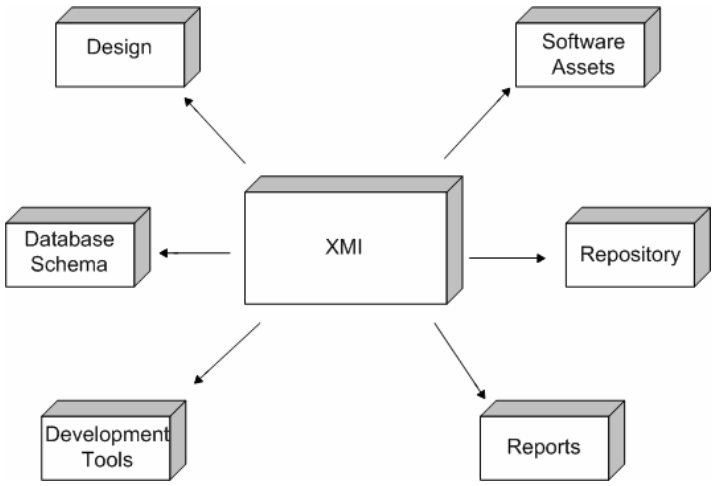


Abbildung 3. 9: XMI-Vision

XMI kann unter anderem weitere Aktivitäten im UML-Umfeld unterstützen.

- Codegenerierung aus objektorientierten Modellen
- Modellvalidierung
- Versionsverwaltung
- Pattern Libraries
- Transfer Format für Data-Warehouse Applikationen

[5] XML - http://www.w3.org/XML/

- u.v.m.

XMI ist ein Standard welcher zur Generierung von XML-Vokabular verwendet werden kann. Die OMG hat mittlerweile 5 Spezifikationen für XMI veröffentlicht. Zum Zeitpunkt der Erstellung der Diplomarbeit ist die Spezifikation XMI 2.1 die aktuelle Version. Unter [OMG01] sind alle Spezifikationen aufgeführt.
XMI ist kein spezifiziertes XML Vokabular, sondern wie oben bereits erwähnt, ein Algorithmus mit dem man ein Vokabular für Metamodelle generieren kann. Es schreibt ein Konzept vor wie „tags“ in XML für Metamodelle definiert werden. Mit anderen Worten kann man XMI als eine Metasprache bezeichnen, mit der es möglich ist Sprachen zu beschreiben. Daher ist es wohl eher als Framework zu sehen.
Dies ist auch einer der Gründe warum XMI nicht gleich XMI ist, denn die Hersteller von Modellierungstools interpretieren das Framework nie gleich. Somit ist das exportierte XMI unterschiedlich und die Interoperabilität ist nicht mehr gegeben. Selbst bei Versionswechseln eines Tools des gleichen Herstellers können Probleme auftreten.

4 Component Architecture CA2.0

Unternehmen müssen sich durch immer schnellere Entwicklungen und sich stetig ändernde Gegebenheiten flexibel und schnell anpassen können. Bestehende Systeme werden weiter entwickelt und müssen in neue Systemlandschaften integriert werden. Neue Anwendungen werden erstellt und zu den schon existierenden hinzugefügt. Alle diese evolutionären Veränderungen führen im Laufe der Zeit zu heterogenen und dezentralen IT-Systemen, deren Komplexität steigt. Durch die Dezentralisierung kommt erschwerend noch die Anzahl der unterschiedlichen Netzwerkprotokolle, Plattformen und verwendete Betriebssystemen hinzu.

4.1 Einführung

Die Component Architecture CA2.0 von BMW ist eine komponentenbasierte Software Architektur. Aufgrund der immer steigenden Komplexität und sich stetig ändernden Entwicklungsstandards wird ein modellgetriebener Ansatz nach MDA in der CA2.0 verfolgt.
Durch die Einführung der CA wird ein konzernweiter Standard für die Entwicklung von verteilten Anwendungen definiert. Die Komponenten selbst sind technologie- sowie plattformunabhängig und können dadurch für unterschiedliche Plattformen verwendet werden. Durch die Einführung der CA soll sich eine Standardisierung und Steigerung der Wiederverwendung von Architekturen in den IT-Projekten bei BMW einstellen. Als Synergieeffekt soll sich der Entwicklungsprozess deutlich beschleunigen und somit die Projektdurchlaufzeiten herabsetzen. Eine detaillierte Ausführung ist in [AB_CA20] zu finden.

4.2 Design

Ein Design einer Architektur, anhand der es möglich sein soll komplexe verteilte Anwendungen zu entwerfen, muss bestimmten Anforderungen gerecht werden. Wichtige Anforderungen, die die Component Architecture berücksichtigt, werden hier aufgeführt:

- Skalierbarkeit
- Klare Trennung von Schichten der Fachlogik
- MDA-Unterstützung
- Durch Definition einer fachlichen Schnittstelle Wiederverwendbarkeit der Komponenten
- Sicherheit / Loadbalancing / Clustering

- Verfügbarkeit
- Plattformunabhängig

Damit diese Anforderungen erfüllt werden können, setzt sich die CA2.0 aus Komponenten zusammen. Komponenten sind in der CA eigenständige, konsistente Einheiten welche weitere Artefakte beinhalten. Komponenten werden in der CA2.0 ausschließlich über ihre fachlichen Schnittstellen verwendet.

Durch den Einsatz von Komponenten und Schnittstellen erreicht man eine Minimierung der Abhängigkeiten, Abstraktion, interne Kohäsion, Contract Base Design und Austauschbarkeit der Implementierung.

Eine weitere gängige Designentscheidung ist die Aufteilung der entwickelten Komponenten in logische Schichten. Jede definierte Schicht deckt einen Teil der Aufgabenbereiche der Gesamtanwendung ab und kann selbst aus mehreren Teilen bestehen.

Die CA2.0 definiert 3 Schichten (eng. Tiers), die Client- , Business- und Integration-Tier welche in der Abbildung 4.1 zu sehen sind. Die Business-Tier, die für diese Diplomarbeit relevant ist, wird nochmals in Ablaufsteuerung, Verhalten, Daten und Schnittstellen zu externen Komponenten aufgeteilt. Die hier vorgenommenen Aufteilungen sind nur logisch und nicht physisch zu sehen. Die Modellierungselemente lassen sich durch die logische Aufteilung gemäß ihren Aufgaben zuordnen.

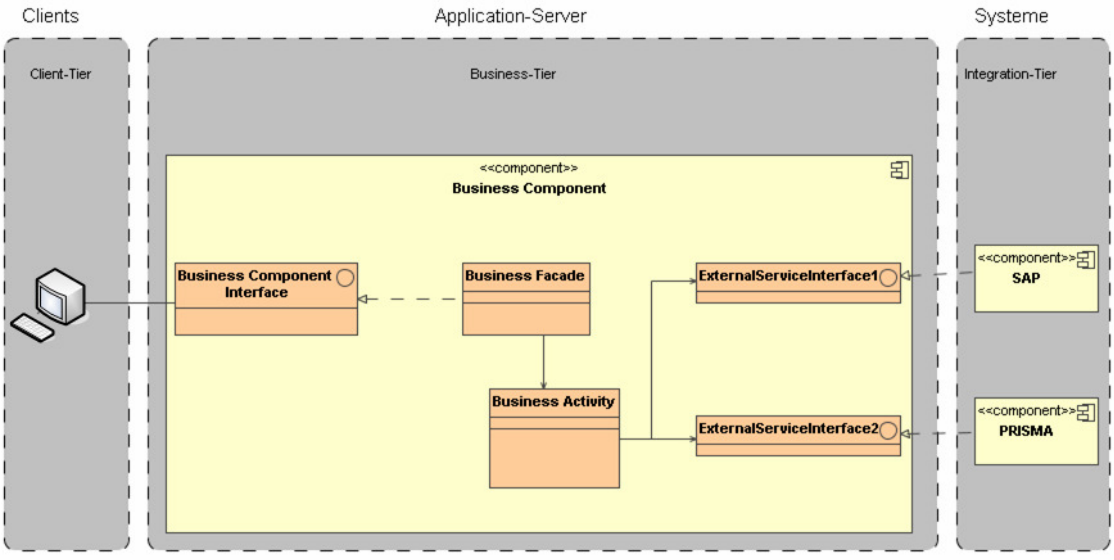


Abbildung 4. 1: Layer-Architektur der CA

4.3 Architektur der CA

Die Component Architecture besteht generell aus drei Elementen:

- Ein Modellierungsprofil (Designsprache), mit dessen Hilfe das PIM und somit das Anwendungsdesign auf Basis von UML modelliert werden kann,
- Transformationsregeln die das plattformunabhängige Modell auf das J2EE konforme Modell und anschließend auf Code abbilden und
- das CA-Framework, welches gewisse Basisfunktionalitäten bereitstellt.

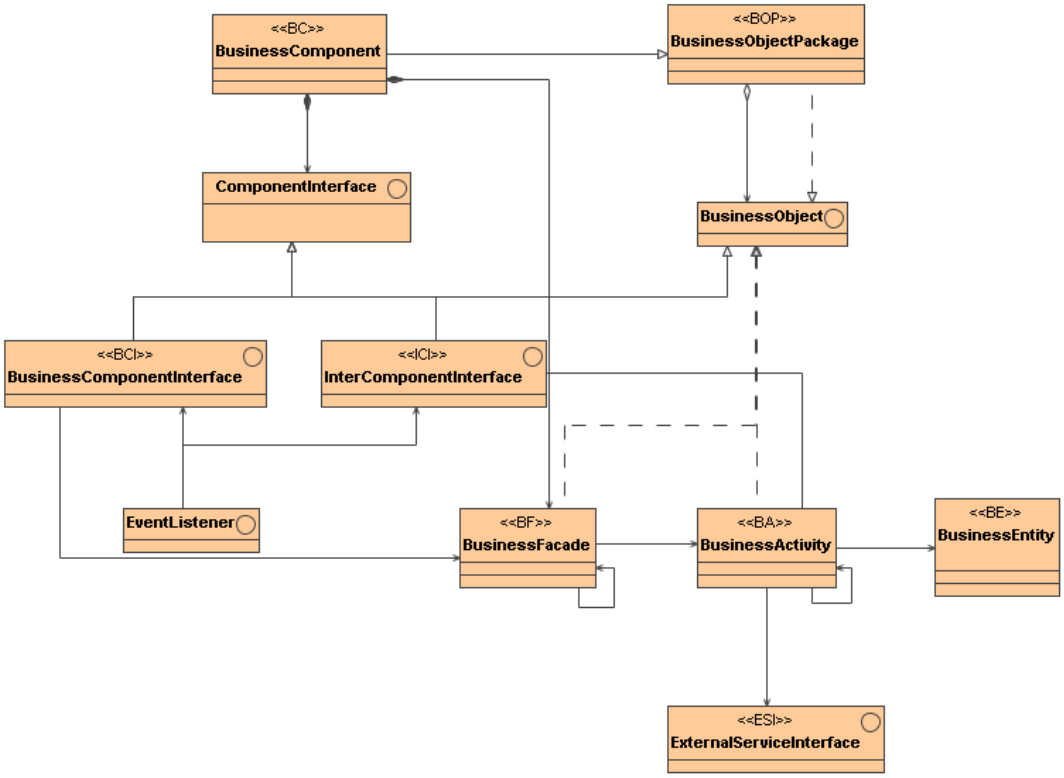


Abbildung 4. 2: Metamodell CA2.0

Wie oben schon erwähnt versteht die CA unter einer Komponente eine Art Container. In solch einem Container können andere Artefakte gruppiert werden und somit eine kohärente Einheit bilden. Im folgenden Schaubild ist eine Komponente, die in der CA als Business Component bezeichnet wird, zu sehen.

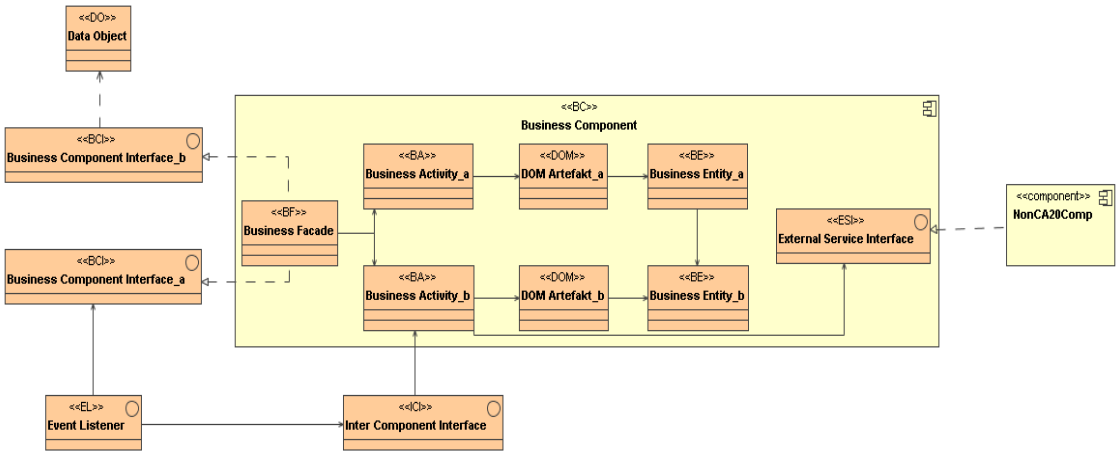


Abbildung 4. 3: CA2.0 Architektur

Die Modellierungselemente der CA2.0 sind einfache Elemente die für die Ablaufsteuerung, das Verhalten, Definition der Schnittsetellen und die Datenhaltung verantwortlich sind.
Bei näherer Betrachtung der CA2.0 fällt eine hohe Ähnlichkeit mit der Komponententechnologie Enterprise Java Beans auf. Dies ist kein Zufall, sondern gewünscht. Dennoch unterscheidet sich die CA2.0 zum Beispiel in der Namensgebung der Elemente und macht gewisse Einschränkung gegenüber EJB beim Entwurf. Durch die Anlehnung der CA2.0 an die Best Practises und Patterns wird das Mapping auf die EJB Technologie einfach.

Folgende Artefakte werden in der Component Architecture definiert:

BC

Die Business Component ist eine fachliche CA-Komponente. Sie bildet einen Container der verschiedene CA-Artefakte gruppieren kann. Man kann sagen, die Business Component ist einfach gesehen ein Strukturierungsmittel.
Eine BC besitzt mindestens eine fachliche Schnittstelle und deren Beschreibung. Diese Kombination entspricht der Spezifikation einer Komponente und muss gegeben sein. Zur Benutzung der Komponente müssen mindestens ein BCI und ein BF existieren.

BCI

Ein Business Component Interface ist eine fachliche Schnittstelle und repräsentiert die fachliche Funktionalität einer Komponente.
Die Schnittstelle definiert den Leistungsumfang und ermöglicht von extern die Möglichkeit zur Verwendung der Komponente und somit den Zugriff auf die Business-Tier. Ein Business Component Interface gehört zur Außenansicht einer Komponente. Das BCI wird in der Spezifikation übernommen und ermöglicht die Wiederverwendbarkeit.

ICI

Ein Inter Component Interface repräsentiert eine fachliche Schnittstelle, welche nur von fachlichen Komponenten oder Clients die in der gleichen Tier (Business-Tier) sind, benutzt werden darf und kann. Das ICI ist nur für lokale Komponenten eines Containers der Business-Tier sichtbar.

Komponenten in der gleichen Schicht können untereinander auf die Funktionalität anderer Komponenten zugreifen und diese verwenden. Allerdings sollen externe Komponenten nicht direkt auf Artefakte einer Komponente zugreifen, da sonst eine zu enge Kopplung zwischen diesen entsteht und die Implementierung nicht ohne weiteres ausgetauscht oder verändert werden kann. Abhilfe schafft hier das Inter Component Interface. Durch das ICI besteht eine feinere Sicht auf die Artefakte einer Business Component.

BF

Die Business Facade ist für die Steuerung einer Komponente zuständig und steuert den Use Case für die jeweilige Komponente. Eine Business Facade kann mehrere Business Activities verwalten. Zu ihren Funktionen gehört unter anderem die Zustandsverwaltung des Clients oder auch die Transaktionssteuerung. Deshalb kann man eine BF mit dem Decorator-Pattern vergleichen. Eine BF bzw. die dazugehörige Komponente befindet sich immer in einem konsistenten Zustand. Die Ausführung ihrer Methoden führt immer von einem konsistenten Zustand A in einen nächsten konsistenten Zustand B. Die BF ist nicht für die Implementierung der Geschäftslogik zuständig, sondern nur für die Koordination der Business Activities.

BA

Eine Business Activity ist für die Abbildung von Aktivitäten oder Teilprozessen zuständig. In der CA2.0 sind BA's grundsätzlich transient und zustandslos. Jede einzelne definierte Methode in der BA ist eine atomare Aktivität und wird entweder ganz oder gar nicht ausgeführt. Alle Methoden der BA werden in einem Transaktionskontext ausgeführt. Zugriff haben nur BF's oder BA's.

Die Methoden der BA sind ausschließlich für die Implementierung des Verhaltens zuständig.

BE

Business Entities sind für die Datenhaltung in der CA2.0 zuständig. BE's halten die Daten der zugehörigen Komponenten. Sie bestehen aus einer zusammengehörigen Gruppierung von Attributen. Außerdem kann eine BE Abfragefunktionalität beinhalten. Diese Methoden müssen jedoch statisch sein.

Auf Business Entities können Business Facades, Business Entities und Business Activities zugreifen. Wichtig hierbei ist, dass explizit nur BA's den Zustand einer BE verändern können. BF haben auf eine BE nur lesenden Zugriff.

EL

Ein Event Listener führt beliebige, synchrone Methoden eines Inter Component Interface oder eines Business Component Interface, asynchron aus. Der Event Listener ist zusammen mit dem BCI die einzigsten 2 Elemente die außerhalb der Business-Tier sichtbar sind. Der Event Listener ähnelt sehr dem ESI, allerdings ist die Richtung des Datenflusses genau umgekehrt. Er empfängt Daten von der Business-, Präsentation oder Integration-Tier und verarbeitet diese asynchron. Mit einer Komponente kommuniziert der Listener nur über ihre technischen Schnittstellen.

ESI

Bei der Entwicklung einer CA2.0-Anwendung ist die Integration von schon bestehenden Systemen bzw. Komponenten sehr wichtig. Für die Integration dieser bestehenden Systeme/Komponenten existiert das External Service Interface, welches eine fachliche Schnittstelle für externe Ressourcen anbietet. Durch die Implementierung dieser Schnittstelle ist die Anbindung CA2.0 nichtkompatibler Komponenten möglich.
Ein ESI kann nur von einer Business Activity verwendet werden und ist selbst zustandslos sowie transient.

DO

Das Data Object entkoppelt externe Clients von der internen Datenstruktur (BE's). Es repräsentiert die Parameter und Rückgabewerte der fachlichen Schnittstelle. Data Objects können als Rückgabewerte und Parameter von BCIs, BFs, BAs, ICIs und ESIs verwendet werden.

4.4 Entwicklungsweg

Der in der CA2.0 gewählte Entwicklungsweg basiert auf den Ideen der Model Driven Architecture der OMG. In den ersten Entwicklungsphasen einer Anwendung interessiert die später eingesetzte Technologie noch nicht. Die Konzentration ist in den ersten Zyklen vollständig auf die Geschäftslogik gerichtet. Es werden funktionale und nicht funktionale Anforderungen erarbeitet und diese mit den bekannten UML-Diagrammen modelliert und ausgedrückt. In dieser Phase sind die Diagramme (Use Case Diagramme, Key Abstraction) noch sehr abstrakt und eignen sich für die Kommunikation mit der Fachabteilung.

Mit den erarbeiteten Anforderungen und den noch sehr abstrakten Diagrammen wird ein Modell erstellt, welches einzig und alleine die funktionalen Anforderungen des Kunden umsetzen soll und von der eingesetzten Plattform zu diesem Zeitpunkt abstrahiert. Das Verhalten und die Geschäftslogik werden in solch einem Platform Independent Model abgebildet. Die iterative Analysephase endet im Design des PIM's.

Um eine lauffähige Anwendung des modellierten Modells zu erhalten, muss das Modell konkretisiert werden und somit plattformspezifische Anreicherungen erfahren. Dieser Schritt resultiert in einem Platform Specific Model. In diesem Modell sind dann je nach gewählter Plattform, technische J2EE- oder .NET-Komponenten zu erkennen. Außerdem verfügt das PSM über Metadaten die in der letzten Stufe der Generierung benötigt werden. In der folgenden Abbildung ist dieses Vorgehen noch einmal bildlich dargestellt.

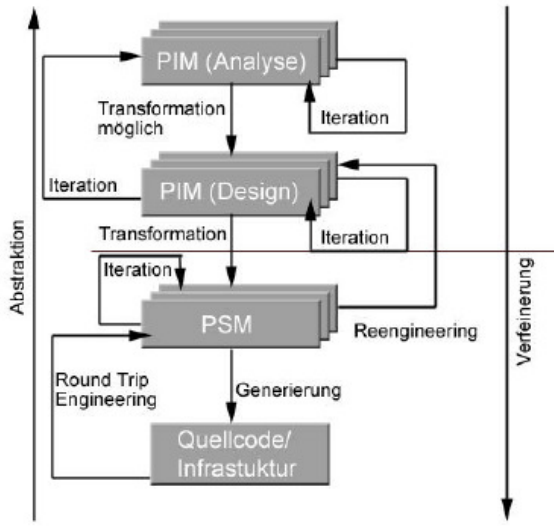


Abbildung 4. 4: Iterative Entwicklung der Modelle

Bei dem hier vorgestellten Entwicklungsweg wird eine CA Komponente mit einem Modellierungs-Tool im PIM modelliert und dann durch die Konkretisierung / Transformation in eine technologiespezifische Komponente überführt. Eine Komponente in der CA2.0 stellt einen Teil der funktionalen Anforderungen des Gesamtsystems zur Verfügung.

In der letzten Phase der Verfeinerung wird das PSM in den Quellcode überführt. Der Quellcode stellt den Implementierungsrahmen da, der noch mit fachlichem Code ergänzt werden muss. Die CA-Artefakte werden im Implementierungsrahmen auf zumeist mehrere OO-Konstrukte (Class, Interface, Package) abgebildet. Dadurch sind die Designelemente der CA nur noch durch die Namensgebung zu erkennen.

4.5 Abbildung der CA2.0 auf EJB2.1

Die Abbildung der fachlichen PIM-Elemente in das J2EE-PSM erfolgt durch Transformation. In den Transformationsvorschriften werden unter anderem folgende Vorschriften definiert:

- Einheitliche Package-Struktur
- Definition der verwendeten J2EE-Patterns
- Namenskonvention
- Abhängigkeiten zwischen PSM-Elementen

Eine technologiefreie Komponente im plattformspezifischen Modell wird dabei auf mehrere technische Komponenten (EJB's) im PSM abgebildet. Die Abbildung der plattformunabhängigen Artefakte des PIM erfolgt unter dem Einsatz der J2EE-Pattern. In der Abbildung 4.5 ist die Abbildung einer technologieneutralen Business Activity im PIM, auf eine technische J2EE Bean im PSM zu sehen. Für jedes in der Designsprache vorkommende Modellelement gibt es technologiespezifische Vorschriften dieser Art.
Es soll hier nicht näher auf die verwendeten Design Patterns und deren Anwendung auf die CA2.0-Artefakte in der Abbildung eingegangen werden. Konkrete Information über die Anwendung und den Einsatz der Pattern findet man in [LB_CA20] oder teilweise im Kapitel Realisierung (vgl. Kapitel 7).

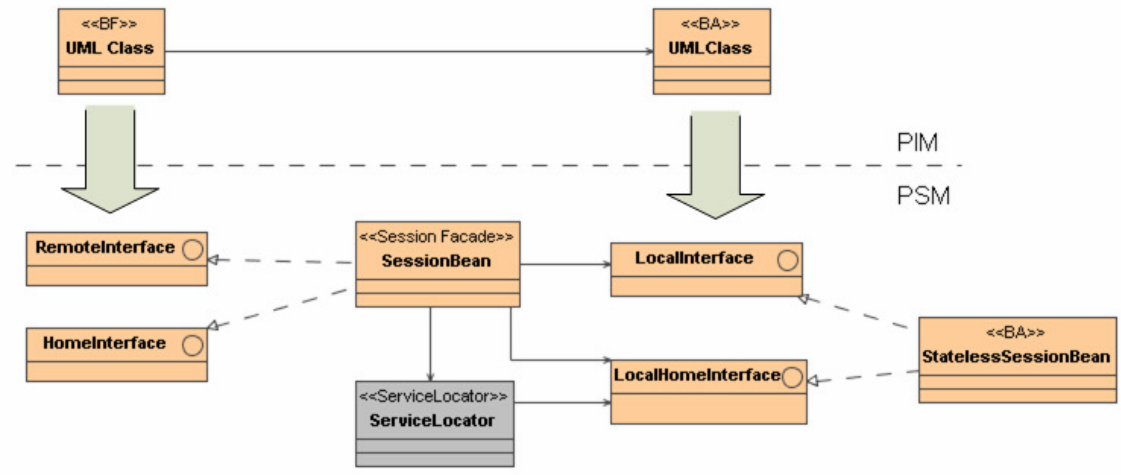


Abbildung 4. 5: Transformationsregel

Bei der Transformation des Platform Independent Model werden die Inhalte einer Komponente eingelesen und anschließend unter Einbindung der Abbildungsvorschriften, die im Metamodell hinterlegt sind, transformiert.

Das Metamodell der CA2.0 definiert die Stereotypen die bei der Erstellung des Anwendungsdesign verwendet werden dürfen. Die Transformation ist iterativ wiederholbar, da der eingesetzte Transformator den manuell hinzugefügten Code nicht überschreibt. Somit wird der Entwickler bei der Ableitung der Artefakte aus den fachlichen Anforderungen unterstützt.

Der nächste Abschnitt in dieser Diplomarbeit behandelt den von BMW entwickelten Generator. Im Zuge dieses Kapitel wird näher auf die Transformationen, das Metamodell der CA2.0 und auf die Abbildungsvorschriften eingegangen.

Die in diesem Kapitel sicherlich aufgekommenen Fragen, unter anderem wie das plattformneutrale Anwendungsdesign in den Sourcecode überführt wird, wird im Folgenden beantwortet.

5 BMW - Transformator

5.1 Einführung

Im Zuge der Entwicklung der Component Architecture bei BMW wurde ein Modelltransformator entwickelt, der den MDA-Ansatz den die CA2.0 folgt, gerecht wird. Der von BMW implementierte Transformator wurde als Plugin für ein grafisches UML-Modellierungstool (Together) entwickelt. Somit kann das in Together modellierte Anwendungsdesign (PIM) in ein plattformspezifisches Modell (PSM) automatisiert überführt werden.

Die Transformation vom PIM ins PSM erfolgt auf Basis eines Metamodells, welches die Abbildung auf die J2EE-Elemente beschreibt. Der Transformator wird durch die im PIM verwendeten Stereotypen, die im Metamodell definiert sind, gesteuert. Im Metamodell sind hinter den Stereotypen PSM-Elemente spezifiziert, welche bei der Transformation für die korrespondierenden PIM-Artefakten erzeugt werden.

5.2 CA2.0 Metamodell (Transformationsvorschriften)

Das in Together erstellte Metamodell für die CA2.0 Transformationen ist an das didaktisch korrekte CA2.0 Metamodell, welches im vorhergehenden Kapitel aufgeführt wurde, angelehnt. Es werden in diesem Metamodell keine konkrete Abhängigkeiten oder Modellierungsvorschriften für das Anwendungsdesign modelliert, sondern es dient den Abbildungsvorschriften. Vor der Transformation wird das Metamodell und die in diesem befindlichen Abbildungsvorschriften vom Transformator eingelesen.

Die Abbildung eines PIM-Elements ist in den Packages des Metamodells hinterlegt. Bei dem Metamodell handelt es sich um Packages mit stereotypbezogenen Abbildungsvorschriften. Über den Stereotyp der PIM-Elemente wird die entsprechende Abbildung des Metamodells ausgewählt und das Element transformiert.

Ein im PIM modelliertes CA-Element wird auf ein PSM-Package mit einer beliebigen Anzahl an Elementen abgebildet. Durch die 1:N Beziehung zwischen PIM und PSM-Elementen ist es möglich, bekannte Design Patterns in die Transformation einfließen zu lassen.

In der folgenden Abbildung ist das Metamodell für die Transformation mittels BMW-Generator aufgeführt.

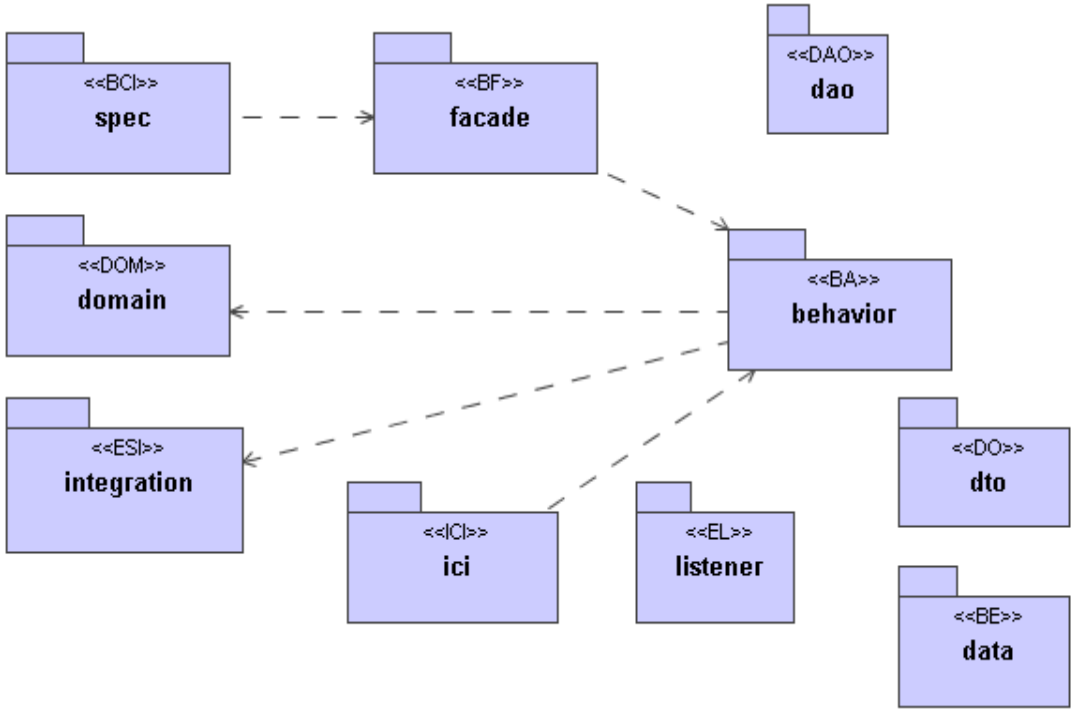


Abbildung 5. 1: Metamodell BMW-Transformator

Die Properties, wie zum Beispiel Attribute und Methoden, des eingelesenen Modellelements werden auf die Platzhalter (CA-Model-Properties) in den Transformations-Elementen übertragen und dann in das PSM übernommen. Das Einlesen der Elemente erfolgt kontextabhängig. Somit ist es möglich, dass ein Attribut welches in einem PIM-Element modelliert wurde auf eine Methode abgebildet wird. Den Kontext kann man in der Transformationsvorschrift über die Angabe eines Stereotyps festlegen. Im Abschnitt „Ablauf einer Transformation“ ist dieses Vorgehen zu sehen.

5.3 Metamodellerweiterung

Bei Bedarf ist es möglich, das mit der CA2.0 ausgelieferte Referenz-Metamodell selbst zu erweitern oder anzupassen. Für die Erweiterungen müssen neue Stereotypen und deren Abbildungen angelegt werden.
Folgende Schritte sind für die Definition einer neuen Abbildungsregel auszuführen:

- Es muss im Package Metamodell ein zusätzliches Package mit eindeutigem Stereotype angelegt werden
- In dem neuen Package werden die Klassen und Interfaces, die bei der Generierung erzeugt werden sollen, erstellt

Nach Ausführung der aufgeführten Schritte wird ein Modellelement im PIM, welches mit dem Stereotyp des neuen Package im Metamodell angelegt wurde, unter Einsatz der Abbildungsvorschrift, in das PSM transformiert.

5.4 CA-Model

Nach [Doc_Meta] ist das CA-Model, welches dem BMW-Transformator für die Instanziierung zugrunde liegt, ein in Java implementiertes Modell das keine Nachimplementierung des MOF Modells ist. Das CA-Model ist ein auf das Einsatzgebiet zugeschnittenes Modell, welches stark vereinfacht ist. Es sind alle nötigen Informationen für die Transformation von den typischen OO-Strukturen wie Assoziation, Aggregation, Klasse, Interface oder Package hinterlegt.
Das CA-Model ist keine direkte Abbildung der CA2.0 Artefakte und somit völlig unabhängig. CA2.0 Artefakte werden nicht durch konkrete CA-Model Elemente, sondern durch ihre Instanzen, abgebildet. Durch diese Festlegung ist das Metamodell durch beliebige Stereotypen erweiterbar und muss nicht angepasst werden. Zum Beispiel wird ein Stereotyp <<*BA*>> als eine CA-Class instanziiert.
Eine Instanz des CA-Models repräsentiert somit bei einem Transformationsvorgang das PIM.

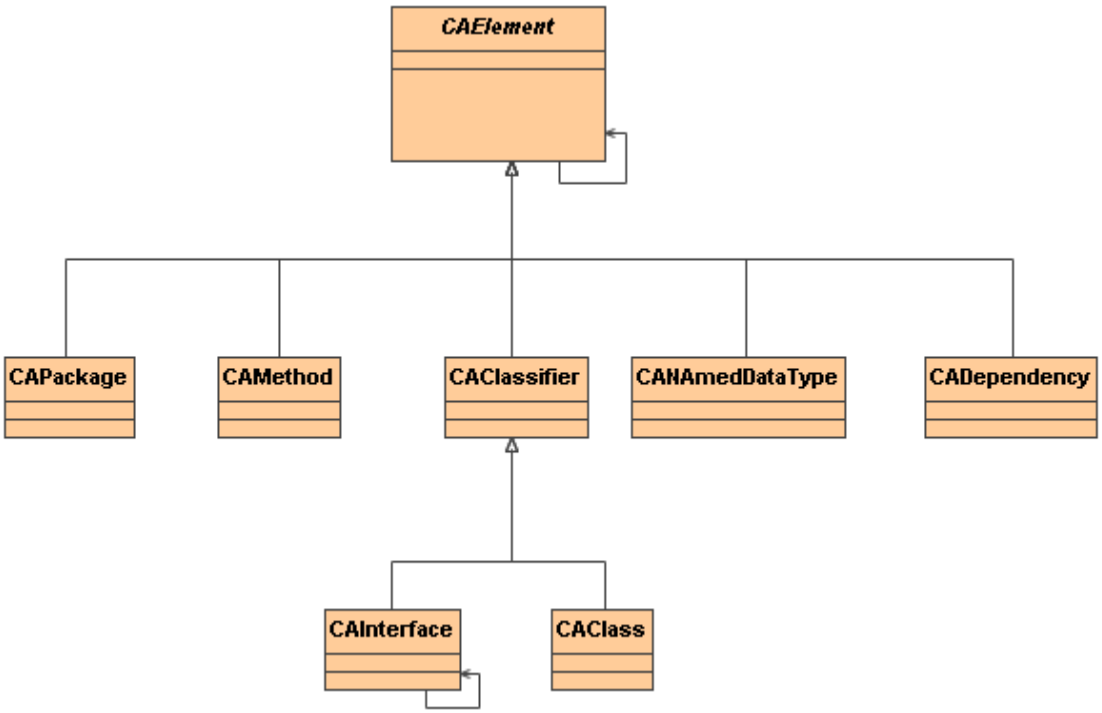


Abbildung 5. 2: CA-Modell des Transformators

Elemente des CA-Model und deren Beschreibung:

CAElement

Das CAElement definiert die Grundfunktionalität aller im CA-Model vorkommenden Elemente. Die CAElement Klasse ist eine abstrakte Klasse.

CAPackage

Ein CAPackage gruppiert CAElemente.

CAClassifier

Hier sind die Grundfunktionalität von CAClass und CAInterface hinterlegt.

CAClass

Repräsentiert eine Klasse des PIM, PSM oder Metamodells.

CAInterface

Repräsentiert ein Interface des PIM´s, PSM´s oder Metamodells.

CAMethod

Repräsentiert eine Methode des PIM´s, PSM´s oder Metamodells.

CANamedDataType

Repräsentiert Attribute einer CAClass oder CAInterface und Parameter oder Rückgabewerte einer CAMethod.

CADependency

Das CADependency Element bildet Association ab. Mit diesem Element sind Aggregationen, Associationen und Kompositionen möglich.

Alle CA-Model Elemente können ihren Zustand in Key-Value Paare (Properties) darstellen. Dieser Mechanismus wird für die Platzhalter im Metamodell, die bei der Transformation durch die Properties der Elemente ersetzt werden, benötigt. Mit diesem Mechanismus kann man einfach über die Platzhalter auf den Zustand der Elemente zugreifen.

5.5 Transformationsablauf

Bei der Transformation liest der Transformator zuerst die Inhalte eines PIM-Package ein und bildet diese dann auf die jeweilig passende CA-Elemente ab. Nach diesem Schritt liegt das PIM als Instanzengeflecht im Speicher vor. Um an den Zustand der jeweiligen PIM-Elemente zu gelangen wird beim BMW-Generator von einem direkten Zugriff auf die Methoden der CAElemente abgeraten und der Weg über die Platzhalter im Metamodell empfohlen.
Wie oben schon erwähnt sind die Abbildungen für die PIM-Elemente im Metamodell definiert welches vor der Transformation eingelesen wird. Der Transformator wählt für den Stereotype des jeweiligen Elements die passende Abbildung und transformiert das PIM-Element in das PSM.

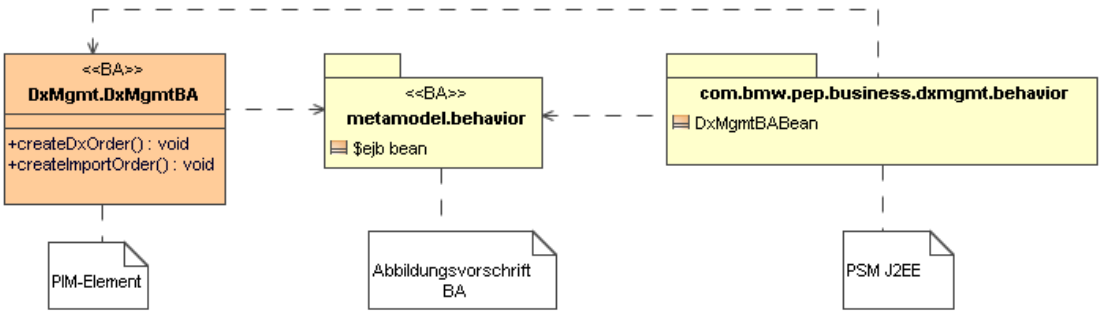


Abbildung 5. 3: Mapping Stereotypen auf ein Metamodell

In dem hier aufgeführten Beispiel kann man das Mapping eines PIM-Element, welches mit dem Stereotype *<<BA>>* markiert ist, auf die PSM-Abbildung unter Verwendung der im Metamodell definierten Abbildungsvorschrift sehen.

Es besteht des Weiteren noch die Möglichkeit Attribute oder Methoden in einem PIM-Element kontextabhängig mit einem Stereotyp zu verknüpfen. Ein häufig aufkommender Fall ist der, dass aus einem Attribut nicht nur eine Membervariable, sondern auch eine Getter- und Setter-Methode transformiert werden soll. In Listing 5.1 ist die angepasste Abbildungsregel für eine Getter-Methode aufgeführt, außerdem kann man die Platzhalter, welche bei der Transformation ersetzt werden erkennen:

```
package metamodel.newlayer;
public class $custom_name {

/**
* @stereotype attribute
*/
private $canameddatatype_attributetype $canameddatatype_name;

/**
* @stereotype attribute
*/
public $canameddatatype_attributetype get$canameddatatype_name(){
return this.$canameddatatype_name;
}
```

Listing 5. 1: Angepasste Abbildung

Nachdem alle Methoden die für die Abbildung eines PIM-Attributs verwendet werden mit dem Stereotyp <<*attribute*>> markiert wurden, wird nach der Transformation ein Attribut auch als Getter im Code vorhanden sein. Nach diesem Prinzip werden die Abbildungen definiert.

6 openArchitectureWare

6.1 Einführung

Die b+m Informatik AG[6] hat mitten der 90er Jahre mit dem Generator Framework ein Produkt entwickelt, welches in der architekturzentrierten Anwendungsentwicklung zum Einsatz kommt. Die AG wurde 1994 mit Hauptsitz in Melsdorf/Kiel gegründet. Das Geschäftsfeld erstreckt sich auf die Planung, Durchführung und das Management komplexer Prozesse im IT-Bereich.

Seit Ende 2003 wird das open b+m Architecture Framework auf der openArchitectureWare[7] Seite gehostet und unter der LGPL(Gnu Lesser General Public License) auf der opensource-Plattform SourceForge.net[8] veröffentlicht. In diesem Zuge wurde auch zur Verdeutlichung des opensource Projektes der Name um das Adjektiv „open“ ergänzt.

openArchitectureWare ist ein Framework zur modellgetriebenen Entwicklung. Es basiert auf einem modularen MDA-Generator-Framework, das arbiträre Formate, Metamodelle und diverse Ausgabeformate unterstützt.
Das modulare Konzept ermöglicht eine spezifische und individuelle Anpassung in den unterschiedlichen Projekten. Zu den Modulen des Frameworks zählt unter anderem ein Metamodell-Generator, welcher es erlaubt, aus Modellen Modelle oder Code zu transformieren.

Über die Jahre hat sich das openArchitectureWare Generator Framework zu einer eigenen kompletten Tool-Suite entwickelt. Für die Erstellung einer generativen Architektur bedient man sich einfach an den Modulen der Tool-Suite und kombiniert diese wie man es benötigt. Deshalb kann man oAW auch als „tool for building MDSD/MDA Tools” bezeichnen.
Im Mittelpunkt der Module findet sich ab der Version 4.0 eine Workflow-Engine wieder, die die gewachsene Anzahl an Modulen zusammenhält und steuerbar macht. Durch Definition von Workflows[9] ist der Generierungsprozess komfortabel steuerbar. Mit diesem Mechanismus sind diverse Module sowie externe Programme in den Transformationslauf integrierbar.
Vordefinierte Workflows, die zum Einlesen und Instanziieren der Modelle oder auch zum Transformieren zu Code genutzt werden können, bestehen bereits und werden mit ausgeliefert.

[6] b+m Informatik AG - http://www.bmiag.de (Stand 10 Oktober 2006)
[7] openArchitectureWare - http://www.openarchitectureware.org (Stand 10 Oktober 2006)
[8] SourceForge - http://sourceforge.net/index.php (Stand 10 Oktober 2006)
[9] Workflow - Eine vordefinierte Abfolge rechnergestützt ablaufender Aktivitäten

6.2 Generative Development Process

6.2.1 Einführung

Architekturzentrierte Softwareentwicklung und entsprechende Vorgehensmodelle haben sich in den letzten Jahren in der Softwareentwicklung etabliert. Leider wird in den meisten Fällen das Potenzial, welches hinter diesen Vorgehen steckt, nur zu einem geringen Teil ausgeschöpft. Um diesem Problem zu begegnen, wurde bei der b+m Informatik AG der architekturzentrierte Generative Development Process (GDP) zur Unterstützung des Entwicklungsprozesses mit dem MDA konformen Generator Framework entwickelt.
Dieser in der Praxis bewährte Prozess ist nicht als Konkurrent bestehender OO-Entwicklungsprozesse, sondern als Ergänzung und Verfeinerung zu sehen. Der GDP ist speziell auf die Anforderungen bei der generativen Entwicklung zugeschnitten und hilft Entwicklungszeit und Entwicklungsaufwand zu reduzieren. Als Synergie verbessert der Prozess, durch die Generierung des Entwicklungsrahmen und der Fokussierung der Entwicklung auf Anwendungsdesign sowie Architektur, die Software-Qualität.
Zwei wesentliche Eigenschaften des Prozesses sind, dass keine starren Entwicklungsumgebungen verwendet oder vorausgesetzt werden, sondern beliebige Ziel-Architekturen, Sprachen, Schnittstellen und Laufzeitkomponenten unterstützt werden. Außerdem wird das PIM mittels Schablonen (Templates) direkt in Sourcecode transformiert und im Gegensatz zur MDA, der pragmatische Weg, ohne den Zwischenschritt über ein Platform Specific Model (PSM) gewählt.

6.2.2 Grundlagen

Bei dem GDP handelt es sich um ein iteratives, inkrementelles Vorgehensmodell, d.h. dass die einzelnen Teilprozesse immer wieder iterativ durchlaufen werden können und so wiederverwendbare Bausteine darstellen.

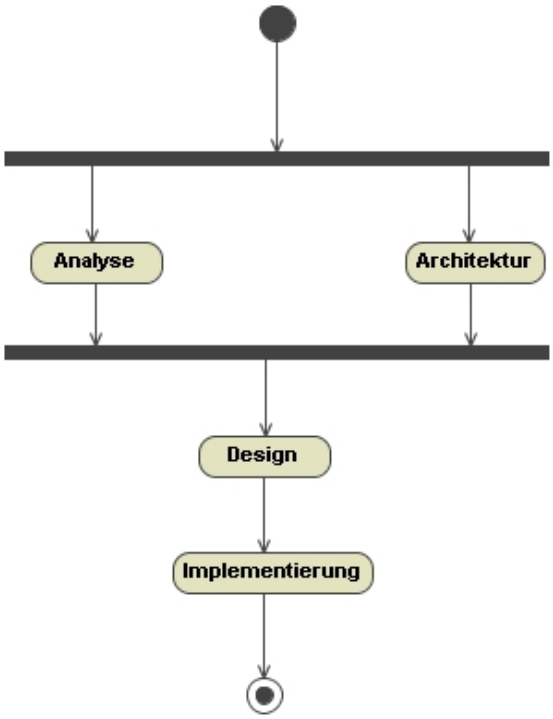


Abbildung 6. 1: Zentrales Grundschema des Vorgehensmodells

Die Entwicklung des fachlichen Modells (Analyse) und technischen Modells (Architektur) kann in zwei getrennt parallelen Prozessen erfolgen. Im Design wird die Fachlichkeit mit der von der Technik zur Verfügung gestellten Sprache (Designsprache) verbunden und ausgedrückt. Anschließend wird das Design automatisch generatorgestützt in einen Implementierungsrahmen überführt.

6.2.3 Architektur

Der Architektur-Teilprozess ist ein spezifisches Merkmal des GDP. Er zeigt den Aspekt der Architekturzentrierung und die Erstellung von Anwendungsfamilien auf.

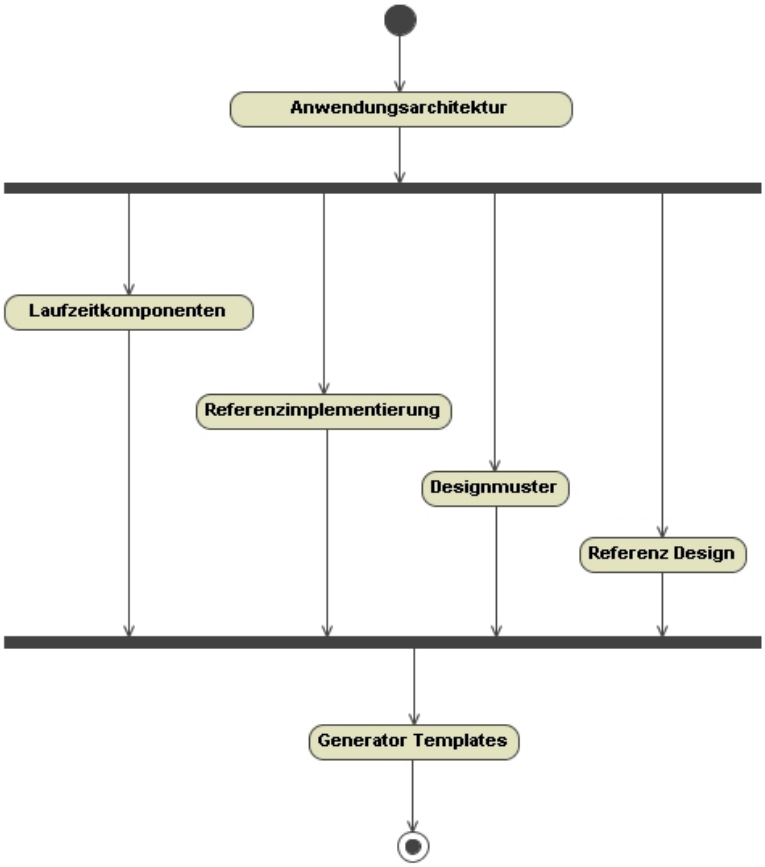


Abbildung 6. 2: Architektur Teilprozess GDP

Anwendungsarchitektur konzipieren

Wichtig im Konzept der Anwendungsarchitektur ist, dass es frei von jeglicher Fachlogik bleibt und somit auch für Projekte, die sich fachlich in einem völlig anderen Kontext befinden, verwendet werden kann. Dies ermöglicht erst die Bildung einer Anwendungsfamilie auf der gegebenen Anwendungsarchitektur.

Typische Aspekte, die sich in einem Architektur-Konzept wieder finden sollten, sind zum Beispiel Technische Schichtung, Locking, Persistenz, Transaktionalität etc.

Laufzeitkomponenten erstellen

Unter dem Begriff Laufzeitkomponente finden sich Frameworks, Basisklassen, Bibliotheken, Utilities etc. wieder. Also technische Komponenten die nicht von der fachlichen Ausprägung beeinflusst werden. Diese Komponenten sind typische Kandidaten zur Wiederverwendung selbst über Anwendungsfamilien hinweg.

Beispiele hierzu sind:

- Persistenz-Framework
- Basisklassen für GUI
- Log-Framework
- Security-Framework

Referenzimplementierung erstellen

Die Referenzimplementierung soll nicht als Quelle für Anregungen bei der Entwicklung dienen. Vielmehr dient sie zusammen mit dem Referenz-Design zur Veranschaulichung und der Anwendung der zugrunde liegenden Designsprache.
Diese Implementierung verdeutlicht den Übergang vom Design zur Realisierung auf der gegebenen Architektur.
In der Referenzimplementierung geht es ausschließlich um die Architektur und nicht um den fachlichen Inhalt. Es sollte ein fachlich trivialer Use-Case verwendet werden, um die Benutzung der Laufzeitkomponenten und der Architektur aufzuzeigen. Aus der Referenzimplementierung werden in einer späteren Iteration die Templates für den Generator abgeleitet.

Designmuster identifizieren

In diesem Schritt wird die architekturzentrierte Designsprache erstellt. Dieser Schritt stellt die wesentliche Abstraktion vom Konzept der Architektur zum Konzept der Anwendungsfamilien dar. Der Entwurf einer Designsprache ist im GDP ein weiterer iterativer Prozess. Die Designsprache sollte so weit wie möglich abstrakt und frei von technologischen Begriffen sein, um ein möglichst technologieunabhängiges PIM zu erstellen. Somit sieht man dem PIM erst im Kontext einer Architektur seine technische Umsetzung an und die Designsprache wird für unterschiedliche Anwendungsfamilien verwendbar.

Referenz Design erstellen

Laut [b+m03] soll das Referenz-Design und die Referenz-Implementierung zusammen exemplarisch die Syntax und Semantik der Designsprache bis auf Implementierungsebene zeigen. Das Referenz-Design ist ein Teilsatz der Designsprache mit der eine Referenzimplementierung ausgedrückt wird.

Generator Templates ableiten

Durch die Templates wird das Anwendungsdesign auf eine gegebene Architektur gebunden. Für eine gegebene Designsprache und Architektur werden für das Anwendungsdesign Templates erstellt. Mittels des gegebenen Programmiermodells (z.B. Java) kann dann das

Anwendungsdesign durch den Generator automatisch in einen Implementierungsrahmen überführt werden.

6.3 Funktionsweise openArchitectureWare

openArchitectureWare ist keine vollständige MDSD/MDA-Entwicklungsumgebung, in der man von PIM-Design über Transformation bis hin zur Codebearbeitung alles abwickeln kann. Auch wenn in der aktuellen Version 4.1 des Frameworks, die Integration in Eclipse weit fortgeschritten ist, sind immer noch diverse weitere MDA-Tools von Nöten, um eine generative Toolarchitektur zu bilden.
Die Minimalinstallation von openarchitectureWare besteht aus zwei Plugins, welche die nötigen Core-Plugins für oAW beinhalten:

- org.openarchitectureware.core.feature.source
- org.openarchitectureware.core.libraries

Das Plugin org.openarchitectureware.core.feature.source beinhaltet die Workfow-Engine, EMF Integration und die Engine für die propritäre Scriptsprache Xpand. Im zweiten Plugin org.openarchitectureware.core.libraries sind die nötigen Bibliotheken für das Feature-Plugin hinterlegt.

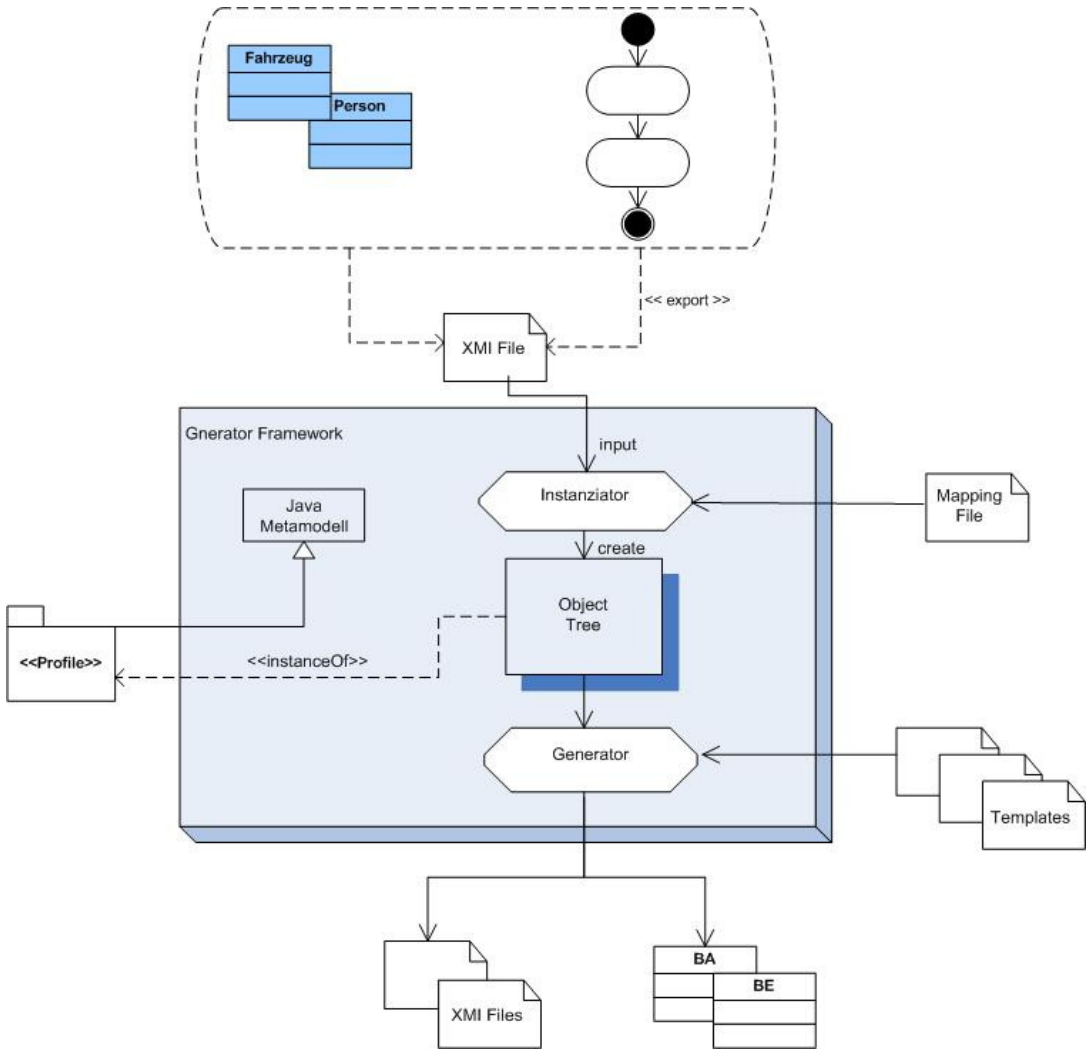


Abbildung 6. 3: Funktionsweise openArchitectureWare Framework

Die Abbildung 6.3 zeigt den generellen Aufbau des Generator Frameworks und das Zusammenspiel der einzelnen Artefakte.

Der Generator besitzt eine flexible Schnittstelle über die Anwendungsmodelle eingelesen werden können. Somit ist das Generator Framework unabhängig von der verwendeten Modellierungssprache und dem verwendetem Tool. Die einzige Anforderung die besteht ist, dass die Modelle in dem von der OMG spezifiziertem Austauschformat XMI (XML Metadata Interchange) vorliegen. Durch eine XML Mapping-Datei, die je nach verwendetem Modellierungstool (Together, MagicDraw, Poseidon etc.) spezifisch ist, werden dann die Elemente des Anwendungsdesigns (PIM) in ein instanziiertes Java-Metamodell überführt.

Der Generator benötigt zusätzlich zum Anwendungsdesign (PIM) eine Java-Implementierung der verwendeten Designsprache (Metamodell). Eine Java-Implementierung des UML-Metamodells für Klassendiagramme, Aktivitätsdiagramme und Zustandsdiagramme ist im Generator Framework beinhaltet und erfordert keine eigene Implementierung. Durch Spezialisierung (Stereotypes) kann das mitgelieferte Metamodell erweitert werden. Somit ist es

möglich das Metamodell auf die zugrunde liegende Domäne anzupassen. Durch die Java-Implementierung des Metamodell sind Änderungen und Erweiterungen in den Metaklassen selbst durchführbar. Von diesem Vorgehen wird jedoch abgeraten. Generell sollte das Metamodell nicht verschmutzt werden. Andere Lösungsansätze für Erweiterungen der Designsprache werden weiter unten aufgeführt.
Der vorgestellte Mechanismus ermöglicht es, jede beliebige Designsprache und Diagrammtypen zu unterstützen, sofern eine Java-Implementierung dafür vorliegt. Im Gegensatz zu anderen auf dem Markt befindlichen MDA-Werkzeugen werden nicht nur statische Diagramme (Klassendiagramm), sondern auch dynamische Diagramme wie zum Beispiel Aktivitätsdiagramme oder Sequenzdiagramme unterstützt.

openArchitectureWare benötigt das Java-Metamodell zur Instanziierung des Anwendungsdesign. Dieses wird vom Generator eingelesen und zur Instanziierung des technischen Anwendungsdesign (PIM) verwendet. Während des Instanziierungsprozesses werden die Model-Elemente des Anwendungsdesign nach den Vorschriften des Meta-Mappings auf das instanziierte Metamodellgeflecht abgebildet. Der Generator sucht für jedes (UML) Element in der XMI-Anwendungsdesign-Datei eine entsprechende Java-Klasse, mit der er das Element instanziieren kann. Ist ein Element mit einem Stereotyp gekennzeichnet, so sucht der Generator in der Metamapping-Datei (metamapping.xml) nach einer entsprechenden Implementierung, die der User zur Verfügung stellt. Falls keine Instanziierungsvorschrift definiert ist, wird der Standard-Metatype verwendet.

Für die Transformation des instanziierten Anwendungsdesign zu Code, entwickelte die b+m Informatik AG eine eigene Templatesprache namens Xpand. Diese ermöglicht eine automatische Umsetzung eines Anwendungsdesigns in einen Implementierungsrahmen auf einer definierten Anwendungsarchitektur. Bei der Templatesprache wurde auf eine einfache und für die Codegenerierung passende Syntax geachtet, um eine geringe Einarbeitungszeit zu gewährleisten.
Das Generator-Backend bindet die Templates dynamisch zur Laufzeit an das instanziierte Anwendungsdesign und ermöglicht dadurch die Navigation.
Um einen iterativen Entwicklungsprozess zu ermöglichen, werden frei definierbare geschützte Bereiche (protectedRegion) in der Templatesprache Xpand unterstützt. Durch geschützte Bereiche ist es möglich in den generierten Code eigenen Code einzubringen, ohne dass dieser bei einem erneuten Generierungslauf verworfen wird. Zu diesem Zweck werden in den Templates Geltungsbereiche mit speziellen XPand-Schlüsselwörtern angelegt.

Dieser Ansatz bringt allerdings einige Nachteile mit sich:

- die Komplexität des Generator steigt, weil dieser für die Verwaltung, Erkennung und Erhaltung der Regionen zuständig ist
- es ist nicht immer einfach die protectedRegion konsistent zu halten. Daher kann es in der Praxis dazu führen, dass Code verloren geht oder überschrieben wird
- Grenzen zwischen generierten und handgeschriebenen Code verlaufen und tragen nicht zur Übersichtlichkeit bei

Aufgrund dieser aufgeführten Punkte sollte eine andere Lösung zur Integration manuell geschriebener Code gewählt werden.
Eine verbreitete Lösung ist, die generierten Artefakte in eigenen abstrakten Klassen/Files zu halten. Manuell implementierte Artefakte erben von den generierten Klassen und überschreiben deren Methoden. Vom System werden dann die nicht abstrakten Klassen, also die die manuell implementiert wurden, instanziiert. In den manuell hinzugefügten Klassen kann dann der leere Implementierungsrahmen der abstrakten Klassen mit Business Logik gefüllt werden.

Dieser vorgestellte Ansatz ist sehr elegant mittels den Gang-of-Four patterns [GHJ+94] umsetzbar. Das in der Diplomarbeit verwendete Pattern ist in der folgenden Abbildung zu sehen.

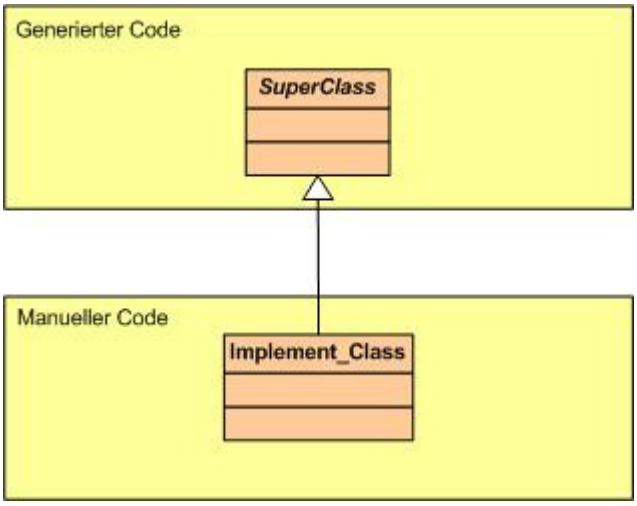


Abbildung 6. 4: Alternatives Konzept zu protected Region

Die Implementierungsklasse erbt von der generierten Elternklasse und überschreibt deren Methoden.

6.3.1 Metamodellierung

6.3.1.1 Modellvalidierung

Mit dem Einsatz von Model Driven Software Development verschiebt sich der Schwerpunkt der Entwicklung, weg von der rein fachlichen Implementierung des Codes, hin zur Modellierung des Anwendungsdesigns auf Basis des dazugehörigen Metamodells. Durch die Generierung des Implementierungsrahmens verringert sich der Zeitaufwand der Programmierung in einem hohen Maße.
Damit aber wie bisher in konventioneller Softwareentwicklung effektiv gearbeitet werden kann, müssen bestimmte Werkzeuge auch schon während der Designzeit vorhanden sein. Ein schwerwiegendes Thema ist hier die Unterstützung der Modellvalidierung. Modelle müssen gegen Constraints, die im Metamodell deklariert wurden geprüft werden können. Somit ist der Designer gezwungen korrekte Modelle zu entwickeln. Je früher die Modellvalidierung in den Entwicklungsprozess eingebunden wird, desto geringer ist die Wahrscheinlichkeit Fehler in das Modell zu modellieren.

Auch das openArchitectureWare Framework unterstützt Modellvalidierung. In früheren Versionen des Frameworks wurde für die Validierung die korrespondierende Metamodellklasse durch das Überschreiben der checkConstraint-Methode verunreinigt. Seit der Version 4.0 ist auch hier eine saubere Lösung eingeführt worden. Mit der Sprache Check werden die Überprüfungen der Constraints in eigene Dateien mit der Endung „chk“ ausgelagert. In den Check-Dateien werden die Constraints definiert, welche über den Workflow in den Generierungsprozess eingebunden und zur Laufzeit ausgeführt werden. Es ist dem Entwickler überlassen, ob dieser bei einem Verstoß eines definierten Constraints die Generierung unterbricht oder nur eine Warnung auf der Konsole ausgibt. Nähere Betrachtung wie Constraints definiert werden folgt weiter unten.
Die von openArchitectureWare unterstützte Modellvalidierung findet im Entwicklungsprozess zur Generierungszeit statt. Damit keine architektonischen Fehler im Design auftreten können, muss die Validierung zur Designzeit in den Entwicklungsprozess eingebunden werden. In Kapitel 8 wird aufgezeigt wie Validierung zur Designzeit möglich ist.

6.3.1.2 Metamodellinstanziierung

Für die Bearbeitung des Anwendungsdesigns durch Templates muss das dazugehörige Metamodell instanziiert werden. Die Metamodellinstanziierung wird über eine Mapping-Datei

gesteuert. Das Generator Framework legt mit den Angaben in der Mapping-Datei das Anwendungsdesign auf ein Metamodellgeflecht im Speicher zur Laufzeit an. Die Mapping-Datei beinhaltet hierzu die nötigen Vorschriften im XML-Format. Im nachfolgenden Listing ist zu erkennen, wie Anwendungsdesign-Elemente auf das vom Entwickler erweiterte Metamodell gemappt werden.

Nachdem das Anwendungsdesign als Instanzengeflecht vorliegt, kann das Generator-Backend dynamisch die korrespondierenden Templates an die passenden Modell-Elemente binden. Durch diesen Mechanismus ist es dem Entwickler möglich, über das instanziierte Anwendungsdesign zu navigieren und die entsprechende Generierung vorzunehmen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MetaMap SYSTEM
"http://www.openarchitectureware.org/dtds/metamap.dtd">
<MetaMap>
      <Mapping>
            <Map>DO</Map>
            <To>pep_ca20_uml_meta.DO</To>
      </Mapping>
      <Mapping>
            <Map>BA</Map>
            <To>pep_ca20_uml_meta.BA</To>
      </Mapping>
      …….
```

Listing 6. 1: Mapping-Datei für die Instanziierung des Metamodells

6.3.2 oAW Features

openArchitectureWare bringt in seiner aktuellen Version 4.x einige neue Features bei seiner Auslieferung mit. Unter anderem gehören diverse Eclipse Plugins zur Auslieferung im Core-Modul dazu, die das Entwickeln in Eclipse effizienter und angenehmer gestalten. Das Installieren des Generator Frameworks ab der Version 4, ist dank dem Plugin Konzept von Eclipse sehr transparent und einfach geworden. Die Installation und Integration vorheriger Versionen wird hier nicht näher beleuchtet. Stattdessen soll hier auf ältere Diplomarbeiten verwiesen werden [DAH01].

6.3.2.1 Workflow Engine

Die im Generator Framework verwendete Workflow-Engine ist deklarativ konfigurierbar. Die Engine wird mit einer auf XML basierten Konfigurationssprache, wie im Beispiel in Abbildung 7.2 zu sehen, konfiguriert. Mit dieser Sprache ist jeder erdenkliche Workflow definierbar.

Ein Workflow, der im oAW Framework logisch als zentrale Steuereinheit gesehen werden kann, besteht aus Workflow-Components. Bei einem Generierungsprozess repräsentiert jede WorkflowComponent einen Teil des Prozesses. In einem Workflow wird das Einlesen und Parsen, die Modellvalidierung, Modelltransformation und Codegenerierung angestoßen, sowie selbst definierte Prozesse.

Seit der Version 4.0 ist der Generierungsprozess und alle dazugehörigen Prozesse durch die Workflow-Engine kontrollierbar. Das Prinzip des Workflows ist essentiell um mit dem Framework arbeiten zu können. Deshalb soll hier detailliert das Prinzip und die Funktionsweise aufgeführt werden.

Im nachfolgenden Beispiel ist eine Workflow-Konfiguration zu sehen. Das Root-Element besitzt den Namen <*workflow*>. Danach folgt die Deklaration zweier Properties und zweier Components.

```
<workflow>
      <property name='genPath' value='/home/user/target'/>
      <property name='model' value='/home/user/model.xmi'/>
      <component class='oaw.emf.XmiReader'>
            <model value='${model}'/>
      </component>
      <component class='oaw.xpand2.Generator'>
            <outlet>
                  <path value='${genPath}'/>
            </outlet>
      </component>
</workflow>
```

Listing 6. 2: Struktur der Workflow-Datei

Die Konfigurationssprache definiert verschiedene Sprachmittel.

Properties

Properties können überall in der Workflow-Datei definiert werden. Nach der Deklaration können diese in den WorkflowComponents und auch in anderen Property-Deklarationen verwendet werden.

- Einfache Properties
  - **<property name='baseDir' value ='./'/>**
- Property Files
  - **<property file='{$baseDir}/myProperty'/>**

Listing 6. 3: Arten von Properties

Nach der Deklaration eines einfachen Property, ist dieses im weiteren Verlauf des Workflows als Attribut verwendbar. Im obigen Beispiel wird das Property „baseDir“ in der folgenden Deklaration eines Property-Files als Attribut verwendet.
Des Weiteren ist es möglich mit Property-Statements Property-Files, wie aus Java bekannt, einzubinden. Das Property-File wird wie gehabt mittels Key/Value Paaren aufgebaut.

Components

Components werden in der Konfigurationssprache als XML-Element definiert. An dem obigen Beispiel ist die Deklaration von Components zu sehen. (ein Workflow ist eine spezielle Component)

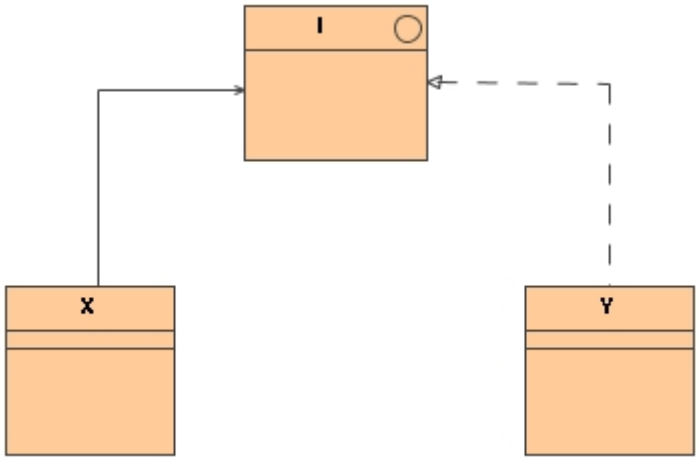


Abbildung 6. 5: Inversion of Control

Hinter der Konfigurationssprache verbirgt sich das Inversion of Control Design Pattern [IOC_P]. Allerdings ist der Begriff in diesem Zusammenhang noch zu allgemein gefasst. Im konkreten Fall handelt es sich um das Dependency Injection Pattern.
Bei Dependency Injection werden die Abhängigkeiten von außen in die Komponente gegeben. Somit wird verhindert, dass die Abhängigkeiten in den Components selbst definiert werden. Die Components selbst implementieren eine setBean-Methode, welche die Abhängigkeit einbindet.
Zum besseren Verständnis ist in der Abbildung 6.6 noch einmal der logische Aufbau durch einen Objektgraphen des oben aufgeführten Beispiels dargestellt.

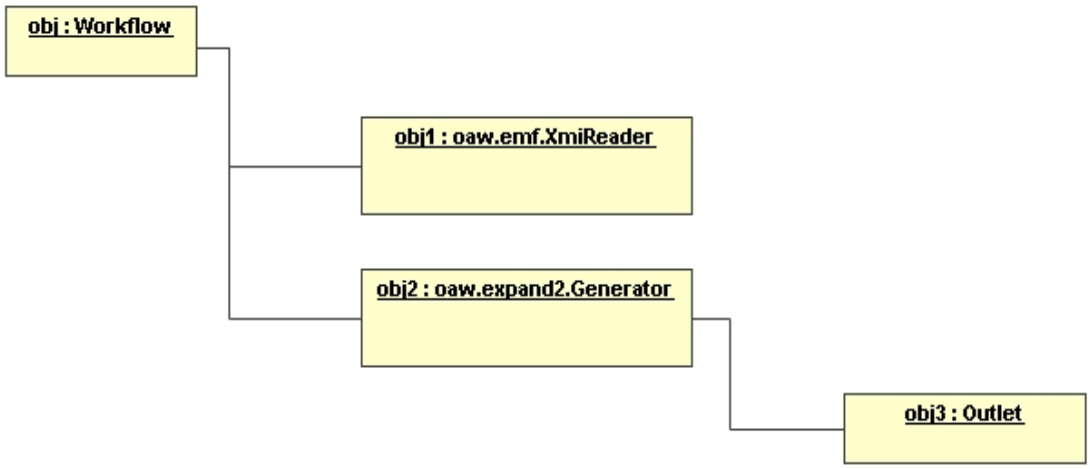


Abbildung 6. 6: Objektgraph der WorkflowComponents

Die Referenzierung von Components untereinander ermöglicht das Attribut „id“ und „idref“. Somit ist es innerhalb des Workflows möglich, auf eine Component die das „id“ Attribut besitzt, zu referenzieren.

Elemente, die nur das Attribut „value“ besitzen, sind einfache Parameter. Einfache Parameter besitzen keine Kindelemente und können auf zwei verschiede Arten deklariert werden. Sie können als XML-Attribut oder als verschachteltes XML-Element mit einem Attribut „value“ angelegt werden.

Vordefinierte Workflow-Components

Bei der Auslieferung von openArchitectureWare sind vordefinierte WorkflowComponents enthalten. Die folgende Tabelle zeigt die mit ausgelieferten WorkflowComponents und deren Funktion.

Component ID	Component Class	
XMIInstantiator	org.openarch…XMIInstantiator	Liest das Model ein und instanziiert dieses auf das Java-Metamodell
DirCleaner	org.openarch…DirectoryCleaner	Löscht alle Artefakte im angegebenen Directory
Generator	org.openarch…Generator	Angaben zur Transformation. - Outlet

Tabelle 6. 1: Ausgelieferte WorkflowComponents

Es besteht die Möglichkeit eigene WorkflowComponents zu entwickeln. Dadurch wird es dem Entwickler unter anderem möglich, explizit in den Generierungsprozess einzugreifen und externe oder selbst geschriebene Programme einzubinden. In der Diplomarbeit wird von diesem Mechanismus Gebrauch gemacht und so das Programm Ant angestoßen, welches mittels einer build-Datei weitere notwendige Metadaten für das Projekt generiert. Doch hierzu wird detailliert im Abschnitt Realisierung eingegangen.

Implementierung eigener WorkflowComponents

Für die Implementierung eigener WorkflowComponents ist es nötig, das Interface org.openarchitectureware.workflow.WorkflowComponent zu implementieren. Das Interface schreibt vier zu realisierende Methoden vor.

```
public abstract interface WorkflowComponent {
      public abstract void invoke(WorkflowContext ctx,
      ProgressMonitor monitor, Issues issues);

      public abstract void checkConfiguration(Issues issues);

      public abstract CompositeComponent getContainer();

      public abstract void setContainer(CompositeComponent
container);
}
```

Listing 6. 4: Interface WorkflowComponent

In der Methode invoke() wird die eigentliche Arbeit der Component implementiert. Ein hilfreicher Parameter ist der Parameter „issues“. Mittels der Klasse Issues kann man Fehler oder Warnungen während des Workflow-Prozesses anzeigen und sich somit nützliche Informationen auf der Konsole ausgeben lassen. Des Weiteren hat die Methode invoke() Zugriff auf den WorkflowContext. Durch den WorkflowContext wird es Components ermöglicht miteinander über Slots zu kommunizieren.

Zur Überprüfung der korrekten Konfiguration der Component ist die Methode checkComponent verantwortlich. Diese wird vor dem Start des Workflow-Prozesses aufgerufen und erlaubt es den Start des Prozesses bei fehlerhafter Konfiguration abzubrechen.

Die beiden Container-Methoden werden für die Kaskadierung der Components verwendet. Mit diesen zwei Methoden wird die Realisierung des Dependency Injection Pattern realisiert.

Workflow-Files kaskadieren (Cartridges)

Seit der Einführung von Workflows in openArchitectureWare ist es dem Entwickler möglich auf einfache Weise Arbeitsschritte, eben Workflows, zu kaskadieren. Aus einem Workflow können durch den Cartridge-Mechanismus weitere selbst- oder vordefinierte Workflows aufgerufen und diesem Parameter übergeben werden. Bei den openArchitectureWare Versionen vor 4.0 musste dieser Schritt mit dem Build-Tool Ant gelöst werden.

6.3.2.2 Instanziator

Der anpassbare Instanziator ermöglicht openArchtiectureWare unterschiedliche Model-Formate einzulesen. Out of the Box unterstützt oAW das Eclipse Modeling Framework (EMF), verschiedene UML-Tools wie z.B. MagicDraw und Poseidon.

Aufgrund unterschiedlicher Interpretierung des XMI-Standards, können sich je nach Hersteller und Toolversion extreme Unterschiede in den XMI-Files ergeben. Deshalb können Probleme bei der Anwendungsdesign-Instanziierung auftreten. Es ist von großer Bedeutung die passende Version der Mapping-Datei zur verwendeten Version des Modellierungstools zu verwenden.

6.3.2.3 Xpand

Das openArchitectureWare Framework beinhaltet eine eigene Templatesprache Xpand. Wie oben schon kurz aufgeführt wird diese Sprache in den Templates für die Steuerung des Output (Generats) verwendet. Die Templates werden in Textdateien mit der Dateierweiterung xpt angelegt, welche mit dem Eclipse-Editor von oAW verknüpft sind. Die Implementierung der Templates legt die konkrete Abbildung auf eine Technologie fest. Durch Einsatz verschiedener technologieabhängiger Templates ist es möglich, das generische Anwendungsdesign (PIM) mit entsprechendem Metamodell nicht nur für eine spezielle Technologie zu verwenden. Will man zum Beispiel nebens der J2EE- auch die .Net-Plattform unterstützen, muss man lediglich in den Templates die Abbildungen anpassen. Somit wird man auch dem Ansatz der OMG gerecht, der besagt, dass Modelle nicht nur eine visuelle Dokumentation sind, sondern als Code angesehen werden können, die von einem Technologiewechsel nicht berührt werden. Selbst bei kleineren Eingriffen in die zugrunde liegende Ausführungsplattform, zum Beispiel durch einen Wechsel des Applikationsserver und sich somit ergebenden Änderungen der Deployment Descriptoren, ist nur eine verhältnismäßig kleine Anpassung in den Templates nötig.
Im nachfolgendem Abschnitt soll auf die Syntax und Semantik dieser Templatesprache eingegangen werden.

Syntax und Semantik

Ein Template besteht grundsätzlich aus zwei Aspekten:
Dem Code, der über die Metamodellinstanz iteriert, sowie dem zu generierenden Quellcode, gegebenenfalls mit Zugriff auf Attribute des Metamodells.
Ein Template besteht aus generischen Anweisungen der Skriptsprache xPand und statischen Anteilen des zu generierenden Programmiermodells. xPand ist auf das Anwendungsgebiet der Codegenerierung zugeschnitten und bietet deshalb nur die notwendigen Sprachfeatures für diesen Bereich an.

In diversen anderen auf dem Markt befindlichen Generator Frameworks werden keine propritäre Skriptsprachen eingesetzt. Bei dem Framework AndroMDA[10] [ANDA] findet die bekannte Skriptsprache Velocity[11] von der Apache Group ihren Einsatz. Jedoch ist auf Grund von überflüssigen Sprachelementen der Lernaufwand wesentlich höher als bei xPand.
xPand bietet als Sprachelemente einen wesentlich kleineren Umfang an. Es werden hauptsächlich bedingte Anweisungen, Schleifen, Operationen und Sprachkonstrukte wie Kommentare zur Navigation und zum Zugriff auf Modellelemente im Speicher befindlichen Anwendungsdesign benötigt. Diese Sprachmittel reichen aus, um den gewünschten Output zu generieren.
Template-Dateien dürfen weitgehend frei benannt werden und stellen selbst Namensräume dar. Polymorphie wird bei der Templatesprache Xpand, durch die in den Templates definierten Metaklassen, unterstützt. Somit kann das Template mit der Superklasse aufgerufen werden und Xpand erkennt über die korrespondierende Metaklasse welches Template ausgeführt werden soll.

Definieren von Templates

xPand Anweisungen werden durch doppelte spitze Klammern « » umschlossen und sind in sich geschlossene Einheiten (wie im Beispiel zu sehen). Anweisungen besitzen ein Start- und End-Schlüsselwort. Der Bereich zwischen den beiden Schlüsselwörtern wird als Scope oder Geltungsbereich bezeichnet. Jeglicher Text, der nicht in den spitzen Klammern steht, ist statischer Text und wird eins zu eins in das Generat übernommen.
Der Aufbau der Templates intern folgt einem bestimmten Schema. Wie auch aus der Programmiersprache Java bekannt, kann man in xPand Namensräume mittels des Schlüsselworts „IMPORT“ einfügen. So ist es nicht von Nöten im Template den vollqualifizierten Namen zu verwenden. Nach den IMPORT-Statements können mit dem Schlüsselwort „EXTENSIONS“, falls verwendet, Extensions eingebunden werden.
Extensions wurden mit der openArchitectureWare Version 4.0 eingeführt und wirken einer Verschmutzung des Metamodells durch Erweiterungen entgegen. Alle eingefügten Erweiterungen mit vorherigen Versionen des Frameworks wurden in den Metaklassen implementiert. Extensions werden weiter unten in einem eigenen Abschnitt näher betrachtet.

[10] AndroMDA - http://www.andromda.org/
[11] Velocity - http://jakarta.apache.org/velocity/

```
«IMPORT meta::model»
«EXTENSION my::ExtensionFile»

«DEFINE javaClass FOR Entity»

      «FILE fileName()»
          package «javaPackage()»;
          public class «name» {
            // implementation
          }
      «ENDFILE»

«ENDDEFINE»
```

Listing 6. 5: Template-Struktur

Nach den Extensions folgen ein oder mehrere Define-Blöcke. Das zentrale Konzept von xPand ist der Define-Block. In der Deklaration des Define-Blockes wird die Metaklasse, für die der Block ausgeführt werden soll, angegeben. Durch diese Angabe kann das Template dynamisch zur Laufzeit an das instanziierte Anwendungsdesign gebunden werden. Wenn das Template an das korrespondierende Metamodel-Element gebunden ist, ist der Zugriff auf dessen Methoden und Beziehungen möglich.

```
«DEFINE templateName(formalParameterList) FOR MetaClass»
      a sequence of statements
«ENDDEFINE»
```

Listing 6. 6: DEFINE Scope

Im Sprachkonstrukt „FILE“ werden der Name und der Speicherort des Generats angegeben. Im Scope des Statements können wieder beliebige Statements stehen wie zum Beispiel Expand.
Das Sprachkonstrukt Expand realisiert ein Sub-Template und kann mit einer Unterroutine verglichen werden. Es wird einfach auf einen anderen Define-Block verwiesen. Falls in der Expand-Anweisung kein explizit angegebener Namensraum steht, wird der aktuelle Namensraum verwendet. Dies kann zur internen Strukturierung von Templates durchaus sinnvoll sein.

Im Abschnitt Realisierung in der Diplomarbeit wird näher auf die Templateentwickung am Beispiel des Projektes eingegangen. Weitere und fundierte Informationen findet man in der Template Spezifikation. [b_mTL]

6.3.2.4 xTend

xTend ist ein weiteres sehr nützliches Feature in openArchitectureWare. Dieses Feature unterstützt das Erstellen von unabhängigen Operationen, welche aus den Templates aufgerufen werden können. Extensions werden in Textdateien mit der Dateierweiterung „ext“ angelegt. Man kann Extensions entweder mit oAW-Ausdrücken implementieren oder innerhalb der Extensions weiter in eine Java Methode verweisen.

```
String asGetter (ModelElement elem) :
      "get"+elem.NameS.toFirstUpper();
```

Listing 6. 7: Extension mit oAW-Ausdrücken

In Listing 6.7 ist eine mit oAW Ausdrücken implementierte Extension zu sehen. Die Extension selbst wird wie ein Methodenaufruf aus dem Template, welches die Extension eingebunden hat, aufgerufen. Im oben aufgeführten Beispiel ist eine Namenskonvention für Getter-Methoden definiert. Die Extension erwartet als Parameter ein Element vom generischen Typ ModelElement. In der Methode wird der Name des Elements ermittelt und diesem der Präfix „get“ vorangestellt.

Die oAW Extensions selbst sind wiederum mit einer Java Extension erweiterbar. Dieser Mechanismus ist bestens dazu geeignet, komplexere Operationen, wie zum Beispiel die Berechnung von Package-Namen, aus den Templates auszulagern.
Bei Verwendung der Java Extensions steht dem Entwickler das volle Funktionsspektrum der Programmiersprache Java zur Verfügung. Im nachfolgenden Listing ist der Aufruf einer Java Extensions, die den Package-Namen einer als Parameter übergebenen Klasse zurück liefert, abgebildet.

```
String getPackageName (Class cls) :
      JAVA pep_ca20_extend.ClassUtil.getPackageName(
org.openarchitectureware.meta.uml.classifier.Class);
```

Listing 6. 8: Java Extension aus einer Extension

6.3.2.5 Check

Es ist sehr wichtig, dass Generatoren korrekten und vollständigen Code generieren. Dafür müssen die Informationen, die der Generator als Input erhält, vollständig und korrekt sein. Zum

Beispiel muss das erstellte Modell konsistent sein. Um dies zu erreichen, hat man die Möglichkeit das Metamodell mittels Constraints anzureichern und über diese das Modell zur Laufzeit zu überprüfen. Eine Alternative zur Implementierung im Metamodell selbst sind Check Files im openArchitectureWare Framework.

Die Syntax, mit der die Check Files implementiert werden, ist ähnlich wie die der Extensions. Sie verwenden die Dateierweiterung „chk“ und werden dazu benutzt Constraints für das Modell zu definieren. Durch diese Lösung wird die Implementierung des Metamodells nicht verschmutzt.

Um eine Validierung, die in einem Check File implementiert ist, aufzurufen, wird im Workflow eine Check-Component integriert. Das Validierungsfile wird in der Check-Component angegeben und bevor der Generierungsprozess startet wird das Modell mit den Validierungsvorschriften überprüft. Es ist dem Entwickler überlassen, ob der Generierungsprozess trotz Auftreten eines Fehlers bei der Validierung startet. Wenn der Prozess nicht unterbrochen werden soll, hat man allerdings immer noch die Möglichkeit mittels einer Warnung/Info/Error Message auf Validierungsfehler hinzuweisen.
Im Beispiel Listing 6.9 ist eine Validierung zu sehen, die überprüft, dass jedes Ende einer Assoziation einen Rollennamen besitzt. Wenn dies nicht der Fall ist, wird der angegebene Text ausgegeben und der Generierungsprozess unterbrochen.

```
context AssociationEnd ERROR Class.NameS+"->"
+Opposite.Class.NameS+":
      Navigable association ends must have a role name" :
      isNavigable ? !isUnnamed : true;
```

Listing 6. 9: Constraints Definition

6.3.2.6 Nützliche Features

Zu openArchitectureWare gehören noch einige weitere nützliche Feature, die hier nicht näher vorgestellt oder im Zuge der Diplomarbeit verwendet werden. Nichtsdestotrotz soll zur Vollständigkeit die wichtigsten Feature und ihre Funktion hier kurz aufgeführt werden.

Recipe

Mit dem Recipe Framework von openArchitectureWare sind Validierungsregeln für Artefakte definierbar, die nicht durch den Generierungsprozess erstellt werden. Beispiel für solche Artefakte sind manuell implementierte Klassen, die nach der Generierung erstellt werden. Die

Entwicklungsumgebung Eclipse liest die während des Generierungsprozess instanziierte Validierungsregeln ein, überprüft diese und kann so den Entwickler bei der Implementierung nach der Generierung leiten.

Model2Mode Transformationssprachel Xtend

Mit der funktionalen Sprache Xtend sind Abbildungen für Modell zu Modell Transformationen beschreibbar.
Weiter Informationen zu xTend unter [oAW+0X].

Integration von Graphical Editor Framework

Mit dem Adapter für das Graphical Modelling Adapter (GMF) ermöglicht oAW zur Designzeit das Überprüfen von in check definierten Constraints.
Näheres zu GMF in einem späteren Kapitel.

Unterstützung für Eclipse UML2Modell

Mit oAW Version 4.0 kommt ein Plugin für UML2 Eclipse mit. Dieses Plugin erweitert die verwendbaren Designsprachen UML und EMF um eine weitere. UML2 für Eclipse ist eine EMF[12] basierte Implementierung des UML 2.x Metamodell der OMG.

XText

Xtext ist ein Framework, mit dem man textuell domänenspezifische Sprachen entwickeln kann. Die Designsprachen lassen sich in bekannter EBNF-Form definieren.
Der Vorteil eines nicht graphisch erstellten Modells ist der, dass bei Änderungen eine Textdatei wesentlich schneller zu bearbeiten ist als ein Modell in einem Modellierungstool. Mehr Informationen über dieses Feature unter [oAW+06].

[12] EMF – Eclipse Modelling Framework http://www.eclipse.org/emf/

7 Realisierung

Dieses Kapitel beschreibt die in der Diplomarbeit verwendeten Werkzeuge und eingesetzte Bibliotheken, die Konfiguration und Verwendung des openArchitectureWare Frameworks sowie die Erweiterung des Metamodells und die Templates. In diesem Kapitel wird außerdem auf die Besonderheiten bei der Modellierung des PIM im von BMW verwendeten Designtool Together eingegangen.
Die Aufgabenstellung wurde in Kapitel 1.2 schon detailliert aufgeführt. Zur kontextrelevanten Übersicht wird die Aufgabe hier noch einmal stichpunktartig erläutert:

Gestellte Aufgaben:

- Ablösung des BMW-Generator
- Einsatz des openArchitectureWare Framework
- Generierung von CA2.0 konforme Artefakte
- Entkoppeln von Together als Designtool
- Verringerung der Generierungsstufen
- Generierung eines Implementierungsrahmen unter Verwendung des PEP-PDM PIM
- Evaluierung der Integration des vollständigen Entwicklungsprozesses in Eclipse

7.1 Verwendete Tools

Die für den Entwicklungsprozess verwendeten Werkzeuge sind fast ausschließlich Open Source Produkte. Together ist das einzigste kommerzielle Werkzeug, welches durch ein Open Source Produkt ersetzt werden kann. Diesem Thema widmen sich die darauf folgenden Kapitel.

7.1.1 Eclipse (IDE)

Für die Implementierung und Projektverwaltung wurde die Eclipse Plattform in der Version 3.2.1 [Eclipse] verwendet.
Eclipse ist seit der Version 3.0 ein Open-Source-Framework zur Entwicklung von Rich Client Applikationen (RCP). Es besteht selbst nur aus einem Kern, der die einzelnen Plugins lädt, welcher die eigentliche Funktion wie z.B. die IDE zur Verfügung stellt. Die Eclipse IDE bringt nützliche Werkzeuge wie Ant, Debugger und den Java-Editor mit. Durch das Plugin-Konzept,

was Eclipse zugrunde liegt, steht einer flexiblen Erweiterung der Entwicklungsumgebung nichts im Wege.

7.1.2 Ant

Für die Steuerung von XDoclet wurde in der Diplomarbeit Ant verwendet. Ant ist ein in Java geschriebenes Werkzeug zur Erstellung von ausführbaren Programmen oder Softwarepakete aus Quellcode, welches mit Make unter Unix verglichen werden kann. Das Open Source Projekt Ant startete als Teil des Jakarta Projekts [Jakarta] und ist ein Apache [Apache] Projekt.
Die Steuerung von Ant wird durch eine XML-Datei konfiguriert. In der build.xml wird ein Projekt definiert welches Targets enthält. Die Targets können von der Entwicklungsumgebung oder auch aus einem Programm aufgerufen werden. Die Targets selbst bestehen aus Tasks die zum Beispiel die Kompilierung von Java Dateien oder den Aufruf von XDoclets ermöglichen.

7.1.3 XDoclet

XDoclet ist eine Open Source Code Engine die attributorientiert arbeitet. Spezielle Tags, die Doclets genannt werden, fügt man als Metadaten in den Code. Die Engine parst die Source-Files und generiert Artefakte wie zum Beispiel Deskriptoren. In der Implementierung der Diplomarbeit findet XDoclet [XDoclet] nur noch für die Generierung der Deskriptoren Verwendung.

7.1.4 Log4J

Als Logging Framework wurde Log4J [Log4J] eingesetzt. Log4j hat sich als Standard bei der Java-Entwicklung dank seiner Ausgereiftheit und Konfigurierbarkeit herausgestellt. Das Open Source Projekt Log4J gehört der Apache Software Foundation an.

7.1.5 Together UML

Together ist das Design-Werkzeug welches für die Anwendungsbeschreibung in CA2.0 verwendet wird. Borland vertreibt das kommerzielle Tool auf seiner Page [Borland]. In Together werden die Platform Independent Models mit der UML erstellt und können durch eine XMI-Export Schnittstelle als XMI-Datei gespeichert werden. Weiterhin werden mittels UML grafisch erstellte Designs im Background von Together automatisch in Code umgesetzt. Umgekehrt wird auch Code ins grafische Design übertragen. Together unterstützt alle UML-spezifischen Diagramme (Class Diagramm, Use Case Diagramm, Sequence Diagramm, u.v.m.).

Ein Problem mit Together ist, dass das exportierte XMI nicht ohne Erstellung einer Mapping-Datei mit dem openArchitectureWare Generator kompatibel ist. Außerdem wird Together im Gegensatz zu anderen Modellierungswerkzeugen nicht von openArchitectureWare unterstützt.

7.1.6 openArchitectureWare Framework

Das openArchitectureWare Framework ist ein Framework zur modellgetriebenen Entwicklung. Der modulare Aufbau erlaubt eine optimale Anpassung und Verwendung in Projekten. Das Framework unterstützt arbiträre Formate, Metamodelle und diverse Ausgabeformate (vgl. Kapitel 6).
Das Framework ist seit Ende 2003 unter LGPL veröffentlicht und kann dort herunter geladen werden. Hilfreiche und zeitnahe Unterstützung bekommt man in verschiedenen Foren. Es existiert auch ein Deutsches Forum unter [itemis].

7.1.7 Hybridlabs Java Beautifier

Die Akzeptanz von generiertem Code hängt nicht zuletzt von der Struktur, Lesbarkeit und Wartbarkeit ab. In der Arbeit wurde ein Java Bautifier von Hybridlabs [Hybridlabs] verwendet damit der generierte Code besser lesbar ist.

7.2 Projektstruktur

Die Projektstruktur und die nicht zu generierenden Artefakte sind im Repository Tool Subversion unter dem Namen pep_ca20 angelegt. Seit openArchitectureWare Teile des Generator-Frameworks als Plugin ausliefert muss das Projekt nicht wie in älteren Version des oAWs aufgesplittert werden.
Die Abbildung 7.1 zeigt das pep_ca20 Projekt als Plugin Projekt.

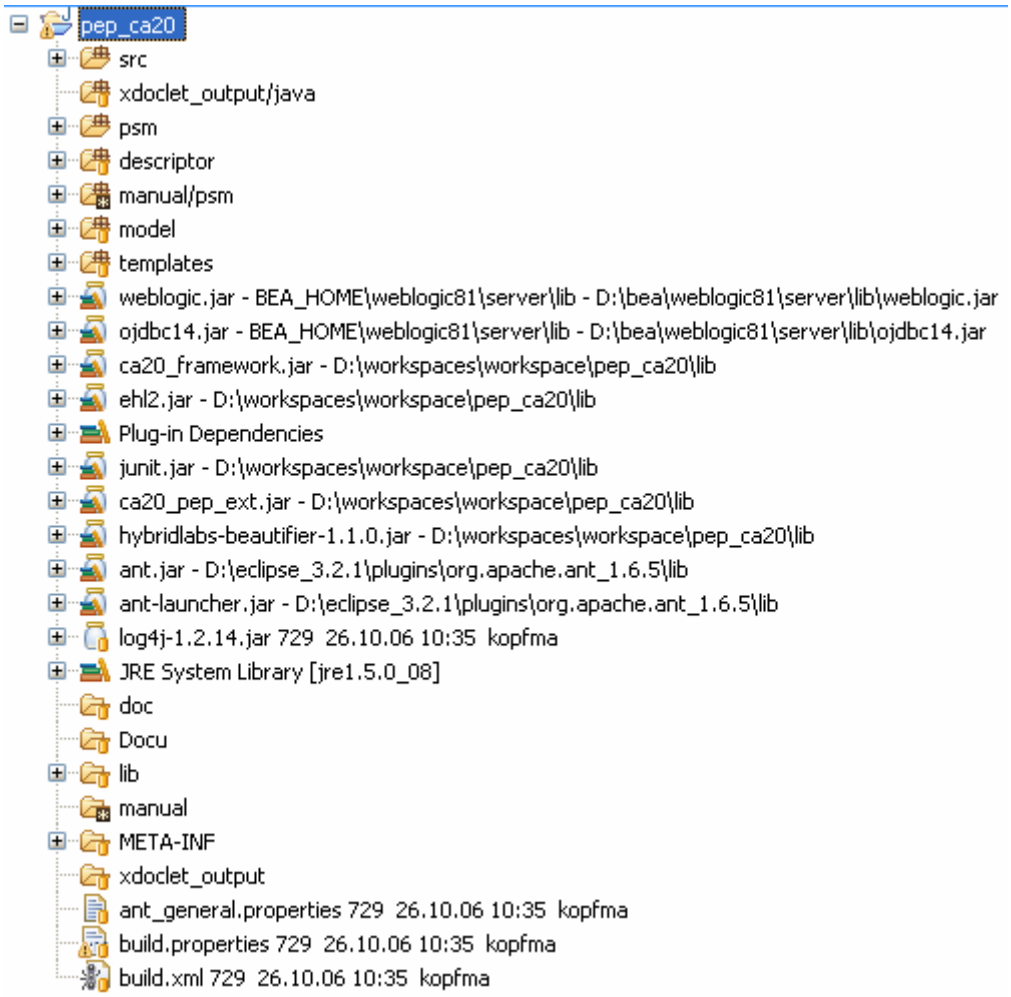


Abbildung 7. 1: Strukturierung des Projektes

7.2.1 Struktur der Generatorartefakte

Bei der Neuanlage des Projektes in Eclipse wird ein Java Projekt angelegt und dieses über die Kontextfunktion zu einem Plugin Projekt konvertiert. Ein Plugin Projekt unterscheidet sich in Eclipse zu einem Java Projekt unter anderem durch den META-INF Ordner in dem das Plugin Manifest zu finden ist.

openArchitectureWare Plugins

Über die MANIFEST-Datei können sämtliche Abhängigkeiten über ein MANIFEST-Formular definiert werden. Es sind folgende Plugins in das Projekt eingebunden.

- org.openarchitectureware.classic.umlMetamodel
  enthält die Elemente des UML Metamodell
- org.openarchitectureware.classic.core
  enthält die Framework-Klassen für openArchitectureWare

- org.openarchitectureware.classic.workflow
  enthält openArchitectureWare “Workflow Components”
- org.openarchitectureware.classic.xmlInstantiator
  enthält XML Parser Komponenten für UML Tools
- org.openarchitectureware.core.check
  enthält Elemente für die Sprache Check von openArchitectureWare
- org.openarchitectureware.classic.xmiInstantiator
  enthält XMI Parser Komponenten für UML Tools
- org.openarchitectureware.core.xpand
  enthält die Xpand Template Engine

Ordner /src

Das Package src wird als zentrales Package für Artefakte des oAW-Generators benutzt. Außer der PIM-Modell-Datei und der Templates-Dateien die jeweils in einem separaten Ordner sind. Folgende Artefakte befinden sich im src-Ordner:

- Mapping Datei (Metamapping)
- Together Mappingdatei (Tool-Mapping)
- Workflow
- Workflow Properties
- Log4J Property Datei

Weiterhin befinden sich noch folgende drei Packages unter dem src-Ordner:

Package /pep_ca20_uml_meta

Das Package pep_ca20_uml_meta enthält alle Erweiterungen des mitgelieferten Metamodells. Die Erweiterungen leiten von den korrespondierenden Metaklassen ab und dadurch die erforderliche Semantik für die Problemdomäne erstellt.

Package /pep_ca20_extend

Im Package pep_ca20_extend sind diverse Utility-Klassen z.B. zum Laden von externen Properties. Außerdem finden sich hier auch die Klassen die für die Verwendung aus den Extension-Files benötigt werden.

Package /pep_ca_20_properties

Das pep_ca_20_properties-Package enthält die Property-Dateien die es ermöglichen den Generator von außen für unterschiedliche Architekturen (EJB2.1/EJB3.0) und weitere diverse

Anpassungen zu konfigurieren. Es dreht sich hier um die Konfiguration der zu generierenden Metadaten für die Deskriptoren. Das konkrete Konzept wird unter dem Abschnitt Properties erläutert.

Package /model

In diesem Package befinden sich die in XMI serialisierten PIM-Modelle, die vom Generator für die Transformation gelesen werden.

Package /templates

In dem Package templates sind alle Template-Dateien für die Transformation mittels des oAW-Generators enthalten. Die Unterpackages sind an die Struktur der Component Architecture angelehnt.

In der folgenden Abbildung ist die Struktur/Anordnung der Templates in den dazugehörigen Packages grafisch aufgeführt:

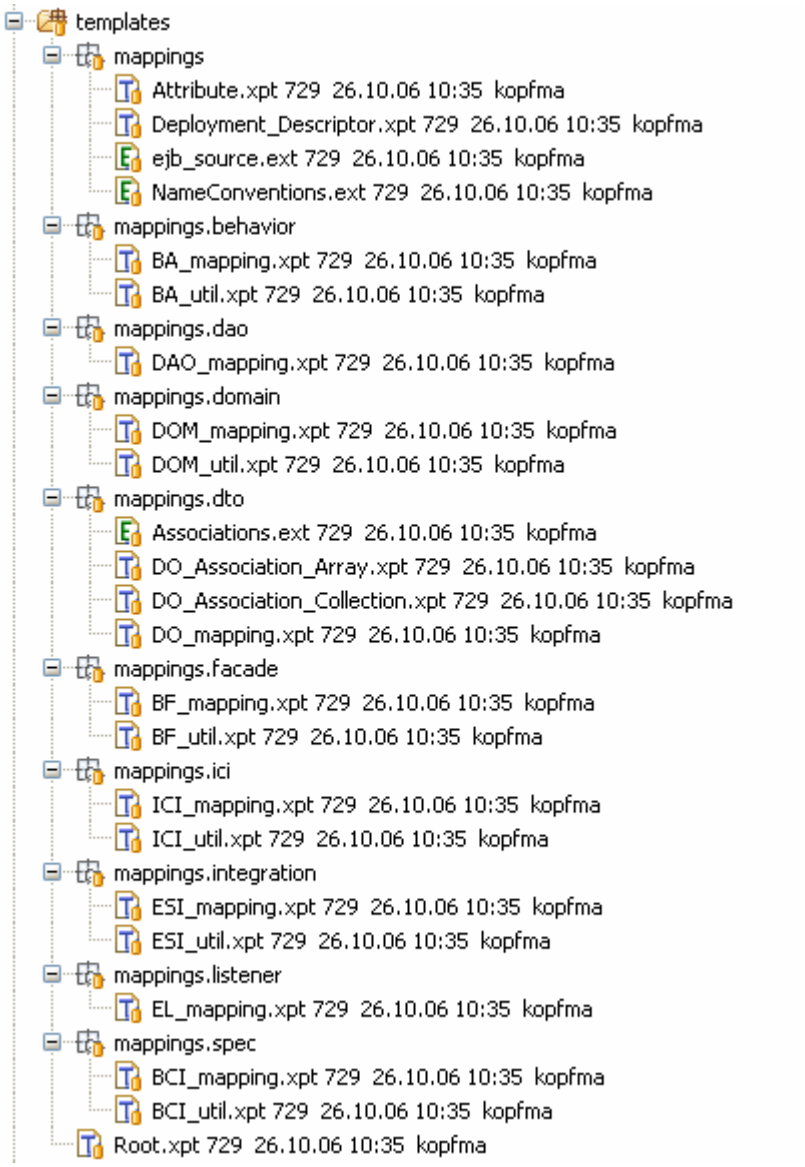


Abbildung 7. 2: Projektstruktur für die Templates

Zusätzliche Funktionen über Extensions /templates

In dem übergeordneten Package mappings finden sich die Extension-Files der Extend Sprache. Extensions sind hauptsächlich zur Erstellung von Namenskonventionen im Projekt zuständig.

Modellvalidierung mit Checks /templates

Außer der Extensions sind auch die Check-Files zur Modellvalidierung in dem Package mappings hinterlegt.

7.2.2 Struktur der PEP-PDM Artefakte

Im Folgenden wird die Struktur der Artefakte von PEP-PDM erläutert. Es handelt sich hier nicht ausschließlich nur um generierte Artefakte, sondern auch um Elemente, die die Anwendung gestellt bekommt. Als Beispiel sind hier unter anderem verschiedene Libraries aufzuführen.

Ordner /psm

In den Ordner psm werden alle generierten Artefakte abgelegt. Dieser Ordner stellt so zu sagen den Implementierungsrahmen zur Verfügung. Konzeptionell werden in der Diplomarbeit auf Protected Regions verzichtet und der Ansatz über Vererbung realisiert. Aus diesem Grund dürfen in den Artefakten die im psm-Ordner liegen keine manuelle Anpassungen oder Implementierungen erfolgen.

Ordner /manual/psm

In diesen Ordner werden die konkreten abgeleiteten Artefakte, korrespondierend zu den Artefakten im Ordner psm, generiert. In diese Implementierungsklassen wird der fachspezifische Code manuell eingefügt. Die Artefakte in diesem Ordner werden bei einer erneuten Generierung nicht gelöscht oder überschrieben.

Ordner /descriptor

In den Ordner descriptor wird, durch den im Workflow des Generatorprozesses integrierten XDoclet-Lauf, der Deployment Descripor für den Application Server generiert.

Ordner /xdoclet/java

In diesen Ordner werden alle für die EJB2.1 Komponenten Architektur benötigte Interfaces generiert. Dieser Ordner wird bei einem iterativen Prozess gelöscht und neu erzeugt.

Ordner lib

Dieser Ordner enthält sämtliche Libraries, die für die Implementierung einer Anwendung benötigt werden.

7.3 Metamodellierung

7.3.1 Java Metamodell

Für die Erstellung eines Anwendungsdesigns und der Möglichkeit zur automatisierten Transformation wird ein Metamodell benötigt. Das Metamodell wird für die vorliegende Domäne durch einen Erweiterungsmechanismus angepasst und zur Instanziierung des Anwendungsdesign verwendet (vgl. 6.3.1). Mit der Erweiterung des Metamodells wird die Semantik des Modells erhöht und einer Architekturfamilie zugeordnet.

Ein an die UML1.4 angelehntes Metamodell, in Java implementiert, bringt das openArchitectureWare Framework bei seiner Auslieferung mit. Die Implementierung umfasst den Code für die Klassendiagramme, Use Case Diagramme und State Diagramme. In der vorliegenden Diplomarbeit wurde lediglich von der Implementierung des Klassendiagramms Gebrauch gemacht, nichtsdestotrotz werden zur Vollständigkeit die anderen beiden Packages aufgeführt.

Die Klassen zu den korrespondierenden Diagrammtypen sind in folgende Packages aufgegliedert:

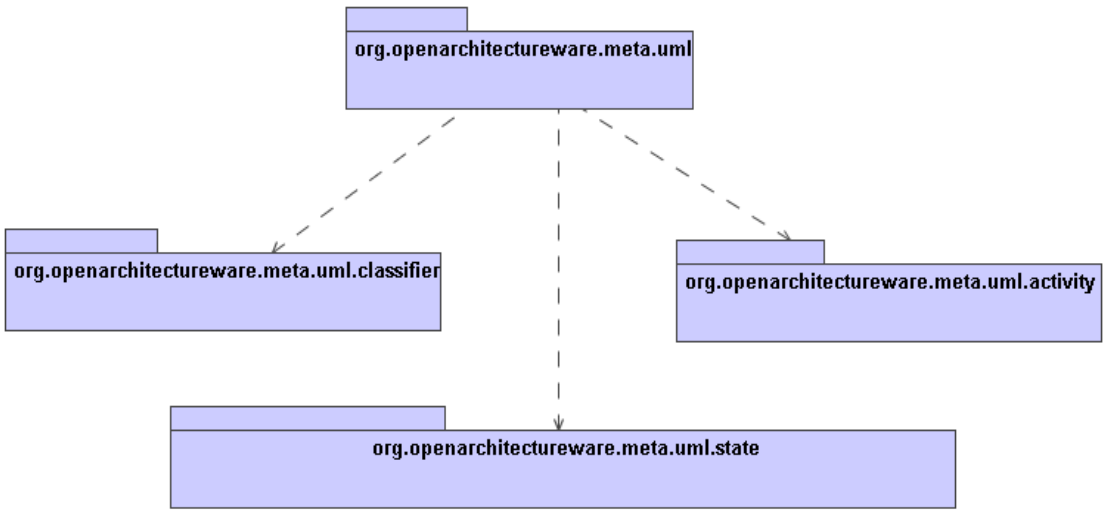


Abbildung 7. 3: Packages der Diagramme

org.openarchitectureware.meta.uml

Das Package *uml* enthält die Basis-Typen des Metamodells, die von den anderen Packages verwendet werden.

org.openarchitectureware.meta.uml.classifier

Das Package *classifier* enthält die Basis-Elemente für die Unterstützung der UML-Klassendiagramme in openArchitectureWare.

org.openarchitectureware.meta.uml.state

Das Package *state* enthält die Basis-Elemente für die Unterstützung der State-Diagramme in openArchitectureWare.

org.openarchitectureware.meta.uml.activity

Das Package *activity* enthält die Klassen für das Metamodell der UML Activity Diagramme in openArchitectureWare.

7.3.2 Erweiterung des Java Metamodells

Für die vorliegende Problemdomäne muss das mitgelieferte Java-Metamodell entsprechend erweitert werden damit der Generator das Anwendungsdesign einlesen und richtig transformieren kann. Das Metamodell im openArchitectureWare wird durch Metaklassen implementiert. Die Erweiterung des Metamodells wird in UML-PIM durch Stereotypen dargestellt.
Die Metaklassen der Erweiterung werden im Metamodell als Klassen angelegt und erben von den jeweilig zu erweiternden UML Metaklassen des mitgelieferten Metamodells. Wie in der Projektstruktur schon aufgeführt, werden alle Metaklassen die das Profil für die Domain Specific Language (DSL) bilden in dem Package pep_ca20_meta gehalten.

Folgende Elemente der Component Architecture sind für die Erweiterung und somit zur Bildung der Domänen-Sprache nötig:

- Business Component Interface (BCI)
- Business Activity (BA)
- Business Facade (BF)
- Business Entity (BE)
- External Service Interface (ESI)
- Inter Component Interface (ICI)
- Data Object (DO)
- Event Listener (EL)

Eine detaillierte Erläuterung der Elemente ist in Kapitel 4.2 zu finden.

Durch die Erweiterung des Metamodells ist es möglich in den Metaklassen zusätzliche Funktionen, wie zum Beispiel Design-Constraints einzubauen. Allerdings wird in dieser Diplomarbeit davon abgesehen um das Metamodell nicht unnötig komplex zu gestalten und zu verschmutzen. Durch den Extensions Mechanismus der vom openArchitectureWare Framework zur Verfügung gestellt wird, ist eine Auslagerung zusätzlicher Funktionen möglich. Design Constraints müssen bei der Verwendung des oAW Frameworks, dank der Sprache Check, auch nicht in den Metaklassen hinterlegt sein. Constraints werden in Textfiles mit der Endung *chk* definiert und über den Workflow bei der Instanziierung des Designs eingebunden (vgl. 6.3.2.5). Durch die Einbindung der Modellvalidierung in den Generierungsprozess, kann noch vor der Transformation überprüft werden, ob sich das zu generierende Anwendungsdesign in einem validen Zustand befindet.

Durch diese sehr hilfreichen Funktionalitäten des openArchitectureWare Frameworks bleibt das erweiterte Metamodell von technischen Anreicherungen frei. Aufgrund der trivialen Implementierung der Metaklassen, sei hier nur eine Erweiterung (Stereotype) exemplarisch aufgeführt. Nötige zusätzliche Funktionen werden in der Arbeit durch Extensions realisiert die weiter unten noch beleuchtet werden.

```
package pep_ca20_uml_meta;
import org.openarchitectureware.meta.uml.classifier.Class;

public class BA extends Class{
      /**
       *    Erweiterung des UML-Element Class.
       */
      private static final long serialVersionUID = 1L;
}
```

Listing 7. 1: Erweiterung des Metamodells

Die mit diesem Mechanismus erstellte domänenspezifische Sprache (Metamodell) wird in der Abbildung 7.4 aufgeführt:

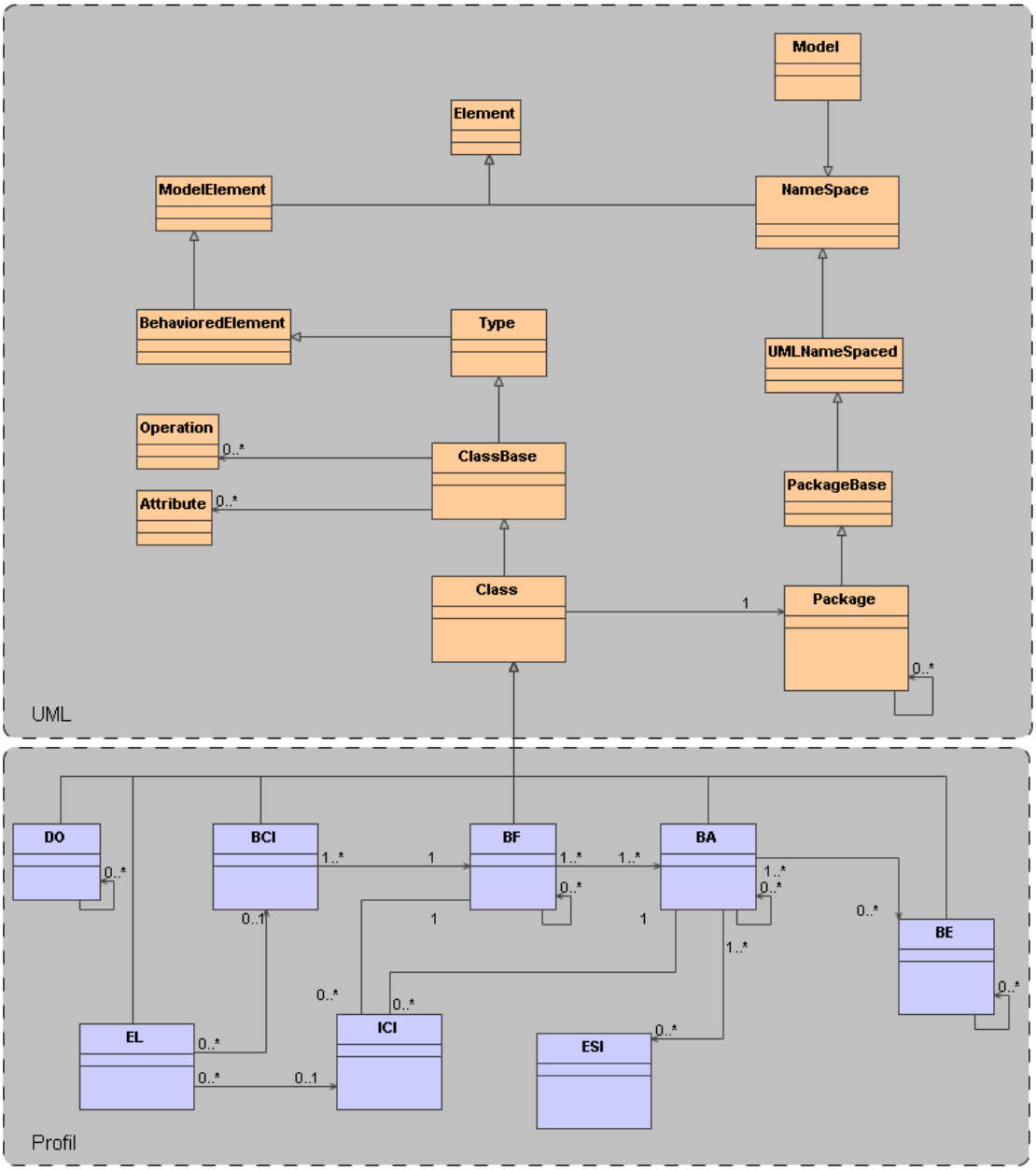


Abbildung 7. 4: Erweiterung des oAW-Metamodells

7.4 PEP-PDM PIM

Die Diplomarbeit basiert auf dem existierenden Projekt PEP-PDM für die BMW-Group. In diesem Abschnitt soll ein kleiner Überblick über die bestehenden Komponenten und die Struktur des PIM's gegeben werden. Die Realisierung sieht das vollständige PIM für die Überführung in den Implementierungsrahmen vor.

7.4.1 Komponenten in PEP

Das PEP plattformunabhängige Modell wird in dem grafischen Modellierungstool Together erstellt. Das Modell selbst ist in Together über Packages strukturiert. Die zu erstellenden Komponenten werden in den jeweiligen Packages, die nach den logischen Layern „business“ und „intermediate“ benannt sind, entworfen (vgl. Kapitel 4.2).

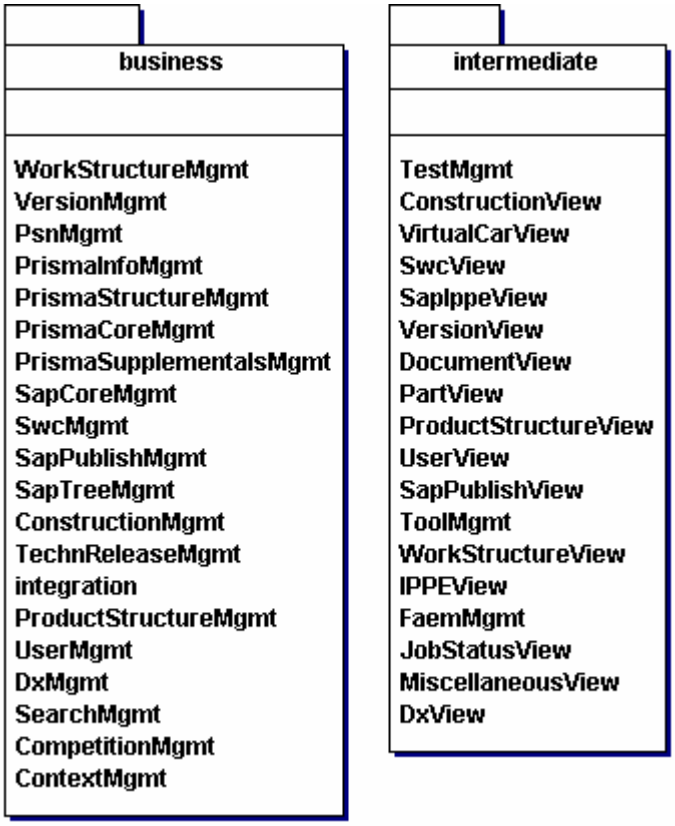


Abbildung 7. 5: Komponenten des Projektes PEP-PDM

Als bekleidendes Beispiel für die Erläuterungen der Realisierung dieser Arbeit wird die SWC-Komponente aus den schon bestehenden Komponenten ausgewählt. Für die Relisierung des SWC’s wird die Komponente SwcView aus der intermediate-Layer und die Komponenten SwcMgmt, SapCoreMgmt und PrismaInfoMgmt aus der business-Layer benötigt. Der Supplier Working Context bietet eine große Anzahl an Use Cases. Es wird hier ein passender Use Case herausgenommen an welchem die zu transformierenden Elemente und vor allem deren Abbildungsvorschriften identifiziert werden.

7.4.2 Use Case „SWC“

Durch den SWC wird eine effizientere Verwaltung von Konstruktionsaufträgen erreicht. Über den Kontext kann ein Zulieferer und interne Mitarbeiter von BMW die Konstruktionsaufgaben verwalten und überschauen. Des Weiteren sind über den Kontext die Berechtigungen für den externen Konstrukteur administrierbar. Der Supplier Working Context ist logisch als ein übergeordnetes Element zu sehen, welches Arbeitsstrukturen für Zulieferaufträge bündelt.

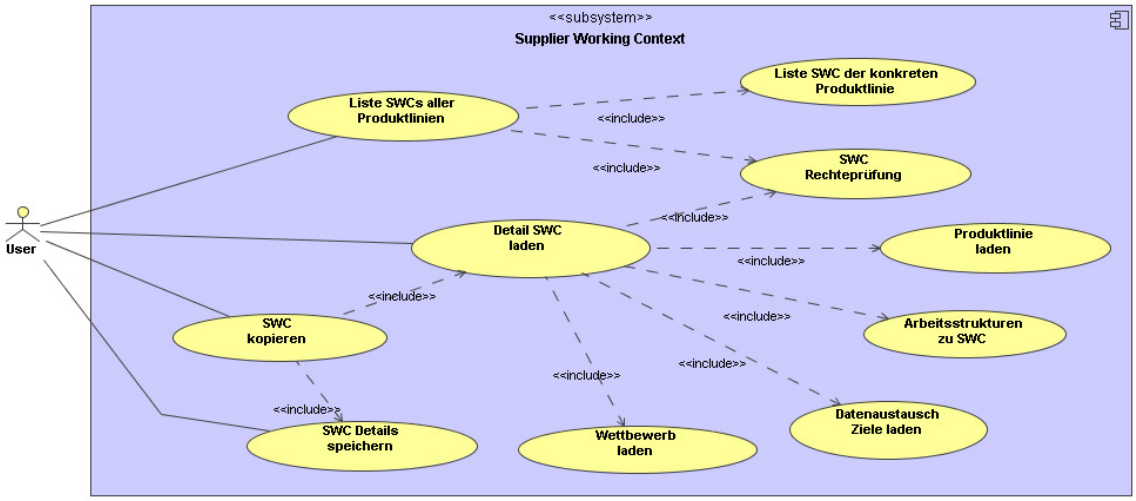


Abbildung 7. 6: Use Case SWC

In der Abbildung 7.6 sind die wichtigsten Fälle, die mit dem Supplier Working Context realisiert werden, aufgezeigt. Im folgenden Teil werden die dazugehörigen Screenshots näher erläutert damit sich ein Gefühl für die Applikation PEP-PDM einstellt.

Unter dem Menüpunkt SWC ist es dem Anwender möglich einen neuen Supplier Working Context anzulegen und näher zu spezifizieren. In der Maske Basisdaten müssen mit einem Stern markierte Pflichtfelder für das erfolgreiche Anlegen eines neuen Kontexts angegeben bzw. ausgewählt werden. In der Abbildung 7.7 ist die zugrunde liegende Maske abgebildet.

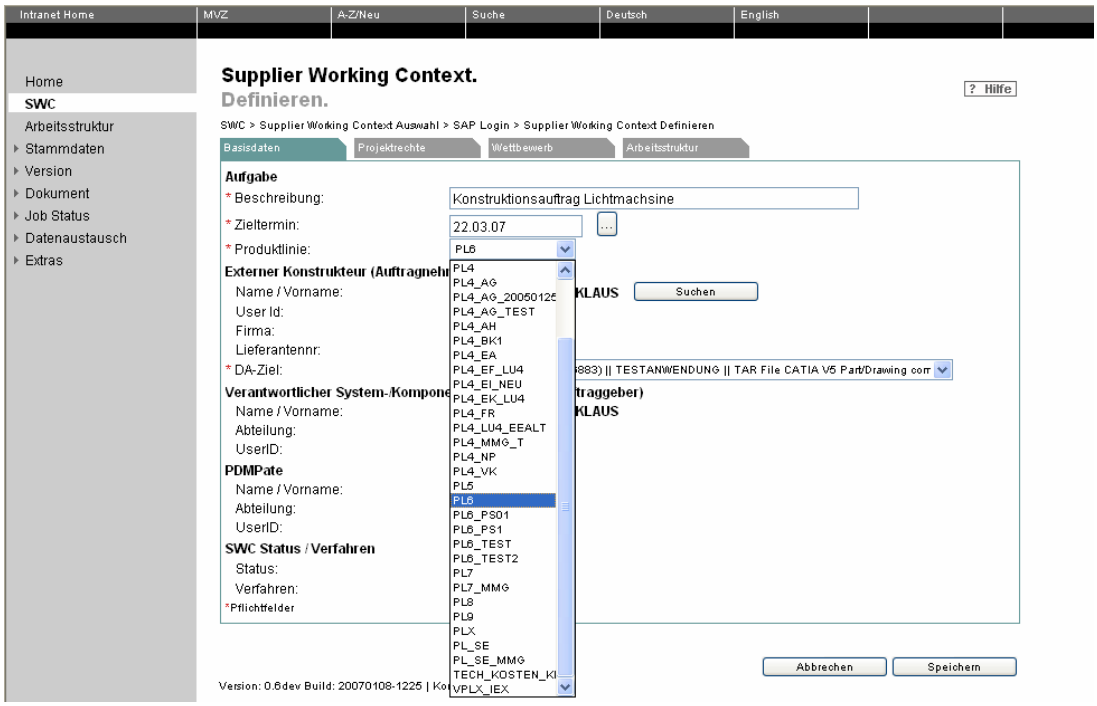


Abbildung 7. 7: Supplier Working Context Definieren

Diverse Pflichtfelder sind durch eine Drop-Down Combobox, bei denen es lediglich einer Selektion bedarf, realisiert. Die Combobox wird, durch im System bestehende Daten, beim Aufruf der Maske befüllt. In der Abbildung 7.7 und Abbildung 7.8 sind die zwei Comboboxen Produktlinie und Datenaustausch Ziel aufgezeigt.

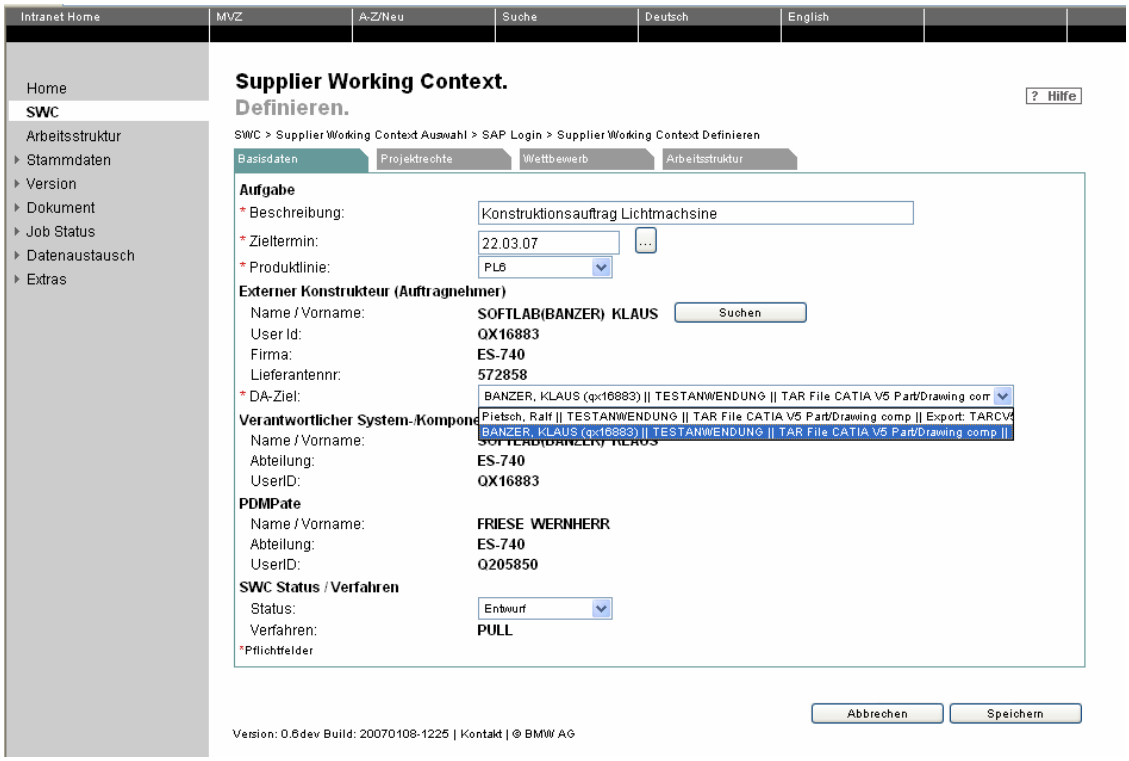


Abbildung 7. 8: Auswahl Datenaustausch Ziel

In der Applikation existieren, wie in der Abbildung 7.9 zu erkennen, viele Such-Masken um dem Anwender das Arbeiten so angenehm wie möglich zu machen. Die folgende Abbildung zeigt die Maske für die Suche von Arbeitsstrukturen.

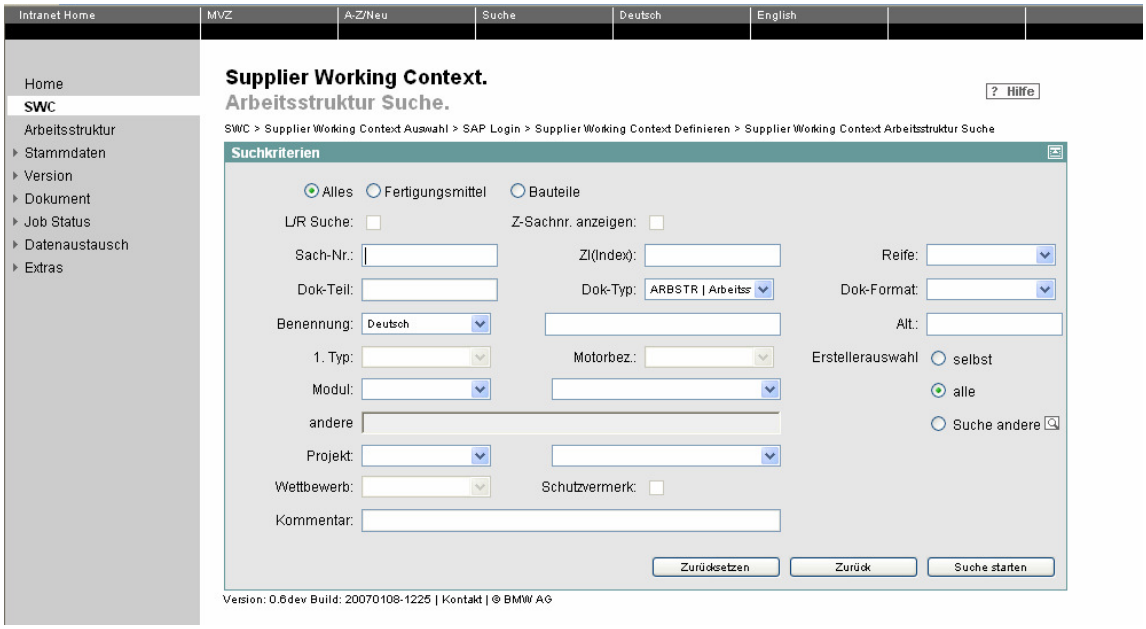


Abbildung 7. 9: Arbeitsstruktur Suche

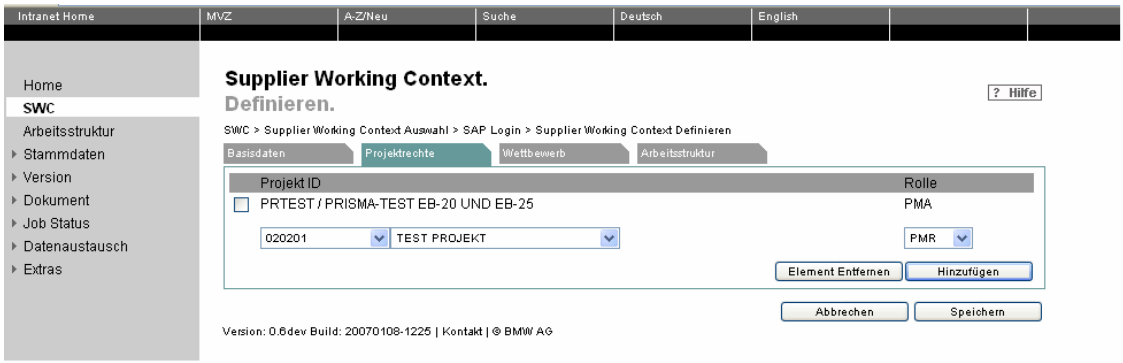


Abbildung 7. 10: Projektrechte

Beim definieren eines neuen Supplier Working Context müssen für die Zulieferfirmen Rechte vergeben und definiert werden. Diese Verteilung der Rechte wird über die Maske Projektrechte, wie in Abbildung 7.10 zu erkennen, realisiert.

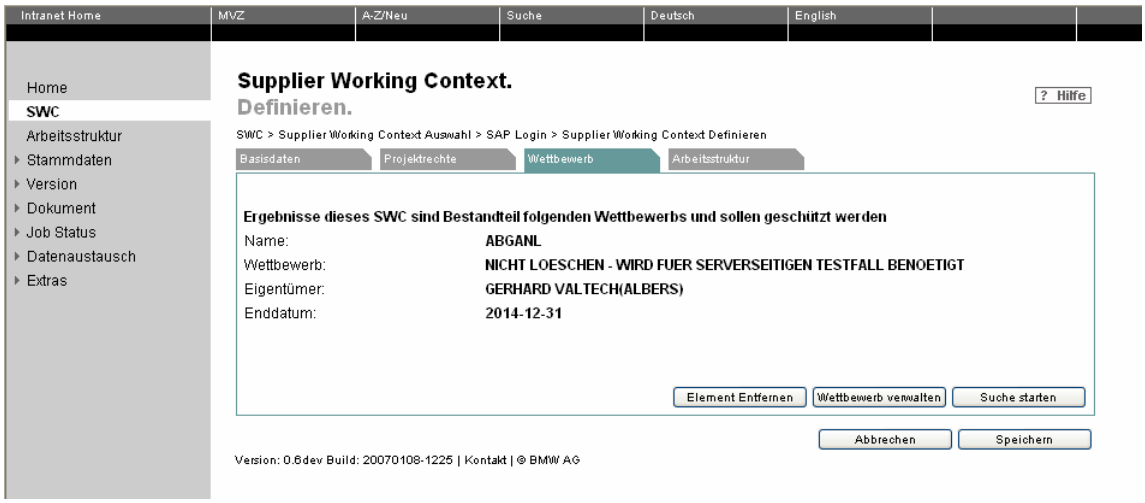


Abbildung 7. 11: Wettbewerb

Mit der Maske Wettbewerb werden die verschiedenen Wettbewerbe verwaltet. Ein Wettbewerb wird immer dann angelegt, wenn BMW mehrere Entwicklungspartner für die Konstruktion eines Bauteils beauftragt, um sich anschließend für das Konzept eines Wettbewerberteilnehmers zu entscheiden. Während des Wettbewerbs stellen die Teilnehmer ihre Ergebnisse in das System, damit BMW-Mitarbeiter diese begutachten können.

Für die Anlage von neuen SWCs wird zumeist ein schon bestehender Kontext kopiert, deshalb wird hier auf den Fall „SWC kopieren“ noch einmal näher und in schriftlicher Form eingegangen.
Man unterscheidet grundsätzlich zwei Arten von SWC’s:

Pull

Beim Pull-SWC kann dem Lieferanten Rechte zugeteilt werden damit dieser selbst Arbeitsstrukturen zusammenstellen und diese sich selbst übermitteln kann.

Push

Bei der Option Push werden dem Lieferanten keine Rechte zugeteilt. Der Lieferant ist somit vom SWC-Ersteller, zumeist ein interner Arbeiter von BMW, abhängig. Die Arbeitsstrukturen werden vom SWC-Ersteller zugeordnet und dann dem Datenaustauschziel übermittelt.

Der Aublauf „SWC-kopieren“ für den Fall Push:

1. Ein Actor kann sich alle existierende SWCs in einem Übersicht-Screen anzeigen lassen. Angezeigt werden allerdings nur die SWCs auf die der User Leserechte hat.
2. Der User wählt anschließend einen gewünschten Kontext aus den er kopieren will. Bei der Auswahl muss allerdings zwischen den zwei Fällen, PUSH und PULL, unterschieden werden. In diesem Ablauf ist der Push-Fall berücksichtigt. Näher Information zu den Fällen ist in [A020] zu finden.
3. Das Datenaustauschziel des SWC wird neu definiert oder ausgewählt. Es wird dem User eine Auswahlliste präsentiert, die die Ziele des SWC-Erstellers beinhaltet.
4. Neuer SWC wird mit folgenden Inhalt angelegt:
    a. Auftragsbeschreibung des Ursprungs-SWCs
    b. Zieltermin des Ursprungs SWC
    c. Produktlinie des Ursprungs SWC
    d. Ausgewähltes DA-Ziel
    e. Verknüpfte Arbeitsstrukturen werden als Alternativen angelegt und mit dem SWC verknüpft
    f. SWC-Ersteller von dem kopierten SWC ist der Autor

In der nachfolgenden Abbildung 7.12 ist der Übersicht-Screen des Anwenders zu sehen. Aus dieser Übersicht ist es dem User erlaubt einen Kontext mit dem aufgeführten Inhalt zu kopieren und sich, wie beschrieben, einiges an Arbeit zu ersparen.

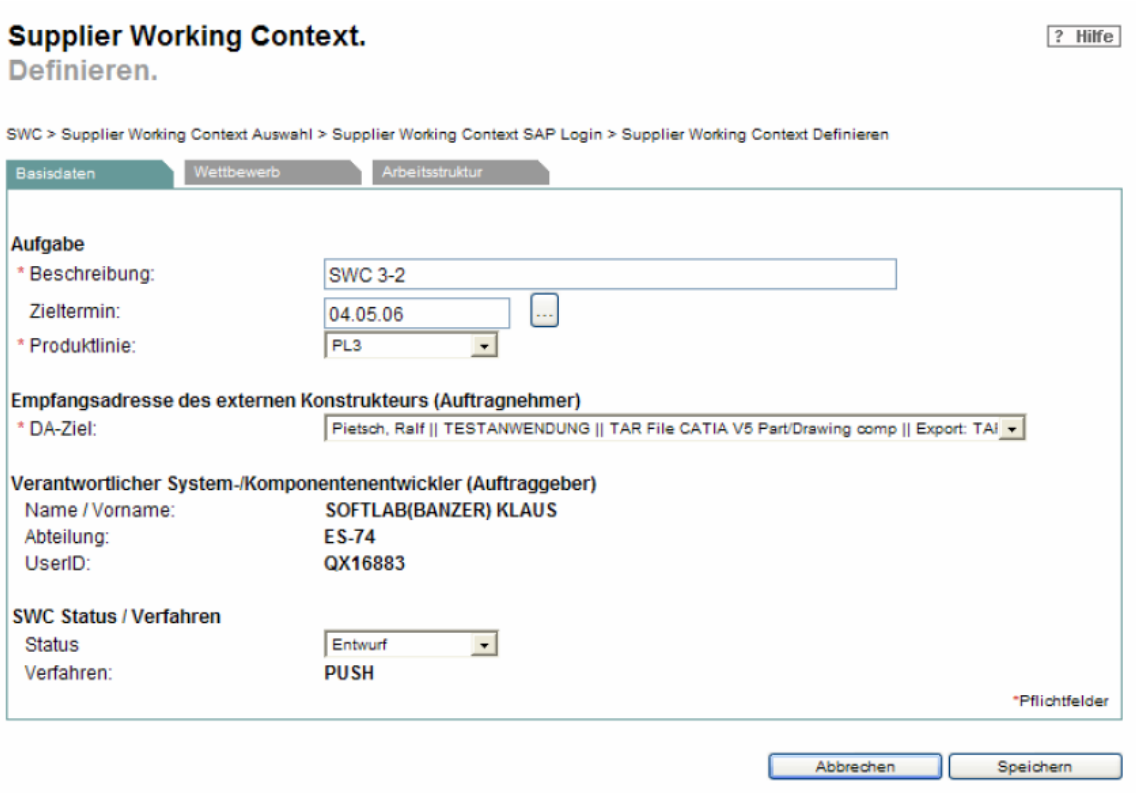


Abbildung 7. 12: Supplier Working Context definieren

7.4.3 SwcView Komponente

Über die SWC Komponente der Intermediate-Layer (com.bmw.pep.intermediate.swcview) und deren fachlichen Schnittstelle SwcViewBCI, werden die benötigten Funktionalitäten für den Supplier Working Context abgebildet.

Die Facade SwcViewFacadeBFBeanImpl referenziert in den Business-Layer über das Inter Component Interface SwcMgmtICI auf die SwcMgmt Komponente (Abbildung 7.13). In der Abbildung wurde ein Teil der Data Objects zur besseren Darstellung entfernt, da diese für das Verständnis nicht unbedingt benötigt werden.

Die Supplier Working Context Facade referenziert noch weitere Komponenten im Business-Layer über deren fachliche Schnittstellen (ICI). Es wird hier allerdings nicht detailliert auf die jeweiligen Objekte und deren Aufgaben eingegangen, da dies für die Erstellung der Diplomarbeit nicht relevant ist. Zum besseren Verständnis soll mittels des Use Case und der ausgewählten Komponente ein Beispiel aufgezeigt werden.

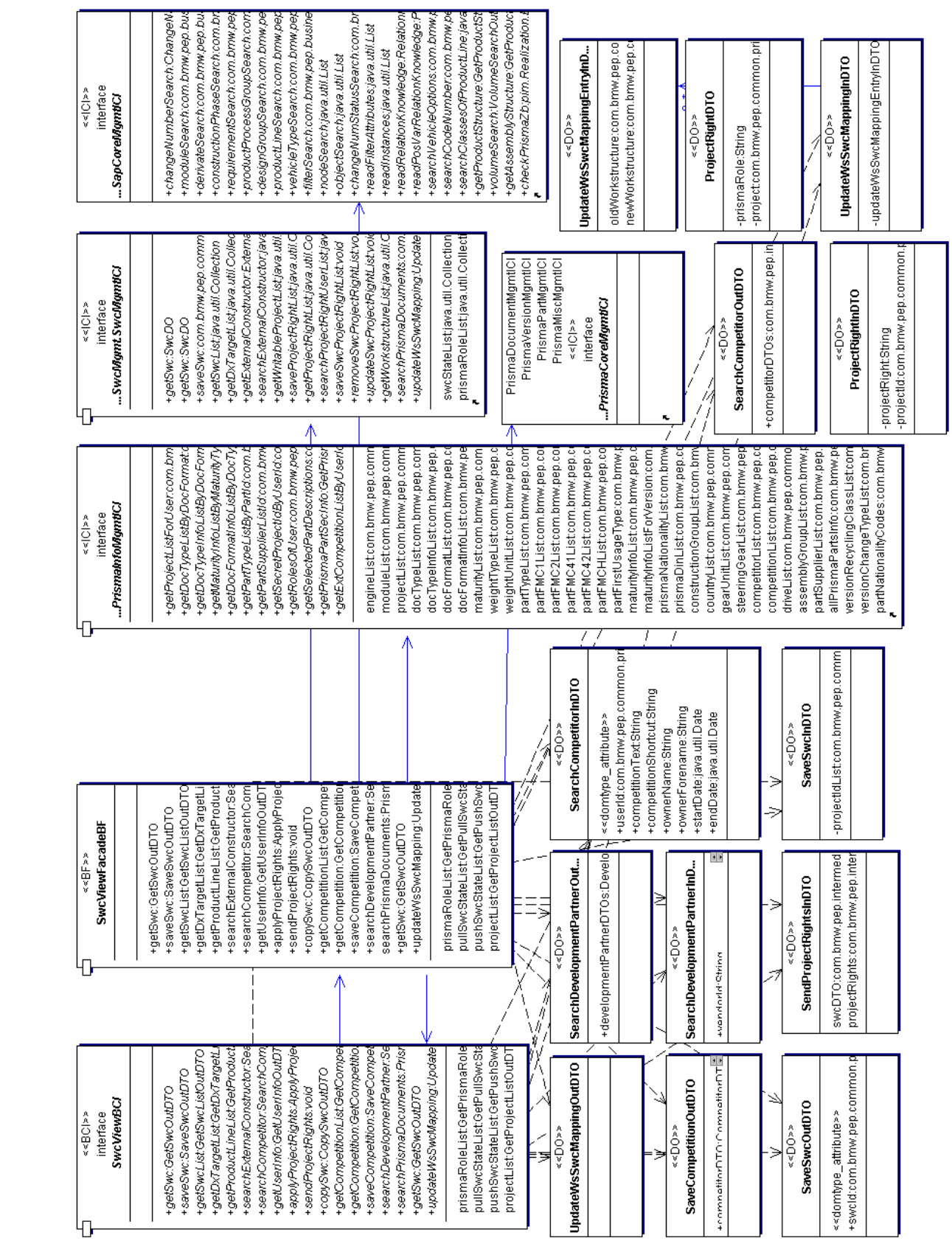


Abbildung 7. 13: Teilauszug der SwcView Komponente des Intermediate-Layers

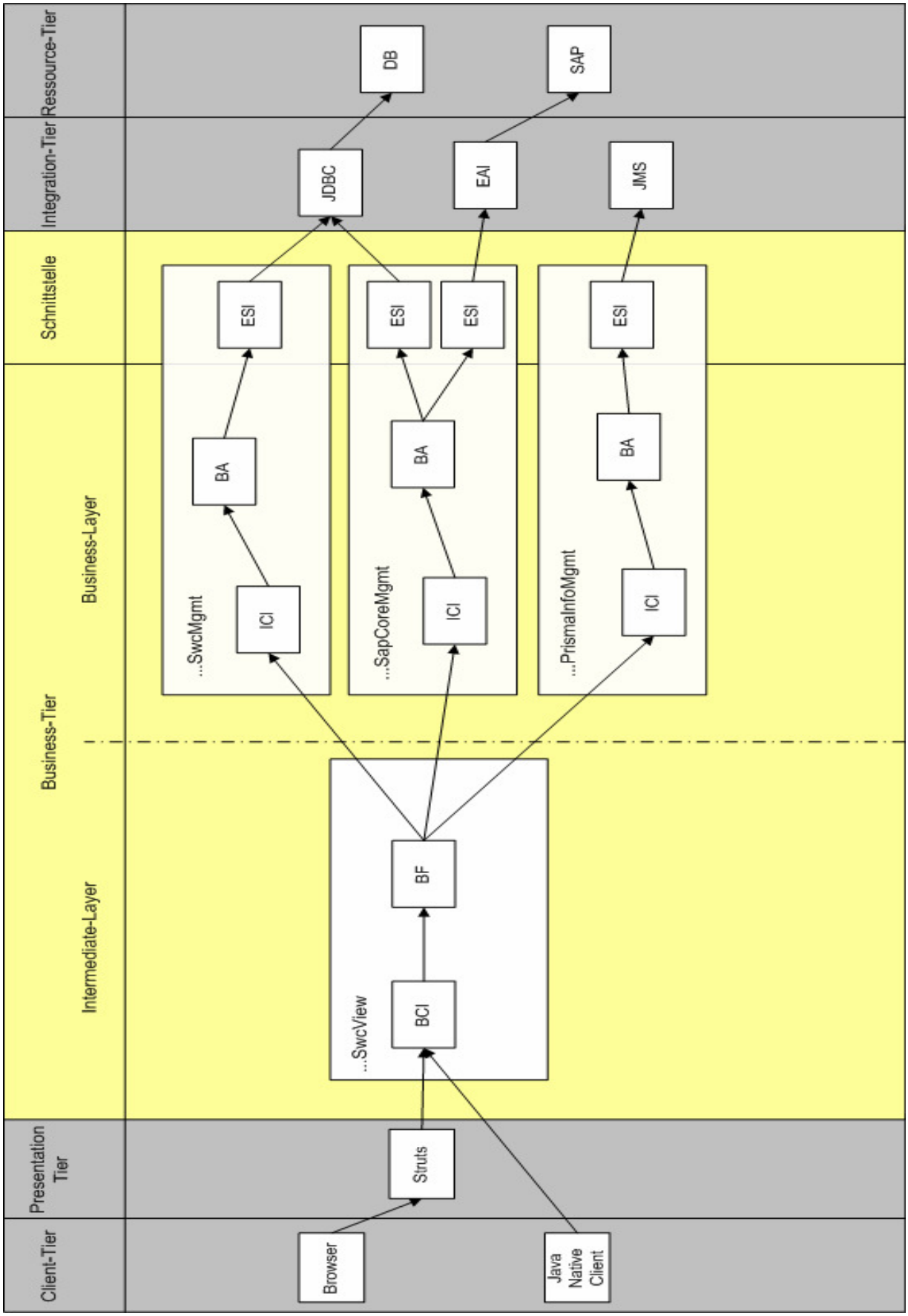


Abbildung 7. 14: SWC und verwendete Komponenten

7.4.4 Falltüren beim Erstellen des Designs

Die Entwicklung mit dem Modellierungstool Together ist nicht immer unproblematisch. Aufgrund diverser Möglichkeiten in Together ein Modell zu modellieren, sowie speziellen Eigenheiten, entstehen unterschiedliche Versionen des XMI-Dialekts der von Together aus dem Modell erstellt wird. Des Weiteren kann es auch Unterschiede geben wenn entweder der grafische Editor oder der Source Code Editor zur Erstellung des Anwendungsdesigns bevorzugt wird.
Im Folgenden werden die Falltüren, die für eine erfolgreiche Generierung mit dem openArchitectureWare Generator zu umgehen sind, erläutert.

- Interface vs. Class
- Spezielle Attribute
- XMI-Export
- Ressourcenintensiv
- Constraints zur Modellierungszeit

Interface vs. Class

In Together hat der Entwickler die Möglichkeit auf unterschiedliche Weisen bei der Erstellung des Anwendungsdesign vorzugehen. Bei der Erstellung des Anwendungsdesign ist es nicht relevant ob eine Schnittstelle als Interface in UML abgebildet wird oder als herkömmliche Klasse. Die plattformspezifische Abbildung wird erst über die Abbildungsvorschriften festgelegt und somit die jeweiligen Konstrukte des verwendeten Programmiermodells. Allerdings ist von Bedeutung, dass die Elemente mit den passenden Stereotypen markiert sind. Deshalb muss darauf geachtet werden, dass keine Stereotypen vergessen werden.
Bei dem Einlesen des Anwendungsdesign in den openArchitectureWare Generator wird keine Unterscheidung der Elemente auf ihren UML-Kontext vorgenommen (Class, Interface). Die Unterscheidung beruht ausschließlich auf den Stereotypen der Elemente.

Spezielle Attribute

Together hinterlegt bei der Modellierung einer Assoziation im Source Code einer Klasse ein privates Attribut „*private PrismaESI lnkPrismaESI*“ mit der Namenskonvention lnk<*targetName*>. Wenn eine Assoziation von einem Interface ausgeht wird richtigerweise kein Attribut angelegt.
In den Templates der Diplomarbeit werden Assoziationen nicht durch ein Attribut wie bei dem BMW-Transformator erkannt, sondern durch die Überprüfung der instanziierten Elemente.
Das Attribut welches Together in den Source Code schreibt kann bei der Generierung zu Problemen führen, da es als Attribut vom oAW-Generator erkannt und instanziiert wird. Somit

wird es normalerweise direkt in den Implementierungsrahmen übernommen, wenn in den Templates Attribute überführt werden. Eine recht unschöne Lösung ist auf den Präfix *lnk*-String zu prüfen und diese vermeidlichen Attribute bei der Abbildung außen vor zu lassen.

XMI-Export

Das exportierte XMI aus Together weißt diverse Mängel zur Spezifikation auf und bereitet somit Probleme. In der ersten Zeile der Datei wird laut Spezifikation das Encoding angegeben. In der exportierten Datei ist dies auch soweit richtig, allerdings besteht in Together nicht die Möglichkeit den Wert des Encoding manuell anzupassen. Dies wäre für den XMI-Parser von oAW eine benötigte Funktion um ohne Fehler das File einzulesen.
Together schreibt in das Attribut *encoding* den Wert *UTF8*, allerdings wird beim Parsen des XMI von oAW der Wert *UTF-8* vorausgesetzt.

Ein weiteres Problem in der XMI-Datei ist die fehlende Protokollangabe bei dem Namespace-Attribut. Auch hier setzt der openArchitectureWare-Parser das in Together fehlende *http://* voraus.
In dem folgenden Listing sind noch einmal die problematischen Zeilen in richtiger Form aufgeführt.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.1' xmlns:UML = 'http://org.omg/UML/1.3'>
```

Listing 7. 2: Problematische XMI Zeilen

Ressourcenintensiv

Together selbst benötigt einen performanten PC um damit aktiv und sinnvoll Anwendungen zu designen und entwickeln. Dieser Wermutstropfen sollte auf keinen Fall unterschätzt werden, da sich die komplette Entwicklung zeitlich nach hinten verlagert. Dieser Punkt wird hier aufgrund zeitintensiver Erfahrung aufgeführt.

Constraints zur Modellierungszeit

Ein sehr problematischer Punkt beim BMW-Ansatz ist die fehlende Validierung der Modelle zur Modellierungszeit. Dieser Punkt soll hier besonders hervorgehoben werden, da dies ein Mechanismus zur frühzeitigen Fehlererkennung und Behebung ist.
In der derzeitigen Version ist dem Entwickler gestattet alle Möglichkeiten, die die UML mitbringt und das Tool Together unterstützt, für die Modellierung zu verwenden. Außerdem existiert, wie erwähnt, keine zeitnahe oder besser gesagt, überhaupt keine Validierung des

Modells. Das konkrete domänenspezifische Metamodell schränkt die Verwendung und die Möglichkeiten der Elemente ebenfalls nicht ein.
Hier werden die Themen EMF/GMF, welche im Ausblick behandelt werden, ansetzen. Bei der Modellierungssprache EMF und der grafischen Darstellung GMF, werden eben genau nur die Möglichkeiten, welche im domänenspezifischen Metamodell definiert wurden, unterstützt. Durch dieses Verhalten ist es dem Entwickler erst gar nicht möglich, zum Beispiel eine Assoziation zwischen 2 Elementen einzuführen, wo dies das Metamodell nicht vorsieht.

In der Realisierung der Diplomarbeit wird über die vom openArchitectureWare Framework mitgebrachte Sprache Check ein Teil der Constraints abgefragt. Durch dieses Vorgehen kann noch vor der Generierung der Artefakte bei Verstoß gegen einen Constraint abgebrochen werden. Diese Möglichkeit bietet der BMW-Ansatz nicht. Direkter Bezug auf die Constraints sind unter Kapitel 7.13 zu finden.

7.5 Erstellung des Workflows

Die Erstellung des Workflows ist ein zentraler Bestandteil bei der Entwicklung mit dem openArchitectureWare Framework. Über den Workflow wird der gesamte Generierungsprozess gesteuert und kann hier auch an zentraler Stelle konfiguriert werden (vgl. Kapitel 6.3.2.1).
Der Workflow besteht aus Components, die ähnlich wie Subroutinen Aufgaben erledigen. Zum Beispiel ist über selbstimplementierte Components es möglich, externe Programme in den Generierungsprozess mit einzubeziehen.

Folgende Aufgaben müssen im Workflow realisiert werden:

- Initiieren der Umgebung damit das „classic“ Metamodell genutzt werden kann
- XMI File parsen und Metamodell instanziieren
- Generator zur Codegenerierung starten
- XDoclet-Lauf starten

7.5.1 Components

Components sind das zentrale Konzept der Workflow Engine. Durch die einzelnen Components die im Workflow definiert werden, wird ein Generierungsschritt repräsentiert.
In der Realisierung verwendete Components:

- XMIInstantiator

- CheckComponent
- DirectoryCleaner
- Generator
- AntStarter

XMIInstantiator

Die Component XMIInstantiator wird zum Parsen des toolspezifischen XMI-File und anschließendem Instanziieren des Metamodells verwendet. Folgende Parameter werden der Component übergeben:

- Instantiatorumgebungs-Slot: In diesem Slot wird das instanziierte Metamodell hinterlegt. Über solche Slots können die Components untereinander kommunizieren. Ein Slot ist eine Art Zwischenspeicher in den Components ihre Ergebnisse ablegen können, damit die andere Components diese verwenden können.
- modelFile: Angabe der XML-Datei des serialisierten Anwendungsdesign.
- xmlMapFile: Angabe des Metamappings-File. Das Mapping-File wird in Kapitell 7.10 näher erläutert.
- toolAdapterClassname: Angabe der passenden Tooladapter Klasse für das verwendete Modellierungstool. Der passende Adapter wird wegen den unterschiedlichen XMI-Dialekte benötigt.

CheckComponent

Durch die Einbindung der CheckComponent kann das instanziierte Anwendungsdesign unter zu Hilfenahme der domänenspezifischen Sprache (Metamodell) und der implementierten Constraints validiert werden.

Angaben die die Component zur Konfiguration benötigt:

- Metamodel: Angabe des zu verwendeten Metamodells
- checkFile: Pfad auf das File in dem die Constraints ausgedrückt sind
- expression: Einstiegspunkt im Modell definieren

DirectoryCleaner

Die DirectoryCleaner Component wird zum Löschen von Verzeichnissen verwendet. In der Realisierung werden die Verzeichnisse, in denen das Generat liegt, mit dieser Component vor der Generierung neuer Artefakte, gelöscht. Das oder die zu löschenden Verzeichnisse werden in der Workflow- oder Property-Datei als Value angegeben. Wenn mehrere Verzeichnisse gelöscht werden sollen, müssen die Verzeichnisse kommasepariert angegeben werden.

Generator

Die Generator Component ist für die Generierung der Artefakte zuständig. Aus dieser Component wird das Java Metamodel, welches bei der Auslieferung von oAW dabei ist, geladen. Anschließend werden die Templates an die passenden Modelelemente gebunden. Die Ausgabeverzeichnisse werden definiert und ein Codebeautifier wird nach der Generierung angestoßen.

Wichtige Elemente in der Generator Component sind:

- expand: Das Expand Element wird für die Bindung der Templates an die Modellelemente herangezogen.
- outlet: Über das outlet-Element können Verzeichnisse definiert werden in denen das Generat abgelegt wird. Der Name des Outlet wird in den Templates zur Referenzierung angegeben.

AntStarter

Die Component AntStarter ist für den Anstoß des XDoclet-Lauf verantwortlich. Dieser Lauf wird für die Generierung der Descriptor-Artefakte benötigt. Zur Verringerung der Generierungsstufen wird dieser Schritt in den Workflow miteingebunden und bei einem Generierungslauf automatisch gestartet. Der Unterschied zu dem Ansatz, der im BMW-Generator verfolgt wird, ist in der Abbildung 7.15 aufgeführt:

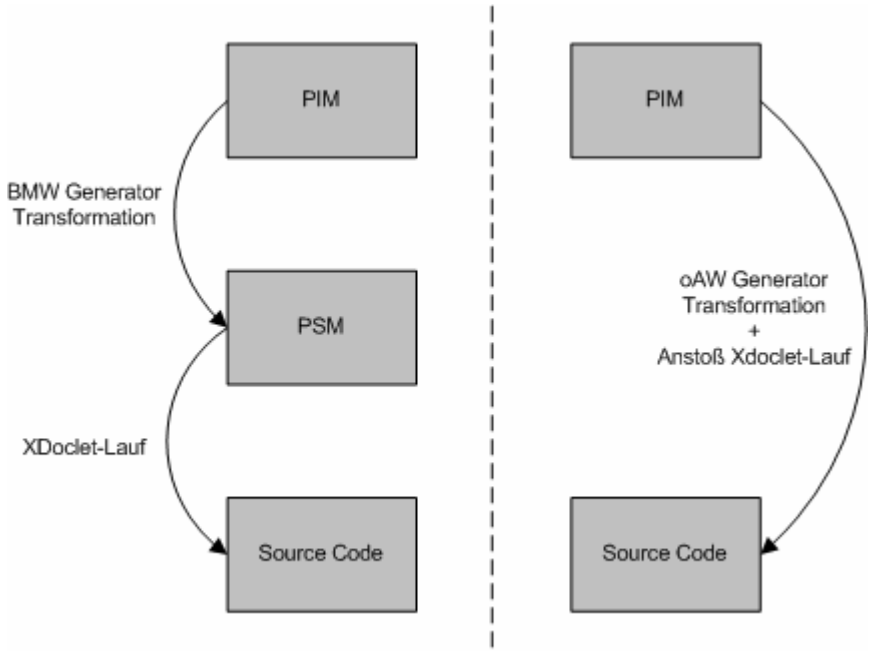


Abbildung 7. 15: Verringerung der Transformationsschritte

Eine nähere Erläuterung warum dieses Konzept gewählt und realisiert wurde erfolgt weiter unten.

7.5.2 Cartridges

Bestehende, in sich abgeschlossene Workflow Skripts, die in den eigenen Workflow eingebunden werden nennt man Cartridges. Es werden die folgenden Cartridges bei der Implementierung inkludiert:

- classicstart.oaw: Initiiert die Slots (Kontext) in welche das instanziierte Metamodell gelegt wird. Konkret wird der InstantiatorEnvironmentSlot und der MetaEnvironmentSlot angelegt und registriert.

- classicinit.oaw: Dieses Cartridge initialisiert das Model welches im Slot hinterlegt ist. Es werden alle Constraints überprüft und beim Aufkommen einer Verletzung eine Fehlermeldung ausgegeben und den Generierungslauf gegebenenfalls unterbrochen.

- classicfinish.oaw: Diese Cartridge ist für die Dump-Ausgabe (Fehlermeldungen) verantwortlich. In der Realisierung wurde selten von der Dump-Ausgabe Gebrauch gemacht. Stattdessen wurde über Log4J ein Logging-Framework eingebunden, welches bei Fehlern aussagekräftige Meldungen erstellt.

Die aufgeführten Cartridges und Components in den hervorgegangen Erläuterungen sind im folgenden Listing aufgeführt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<workflow>
    <property file="workflow.properties"/>
    <property name="antStarter" value="pep_ca20_extend.AntStarter"/>

        <cartridge file="org/openarchitectureware/workflow/oawclassic/classicstart.oaw">
            <metaEnvironmentSlot value="me"/>
                <instantiatorEnvironmentSlot value="ie"/>
        </cartridge>

        <!-- Model xmi file will be instanciate. -->
    <component
class="org.openarchitectureware.core.frontends.xmi.workflow.XMIInstantiator">
                <instantiatorEnvironmentSlot value="ie"/>
                <modelFile value="${model.xmi}"/>
                <xmlMapFile value="${toolMappingFile}"/>
                <metaMapFile value="${metaMapFile}"/>
                <toolAdapterClassname value="${toolAdapterClassname}"/>
        </component>

        <cartridge file="org/openarchitectureware/workflow/oawclassic/classicinit.oaw">
            <metaEnvironmentSlot value="me"/>
        </cartridge>

        <component id="dirCleaner"
class="org.openarchitectureware.workflow.common.DirectoryCleaner" skipOnErrors="false">
                <directories value="${srcGenPath}"/>
        </component>

        <component id="generator" class="org.openarchitectureware.xpand2.Generator"
skipOnErrors="false">
            <metaModel class="org.openarchitectureware.type.impl.java.JavaMetaModel">
                        <typeStrategy
class="org.openarchitectureware.type.impl.oawclassic.OAWClassicStrategy"
convertPropertiesToLowerCase="false"/>
            </metaModel>
           <fileEncoding value="ISO-8859-1"/>
              <expand value="Root::Root FOREACH me.getElements('Model')"/>
                <outlet path='main'/>
                <outlet name='PSM' path='psm' overwrite='true'/>
        <outlet name='XDoclet' path='xdoclet_output/java' overwrite='true'/>

        <outlet name='MANUAL' path='manual/psm' overwrite='false'/>

        <beautifier class="org.hybridlabs.source.formatter.JavaImportBeautifier"
                    conventionFilePath=""
                    organizeImports="false"
                    format="true"/>
                <beautifier
class="org.openarchitectureware.xpand2.output.XmlBeautifier"/>
        </component>

        <cartridge file="org/openarchitectureware/workflow/oawclassic/classicfinish.oaw">
                <instantiatorEnvironmentSlot value="ie"/>
                <dumpfile value="${dumpfile}"/>
        </cartridge>

        <component class="${antStarter}"></component>
</workflow>
```

Listing 7. 3: Workflow

7.6 Templates

Für die Transformation in den Implementierungsrahmen auf Basis der Component Architecture und J2EE wird der Template-Mechanismus des openArchitectureWare Frameworks verwendet (vgl. Kapitel 6.3.2.3). In den Templates werden die Abbildungsvorschriften hinterlegt und Gebrauch von Design-Patterns gemacht.

Bei einer Implementierung „From Scratch“, würden die Templates von der Referenzimplementierung abgeleitet werden. Der Ausgangspunkt in dieser Diplomarbeit ist allerdings ein anderer und die Templates können zumindest teilweise aus den Abbildungsvorschriften von Together abgeleitet werden.

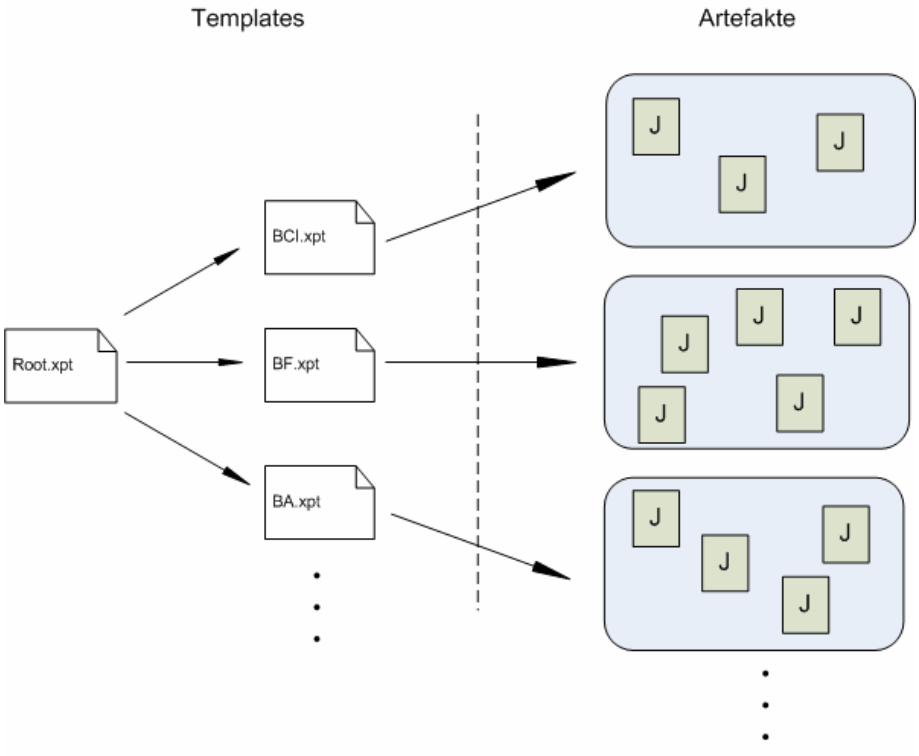


Abbildung 7. 16: Template-Expansion

Template-Dateien sind Namensräume in denen selbst wieder Templates definiert werden können. Es muss und sollte nicht für jedes Template ein extra Template-File existieren. Viel mehr macht es Sinn, zusammengehörige Templates (im Normalfall die Templates für ein CA-Element) in einem generischen Template zusammen zu halten und aus diesem Template, über ein weiteres Util-Template, Funktionen die zur Übersicht ausgelagert wurden, zu inkludieren.

Die Package-Struktur, in der die Templates abgelegt sind, ist an den Aufbau der Projektstruktur von PEP-PDM angelehnt. Das heißt, die Templates werden in gleichnamigen Packages, wie die Artefakte, die aus den Templates generiert werden, abgelegt.

Package spec

In dem Package „spec“ sind die fachliche Schnittstellen einer Business Component hinterlegt.

- BA.xpt
- BA_util.xpt

Package facade

Das Package „facade“ beinhaltet die Business Facade Templates der Business Component. Die Facade ist für die Ablaufsteuerung der BC verantwortlich.

- BF.xpt
- BF_util.xpt

Package behavior

Das Package „behavior“ beinhaltet die Business Activities Templates, die für Teilprozesse oder Aktivitäten verantwortlich sind.

- BA.xpt
- BA_util.xpt

Package dto

Das Package „dto“ beinhaltet die Templates und deren Abbildung der Data Objects.

- DO.xpt

Package ici

Das Package „ici“ beinhaltet die Templates und deren Abbildung des Inter Component Interface.

- ICI.xpt
- ICI_util.xpt

Package integration

Das Package “integration” beinhaltet die Templates und deren Abbildung des External Service Interface.

- ESI.xpt
- ESI_util.xpt

Grundlegendes Konzept des Aufbaus der Templates ist in der Abbildung 7.17 dargestellt:

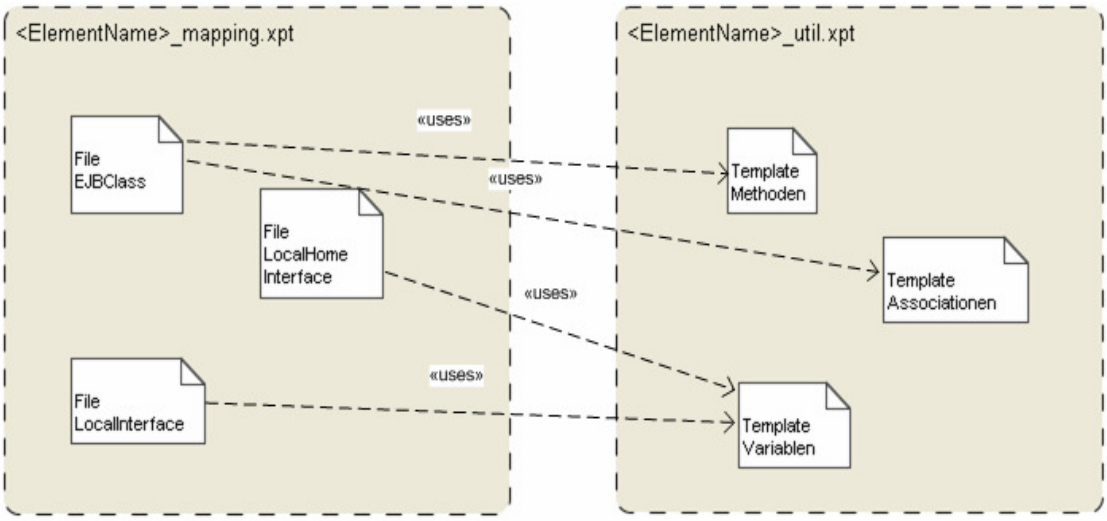


Abbildung 7. 17: Template-Konzept

Die Realisierung beinhaltet folgende Artefakte:

Klassen und Interfaces

- Business Component Interface
  - Factory
  - Business Interface
  - Session Bean
  - Mock-Artefakt
  - Test-Artefakt

- Business Facade
  - Delegate Session Bean
  - Session Bean
  - Remote/Home Interface
  - Session Class
  - Utility Class

- Business Activity
  - Session Bean
  - Remote/Home Interface
  - Session Class
  - Utility Class

- Inter Component Interface
    - Factory
    - ICI Interface
    - Session Bean
    - Mock-Artefakt
    - Remote/Home Interface

- External Service Interface
    - Factory
    - ESI Interface
    - Mock-Artefakt
    - Service Class

- Event Listener
    - Message Driven Bean

- Data Object
    - POJO[13]

7.6.1 Root-Template

Der zentrale Einstiegspunkt für den Generator ist das Root-Template. Das Root-Template wird, wie oben aufgeführt, über den Workflow als Einstiegspunkt angegeben (vgl. Kapitel 7.5). Der Generator schaut im Classpath nach einem File Root.xpt in welchem ein oder mehrere Root-Elemente definiert sind.
Im Define-Ausdruck werden die jeweiligen Elemente (Model, Package, Object, DO, BA, u.v.m.), die für diesen Block expandiert werden sollen, angegeben (vgl. Kapitel 6.3.2.3). Hier wird der Polymorphie-Mechanismus von Xpand verwendet um rekursiv jedes Element im Model zu expandieren.

Benutzter Einstiegspunkt in der Realisierung:

- Model: Es wird im Modell an oberster Stelle der Einstiegspunkt definiert und dann rekursiv der Hierarchie gefolgt

[13] POJO – Abkürzung für Plain Old Java Object, also ein ganz normales Objekt im Programmiermodell Java.

Bestehende Einstiegspunkte:

- BA: Der Business Activity Einstiegspunkt expandiert die Templates, die nur für die Business Activity zuständig sind

- ICI: Der Internal Component Interface Einstiegspunkt expandiert die Templates, die ausschließlich für die Abbildung der ICI Elemente zuständig sind

- DO: Der Data Object Einstiegspunkt expandiert die Templates, die ausschließlich für die Abbildung der DO Elmente zuständig sind

- BF: Der Business Facade Einstiegspunkt expandiert die Templates, die ausschließlich für die Abbildung der BF Elemente zuständig sind

- BCI: Der Business Component Interface Einstiegspunkt expandiert die Templates, die ausschließlich für die Abbildung der BCI Elemente zuständig sind

- ESI: Der External Service Interface Einstiegspunkt expandiert die Templates, die ausschließlich für die Abbildung der ESI Elemente zuständig sind

Die hier definierten Einstiegspunkte kommen im Projektumfeld zum Einsatz. Durch diese feine Gliederung können bei einer inkrementellen Implementierung ausschließlich die Elemente mit dem Einstiegspunkt-Element generiert werden und somit die Zeit der Generierungsphase bei größeren Modellen enorm verkürzen.
Diese definierbaren Einstiegspunkte sind, für die Verwendung der generativen Architektur im Projektumfeld, sicherlich ein Entscheidungskriterium für die Akzeptanz.

```
«IMPORT org::openarchitectureware::core::meta::core»
«IMPORT org::openarchitectureware::meta::uml::classifier»
«IMPORT pep_ca20_uml_meta»

«DEFINE Root FOR Model»
      «EXPAND Root FOREACH OwnedElement»
«ENDDEFINE»

«DEFINE Root FOR Package»
      «EXPAND Root FOREACH OwnedElement»
«ENDDEFINE»

«DEFINE Root FOR Object»
«ENDDEFINE»

«DEFINE Root FOR DO»
      «EXPAND mappings::dto::DO_mapping::DOClass»
«ENDDEFINE»

«DEFINE Root FOR BF»
      «EXPAND mappings::facade::BF_mapping::BFBeanClass»
«ENDDEFINE»

«DEFINE Root FOR BCI»
      «EXPAND mappings::spec::BCI_mapping::BCIClass»
«ENDDEFINE»

«DEFINE Root FOR BA»
      «EXPAND mappings::behavior::BA_mapping::BAClass»
«ENDDEFINE»

«DEFINE Root FOR DAO»
      «EXPAND mappings::dao::DAO_mapping::DAOClass»
«ENDDEFINE»

«DEFINE Root FOR EL»
      «EXPAND mappings::listener::EL_mapping::ELClass»
«ENDDEFINE»

«DEFINE Root FOR ESI»
      «EXPAND mappings::integration::ESI_mapping::ESIClass»
«ENDDEFINE»

«DEFINE Root FOR ICI»
      «EXPAND mappings::ici::ICI_mapping::ICIClass»
«ENDDEFINE»

«DEFINE Root FOR DOM»
      «EXPAND mappings::domain::DOM_mapping::DOMClass»
«ENDDEFINE»
```

Listing 7. 4: Root-Template

7.6.2 BA-Templates

Im Package *mappings.behavior* ist der Namensraum für die Business Activity definiert. Business Activities bilden in der Component Architecture fachliche Teilprozesse oder Aktivitäten (Use Cases) ab. Folgende Artefakten werden aus den Templates der Business Activity erstellt.

Bean-Klasse

Für ein <<*BA*>> Element im PIM wird eine stateless Session Bean <*Elementname*>BABean.java generiert. Die Bean Klasse implementiert die Geschäftsmethoden aus dem lokalen Komponenten Interface.

Bean-Klasse-Impl

Die Klase <*Elementname*>BABean_protectedImpl.java erbt von der generierten Bean-Klasse <*Elementname*>BABean.java. Diese Klasse wird bei einem iterativen Entwicklungsprozess nicht überschrieben und wird deshalb für den manuell hinzugefügten Code verwendet. Durch diese Erweiterung der Bean Klasse werden die „protected Regions“ umgangen (vgl. Kapitel 6.3).

Bean-LocalHome Interface

Das Local Home Interface <*Elementname*>BALocalHome.java deklariert Methoden, die für den Lebenszyklus der Bean zuständig sind, wie zum Beispiel die create()-Methoden mit der der Client eine Referenz auf das Komponenten Interface erhält.

Bean-Local Interface

In dem Local Interface <*Elementname*>BALocal.java sind die Geschäftsmethoden deklariert. Ein lokaler Client kann dieses Interface für den Aufruf der Methoden verwenden.

Bean-Session

Die Klasse <*Elementname*>BASession leitet von der Bean-Klasse <*Elementname*>BABean ab. Sie beinhaltet die Methode ejbCreate(), die mit der Methode des Home Interfaces der Bean-Klasse korrespondiert.

Bean-Util

Die Klasse <*Elementname*>BAUtil implementiert statische Methoden, die für die Realisierung von Lazy Loading verwendet werden. In dieser Klasse wird das Local Home in einer statischen Referenzvariabel gecached. Es wird die Anzahl der Lookups verringert und somit eine bessere Perfomance erreicht.

7.6.3 BF-Templates

Im Package *mappings.facade* ist der Namensraum für die Business Facade definiert. Die Session Facade stellt den externen Clients die fachlichen Dienste der Komponente zur Verfügung. Die Clients dürfen ausschließlich nur über das Delegate BCI auf die Facade zugreifen. Folgende Artefakte werden aus den Templates der Business Facade erstellt.

Bean-Klasse Delegate

Für ein <<*BF*>> Element im Anwendungsdesign (PIM) wird eine stateless Session Bean <*Elementname*>BFBean.java generiert. Die <*Elementname*>BFBean.java ist ein Delegate, der als Interceptor für Facade-Aufrufe genutzt werden kann.

BeanImpl

Die Klasse <*Elementname*>BFBeanImpl implementiert das Delegate Interface und ist somit für die Implementierung der Facade zuständig.

BeanImpl-Impl

Die Klase <*Elementname*>BFBeanImpl_protectedImpl.java erbt von der generierten Bean-Klasse <*Elementname*>BFBean.java. Diese Klasse wird bei einem iterativen Entwicklungsprozess nicht überschrieben und wird deshalb für den manuell hinzugefügten Code verwendet.

Bean Remote Home Interface

Das Remote Home Interface <*Elementname*>BFHome.java deklariert Methoden, die für den Lebenszyklus der Bean zuständig sind, zum Beispiel die create()-Methoden mit der der Client eine Referenz auf das Komponenten Interface erhält.

Bean-Remote-Interface

Das Interface <*Elementname*>BF.java deklariert die Geschäftsmethoden des Bean, die von dem Remote Client aufgerufen werden können.

Bean-Session

Die Klasse <*Elementname*>BFSession beinhaltet die Methode ejbCreate() die mit der Methode des Home-Interfaces der Bean-Klasse korrespondiert. Die Klasse erbt von der Bean-Klasse <*Elementname*>BFBean.

Bean-Util

Die Klasse <*Elementname*>BFUtil implementiert statische Methoden die für die Realisierung von Lazy Loading verwendet werden. In dieser Klasse wird das Local Home in einer statischen Referenzvariabel gecached.

7.6.4 BCI-Templates

Im Package *mappings.spec* ist der Namensraum für das Business Component Interface definiert. Nach der Abbildung auf J2EE entspricht das BCI dem Business Delegate Design Pattern. Das Business Delegate bietet eine Abstraktionsschicht für die Implementierung der Businessschicht (API) und verhindert zum Beispiel eine Kopplung zwischen Presentation- und Business-Tier. Es verwendet einen Service Locator, der im CA2.0 Framework mit ausgeliefert wird, für den Zugriff auf die Business Facade. Folgende Artefakten werden aus den Templates des Business Component Interface erstellt:

Interface

Das <*Elementname*>BCI.java Interface stellt die fachliche Schnittstelle als plattformunabhängiges Java Interface zur Verfügung. Für die Implementierung sind nur J2SE-Exeption und neutrale Parameter sowie Rückgabewerte zulässig.

Factory-Klasse

Die Factory ist für die transparente Erzeugung der Business Delegate Implementierung zuständig. Die Factory Klasse selbst ist als Singelton implementiert. Der Factory ist nur das Interface bekannt, deshalb kann die konkrete Implementierung unbemerkt zum Beispiel für Testzwecke ausgetauscht werden.

EJBBean-Klasse

Die EJB-Bean Klasse EJB<*Elementname*>BCI.java realisiert das Business Delegate Interface und besorgt sich im Konstruktur die Referenzen auf die Business Facade.

Mock-Klasse

Das Mock Objekt implementiert das Business Delegate Interface. Dieses Dummy Objekt wird für Unit Tests verwendet.

Test-Klasse

Die Test Klasse <*Elementname*>BCITest.java erbt von dem JUnit Framework welches bei der CA2.0 mit ausgeliefert wird.

7.6.5 DO-Templates

Im Package *mappings.dto* ist der Namensraum für das Data Object definiert. Die DOs repräsentieren die fachlichen Parameter bzw. Rückgabewerte. Folgende Artefakte werden aus den Templates der Data Object erstellt:

DOClass

Ein DO wird als einfaches POJO (Plain old java object) <*Elementname*>DO.java oder <*Elementname*>DTO.java mit den im Modell modellierten Membervariablen sowie deren Getter und Setter abgebildet. Das DO dient als Parameter und Rückgabewert der fachlichen Schnittstelle. Die DO Klassen erben von dem AbstractPepValueObject, welches seinerseits das Serializable Interface implementiert, somit sind die DOs serialisierbar und können für Remote Clients über RMI-IIOP[14] verwendet werden.

7.6.6 EL-Templates

Im Package *mappings.listener* ist der Namensraum für die Event Listener definiert. Der Event Listener ermöglicht die asynchrone Ausführung der BCI und ICI Methoden. Standardmäßig wird der Event Listener auf eine Message Driven Bean abgebildet. Dies kann aber nur erfolgen wenn die Reihenfolge der ankommenden Messages keine Rolle spielt.
Bei der Message Driven Bean handelt es sich um einen technischen Client der über das Interface des Business Delegates oder ICI mit der Komponente kommuniziert. Bei einer fachlichen Transaktion muss die Message Driven Bean eine Transaktion der Komponente übernehmen können. Die Message Driven Bean besorgt sich aus der JMS[15]-Destination die Message und packt die enthaltenen Parameter aus.

Folgende zwei Artefakte werden durch das Template erstellt:

Container Managed Message Driven Bean

Kommuniziert die Message Driven Bean über das Business Delegate mit einer Business Facade muss die Message Driven Bean keine Transaktion initiieren können. Somit wird in diesem Fall eine Container Managed Message Driven Bean CM<*Elementname*>Bean.java verwendet.

Bean Managed Message Driven Bean

Die Transaktionssteuerung wird bei der Kommunikation mit einer Business Activity, die kein Transaktionshandling besitzt, über ein Inter Component Interface von der Message Driven Bean

[14] RMI-IIOP – Remote Methode Invocation over Internet Inter-ORB Protocol
[15] JMS – Java Message Service

übernommen. Aus diesem Grund wird eine Bean Managed Message Driven Bean <*Elementname*>Bean.java verwendet.

7.6.7 ESI-Templates

Im Package *mappings.integration* ist der Namensraum für das External Service Interface definiert. Dieses Interface ist für die Integration nicht kompatibler CA2.0 Komponenten verantwortlich. Die Realisierung erfolgt nach dem klassischen Adapter Pattern. Folgende Artefakte werden aus den Templates des External Service Interface erstellt.

Interface

Die Methoden des PIM ESI Elements werden in das PSM ESI Interface <*Elementname*>ESI.java übernommen.

Factory-Klasse

Die Factory des ESIs ist für die Erzeugung der Implementierung (Service<*Elementname*>ESI) des ESI Interfaces zuständig.

Mock-Klasse

Das Mock Objekt Mock<*Elementname*>ESI.java implementiert das External Service Interface <*Elementname*>ESI.java. Dieses Dummy Objekt wird für Unit Tests verwendet.

Service

Die Service Klasse Service<*Elementname*>ESI.java implementiert die Java Schnittstelle und stellt den Service zur Verfügung. In der Service Klase wird auf die vorhandene Backend Ressource zugegriffen.

7.6.8 ICI-Templates

Im Package *mappings.ici* ist der Namensraum für das Inter Component Interface definiert. Das ICI erlaubt den Zugriff auf die Realisierung der Komponente innerhalb der Business-Tier (i.d.R. Business Activity). Es wird mittels des Factory Pattern auf J2EE umgesetzt. Folgende Artefakte werden aus den Templates das Inter Component Interface erstellt.

Interface

Das Interface <*Elementname*>ICI.java ist die fachliche Schnittstelle des ICIs. Die definierten Methoden im PIM werden in die Java-Schnittstelle übernommen.

Factory-Klasse

Die Factory kapselt die Erzeugung der Implementierung des ICI´s-Interfaces.

Mock-Klasse

Das Mock Objekt implementiert das Business Delegate Interface. Dieses Dummy Objekt wird für Unit-Tests verwendet.

Bean-Klasse

Die EJB-Bean Klasse EJB<*Elementname*>BCI.java realisiert das Inter Component Interface und besorgt sich im Konstruktur die Referenzen auf die Local und Local Home Interfaces der Business Activity.

7.7 Mapping-Datei

Der openArchitectureWare Generator benötigt für die richtige Abbildung der Stereotypen im Anwendungsdesign, auf das erweiterte Java-Metamodell, ein Mapping. Welcher Stereotyp mit welcher Metamodellklasse instanziiert wird, ist in einer XML Datei hinterlegt.

Das erweiterte Metamodell ist in dem Package *pep_ca20_uml_meta* im Projekt hinterlegt. In diesem Package sind alle im Anwendungsdesign verwendeten Stereotypen als Erweiterung des mitgelieferten Java-Metamodell von oAW implementiert. Die in dieser Arbeit verwendeten Erweiterungen leiten ausnahmslos alle von *org.openarchitectureware.meta.uml.classifier.Class* ab.
In der folgenden Abbildung ist das Mapping des Profils auf die erweiterten Meta-Klassen abgebildet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MetaMap SYSTEM
"http://www.openarchitectureware.org/dtds/metamap.dtd">
<MetaMap>
	<Mapping>
		<Map>DO</Map>
		<To>pep_ca20_uml_meta.DO</To>
	</Mapping>
	<Mapping>
		<Map>BA</Map>
		<To>pep_ca20_uml_meta.BA</To>
	</Mapping>
	<Mapping>
		<Map>BCI</Map>
		<To>pep_ca20_uml_meta.BCI</To>
	</Mapping>
	<Mapping>
		<Map>BE</Map>
		<To>pep_ca20_uml_meta.BE</To>
	</Mapping>
	<Mapping>
		<Map>BF</Map>
		<To>pep_ca20_uml_meta.BF</To>
	</Mapping>
	<Mapping>
		<Map>DAO</Map>
		<To>pep_ca20_uml_meta.DAO</To>
	</Mapping>
	<Mapping>
		<Map>DOM</Map>
		<To>pep_ca20_uml_meta.DOM</To>
	</Mapping>
	<Mapping>
		<Map>EL</Map>
		<To>pep_ca20_uml_meta.EL</To>
	</Mapping>
	<Mapping>
		<Map>ESI</Map>
		<To>pep_ca20_uml_meta.ESI</To>
	</Mapping>
	<Mapping>
		<Map>ICI</Map>
		<To>pep_ca20_uml_meta.ICI</To>
	</Mapping>
</MetaMap>
```

Listing 7. 5: MetaMapping-Datei

7.8 Externe Konfiguration

Durch den vorherzusehenden Wechsel des Komponenten Konzeptes von derzeit EJB 2.1, auf das neue mit JEE5 aufkommende EJB3.0 Konzept, stellt sich bei der Erstellung der generativen Architektur die Frage, wie weit kann man über Property-Files eine externe Konfiguration in die Architektur einbringen.

Mögliche Artefakte die über Property-Files in den Generierungsprozess eingebunden werden können sind die Metadaten für XDoclet:

- EJB-tags
- Weblogic-tags

Die Metadaten die als spezielle JavaDoc Tags in den Code eingefügt werden, können aus Property-Files während des Generierungsprozesses inkludiert werden. In den Templates des openArchitectureWare Frameworks werden Platzhalter eingefügt die durch die Werte der Property-Files ersetzt werden. Es ist somit nicht mehr nötig direkt im Code oder in den Templates Anpassungen bei Änderungen der Initialen-Metadaten vorzunehmen.
Sollen allerdings für einzelne generierte Beans spezielle Werte eingefügt werden, muss der Entwickler dies direkt im Code vornehmen. Bei Besonderheiten dieser Art ist die Anpassung im Code jedoch kein großes Problem, da der Entwickler die Initialeinstellungen nur bei fundiertem Wissen dessen was er tut machen soll.

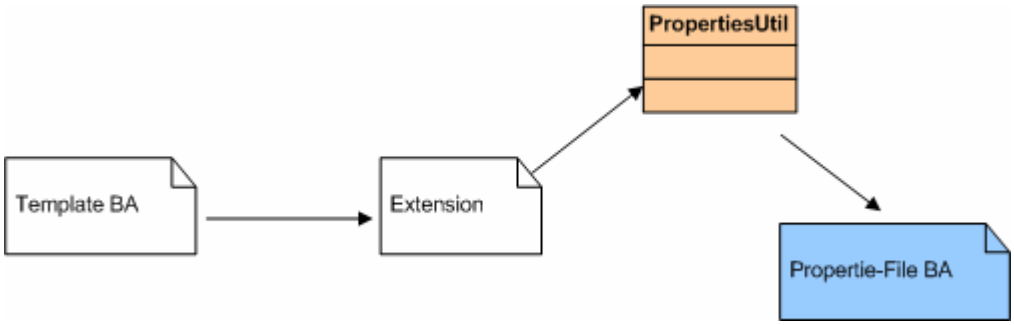


Abbildung 7. 18: Integration der Platzhalter

In den Templates die für die Generierung von Bean-Artefakten zuständig sind (vgl. Kapitel 7.6), werden die Werte der XDoclet-tags über den Extension-Mechanismus des openArchitectureWare Frameworks eingebunden. Aus dem Template wird die Methode der ejb_source Extension aufgerufen. Die Methode erwartet zwei Parameter damit zwischen dem Stereotyp und dem gewünschten Key unterschieden werden kann. Hierzu siehe auch Listing 7.6.

```
String getProperties (Class cls, String key) :
      JAVA pep_ca20_extend.PropertiesUtil.getProperties(org.
openarchitectureware.meta.uml.classifier.Class,java.lang.String);
```

Listing 7. 6: Extension ejb_source

7.8.1 Konzept der Property-Dateien

Die Properties werden mittels einer Util Klasse eingelesen. Durch Übergabe des Stereotype (Parameter) und Key werden die passenden Werte eingelesen und im Generierungsprozess eingefügt.
Für jedes Element in der Designsprache, das über die Transformation auf eine Bean abgebildet wird, existiert ein eigenes Property-File.

- ejb2.1_BA.properties
- ejb2.1_BE.properties
- ejb2.1_BF.properties
- ejb2.1_EL.properties

Des Weiteren existiert noch ein grundlegendes Property-File das *config.properties*. In diesem File können grundlegende Konfigurationsmöglichkeiten eingestellt werden. Zum Beispiel kann hier der Pfad der Bean Property Files angegeben oder geändert werden.

ejb.properties.path.BA=src/pep_ca20_properties/ejb2.1_BA.properties

7.8.2 Aufbau der Bean Property Files

In den Property-Files selbst existieren zwei Bereiche (Region). Im Bereich Class-Region werden die Metadaten, welche auf Klassenebene eingefügt werden, angegeben und ein Bereich BusinessMethode-Region, in dem die Werte für die Methoden-Ebene hinterlegt werden.

```
################################################
#Class-Region
################################################
#tag: @ejb.transaction
transaction.Type=Mandatory

#tag: @ejb.bean
bean.type=Stateless
bean.view-type=local

#tag: @weblogic.pool
pool.initial-beans-in-free-pool=10
pool.max-beans-in-free-pool=100

################################################
#BusinessMethode-Region
################################################
permission.unchecked=true
interface-method.view-type=local

#weblogic
weblogic.enable-call-by-reference=True
pool.initial-beans-in-free-pool=100
pool.max-beans-in-free-pool=1000
```

Listing 7. 7: Property File Business Facade

Generell sind jegliche Arten an Metadaten über die Property-Files inkludierbar. Lediglich der String, welcher im Template in die Extension übergeben wird, muss mit dem Key für den passenden Wert im Property-File übereinstimmen.

Mit der Einführung der Property Files ist der Generierungsprozess von außen konfigurierbar und somit kein gravierender Eingriff in die Templates oder im Code (evt. PSM) bei grundlegenden Änderungen nötig. Der explizite Generierungsschritt, wie nach OMG MDA, auf ein plattformspezifisches Modell, wird in der Arbeit bewusst übersprungen (vgl. Kapitel 3).

Als einen kleinen Ausblick zur Überführung des JEE5 EJB3.0 Konzept soll hier angemerkt werden, dass über die *config.properties*-Datei die Angabe des EJB-Konzeptes angegeben werden kann. Im Generierungsprozess kann dann zwischen den Templates, die konform zu EJB2.1 sind und den konformen EJB3.0 Templates, ausgewählt werden. Hierzu wäre nur ein marginaler Eingriff in das Root-Template nötig für die Einbindung von EJB3.0 konformen Templates.

7.9 Extensions

Mit dem Feature Extension des openArchtitectureWare Frameworks können hilfreiche externe Operationen erstellt und aus den Templates verwendet werden. Durch die Verwendung von Extensions stehen dem Entwickler nicht nur die Sprachmittel der Expression Language von oAW zur Verfügung, sondern durch in den Extensions definierten Erweiterungen in Java-Methoden der komplette Sprachumfang des Java Programmiermodells (vgl. Kapitel 6.3.2.4). Die Methoden welche aus der Extension aufgerufen werden, müssen *public* und *static* definiert sein.
Im vorhergehenden Kapitel 7.9 wurden Extensions für den Zugriff aus dem Template auf Property-Files und deren Werten verwendet. In diesem Kapitel werden noch weitere Verwendungen, die in der Realisierung mit eingeflossen sind, aufgeführt.

7.9.1 Namenskonventionen

Eine gängige Verwendung der Extension ist die Festlegung von Namenskonventionen in einem Extensions-File. Aus den Templates kann dann wie auf eine normale Methode auf die Methoden zur Erstellung der Namen zugegriffen werden. So ist eine durchgängige Implementierung der Benennung garantiert. Sollte sich im Laufe der Entwicklung die Namenskonventionen aus irgendwelchen Gründen ändern, muss lediglich an zentraler Stelle die Änderungen durchgeführt werden.

Namenskonventionen wurden in der Realisierung für folgende Elemente verwendet:

- Methoden Getter/Setter
- Parameter
- Instanzvariable

Methoden Getter/Setter

Bei der Generierung von *Getter*- und *Setter*-Methoden wird vor den Methodennamen, der aus dem Anwendungsdesign übernommen wird, entweder der String *get* oder *set* vor den Methodenname konkateniert. Außerdem wird der erste Buchstabe der Methode mit *toFirstUpper()* großgeschrieben.

Parameter und Instanzvariablen

Parameter und Instanzvariablen unterliegen einer gewissen Namenskonvention. Der erste Buchstabe eines Parameters wird grundsätzlich groß geschrieben. Das genaue Gegenteil erfolgt bei Instanzvariablen.

7.9.2 Package Berechnung und Typ Ermittlung

Für die Erstellung der Packages sind Extensions unumgänglich. Der benötigte Funktionsumfang bietet die Expression Language selbst nicht, deshalb muss aus den Extensions-Files in Java-Methoden referenziert werden.

Über den Extension Mechanismus wird die komplette Package-Struktur des PEP-PDM Projektes realisiert. Durch diverse Konventionen bei der Modellierung und Strukturierung des Anwendungsdesigns in Together kann die Package-Struktur nicht vollständig durch Rekursion aufgebaut werden. Die identische Strukturierung ist im PIM auch nicht gewünscht, da die Strukturierung unter Umständen schon plattformspezifische Details enthält.
Die teilweise nicht konsequente Einhaltung der Namenskonventionen bei der Modellierung des Anwendungsdesigns verursachten bei der Package-Strukturierung große Probleme. Selbst unbeabsichtigte Leerzeichen, die in den Namen der Elemente existierten, mussten ausgefiltert werden.
Die Expression Language von oAW ist für diese Ausnahmen zu überprüfen und zu behandeln nicht konzipiert. In der Realisierung wurden deshalb Java-Pattern (reguläre Ausdrücke) für die Validierung der Packagenamen verwendet.

Verwendete Extensions-Funktionen der Extension *NameConventions* für die Package und Typ-Ermittlung:

Extensions-Funktion	Beschreibung
getPackageName_Attributes(Class cls)	Zuständig für die Standardstruktur der Packages bis auf ihre Stereotype Unterscheidung
paraName(Parameter para)	Liefert die vollständige Packagestruktur des übergebenen Parameters zurück
getTypePackageName(Type value)	Liefert den vollqualifizierten Rückgabetyp einer Methode
packagePath(Class cls)	Ermittelt rekursiv durch Verwendug der packageName()-Methode die vollqualifizierte Packagestruktur und konvertiert diese in einen Pfad
packageName(Class cls)	Ermittelt rekursiv die vollqualifizierte Packagestruktur

Tabelle 7. 1: Extensions und Beschreibung

7.9.3 Zusatzfunktionen in den Metaklassen

In der Vorgängerversion des openArchitectureWare Frameworks wurden viele der Aufgaben, die in der aktuellen Version durch Extensions oder Check-Files übernommen werden, in den selbst implementierten Metaklassen definiert.

Diese Funktionen haben aber im Metamodell-Kontext im engeren Sinne nichts verloren, sondern sind ausschließlich Hilfsfunktionen für den Generierungsprozess und gehören somit nicht in den Kontext der Designsprache. Laut [Völ02] soll eine Verschmutzung des Metamodells ausdrücklich verhindert werden und somit die Übersicht des Modells und dessen Profils gewahrt bleiben.

7.10 Design Constraints

Wie oben schon erläutert, sollen im erweiterten Metamodell keine unnötigen Zusatzfunktionen, die unter Umständen auch nichts mit dem Kontext des Metamodells zu tun haben, realisiert werden. Constraints die zur Validierung des Anwendungsdesigns verwendet werden haben zwar einen direkten Bezug zum Metamodell, allerdings ist es architektonisch sinnvoll, diese Funktionalitäten in externen Files auszulagern. So ist zu jeder Zeit eine saubere Trennung und dadurch auch eine gewisse Übersicht vorhanden.

In der Arbeit wurde für die Erstellung von Constraints die neue Sprache Check von openArchitectureWare verwendet (vgl. Kapitel 6.3.2.5). Die Syntax der Check Files

unterscheidet sich nicht von der Syntax der Extensions. Beide Sprachen basieren auf dem Expression Framework von openArchitectureWare.
Es besteht die Möglichkeit über die Angabe (*WARNING* | *ERROR*) den Generierungsprozess bei einer Verletzung der Constraints abzubrechen oder nur eine Warnung in die Konsole zu schreiben.

Im nachfolgenden Listing 7.8 ist die Implementierung eines Constraints für die Business Facade abgebildet.

```
context BF WARNING "Association BF --> BCI is missing in the
model." :
      check_Association(this)? true : false;
```

Listing 7. 8: Constraint File

Es besteht auch in den Check-Files die Möglichkeit auf Extensions zu referenzieren. In dem dargestellten Beispiel wird diese Möglichkeit verwendet. Es ist im Allgemeinen angenehmer über Extensions auf Java-Methoden zu referenzieren und diese dann wie normale Methoden in den Templates oder Check-Files zu verwenden.

Im nachfolgenden Listing ist eine Methode aus der *Check_Util.java* Klasse aufgelistet. In der Methode *check_Association(BF)* wird überprüft, ob eine Assoziation zwischen dem Element Business Facade und dem Business Delegate (BCI) im Anwendungsdesign existiert.

```
/**
*  Checking for Business Delegate Association. The association is
*  used for interception (proxy).
*/
public static boolean check_Association(pep_ca20_uml_meta.BF
clazz){
      boolean value = false;

      for(Object col : clazz.AssociationEnd()){
            if(((AssociationEnd)col).Opposite().Class() instanceof
            pep_ca20_uml_meta.BCI){
                  value = true;
            }
      }
      return value;
}
```

Listing 7. 9: Klasse Check_Util mit Methode

Darüber hinaus wurden noch weitere Constraints für die Modellvalidierung eingefügt. Hier soll allerdings das obige Beispiel zur Veranschaulichung ausreichen.

Weitere Constraints:

- Das Inter Component Interface darf nur auf eine Business Activity referenzieren
- Business Facade referenziert auf eine oder mehrere Business Activity oder eine weitere Business Facade
- Das External Service Interface darf nur von einer Business Activity referenziert werden
- u.v.m.

7.11 Verbesserungen zu BMW-Transformation im Überblick

Zum Abschluss des Kapitels über die Realisierung sollen die Verbesserungen und Möglichkeiten des neuen Ansatzes aufgeführt werden.

7.11.1 Verbesserungen

Folgende Verbesserungen gegenüber des BMW-Transformators unter Verwendung von Together als Modellierungstool wurden erzielt:

- Modellvalidierung: Bei dem Ansatz BMW-Transformator mit Together existierten keinerlei Überprüfungen des Anwendungsdesign. Somit kann alles modelliert werden was mit der Unified Modelling Language und des Tools Together möglich ist. Durch die Modellvalidierung kann jetzt zumindest zu einem sehr frühen Zeitpunkt, jedoch nicht zur Modellierungszeit, auf Verletzungen von Constraints überprüft werden.

- Zentraler Konfigurationspunkt des Generators: Mit dem Konzept des Workflows in oAW ist eine zentrale Konfiguration des Generators, und allen anderen Prozessen aus denen sich der Generierungsprozess zusammensetzt, möglich. Derartige Konfigurationsmöglichkeiten gibt es zwar auch bei dem BMW-Ansatz, allerdings sind diese Möglichkeiten in den dutzenden dezentralen Property-Dateien versteckt.

- Flexibel erweiterbarer Generierungsprozess: Der Generierungsprozess ist durch die Workflow-Engine leicht erweiterbar. Diese Flexibilität wird unter anderem für die Inkludierung des Deskriptor-Generierungslauf mittels XDoclet benutzt. Der BMW-Transformator bietet hier keinerlei Möglichkeiten, um dessen Generierungsprozess zu erweitern.

- Verwendung weiterer Modellierungstools: Die Realisierung mit dem oAW-Framework erlaubt mit geringem Aufwand einen Wechsel des Modellierungstools. Es muss lediglich die Mapping-Datei und eine Adapterklasse für das neue Modellierungstool angepasst werden. openArchitectureWare unterstützt hier einige bekannte Tools, so dass im Grunde nur die Konfigurationsdatei im Workflow angepasst werden muss (vgl. 6.3.2.2).

- Externe Konfiguration der Doclet-Tags: Initiale Konfiguration verschiedener Doclet-Tags (vgl. Kapitel 7.8) sind in Property-Dateien ausgelagert und können komfortabel von extern konfiguriert werden.

- Logging Framework: In der Realisierung ist das Logging Framework Log4J eingesetzt. Somit sind aussagekräftige Debug- und Info-Messages für die Ausgabe auf der Konsole möglich. Bei der Fehlersuche ist das Logging Framework für den Entwickler eine sehr große Erleichterung.
  Der BMW-Ansatz beinhaltet keinerlei Logging für den Transformationsprozess, so dass sich die Fehlersuche um ein Vielfaches erschwert.

- Abnabelung von Together: Durch die flexible Schnittstelle des oAWs zum Einlesen von XMI-Files, kann das exportierte Anwendungsdesign aus Together instanziiert werden.

7.11.2 Möglichkeiten

An dieser Stelle soll noch ein Ausblick auf weitere Möglichkeiten, die mit dem openArchitectureWare Framework möglich sind, gegeben werden.

- Recipe Framework
  Das Recipe Framework bietet, ähnlich wie die Constraints, prüfende Funktionalitäten an. Es können Regeln definiert werden die nach der Generierung eingehalten werden müssen. Ein Beispiel wäre die Erstellung einer Implementierungsklasse, die von einer generierten Klasse ableitet (vgl. Kapitel 6.3).
  Der Entwickler wird durch das Framework darauf hingewiesen, dass eine Implementierungsklasse von einem bestimmten Typ existieren muss. Mit diesem Konzept ist es möglich, den Entwickler bei der Realisierung der fachspezifischen Logik an die Hand zu nehmen.

- UML2Ecore (M2M)

  Mit dem openArchitectureWare Modul UML2Ecore ist es möglich, bestehende UML-Modelle auf das Eclipse Modelling Framework abzubilden. Dies ermöglicht die Modellierung in einem konventionellen Modellierungstool anstatt in einem baumbasierten EMF-Editor vorzunehmen oder die Überführung bestehender Modelle in das Ecore-Format von Eclipse. Einen theoretischen Einblick wird im Kapitel Ausblick gegeben.

- Graphical Modelling Framework

  Nach der Modell zu Modell Transformation eines UML-Modells, ist es in Eclipse mit GMF möglich, für das zugrunde liegende Anwendungsdesign (Ecore-File) einen grafischen Editor zu erstellen. Ab diesem Zeitpunkt würde einer vollständigen Entwicklung, angefangen bei der Modellierung bis zur Implementierung fachspezifischer Teile, in Eclipse nichts mehr im Wege stehen.

  Zu diesen zwei letzten Punkten EMF und GMF wird im Ausblick näher eingegangen.

8 Ausblick

Im Ausblick dieser Diplomarbeit soll nicht, wie in vielen anderen zuvor geschriebenen Arbeiten, über die Chancen und Möglichkeiten des MDA-Ansatzes geschrieben werden. Durch jahrelangen Einsatz von MDA in erfolgreichen Projekten dürfte es klar sein, dass MDA überlegt eingesetzt einen gewissen Schritt zum Erfolg beisteuert. Natürlich ist der modellgetriebene Ansatz nicht in jeder Problemdomäne umsetzbar, jedoch existieren bei der herkömmlichen Softwareentwicklung viele Bereiche in denen durch MDA eine effizientere Entwicklung möglich ist.

In diesem Ausblick soll der nächste evolutionäre Schritt, der beim Einsatz einer generativen Architektur folgen wird, aufgeführt und mit aktuellen Technologien beleuchtet werden.

Der generelle Trend in der Softwareentwicklung auf Basis einer generativen Architektur geht zur vollständigen Integration in ein Entwicklungstool und von der Verwendung einer Toolchain weg. Angefangen bei der Erstellung der domänenspezifischen Designsprache, über Erstellung des Anwendungsdesign und der Transformation in einen Implementierungsrahmen, bis hin zur Implementierung des fachspezifischen Codes. Durch diese Integration des kompletten Prozesses löst man sich von bestehenden Problemen wie es zum Beispiel das XMI Framework (vgl. Kapitel 3.4.4) mit sich bringt.

In dieser Diplomarbeit werden explizit die Technologien Eclipse Modeling Framework und das Graphical Modeling Framework zur Integration des Softwareentwicklungsprozesses als Ausblick aufgeführt.

8.1 Eclipse Modeling Framework (EMF)

Das Eclipse Modeling Framework ist, wie der Name schon suggeriert, ein Modeling Framework mit welchem es möglich ist Code zu generieren. EMF selbst ist ein Open Source Java Framework unter der Leitung der Eclipse Open Source Communtity.

Das Eclipse Modeling Framework (EMF) stellt ein Metametamodell namens Ecore, zur Beschreibung von Metamodellen zur Verfügung. Das Metametamodell ist an das von der OMG spezifizierte MOF angelehnt jedoch nicht ganz so mächtig. Somit erlaubt EMF, mittels eines baumbasierten Editors, die Definition von Metamodellen die dann zur Verwendung bei der Erstellung des Anwendungsdesign benötigt werden.

Das mit Ecore modellierte Metamodell kann mit EMF in Implementierungsklassen überführt werden, die so, eine API zur Erstellung von Instanzen des Metamodells zur Verfügung stellen. Außer den Implementierungsklassen werden noch weitere Artefakte generiert, die für einen

Editor sowie zum Editieren des Modells Verwendung finden. Hierzu zählt zum Beispiel die Fähigkeit, andere Klassen über Änderungen im Modell zu informieren oder eine effiziente API, um Modelle während der Laufzeit zu verändern. Es existieren für EMF zusätzliche Erweiterungen, die z. B. die Validierung von Modellen erleichtert.

8.1.1 Ecore (Meta) Model

Das Modell, welches zur Darstellung von Modellen in EMF verwendet wird, nennt sich Ecore. Ecore selbst ist ein EMF-Modell und beschreibt sich selbst. In der Schichtenarchitektur würde sich das Ecore-Modell also auf der Layer 2 und Layer 3 wieder finden (vgl. Kapitel 3.4.1).
Aus der definierten domänenspezifischen Sprache in Ecore kann EMF einen baumbasierten Editor für die Erstellung des Anwendungsdesign, ein Editing Framework zum Editieren des Modells und natürlich das Modell als Code generieren. Für die Generierung dieser doch sehr hilfreichen Artefakte ist nichts Weiteres als ein Knopfdruck nötig und der mitgebrachte Generator JET[16] in EMF erstellt den nötigen Code. In der folgenden Auflistung werden nochmals alle Artefakte die durch Generierung erstellt werden können genant und deren Zweck aufgeführt:

- Implementierungsklassen des Metamodells: Jedes Element das im Metamodell definiert wurde hat im Code ein korrespondierendes Interface und eine Implementation
- Das .edit Projekt: In diesem Projekt sind einige Hilfsmittel zur Erstellung von Editoren
- Das .editor Projekt: Beinhaltet einen baumbasierten Editor der auf dem zugrunde liegende Metamodell basiert
- Das .test Projekt: Beinhaltet eine Vielzahl an Tests für das Metamodell
- Plugin.xml: Für die Verwendung des Projektes als Plugin. Das Plugin kann in das Plugin-Verzeichnis von Eclipse hinzugefügt werden

[16] JET – Java Emitter Templates

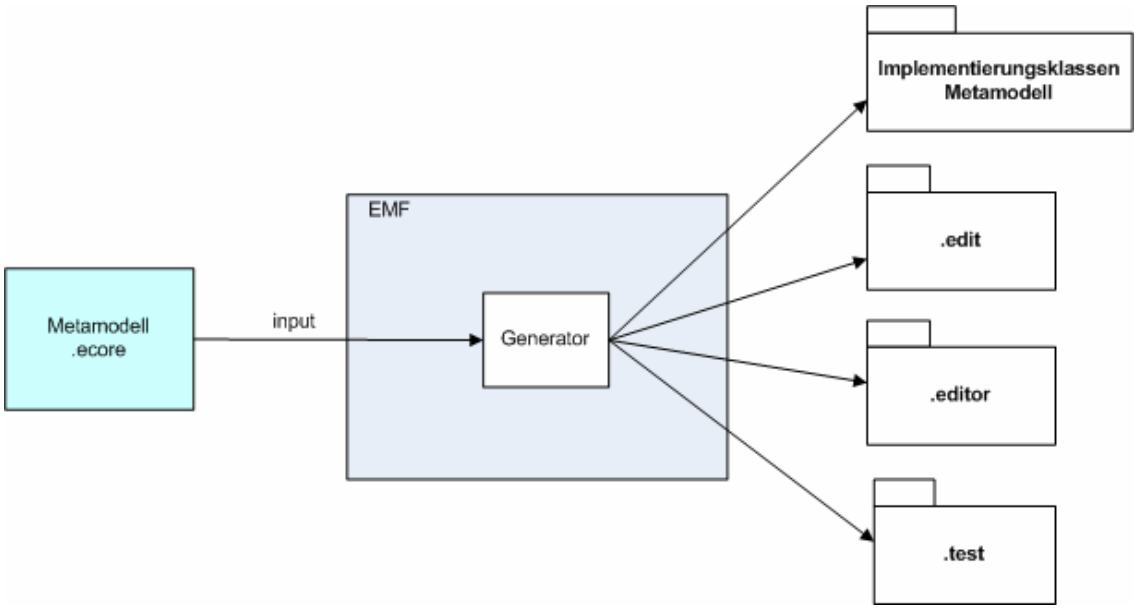


Abbildung 8. 1: EMF Generat

Nachdem die Packages und Projekte aus dem domänenspezifischen Ecore-Model generiert wurden kann der Anwender, entweder durch das Starten einer neue Workbench oder über hinzufügen des Plugins in den Plugin-Ordner, das Anwendungsdesign auf Basis des Metamodells entwerfen.
Es wird allerdings nicht näher auf diese Variante eingegangen, da der Fokus auf die grafische und nicht die baumbasierte Darstellung gelegt wird. Nähere Informationen zur Erstellung des Anwendungsdesign über die klassische Weise kann man in [EMF_DOC] finden.

8.1.2 UML2Ecore

Metamodelle mit dem von Eclipse mitgebrachten baumbasierten Editor zu erstellen ist nicht besonders anwenderfreundlich (vgl. Kapitel 6.3.1). Auch der GMF Ecore Editor, welcher unten näher erläutert wird, ist nicht immer praktikabel.
Mit der Cartridge UML2Ecore von openArchitectureWare ist es möglich weiterhin das Metamodell in einem herkömmlichen Modellierungstool zu modellieren. Das verwendete Tool muss allerdings das Modell über eine Export-Schnittstelle auf das Format EMF UML2 XMI zur Verfügung stellen können. Für die Evaluierung von EMF/GMF wurde in der Diplomarbeit Magic Draw Version 11.6 zur Erstellung des Metamodells verwendet. Über zu Hilfenahme von UML2Ecore wird dann das Modell auf eine Ecore-Instanz abgebildet.

In dem vordefinierten Workflow (Cartridge) des UML2Ecore-Plugin ist eine Abbildung auf das Ecore-Modell vorhanden. Für die Evaluierung der Technologie und deren Möglichkeiten reichen diese Default-Einstellungen. Es muss allerdings darauf hingewiesen werden, dass diese

Default-Transformation keine Besonderheiten in der Abbildung berücksichtigt. Die Abbildung wird mit dem Extension Mechanismus, der in Kapitel 6.3.2.4 erklärt wurde, realisiert. Hier wird nur ein Einblick gegeben, welche Punkte bei der Model2Model Transformation zu berücksichtigen sind. Nähere Informationen sind unter [M2M_Ecore] zu finden.

- Metamodell in Designtool entwerfen und in EMF UML2 Format exportieren
- Workflow einbinden und anpassen
- Definition der Abbildung welches UML2-Element auf welches Ecore-Element transformiert werden soll
- Definition der Constraints
- Test-Workflow ausführen zur Überprüfung der Modelltransformation

Die aufgeführten Aufgaben werden ausnahmslos mit Sprachen von openArchitectureWare definiert die auch schon in vorangegangenen Kapiteln verwendet wurden. Der Lernaufwand ist somit marginal und es muss nur das Konzept hinter den Mechanismen verstanden werden.

Im Anschluss der Transformationen stellt sich die Frage, ob es nicht möglich ist in Eclipse, basierend auf einem im Ecore-Format vorliegenden Metamodell, eigene Anwendungsdesigns grafisch zu modellieren? Es wird ein grafischer Editor benötigt, der auf Basis des erstellten domänenspezifischen Modells arbeitet. Mit solch einer Möglichkeit kann der Softwareentwicklungsprozess komplett in Eclipse integrierbar sein. Die Antwort hierauf ist positiv allerdings die Technologie noch in den Kinderschuhen.
Im folgenden Kapitel wird genau diese Technologie erläutert und aufgeführt, wie mittels GMF basierend auf einem domänenspezifischen Modell (Metamodell), ein grafischer Editor zum Modellieren erstellt werden kann.

8.2 Graphical Modeling Framework (GMF)

Das Graphical Modeling Framework [GMF] erweitert die Fähigkeiten von EMF um eine grafische Darstellungs- und Bearbeitungsmöglichkeit. GMF basiert auf den zwei Frameworks EMF, wurde oben erläutert, und dem Graphical Editing Framework (GEF). Das Graphical Editing Framework ermöglicht grafische Oberflächen zu erstellen und zu bearbeiten. Eine detailierte Ausführung von GEF würde den Rahmen der Diplomarbeit sprengen. Nähere Informationen über GEF sind unter [GMF] zu finden.
GMF selbst lässt sich in ein Runtime- und ein Entwicklungswerkzeug-Paket aufteilen. In dem Werkzeugpaket befinden sich Editoren mit denen die Eigenschaften des grafischen Editors angepasst werden und Generatoren, die für die Feature der Runtime zuständig sind. Die

Runtime von GMF bringt selbst einige nützliche Funktionalitäten mit ohne dass man eine Zeile Code schreiben muss. Hierzu zählen unter anderem Features wie Zoom, automatisches Anordnen der Elemente oder auch eine Export-Funktion.

8.2.1 Konzept

Zur Erstellung eines grafischen Editors mit GMF wird von dem Metamodell (Ecore-File), wurde im Abschnitt EMF erläutert, ausgegangen. Die .edit sowie .editor Artefakte müssen somit generiert sein und zur Verfügung stehen.
GMF selbst setzt sich aus mehreren Modellen, die im nächsten Punkt aufgeführt werden, zusammen. In den Modellen wird die grafische Darstellung der im Metamodell definierten Elemente im Editor bestimmt, sowie welche Werkzeuge in der Tool-Leiste des generierten Editors aufgelistet werden.
In der nachfolgenden Abbildung ist das Konzept der Modelle die GMF benötigt und der Ablauf in Richtung Editor Code aufgeführt.
Nach der Definition der Modelle und dem Mapping wird das Gen Model automatisch erstellt. Aus diesem Wrapper-Model (beinhaltet zusätzliche Metadaten) kann anschließend automatisch der Code für den grafischen Editor generiert werden. Der Code wird in einem <*Projektname*>.diagram Projekt abgelegt:

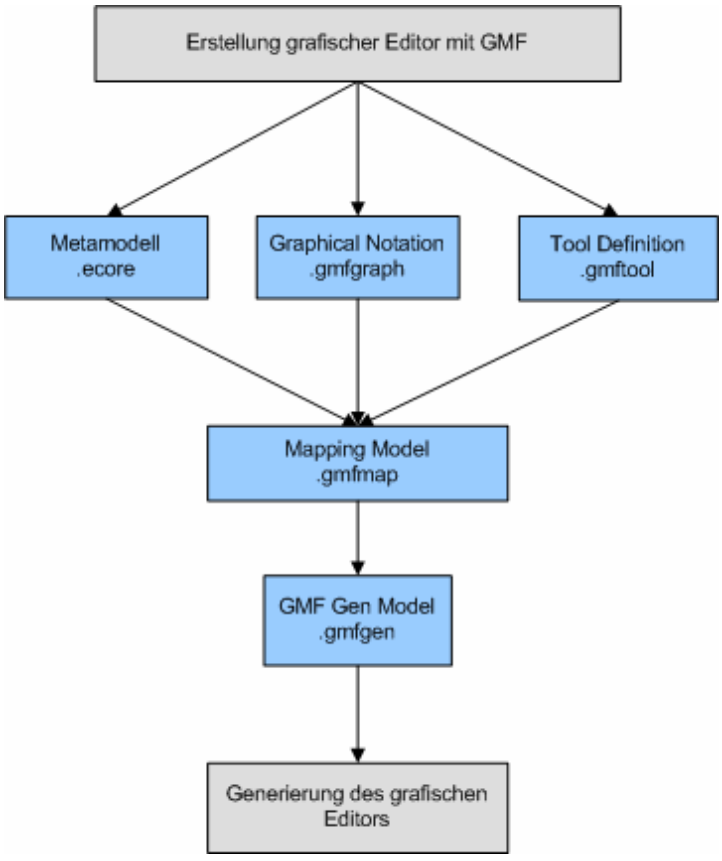


Abbildung 8. 2: Zusammenspiel der GMF Modelle

8.2.2 GMF-Modelle

Für die Erstellung eines grafischen Editors werden verschiedene Modelle benötigt. Es reicht nicht aus, dem Editor nur das Domänenmodell zu geben. Dem Editor würden sämtliche Informationen über die grafische Repräsentation fehlen. Deshalb werden bei der Erstellung des grafischen Editors folgende Informationen in Form von Modellen dem GMF Framework zur Verfügung gestellt:

EMF Domain Model

Das EMF Domain Model ist das domänenspezifische Modell (Metamodell). Dieses Modell ist die Basis auf der die Erstellung des grafischen Editors aufbaut. Es enthält die Informationen der Modelle und deren Abbildung untereinander.

Tool-Model

Das Tool-Modell beinhaltet die Information für die Tool-Leiste. In diesem Modell muss definiert werden, welche Modellelemente rechts im grafischen Editor in der Tool-Leiste zur Verfügung stehen sollen. Des Weiteren wird im Modell auch die Gruppierung der Elemente in der Leiste festgelegt unter der später die einzelnen Elemente in der Tool-Leiste wieder zu finden sind.

Graph-Model

Mit dem Graph Modell wird das Aussehen der Domain-Model-Elemente im Editor definiert. In diesem Modell werden die grafischen Figuren definiert, die das Aussehen des grafischen Editors später repräsentieren. Es werden jegliche geometrische Figuren unterstützt sowie die Möglichkeit der farblichen Repräsentation. Eine weitere Option sind eigene entworfene Bilder im GIF-Format einzubinden mit welchen das Element dann im Editor angezeigt wird. Diese Möglichkeit ist vor allem bei Diagrammen wie State Machines und dergleichen von Nutzen, da hier ein Bild sicherlich schöner anzuschauen ist als eine geometrische Figur.

Mapping Modell

Nachdem alle Modelle die GMF benötigt erstellt wurden, wird durch das Mapping alles zu einem Ganzen zusammen geführt. Die Elemente im Domänen Metamodell werden auf ihre grafische Abbildung, die im Graph-Model definiert sind, gemappt. Des Weiteren werden die Elemente die in der Tool-Leiste benötigt werden, zur Erstellung des Anwendungsdesigns im Editor, auch im Mapping-Model festgelegt.

Generate Model

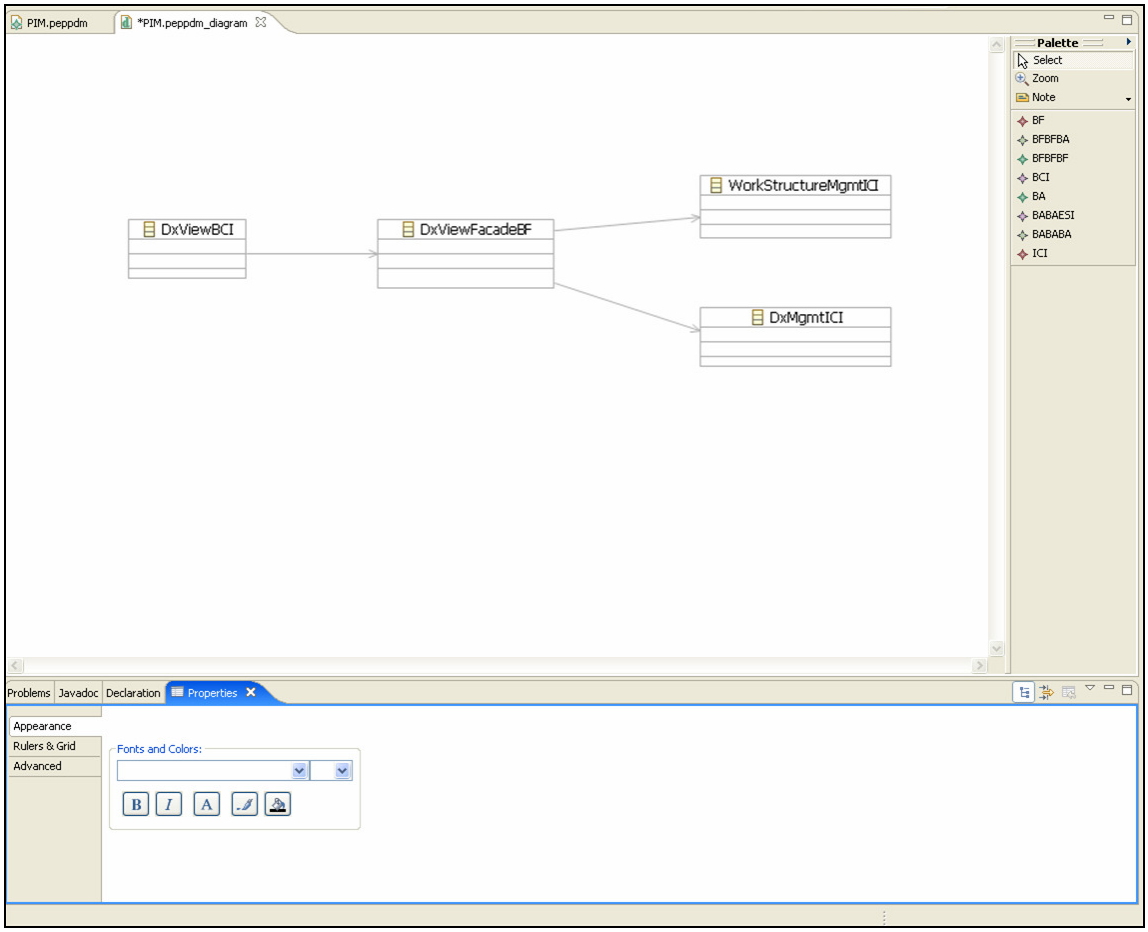
Das Gen-Model wird aus dem Mapping-Model erstellt. In diesem Modell können weitere Metadaten, die für die Generierung des Codes benötigt werden, angegeben werden. Die hier definierten Informationen sind ausschließlich für die Generierung notwendig und sind keine Daten, die in den anderen aufgeführten Modellen benötigt werden. Zum Beispiel kann hier die File Extension für das Diagramm angepasst oder ein Prefix für die Benennung der Packages angegeben werden.

8.2.3 Grafischer Editor

Der grafische Editor der generiert wird bringt schon einige, aus herkömmlichen grafischen Editoren bekannte, nützliche Features mit sich. Oben wurden schon die Feature des Zoomens oder der Export aufgeführt, allerdings sind noch weitaus mehr Features hier aufzulisten wie zum Beispiel:

- Overview Pane
- Properties View
- Formatierung
- u.v.m.

Der Editor selbst ist im gewohnten Look&Feel von Eclipse aufgebaut. Die Anordnung der Views ist flexibel wie aus Eclipse bekannt. Beliebige Erweiterungen können in dem Editor-Code eingebaut werden, jedoch ist hierzu das Verständnis von GEF notwendig.
Eine Veranschaulichung des generierten Editors ist im folgenden Screenshot zu bekommen.



Abbidlung 8. 3: GMF Editor

8.3 Fazit GMF

Abschließend bleibt zu sagen, dass mit GMF eine weitere sehr interessante Technologie, die durchaus ein großes Potential mit sich bringt, zu den Frameworks von Eclipse hinzugekommen ist. GMF kann viel Arbeit bei der Erstellung eines grafischen Editors ersparen und nach einiger Zeit Erfahrung ist auch ein zügiges Vorankommen möglich. Mit GMF wird die Lücke zwischen EMF und GEF geschlossen und erleichtert somit die Entwicklung eines Editors.

Sollen Funktionen eingebaut werden die nicht standardmäßig mit generiert werden ist ein fundiertes Wissen in GEF unabdingbar. Genau hier ist auch der Knackpunkt für die Verwendung von GMF. Sich in die Frameworks einzuarbeiten ist ein hartes Stück Arbeit da die Lernkurve sehr flach ist. Ist dieser Schritt durchlaufen kann aber sehr schnell ein mächtiger grafischer Editor implementiert werden.

Weiterhin kann durch den Einsatz des openArchitectureWare Frameworks, unter Verwendung eines GMF-Editors, Modellvalidierung sogar schon zur Designzeit realisiert werden. Dieser Punkt ist wohl einer der wichtigsten im Bezug auf eine vollständige Integration des

Entwicklungsprozesses. Wie oben schon erläutert ist es in modellgetriebenen Architekturen wichtig die Validierung zum frühstmöglichen Zeitpunkt zur Verfügung zu stellen. Durch die Validierung zur Designzeit gehören zeitraubende Iterationen, aufgrund fehlerhafter Modelle, der Vergangenheit an. Generell ist die Integration von Modulen, die bei dem openArchitectureWare Framework mit ausgeliefert werden bei einem vollständig integrierten Entwicklungsprozess in Eclipse sehr effizient.

Während der Erstellung der Diplomarbeit wurde bei BMW die Nachfolger-Architektur CA3.0 entwickelt. Die CA3.0 basiert auf JEE 5 Technologien und verwendet EMF als Modellierungssprache sowie das openArchitectureWare Framework als Generator Framework. Das Anwendungsdesign (PIM) wird derzeit im baumbasierten Editor von EMF entwickelt und ist daher recht anwenderunfreundlich. Genau hier würde der Einsatz von GMF eine deutliche Verbesserung bringen. Die Anwender könnten das PIM in einem grafischen Editor erstellen und somit das lästige Arbeiten im EMF Editor umgehen. Würde das UML2Ecore Modul von openArchitectureWare hier noch zum Einsatz kommen, wäre es möglich, das Anwendungsdesign in einem gewöhnlichen und komfortablen Designtool, wie zum Beispiel Magic Draw, zu entwickeln.

Solch eine vollständige Integration wurde zum Teil evaluiert jedoch nicht in die Realisierung dieser Arbeit mit aufgenommen. Diese Themen sind prädestiniert für eine Diplomarbeit, die auf dieser aufsetzen könne und eine vollständige Integration mit den aufgeführten Technologien realisiert.

9 Bewertung

Rückblickend kann man sagen, dass der Einsatz eines modellgetriebenen Ansatzes im Umfeld der Technologie J2EE und der Generierung von Business Tier Artefakten einige Vorteile mit sich bringt. Es ist möglich aus einem Anwendungsdesign auf Basis einer domänenspezifischen Sprache einen Architekturrahmen zu generieren. Der Mehrwert der Generierung wird sehr stark deutlich wenn man sich die Anzahl der Artefakte die in der EJB-Technologie benötigt werden anschaut. An dieser Stelle wird dem Entwickler viel Copy and Paste-Arbeit abgenommen und somit auch Fehler, die bei der Implementierung entstehen können, reduziert.
Allerdings ist die Komplexität die ein modellgetriebener Ansatz mit sich bringt nicht zu unterschätzen. Konzepte wie Metamodell, PIM, PSM, UML und diverse andere müssen erst verstanden werden bevor effektiv damit gearbeitet werden kann.

Die Technologieersetzung des BMW-Transformators mit dem openArchitectureWare Framework zeigte durchgehende Verbesserungen und erweiterte Möglichkeiten auf. Dies resultierte unter anderem in der zusätzlichen Funktion der Modellvalidierung zur Generierungszeit. Weiterhin wurden die Freiheiten des Entwicklers mit dem oAW erweitert, so dass über diverse Module die das Framework mitbringt weitere Funktionalität eingebaut werden kann. Die Transparenz des Generierungsprozesses ist im Gegensatz zu der Generierung mit dem BMW-Generator ein weiteres Merkmal welches gerade für neue Teammitglieder zur schnellen Einarbeitung wichtig ist. Allerdings konnte die erhoffte Ersetzung des BMW-Generators im Projekt PEP-PDM nicht realisiert werden. Das Projekt ist zu weit fortgeschrittenen. Die Protected Regions des BMW-Generators würde der openArchitectureWare Generator nicht berücksichtigen und jeglicher erstellter fachspezifischer Code würde bei einer Iteration des Generierungsprozesses verloren gehen. Die erstellte generative Architektur kann aber bei einem neuen Projekt, basierend auf der Component Architecture, sofort eingesetzt werden.
Geht man von dem Stand der Möglichkeit aus, den Entwicklungsprozess komplett in Eclipse zu integrieren, würden sich Probleme die durch das XMI-Framework entstehen auflösen. Außerdem würde nur noch eine Entwicklungsumgebung benötigt und nicht wie zurzeit viele verschiedene die nur selten ohne Probleme zusammenarbeiten.
Hier hat openArchitectureWare seit der Version 4 einen wichtigen Schritt in diese Richtung getan. Es wird zwar nach wie vor der klassische Weg auf Basis von UML-Modellen unterstützt, jedoch das Augenmerk hauptsächlich auf das EMF und GMF Framework gelegt.
Durch die ständige Weiterentwicklung des openArchitectureWare Frameworks, sowie das noch sehr junge GMF Projekt, bleibt mit Spannung abzuwarten was sich hier in diesen Bereichen

noch ergeben wird. Die ersten generativen Architekturen, die einen grafischen Editor unter GMF verwenden, sind auf jeden Fall sehr viel versprechend.

A Glossar

Anwendungsfamilien

Die Anwendungsfamilie ist nicht nur die Menge der Anwendungen mit gleicher Plattformbindung, sondern auch operationaler Bestandteil des Entwicklungsprozesses. Sie beinhalten insbesondere alle für den Generator-Einsatz notwendige Bausteine, um architekturzentrierte Modelle in Implementierungsrahmen überführen zu können.

Designsprache

Die Designsprache einer Anwendungsfamilie ermöglicht die abstrakte Modellierung schematischer Architekturaspekte. In den meisten Fällen wird die Modeling Unified Language (UML) mit einer semantischen Anreicherung der Stereotypen verwendet.

EJB (Enterprise Java Beans)

Enterprise Java Beans sind standardisierte Komponenten im Kontext von J2EE. Sie ermöglichen die Entwicklung komplexer verteilter Softwaresysteme.

EMF (Eclipse Modeling Framework)

Das Eclipse Modeling Framework ist ein Framework mit welchem es aus strukturierten Modellen möglich ist Code zu generieren. EMF verwendet zur Erstellung des Metamodells das Metametamodell Ecore. Ecore kann mit der Unified Modeling Language verglichen werden, allerdings mit einem wesentlich geringeren Umfang.

Fachliches Coding

Das fachliche Coding erfolgt ausschließlich in den geschützten Bereichen (protectedRegion), die in den Templates entsprechend definiert wurden, oder wie in der Realisierung dieser Diplomarbeit über einen architektonischen Ansatzes mittels Spezialisierung realisiert wurde.

Generator Backend

Das Generator Backend interpretiert die Templates, bindet diese an das Java-Metamodell und nimmt die Sourcecode-Expansion vor. Unter Einsatz von geschützten Bereichen wird bereits bestehender fachlicher Code übernommen.

GMF (Graphical Modeling Framework)

Das Graphical Modeling Framework der Eclipse Community ist ein Framework, welches das Erstellen von grafischen Editoren in Eclipse erleichtert. GMF basiert auf dem Eclipse Modeling Framework und des Graphical Editing Framework.

GEF (Graphical Editing Framework)

Das Graphical Editing Framework ist ein Framework zur grafischen Darstellung von Modellen. Zumeist werden als Modell ein EMF Modell verwendet ist aber nicht zwingend notwendig. GEF erlaubt dem Entwickler mit dem Modell zu interagieren und bietet nützliche Workbench Funktionen.

IDE (Integrated Development Environment)

IDEs sind Entwicklungsprogramme die zur Erstellung von Software verwendet werden.

Implementierungsrahmen

Der Implementierungsrahmen ist architekturspezifischer Code, der um die Implementierung der Fachlogik ergänzt werden muss.

Instanziator

Der Instanziator parst den XMI-Export des Anwendungsdesigns und instanziiert auf Basis der Instanziierungsvorschriften das Java-Metamodell.

Instanziiertes Java-Metamodell

Das instanziierte Java-Metamodell dient zur Abbildung des Anwendungsdesigns auf ihre korrespondierende Metaklassen. Über das resultierende instanziierte Anwendungsdesign können die Templates vom Generator Backend angebunden werden.

J2EE (Java 2 Enterprise Edition)

J2EE ist die Spezifikation einer Softwarearchitektur für multi-tier Applikationen. J2EE vereinfacht das Entwerfen von Applikationen die portabel, skalierbar und leicht integrierbar sind.

MOF (Meta Object Facility)

Die Meta Object Facility beschreibt eine abstrakte Sprache zur Erstellung von plattformunabhängigen Metamodellen. Eine Instanz der MOF wäre zum Beispiel die Unified Modeling Language (UML) der OMG.

XMI (XML Metadata Interchange)

XMI dient als Austauschformat der in den UML-Werkzeugen erstellten Modelle.

B Bibliographie

[A020] IT-Konzept AppServer / Presentation / EAI
Stand: 16.10.2006

[AB_CA20] Architekturbaustein der CA 2.0
Stand: 08.12.2006

[ANDA] AndroMDA
http://www.andromda.org/
Stand: 20.11.2006

[Ant] Apache Ant Homepage
http://ant.apache.org/index.html
Stand: 08.12.2006

[Borland] Borland Homepage
http://www.borland.com/de/products/together/index.html
Stand: 08.12.2006

[b_mTL] Template Spezifikation
openArchitectureWare Eclipse Page
http://www.eclipse.org/gmt/oaw/
Stand: 08.12.2006

[DAH01] Diplomarbeit Joachim Hengge
Thema:
Erweiterung eines metamodell-basierten Generator-Frameworks zur Erzeugung von Quellcode für eine mehrschichtige Client/Server Architektur am Beispiel von J2EE

[Doc_Meta] Dokumentation Metamodell CA2.0
Stand: 08.12.2006

[Eclipse] Eclipse Homepage
http://www.eclipse.org/
Stand: 08.12.2006

[EMF_DOC] Eclipse Modeling Framework Dokumentation
http://www.eclipse.org/emf/docs/
Stand: 08.12.2006

[GEF] Graphical Editing Framework
http://www.eclipse.org/gef
Stand: 08.12.2006

[GMF] Graphical Modeling Framework
http://www.eclipse.org/gmf/
Stand: 08.12.2006

[Hybridlabs] Java Beautifier
http://www.hybrid-labs.de/
Stand: 08.12.2006

[itemis] Itemis Homepage / Deutsches Forum openArchitectureWare
http://oaw.itemis.de/
Stand: 08.12.2006

[IOC_P] Inversion of Control Design Pattern
http://www.martinfowler.com/articles/injection.html
Stand: 08.12.2006

[Jakarta] Jakarta Homepage
http://jakarta.apache.org/
Stand: 08.12.2006

[LB_CA20] Lösungsbaustein CA 2.0
Stand: 08.12.2006

[Log4J] Log4J Homepage
http://logging.apache.org/log4j/docs/
Stand: 08.12.2006

[MDA00] Model Driven Architecture Guide 1.0.1 Object Management Group
http://www.omg.org/docs/omg/03-06-01.pdf
Stand: 25.08.2004

[M2M_Ecore] openArchitectureWare Beschreibung für Modell zu Modell Transformation
http://www.eclipse.org/gmt/oaw/doc/4.1/52_m2mWithUML2Example.pdf
Stand: 08.12.2006

[OMG_01] XMI Spezifikation
Object Management Group
http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI
Stand: 08.12.2006

[UML_01] Unified Modeling Language
Object Management Group
http://www.omg.org/technology/documents/formal/uml.htm
Stand: 08.12.2006

[Völ01] Markus Völter
Sprachen / Modellen / Fabriken
http://www.voelter.de/data/articles/lmf.pdf
Stand: November 2006

[Völ02] Markus Völter
Model Driven Software Development
Stand: 28.11.2006

[XDoclet] XDoclet Homepage
http://xdoclet.sourceforge.net/xdoclet/index.html
Stand: 08.12.2006

CD-ROM Diplomarbeit

Inhalte der CD-ROM der vorliegenden Diplomarbeit:

- Diplomarbeit (PDF)
- Artikel (PDF)
- Eclipse mit Projekt und oAW
    - Generator mit Templates, Constraints und Extensions
    - Serialisiertes PEP-PDM PIM