

FACHHOCHSCHULE WEDEL

– University of Applied Sciences –

SEMINARARBEIT

Thema:

Modellgetriebene Softwareentwicklung mit openArchitectureWare

Eingereicht von: Björn Peemöller
Borgfelde 4a
22869 Schenefeld
E-Mail: inf6254@fh-wedel.de

Sebastian Reese
Moorende 43a
21635 Jork
E-Mail: ms6233@fh-wedel.de

Abgegeben am: 30. Dezember 2008

Dozenten: Prof. Dr. Ulrich Hoffmann
Prof. Dr. Uwe Schmidt

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
Listingverzeichnis	IV
Abkürzungsverzeichnis	V
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau der Arbeit	1
2 Modellgetriebene Softwareentwicklung	2
2.1 Definition und Einordnung	2
2.1.1 Definition	2
2.1.2 Motivation	3
2.1.3 Konzepte	5
2.2 Modellierung	5
2.2.1 Begriffsbildung	5
2.2.2 Leitfaden zur Modellierung	8
2.3 Transformationen	10
2.3.1 Modell-zu-Modell-Transformation	10
2.3.2 Modell-zu-Code-Transformation	11
2.3.3 Cartridges	14
3 openArchitectureWare	15
3.1 Grundlagen	15
3.2 Workflow	20
3.2.1 Cartridge	21
3.3 Xpand	22
3.3.1 Metaprogrammierung	22
3.3.2 Gründe für Xpand	23
3.3.3 Sprachelemente	25
3.4 Einheitliches Typ- und Expression-System	28
3.4.1 Typsystem	28
3.4.2 Expression-System	29
3.5 Xtend	30
3.5.1 Aufruf	32
3.5.2 M2M-Transformation	34
3.6 Check	35
3.7 Xtext	36

4	Beispiel – Stammdatenpflege mit openArchitectureWare	42
4.1	Zweck und Ziel des Beispiels	42
4.2	Modellgesteuerte Stammdatenpflege	43
4.3	Umsetzung mit openArchitectureWare	47
4.3.1	Entwicklung des Metamodells	47
4.3.2	Entwicklung des Generators	50
4.3.3	Modellierung der Seminarverwaltung	54
4.4	Mögliche Weiterentwicklungen	56
5	Fazit	58
5.1	Modellgetriebene Softwareentwicklung	58
5.2	openArchitectureWare	58
6	Literaturverzeichnis	60
7	Weitere Internetquellen	62

Abbildungsverzeichnis

1	Begriffsbildung Modellierung (in Anlehnung an [SVEH07, S. 28])	6
2	Modellebenen in der modellgetriebenen Softwareentwicklung	8
3	Dreistufige Vererbung	13
4	openArchitectureWare-Framework Überblick (vgl. [Gen08b, Folie 4]) . . .	15
5	Erstes Beispiel: Klassendiagramm	16
6	Built-in Types	29
7	Higher order functions	30
8	Validierung generierter EMF-Editor	37
9	Überblick Xtext-Framework	38
10	Grammatikregel	39
11	Beispiel Editor	41
12	Beispieldatenmodell Seminarverwaltung	42
13	3-Schichten-Architektur der Stammdatenpflege	44
14	Modelle in der Stammdatenpflege	45
15	Generierter Editor für die Sprache SDDSL	50
16	Workflow-Struktur	51
17	Dialog zur Personenpflege	55

Listungsverzeichnis

1	Protected Region in einer Java-Methode	13
2	Erstes Beispiel: Workflow	17
3	Erstes Beispiel: Template	18
4	Erstes Beispiel: Ausgabe	18
5	Erstes Beispiel: Java Datei	19
6	Beispiel Workflow	21
7	Beispiel einer Xpand-Datei	25
8	DEFINE	26
9	XPAND	26
10	FILE	26
11	PROTECTED	27
12	Generierte Datei mit geschütztem Bereich	27
13	AROUND	28
14	Beispiel-Extension	30
15	Beispiel Java-Extension	31
16	Cached Extension	32
17	Beispiel Aufruf der Extension	33
18	Beispiel Ausgabedatei <i>Model.sddsl.Model</i>	33
19	Aufruf über WorkflowComponent	33

20	M2M-Transformation mit Xtend	34
21	Check-Constraint Entity	35
22	Beispiel Entität	38
23	Notation einer Entität	39
24	Usecase für die Bearbeitung von Personen	46
25	Dialogkonfiguration für die Bearbeitung von Personen	46
26	Xtext-Modell für das Datenmetamodell	47
27	Check-Constraints für das Datenmetamodell	49
28	Modell-Modifikation: Hinzufügen eines ID-Attributs	52
29	Template zur Erzeugung der Hibernate-Mappings (Ausschnitt)	53
30	Datenmodell der Seminarverwaltung als SDDSL-Modell	54
31	Workflow der Seminarverwaltung	54

Abkürzungsverzeichnis

CASE	Computer Aided Software Engineering
CUF	Client Utilities & Framework
DSL	Domänenspezifische Sprache (Domain Specific Language)
GUI	Graphical User Interface
MDS	Modellgetriebene Softwareentwicklung (Model Driven Software Development)
SDDSL	Domänenspezifische Sprache der Stammdatenpflege
UML	Unified Modeling Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

1 Einleitung

1.1 Motivation

Bei der Entwicklung von Software wird heutzutage eine Vielzahl an Frameworks eingesetzt, um Standardaufgaben zu lösen. Gleichzeitig steigt die Komplexität der Software, sei es durch vermehrten Konfigurationsaufwand oder die Vielzahl an abzubildenden Aspekten. Somit ist es notwendig und sinnvoll, Software auf einem höheren Abstraktionsniveau als dem Quellcode zu entwickeln. Eine solche Möglichkeit bietet die modellgetriebene Softwareentwicklung, bei der ausgewählte Softwareaspekte durch Modelle abstrakt beschrieben werden. Das Framework openArchitectureWare unterstützt dieses Vorgehen durch entsprechende Werkzeuge.

Diese Ausarbeitung zu dem Vortrag „Modellgetriebene Softwareentwicklung mit openArchitectureWare“ stellt das Framework openArchitectureWare vor, das gewissermaßen einen Werkzeugkasten für die modellgetriebene Softwareentwicklung bildet. Der Vortrag ist eingegliedert in das Informatik-Seminar „Linux und Netzwerke, Softwareentwicklung mit Eclipse“ [1] von Prof. Dr. Ulrich Hoffmann und Prof. Dr. Uwe Schmidt an der FH Wedel im Wintersemester 2008/2009.

1.2 Aufbau der Arbeit

In Abschnitt 2 wird zunächst die modellgetriebene Softwareentwicklung als Konzept der Softwareentwicklung vorgestellt, die die Basis für weitere Ausführungen bildet. Die Ausführungen umfassen sowohl die theoretischen Grundlagen als auch Hinweise für die Anwendung. Anschließend wird im Abschnitt 3 das Framework openArchitectureWare detaillierter vorgestellt. Dazu werden die einzelnen Bestandteile des Frameworks mit Hilfe von Beispielen sukzessive eingeführt. Im Anschluss hieran wird im Abschnitt 4 openArchitectureWare für ein größeres Anwendungsbeispiel eingesetzt. Exemplarisch soll eine Stammdatenpflege modellgetrieben entwickelt werden, um eine erste Evaluierung des Frameworks im realen Einsatz zu ermöglichen.

Abschließend werden im Abschnitt 5 die modellgetriebene Softwareentwicklung im Allgemeinen und openArchitectureWare im Besonderen zusammenfassend betrachtet und bewertet.

2 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung ist eine Technik der Softwareentwicklung, bei der Teile des zu entwickelnden Softwaresystems mit Hilfe von Modellen auf einem höheren Abstraktionsniveau beschrieben werden (vgl. [SVEH07, S. 4 u. S. 11]). Unter einem Modell soll hierbei eine Menge von Aussagen über ein betrachtetes System verstanden werden (vgl. [Met05, S. 19]). Bei Modellen wird in der Regel von Systemanteilen abstrahiert, die für die jeweilige Betrachtung nicht relevant sind, um die Komplexität zu verringern und eine Fokussierung auf die relevanten Aspekten zu erlauben.

Die Verwendung von Modellen zur Softwareentwicklung an sich ist nicht neu und hat seit der Definition der Unified Modeling Language (UML) noch zugenommen. Diese Modelle dienen jedoch oftmals lediglich zum Design bzw. zur Dokumentation der Software, was nach Stahl et al. [SVEH07] als *modellbasierte* Softwareentwicklung bezeichnet werden soll (vgl. [SVEH07, S. 3]). Hierbei besteht zwischen dem Modell und der Implementierung lediglich eine gedankliche Verbindung. Dies kann zum einen dazu führen, dass im Entwicklungsprozess gewonnene Erkenntnisse sowie daraus resultierende Änderungen im Modell nachgetragen werden müssen. Dieser Prozess ist jedoch aufwändig und wird daher in der Praxis oftmals vernachlässigt, sodass Inkonsistenzen zwischen dem Modell und der Implementierung entstehen. Zum anderen liegt die Aufgabe der Transformation des Modells in die Implementierung vollständig beim Entwickler, sodass das Design zum Teil „nachimplementiert“ wird. Ein solcher Prozess ist nicht nur fehleranfällig, sondern birgt darüber hinaus die Gefahr unterschiedlicher Interpretationen seitens der Entwickler und damit auch die Gefahr unterschiedlicher (und damit inkonsistenter) Umsetzungen.

Im Unterschied dazu stehen bei der *modellgetriebenen* Softwareentwicklung die Modelle gleichberechtigt neben dem Quellcode und fließen direkt in die Implementierung der Software mit ein. Das Adjektiv „getrieben“ soll dabei die zentrale Rolle der Modelle im Entwicklungsprozess hervorheben. Durch die direkte Integration der Modelle in die Software ergibt sich zudem die Möglichkeit der Automatisierung dieses Prozesses.

2.1 Definition und Einordnung

2.1.1 Definition

Die modellgetriebene Softwareentwicklung wird in der Literatur nicht einheitlich definiert (vgl. [ZW06, S. 22]; [Met05, S. 2]; [SVEH07, S. 11]). Die in der oben genannten

Literatur aufgeführten Definitionen der MDSO haben jedoch einen gemeinsamen Kerngedanken: Die Modelle stellen einen wesentlichen Aspekt der MDSO dar und fließen in die Software als Endprodukt ein. Für die weiteren Ausführungen soll die umfassendere Definition von Stahl et al. [SVEH07] verwendet werden, die auch die Art der Integration der Modelle in die Software umfasst:

Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSO) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen. [SVEH07, S. 11]

Diese Definition beinhaltet die folgenden drei Aspekte (vgl. [SVEH07, S. 11ff.]):

Formale Modelle: Unter einem formalen Modell ist eine vollständige Beschreibung eines abgegrenzten Teils der Software zu verstehen, wobei genau definiert ist, welche Anteile der Software beschrieben werden. Die Modelle können in einer beliebigen Notation (z. B. textuell oder grafisch) vorliegen.

Erzeugung lauffähiger Software: Die Modelle fließen direkt in die Software als Endprodukt der Softwareentwicklung ein. Ein formales Modell, das lediglich zur Dokumentation der Software dient, stellt somit kein Modell im Sinne der MDSO dar.

Automatisierung der Erzeugung: Der Prozess vom Modell zur ausführbaren Software soll automatisch und ohne manuelle Eingriffe erfolgen. Allgemeiner betrachtet stellen die Modelle einen Teil der Eingabe des Build-Prozesses¹ dar. Die Erzeugung von manuell auszufüllenden Quelltextstrukturen, die beispielsweise von CASE-Tools erzeugt werden können, ist somit keine automatische Erzeugung im Sinne der MDSO.

Die Einordnung, ob ein Vorgehen modellgetrieben ist oder nicht, kann dabei nicht immer ganz trennscharf getroffen werden. Hervorzuheben ist, dass MDSO nicht auf einzelne Sprachen oder Werkzeuge eingeschränkt ist, sondern einen allgemeinen Ansatz darstellt.

2.1.2 Motivation

Die übergeordneten Ziele der MDSO sind die Verbesserung der Softwarequalität und der Wiederverwendbarkeit sowie die Erhöhung der Entwicklungseffizienz. Diese Ziele sollen durch den Einsatz der MDSO wie folgt erreicht werden (vgl. [SVEH07, S. 13ff.]):

¹Unter dem Build-Prozess soll der Prozess der Erstellung der ausführbaren Software ausgehend von Quellcode sowie weiteren Quellen verstanden werden. Dieser Prozess umfasst beispielsweise eine Kompilierung sowie das Linken von Objektcode.

Einheitliche Architektur Durch die automatisierte Erzeugung der modellierten Softwareteile wird gewährleistet, dass diese Teile einheitlich und nach einem festgelegten Schema erzeugt werden und so zwingend dieselbe Architektur und Implementierung besitzen. Dieses Vorgehen erhöht die Konsistenz des Systems und führt zu einer klareren Struktur. Dies bedeutet dabei nicht, dass die Architektur bereits zu einem frühen Zeitpunkt endgültig festgelegt sein muss. Vielmehr werden nachträgliche Architekturänderungen erleichtert und können konsistent durchgeführt werden. Die Änderungen müssen lediglich an einer zentralen Stelle (der Festlegung, wie die Software ausgehend von den Modellen zu erzeugen ist) vorgenommen werden.

Entwicklungseffizienz Auf den ersten Blick ließe sich eine Effizienzsteigerung darauf zurückführen, dass bei der MDS, im Vergleich zu einem herkömmlichen Vorgehen, der Anteil an zu schreibendem Quellcode durch die automatische Erzeugung von Codeteilen wesentlich geringer ist. Gerade in größeren Projekten entfällt auf das eigentliche Schreiben des Quelltextes jedoch nur ein sehr geringer Teil der Zeit, sodass hier lediglich ein kleiner Vorteil zu erwarten ist.

Ein wesentlich größerer Vorteil ergibt sich mittelfristig bei der Weiterentwicklung bzw. in der Wartungsphase, wenn bereits mehrere Änderungen am System durchgeführt wurden. Durch die Automatisierung ist das System auch nach mehreren Änderungen konform zu der ursprünglichen Architektur, wodurch auch nachträgliche Erweiterungen einfach integriert und so potentielle Fehler reduziert werden können. Zudem kann das Expertenwissen, das während der Entwicklung der Automation in diese eingeflossen ist, auch zukünftig weiterverwendet werden.

Wiederverwendbarkeit Die Architekturen, Modellierungsmechanismen sowie Automationsschritte können zur Herstellung mehrerer verschiedener Softwaresysteme verwendet werden. Insbesondere wenn eine Reihe von sehr ähnlichen Systemen entwickelt werden soll, ergibt sich ein großes Wiederverwendungspotential: Durch die Abstraktion von den Unterschieden der Systeme und die Auslagerung dieser Unterschiede in Modelle sind nur geringe Anpassungen notwendig, um mit einer Implementierung verschiedene, gleichartige Systeme zu erstellen.

2.1.3 Konzepte

Softwaresysteme bilden unterschiedliche fachliche oder technische Geltungsbereiche ab. Kerngedanke der MDSO ist es, eine für den jeweiligen Bereich spezifische Abstraktion zu finden, um diesen auf Basis der Abstraktion formal beschreiben zu können. Mit Hilfe dieser formalen Beschreibung wird es möglich, die Aspekte des Geltungsbereichs automatisiert in den Entwicklungsprozess zu integrieren. Für diese abstrakte Beschreibung sowie für die automatisierte Integration existieren in der MDSO zwei grundlegende Konzepte (vgl. [BCT05, S. 2]):

- Bei der *Modellierung* werden die Geltungsbereiche der zu entwickelnden Software abstrakt mit Hilfe von Modellen beschrieben.
- *Transformationen* dienen zur Verarbeitung der Modelle; die Modelle fließen so direkt oder indirekt (über Zwischenmodelle) in die zu entwickelnde Software ein.

2.2 Modellierung

2.2.1 Begriffsbildung

Die Modellierung im Kontext der MDSO sowie die in diesem Zusammenhang verwendete Terminologie sollen im Folgenden erläutert werden (vgl. [SVEH07, S. 28ff.]). Abbildung 1 zeigt eine Einordnung der einzelnen Begriffe sowie deren Zusammenhänge untereinander.

Domäne Eine Domäne bezeichnet einen abgegrenzten Interessens- oder Wissensbereich, der durch ein Modell beschrieben werden soll. Unterschiedliche Domänen können dabei jeweils durch eigene Modelle beschrieben werden, sollten jedoch klar voneinander abgegrenzt werden. Zur Verringerung der Komplexität großer Domänen können diese zusätzlich in Subdomänen aufgeteilt werden. Domänen können dabei sowohl fachlich als auch technisch orientiert sein: Die Speicherung von Objekten in Datenbanken stellt eine technische Domäne dar, während die Regeln eines Kreditrankings einer Bank eine fachliche Domäne darstellen.

Metamodell Um eine Domäne formal beschreiben zu können, muss ihre Struktur formalisiert werden. Die formale Beschreibung dieser Domänenstruktur wird als Metamodell bezeichnet. Da Domänen durch Modelle beschrieben werden, beschreibt ein Metamodell zugleich auch die Struktur der Modelle der entsprechenden Domäne. Ein Metamodell

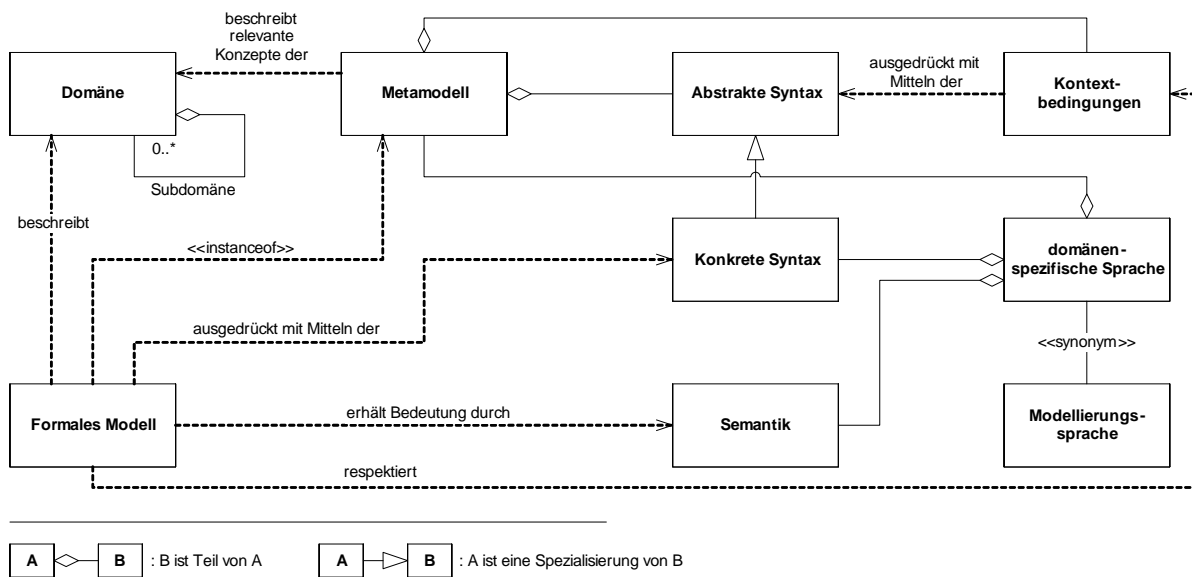


Abbildung 1: Begriffsbildung Modellierung (in Anlehnung an [SVEH07, S. 28])

legt zudem die abstrakte Syntax sowie die Kontextbedingungen einer Sprache fest, mit der die Domäne beschrieben werden kann.

Abstrakte und konkrete Syntax Die Syntax einer Sprache legt die Regeln für die korrekte Konstruktion von Ausdrücken einer Sprache fest. In Bezug auf das Metamodell beschreibt die *abstrakte Syntax* die Regeln zur korrekten Konstruktion von Modellen auf Basis der Elemente des Metamodells.

Die *konkrete Syntax* beschreibt die Regeln für eine konkrete Repräsentation der abstrakten Syntax, beispielsweise in einer textuellen oder grafischen Notation, mit denen die Modelle letztendlich geschrieben werden können. Eine abstrakte Syntax kann dabei durch eine oder auch mehrere konkreten Syntaxen repräsentiert werden. Die Definition einer Syntax kann beispielsweise durch eine kontextfreie Grammatik oder mit Hilfe der Backus-Naur-Form erfolgen.

So definiert die abstrakte Syntax der Sprache Java das Konstrukt der Klassen, die durch einen Namen identifiziert werden. Klassen können zudem von einer anderen Klasse abgeleitet werden sowie Interfaces implementieren. Die konkrete Syntax hingegen legt fest, dass eine Klassendefinition mit dem Schlüsselwort *class* beginnt und die Oberklasse mit *extends* sowie die implementierten Interfaces mit *implements* angegeben werden.

Kontextbedingungen Die Kontextbedingungen² einer Sprache beschränken deren Syntax und beschreiben die Menge der wohlgeformten (gültigen) Sprachausdrücke (vgl. [HR00, S. 14]). Die Bedingungen werden dabei mit Bezug auf die abstrakte Syntax definiert und sind somit von ihr abhängig. Ein Beispiel für Kontextbedingungen von Programmiersprachen ist die Bedingung, dass Variablen vor der ersten Benutzung definiert werden müssen.

Semantik Im Gegensatz zur Syntax, die Regeln über den korrekten Aufbau von Ausdrücken einer Sprache festlegt, legt die *Semantik* die Bedeutung der Sprachelemente auf Basis der abstrakten Syntax fest. Die Semantik kann dabei formal spezifiziert sein oder intuitiv abgeleitet werden. Eine gängige Semantik des binären Operators „+“ beispielsweise ist die mathematische Operation der Addition.

Domänenspezifische Sprache Eine domänenspezifische Sprache (domain specific language, DSL) entspricht einer Programmiersprache der jeweiligen Domäne, mit der Modelle geschrieben werden können. Eine domänenspezifische Sprache basiert auf einem Metamodell der Domäne mit dazugehöriger abstrakter Syntax und Kontextbedingungen. Die konkrete Syntax legt die Notation der domänenspezifischen Sprache fest, während die Bedeutung der Ausdrücke durch die Semantik bestimmt wird.

Formales Modell Ein formales Modell ist eine Beschreibung einer Domäne und wird in seiner Struktur wiederum durch ein Metamodell beschrieben. Ein formales Modell kann als ein in einer domänenspezifischen Sprache geschriebenes Programm angesehen werden, das automatisiert in die ausführbare Software überführt werden kann (vgl. [SVEH07, S. 31]).

Metametamodell Jedes Modell kann in seiner Struktur ebenfalls wieder durch ein Modell beschrieben werden, das Metamodell des Modells. Das Modell bildet dabei eine Instanz des Metamodells. Prinzipiell sind beliebig viele Metaebenen möglich, in der MDSD wird üblicherweise eine dreistufige Modellhierarchie verwendet: Modell, Metamodell und Metametamodell (vgl. [SVEH07, S. 31]). Das *Metametamodell* beschreibt dabei

²Stahl et al. [SVEH07] sprechen an dieser Stelle von der *statischen Semantik* einer Sprache. Dieser Begriff ist jedoch irreführend, da die Einschränkung der Syntax nicht die Semantik (die Bedeutung einer Sprache) betrifft. Daher wird Harel und Rumpe [HR00] folgend der geeignetere Begriff der *Kontextbedingungen* einer Sprache verwendet (vgl. [HR00, S. 14]).

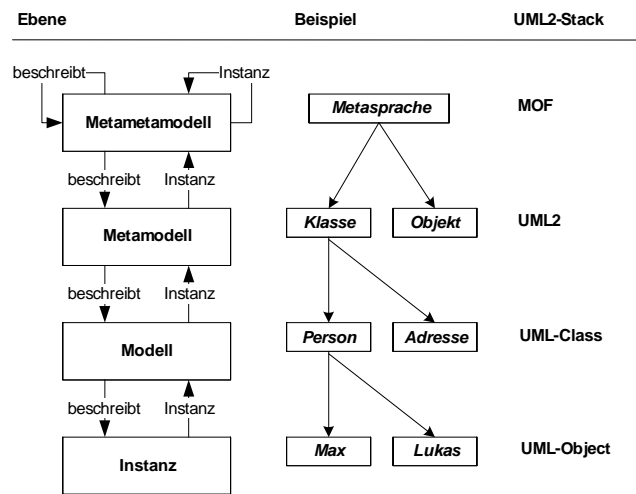


Abbildung 2: Modellebenen in der modellgetriebenen Softwareentwicklung

die Struktur des zu einer Domäne gehörigen Metamodells. Durch die Verwendung einheitlicher Metametamodelle wird es möglich, unterschiedliche Metamodelle mit denselben Softwarewerkzeugen zu erstellen sowie zu verarbeiten. Die Beschreibung von Metametamodellen kann durch weitere, unabhängige Metamodelle oder rekursiv erfolgen. In dem letzteren Fall beschreibt sich das Metametamodell somit selbst.

2.2.2 Leitfaden zur Modellierung

Domänen Modelle dienen dazu, Domänen der Software abstrakt zu beschreiben. Somit kommt der Auswahl der zu beschreibenden Domänen eine zentrale Bedeutung zu (vgl. [HSS05, S. 113]). Prinzipiell kann jede beliebige Domäne modelliert werden, für die dies sinnvoll ist. Schließlich ist die Allgemeingültigkeit des Modellierungsansatzes ein zentraler Aspekt der MDS. Es existieren jedoch einige typische Anwendungsbereiche, die an dieser Stelle vorgestellt werden sollen (vgl. [Gen08a]). Diese Bereiche lassen sich zudem oftmals sinnvoll miteinander kombinieren.

- *Fachliche Aspekte:*

Durch die Modellierung der fachlichen Aspekte der Software kann eine weitgehende Unabhängigkeit von Fachlichkeit und Technik erreicht werden, was dem Ansatz der Trennung von Zuständigkeiten Rechnung trägt. Dadurch wird es möglich, die Software auf einfache Weise um zusätzliche fachliche Funktionen zu erweitern oder im

Sinne von Produktfamilien verwandte Aufgabenstellungen umzusetzen. Beispiele für fachliche Modelle sind

- Entitätstypen, Beziehungstypen
- Regelwerke (z. B. für Kreditranking)

- *Technische Aspekte:*

Durch die Modellierung von technischen Aspekten kann zum einen eine gewisse Unabhängigkeit von der technischen Realisierung gewonnen werden. Zum anderen können verwendete Frameworks über eine einheitliche und klar definierte Schnittstelle benutzt werden. Beispiele für fachliche Modelle sind

- Persistenzmodelle (z. B. OR-Mappings wie Hibernate)
- Eingabemasken (z. B. Swing-GUIs)
- Ablaufsteuerungen (z. B. Transitionen in Web-Anwendungen)

Abstraktionsniveau Eine wichtige Entscheidung beim Entwurf eines Metamodells ist die Wahl des Abstraktionsniveaus. Ist das Modell zu konkret, so ergibt sich kein zusätzlicher Nutzen durch das Modell, die Zielsprache wird nachmodelliert. Zudem führt die damit einhergehende Vielzahl an Aspekten zu Unübersichtlichkeit. Ist das Modell jedoch zu abstrakt, so werden wesentliche Fragestellungen nicht im Modell abgebildet, was wiederum die Generierung erschwert. Insofern ist die Wahl des Abstraktionsniveau keine triviale Entscheidung, sondern kontextabhängig zu treffen. Dafür können die folgenden Empfehlungen gegeben werden:

- Plattformspezifische Modellteile (z. B. Persistenzinformationen) sollten in Konfigurationen oder separate Modelle ausgelagert werden.
- Identische Code-Anteile sollten nicht durch identische Modellelemente erzeugt werden müssen. Stattdessen sollte der entsprechende Code in Bibliotheken ausgelagert werden.

Insgesamt sollte eine Tendenz zu schlanken Modellen vorliegen, die eventuell durch weitere Modelle erweitert werden.

2.3 Transformationen

Eine Transformation ist ein Arbeitsschritt, bei dem ausgehend von einem oder mehreren formalen Modellen als Quelle ein Ergebnis erzeugt wird. Transformationen lassen sich anhand des Transformationsergebnisses in zwei Arten unterscheiden:³

Modell-zu-Modell-Transformation: Als Transformationsergebnis wird ein Modell erzeugt, das anschließend wiederum transformiert werden kann. Die Metamodelle der Modelle können dabei identisch sein, sodass das Quellmodell ergänzt oder modifiziert werden kann. Bei unterschiedlichen Metamodellen können beispielsweise von der Implementierung unabhängige Modelle in Modelle transformiert werden, die Implementierungsaspekte beinhalten.

Modell-zu-Code-Transformation: Als Ergebnis der Transformation wird Quellcode der zu erstellenden Software erzeugt, der auf der Zielplattform der Software basiert. Die Modell-zu-Code-Transformation beinhaltet auch die Möglichkeit, das Modell zur Laufzeit zu interpretieren.

2.3.1 Modell-zu-Modell-Transformation

Modell-zu-Modell-Transformationen stellen einen wesentlichen Bestandteil der MDSD dar, da sie die Lücke zwischen verschiedenen Abstraktionsniveaus überbrücken (vgl. [SVEH07, S. 195]). In einer MDSD ohne Modelltransformationen existieren lediglich zwei Abstraktionsebenen – die Ebene des Modells und die Ebene des Quellcodes. Diese Einschränkung führt jedoch zu zwei Problemen (vgl. [SVEH07, S. 195]):

- Die Konkretisierung vom Modell zu Code muss in einem Schritt erfolgen.
- Die Konzepte der Generatoren müssen dieselben wie die Konzepte der Modelle sein.

Modell-Transformationen hingegen erlauben es, mehrere Abstraktionsebenen miteinander zu verbinden. Dadurch wird es beispielsweise möglich, fachliche Datenmodelle schrittweise in technische Persistenzmodelle zu überführen, aus denen dann OR-Mappings generiert werden können. Modell-Transformationen können in zwei Kategorien eingeteilt werden (vgl. [SVEH07, S. 199f.]):

³Für eine detailliertere Klassifikation von Transformationen siehe [Met05].

- Bei der *Modellmodifikation* wird ein bestehendes Modell verändert, die Modellmodifikation bewegt sich folglich innerhalb eines Metamodells. So können in einem Datenmodell beispielsweise zusätzliche Standardattribute hinzugefügt werden (createdAt, createdBy). Ein Spezialfall der Modellmodifikation ist das *Model-Weaving*, bei dem mehrere Modelle miteinander verbunden werden.
- Bei der *Modelltransformation* wird ausgehend von einem oder mehreren Modellen ein neues Modell erstellt. Die Metamodelle können dabei unterschiedlich sein. Diese Form wird oft verwendet, wenn Modelle in Modelle eines anderen Metamodells übersetzt werden sollen.

2.3.2 Modell-zu-Code-Transformation

Für die Modell-zu-Code-Transformation existieren zwei funktional äquivalente Ansätze: Die Generierung von Quellcode ausgehend vom Modell sowie die Interpretierung des Modells zur Laufzeit.

Generierung: Ausgehend von den Modellen wird bei der Generierung Code erzeugt, der später zusammen mit dem manuell erstellten Quellcode kompiliert wird. Die Integration der Modelle erfordert folglich einen zusätzlichen vorgeschalteten Schritt im Build-Prozess.

Die Generierung erfolgt dabei anhand festgelegter Transformationsregeln. Diese können beispielsweise in einer imperativen Sprache (z. B. Java) ausprogrammiert oder mit Hilfe einer Templatesprache angegeben werden. Ausprogrammierte Generatoren bieten vielfältige Möglichkeiten der Codeerzeugung, in der Regel jedoch nur eine eingeschränkte Unterstützung von Zeichenketten, die statische Anteile des zu erzeugenden Codes abbilden. Sollen große statische Anteile erzeugt werden, so sind Templatesprachen besser geeignet. Ein Beispiel für eine Templatesprache zur Transformation von XML-Modellen ist XSLT [2].

Interpretierung: Bei der Interpretierung wird das Modell im Gegensatz zur Generierung erst zur Laufzeit eingebunden, die Anwendung wird ohne Nutzung der Informationen der Modelle kompiliert. Das Modell wird von der Anwendung eingelesen und die Anwendung verhält sich entsprechend der Informationen im Modell.

Die oben genannten Methoden weisen dabei unterschiedliche Vor- und Nachteile auf:

- *Performance:*

Zur Laufzeit bietet die Generierung prinzipiell eine bessere Performance, da die Modellinformationen bereits in einer für die Ausführung optimierten Form vorliegen (Bytecode, Maschinencode). Insbesondere Modelle mit zahlreichen Varianten führen zu einem Performancenachteil bei der Interpretierung, da die Varianten durch entsprechende Kontrollstrukturen aufwändig abgebildet werden müssen.

- *Laufzeitflexibilität:*

Ein großer Vorteil der Interpretierung ist ihre Laufzeitflexibilität. Während bei der Generierung das Verhalten statisch erzeugt wird und anschließend festgelegt ist, können bei der Interpretierung Modelländerungen zur Laufzeit berücksichtigt werden (vgl. [VD07]). Des Weiteren erfordern Modelländerungen oder Erweiterungen bei der Interpretierung keinen erneuten Build-Prozess, was das Deployment erleichtert.

- *Programmkomplexität und -qualität:*

Generierter Code ist in der Regel weniger komplex als der Code vergleichbarer Interpreter, da die Interpreter Kontrollstrukturen für sämtliche möglichen Varianten des Modells enthalten müssen. Dadurch werden bei generiertem Code prinzipiell die Programmüberprüfung sowie das Debugging erleichtert. Dieser Vorteil ist jedoch auch stark von der Qualität des erzeugten Codes abhängig, insbesondere hinsichtlich der Lesbarkeit des Codes. Werden beispielsweise Variablen aufgrund fehlender semantischer Informationen durchnummeriert, beeinträchtigt dies die Lesbarkeit des erzeugten Codes. Insofern kommt der Nutzung von Plattformidiomen und der Generierung qualitativ hochwertigen Codes eine große Bedeutung zu.

Integration von generiertem und manuellem Code In der Regel steht der erzeugte Code (oder das zu interpretierende Modell) nicht alleine, sondern muss in weiteren, manuell geschriebenen Code integriert werden. Da generierter Code in der Regel mehrfach erzeugt und auch wieder gelöscht wird, ist eine manuelle Erweiterung nicht zielführend. Somit stellt die Integration von generiertem und manuellem Code eine zentrale Herausforderung der MDSD dar. An dieser Stelle sollen drei mögliche Vorgehensweisen kurz vorgestellt werden; für eine detailliertere Diskussion sei auf Stahl et al. [SVEH07, S. 159ff.] verwiesen.

- *Protected Regions:*

Protected Regions (geschützte Bereiche) sind durch Kommentare der Zielsprache gekennzeichnete Bereiche, die bei einer erneuten Generierung nicht überschrieben werden und somit erweitert werden können (vgl. Listing 1). Der generierte Code wird somit direkt manuell verändert. Dieser Ansatz ist methodisch jedoch eher unsauber, da er die Grenzen zwischen generiertem Zwischenergebnis und Quellcode verwischt.

```
1 public void doIt(Object[] args) {
2   // PROTECT START ID(42)
3   // your code goes here
4   // PROTECT END ID(42)
5 }
```

Listing 1: Protected Region in einer Java-Methode

- *Dreistufige Vererbung:*

Bei der dreistufigen Vererbung erfolgt die Integration über den Vererbungsmechanismus von objektorientierten Sprachen. Dazu wird in der Regel die in Abbildung 3 dargestellte Hierarchie umgesetzt: Eine abstrakte Basisklasse implementiert von

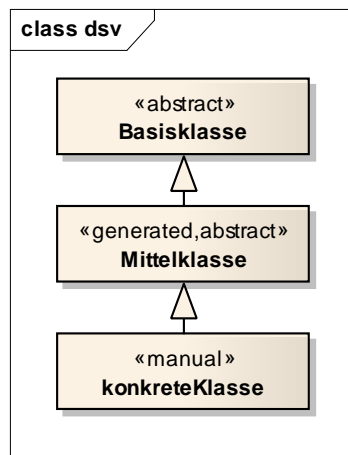


Abbildung 3: Dreistufige Vererbung

der eigentlichen Anwendung unabhängige Funktionalität. Eine generierte, aber ebenfalls abstrakte Mittelklasse implementiert alle Aspekte, die aus dem Modell

abgeleitet werden können, während eine manuell implementierte Klasse schließlich zusätzliche fachliche Methoden bereitstellt. Dieser Ansatz wird beispielsweise im Web-Framework „symfony“ für die Persistenz verwendet.

- *Entwurfsmuster-basiert:*

Bei der entwurfsmuster-basierten Integration werden verschiedene Formen der Vererbung und Delegation verwendet. So kann manueller Code generierten Code aufrufen bzw. von diesem aufgerufen werden. Um die Abhängigkeiten besser zu strukturieren, können entsprechende Erzeugungsmuster eingesetzt werden.

2.3.3 Cartridges

Cartridges stellen zwar keine konzeptuell neue Transformation dar, sind jedoch eine logische Weiterentwicklung der vorgestellten Transformationen und sollen daher an dieser Stelle erwähnt werden. Cartridges (Steckmodule) sind fertige Generatorkomponenten für einzelne Domänen (vgl. [SVEH07, S. 197]) und umfassen neben einem Metamodell fertige Modell-zu-Code-Transformationen. Cartridges erlauben es beispielsweise, technische Details für verwendete Frameworks wie Spring oder Hibernate zu kapseln. Für die Nutzung einer solchen Cartridge muss ein dem Metamodell der Cartridge entsprechendes Modell erzeugt werden – typischerweise mit Hilfe einer Modelltransformation. Die Erzeugung des Generierungsergebnisses geschieht dann innerhalb der Cartridge. Diese Vorgehensweise erlaubt es, oft verwendete Generatoren durch eine einheitliche Schnittstelle wiederzuverwenden. Eine Austauschmöglichkeit für Cartridges bietet beispielsweise die Fornax-Plattform [3].

3 openArchitectureWare

3.1 Grundlagen

openArchitectureWare bezeichnet ein freies Generator-Framework für die modellgetriebene Softwareentwicklung. Es wurde in Java geschrieben und bündelt verschiedene Werkzeuge, um aus Modellen Text zu generieren. Das Framework ist also nicht auf die Erzeugung von Quellcode einer vorgegebenen Sprache festgelegt, sondern auf die Transformation von Modellen im Allgemeinen (vgl. [Völ07, S. 1]).

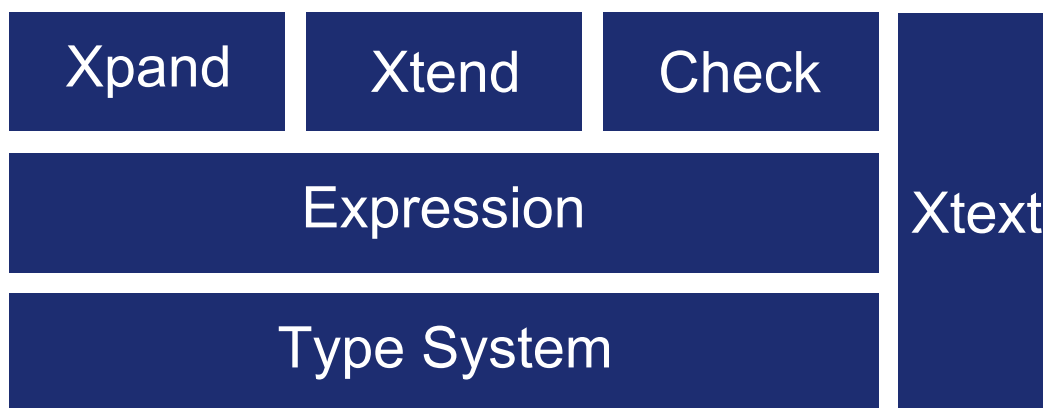


Abbildung 4: openArchitectureWare-Framework Überblick (vgl. [Gen08b, Folie 4])

In Abbildung 4 sind die Hauptbestandteile des Frameworks dargestellt. openArchitectureWare besteht aus den drei Sprachen *Xpand*, *Xtend* und *Check*. Die Sprachen basieren alle auf einem gemeinsamen Typ- und Expressions-System, so dass dieses nur einmal gelernt werden muss und dann sofort in den anderen Sprachen angewandt werden kann. Zu openArchitectureWare gehört außerdem das Xtext-Framework, mit dem textuelle DSLs erstellt werden können.

Die Sprachen von openArchitectureWare

Xpand ist eine Template-Sprache zur Beschreibung der generierten Texte

Xtend ermöglicht Erweiterung von Xpand und Check

Check mit Check lassen sich Randbedingungen der Metamodelle formulieren.

Um in das Thema openArchitectureWare und die damit verbundenen Komponenten besser einsteigen zu können, wird an dieser Stelle ein einführendes Beispiel besprochen.

Wie die Komponenten dabei genau funktionieren und zusammenspielen wird in den folgenden Abschnitten besprochen werden. In Abbildung 5 ist ein einfaches Klassendiagramm dargestellt.

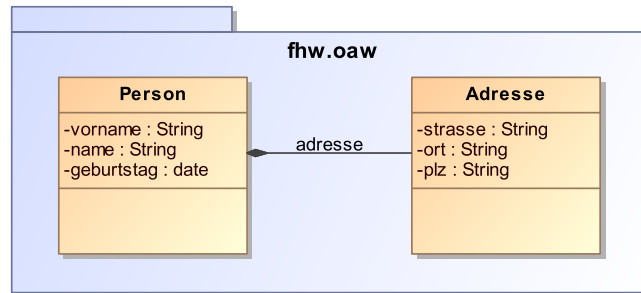


Abbildung 5: Erstes Beispiel: Klassendiagramm

Um aus dem dargestellten UML Diagramm Java Quelltext zu generieren sind die folgenden Schritte notwendig:

- Einlesen des Modells
- Übergabe des Modells zum Generator
- Generieren des Quelltextes
- Ausgabe in `.java` Dateien

Die Ausführung und Definition dieser Schritte erfolgt dabei in einem sogenannten *Workflow*. In Listing 2 ist ein Workflow dargestellt. Über die *XmiReader-Komponente* wird das Modell eingelesen. Anschließend wird eine Generatorkomponente gestartet, die das Modell anhand eines Templates (mit dem Namen `Root`) expandiert und in Quelltext umwandelt. Das Ergebnis wird in den Ordner `src-gen` geschrieben.

```
1 <workflow>
2   ...
3   <component class="oaw.emf.XmiReader">
4     <modelFile value="OAW_Einführendes_Beispiel.uml"/>
5     <outputSlot value="theModel"/>
6   </component>
7
8   <component id="generator"
9     class="org.openarchitectureware.xpand2.Generator"
10    skipOnError="false">
11
12     <expand value="xpand::Root::Root FOR theModel"/>
13     <outlet path="src-gen">
14       <postprocessor class="org.openarchi...output.JavaBeautifier"/>
15     </outlet>
16
17     ...
18
19   </component>
20   ...
21 </workflow>
```

Listing 2: Erstes Beispiel: Workflow

Einen ersten Eindruck darüber, wie diese Regeln formal notiert werden ist in Listing 3 zu sehen. Es werden dort, ähnlich zu XSLT, bestimmte Templates beschrieben, die dem Generator mitteilen, wie mit welchen Elementen zu verfahren ist. Die Kommandos sind dabei in « und » geklammert. Alles was außerhalb steht, wird als Text in die generierten Dateien übernommen. Es ist außerdem zu erkennen, dass aus einem Template weitere Funktionen aufgerufen werden können (z.B. `asPackageName()`).

```

1  ...
2  « DEFINE Root FOR Class »
3    « FILE this.name+".java" »
4    package « asPackageName() »;
5
6    public class « asClassName() » {
7      « FOREACH ownedAttribute.typeSelect(Property) AS p »
8        private « p.type » « p.name.toLowercase() »;
9
10     public void set« p.name.toFirstUpper() »(« p.type » «
11         p.name.toLowercase() ») {
12         this.« p.name.toLowercase() » = « p.name.toLowercase() »;
13     }
14
15     public « p.type » get« p.name.toFirstUpper() »() {
16         return this.« p.name.toLowercase() »;
17     }
18     « ENDFOREACH »
19 }
20 « ENDFILE »
21 « ENDDDEFINE »
22 ...

```

Listing 3: Erstes Beispiel: Template

In Listing 4 ist die Ausgabe zu sehen, die bei der Ausführung des Workflows entsteht.

```

1 WorkflowRunner      - -----
2 WorkflowRunner      - openArchitectureWare 4.3.0, Build 20080508-1430PRD
3 WorkflowRunner      - (c) 2005-2008 openarchitectureware.org and contrib
4 WorkflowRunner      - -----
5 WorkflowRunner      - running workflow: generator.oaw
6 WorkflowRunner      -
7 CompositeComponent  - XmiReader: file 'OAW_Einfuehrendes_Beispiel.uml'
8 CompositeComponent  - Generator(generator): generating
9                      - 'xpanse::Root::Root FOR theModel'
10 Generator           - Written 2 files to outlet [default](src-gen)
11 WorkflowRunner      - workflow completed in 1515ms!

```

Listing 4: Erstes Beispiel: Ausgabe

Dabei werden die beiden Dateien *Person.java* und *Adresse.java* erstellt. Die generierte Personen-Klasse ist in Listing 5 zu sehen.

```
1 package fhw.oaw;
2
3 public class Person {
4
5     private String vorname;
6
7     public void setVorname(String vorname) {
8         this.vorname = vorname;
9     }
10
11    public String getVorname() {
12        return this.vorname;
13    }
14
15    private String name;
16
17    public void setName(String name) {
18        this.name = name;
19    }
20
21    public String getName() {
22        return this.name;
23    }
24
25    private String geburtstag;
26
27    public void setGeburtstag(String gGeburtstag) {
28        this.geburtstag = geburtstag;
29    }
30
31    public String getGeburtstag() {
32        return this.geburtstag;
33    }
34
35    private fhw.Adresse Adresse;
36
37    public void setAdresse(fhw.Adresse adresse) {
38        this.adresse = adresse;
39    }
40
41    public fhw.Adresse getAdresse() {
42        return this.adresse;
43    }
44
45 }
```

Listing 5: Erstes Beispiel: Java Datei

3.2 Workflow

Die Workflow-Engine des *openArchitectureWare* Projektes ist die zentrale Steuereinheit des Frameworks. Mit einem Workflow werden alle Arten von Generatoren der *openArchitectureWare* gesteuert. Ein Workflow ist in einfacher XML-Syntax beschrieben und besteht aus verschiedenen Komponenten, den sogenannten *WorkflowComponents*. Diese Komponenten werden in der Konfigurationsdatei in einer Sequenz zusammengesetzt und nacheinander ausgeführt. In der unten stehenden Liste sind einige Standardaufgaben für solche Workflow-Komponenten aufgeführt (vgl. [EFH⁺08, S. 47]).

- Laden von Modellen
- Leeren der Zielverzeichnisse
- Aufbereiten d. generierten Codes
- Starten von Generatoren
- Bedingte Komponenten

Die einzelnen Komponenten können über einen gemeinsamen Kontext die Laufzeitumgebung des Workflows abfragen. Diese Laufzeitumgebung wird *WorkflowContext* genannt und stellt sogenannte *Slots* bereit. In den Slots können Werte für nachfolgende Komponenten abgelegt werden. In Listing 6 ist ein beispielhafter Workflow mit Kommunikation über Slots aufgezeigt (vgl. [EFH⁺08, S. 65]).

```
1 <workflow>
2   <!-- define an output folder -->
3   <property name='target' value='src-gen/'/>
4
5   <!-- load the model -->
6   <component class="org.openarchitectureware.emf.XmiReader">
7     ...
8     <outputSlot value="model"/>
9   </component>
10
11  <!-- check consistency -->
12  <component class="datamodel.generator.Checker">
13    ...
14    <modelSlot value="model"/>
15  </component>
16
17  <!-- generate some code -->
18  <component class="oaw.xpand2.Generator">
19    ...
20    <modelSlot value="model"/>
21    <output path="{target}">
22  </component>
23 </workflow>
```

Listing 6: Beispiel Workflow

Zunächst wird eine Variable mit dem Namen *target* und dem Wert *src-gen/* definiert. Die Generatorkomponente wird später auf den Wert zugreifen und unter dem angegebenen Pfad den generierten Code ablegen. Nachdem der *XmiReader* ein Modell eingelesen hat, wird dieses im Kontext unter dem Namen "model" abgelegt. Die nachfolgenden Komponenten *Checker* und *Generator* greifen über diesen Namen wieder auf das Modell zu. In dem Beispiel ist gut zu erkennen, dass die Workflow-Komponenten jeweils einer Java-Klasse entsprechen. Während ein Workflow ausgeführt wird, müssen die entsprechenden Klassen in der *Classpath*-Variablen von Java auffindbar sein. Der Workflow kann über eine von openArchitectureWare vorgegebene Klasse *WorkflowRunner* gestartet werden. Auch ein indirekter Aufruf über die Eclipse-Plattform oder Apaches Ant ist möglich.

3.2.1 Cartridge

Bei der modellgetriebenen Entwicklung von Software ist darauf zu achten, dass die entwickelten Metamodelle keine Informationen über die Zielplattform enthalten, damit ein genügendes Abstraktionsniveau erhalten bleibt (vgl. [Pee07, S. 26]). Letztendlich muss

aber Information für die Zielplattform bereitgestellt werden. Hierfür werden in openArchitectureWare sogenannte *Cartridges* genutzt. In einer Cartridge können plattformspezifische Details getrennt von der fachlichen Seite modelliert werden. Um die Metamodelle der beiden Seiten zu vereinen kann eine M2M-Transformation genutzt werden, um so das fachliche Modell in das implementierungsabhängige Modell zu überführen (vgl. [SVEH07, S. 197]).

Als Beispiel soll hier eine Anwendung dienen, die mit Hilfe von Hibernate Entitäten in einer Datenbank persistent speichert. Für die Integration von Hibernate in die Anwendung ist es notwendig einige plattformspezifische Elemente mit in das Metamodell aufzunehmen. Es muss formal festgelegt werden, was z.B. als Primärschlüssel zu verwenden ist, und nach welcher Strategie ein nächster Wert zu ermitteln ist. Diese Angaben haben aber keinen Bezug zur fachlichen Domäne des Metamodells. Durch die Kapselung der Funktionalität für Hibernate in eine Cartridge wird dieses Problem gelöst. Innerhalb der Cartridge wird ein eigenes Metamodell mit plattformspezifischen Aspekten entwickelt. Soll nun die Beispielanwendung mit Hibernate arbeiten, kann die Methodik der M2M-Transformation angewandt werden, um die DSL der Anwendung um spezifische Aspekte der Hibernate-Cartridge zu erweitern und in ein Modell der Cartridge umzuwandeln (vgl. [SVEH07, S. 198]).

Auf der Seite des *Fornax-Projektes* [3] gibt es frei verfügbare Cartridges, die in das eigene Projekt eingebunden werden können. Hier wird z.B. eine Hibernate-Cartridge bereitgestellt. Somit muss das eigene Entwicklerteam nicht mehr über ein spezialisiertes Wissen über die Zielplattform verfügen.

3.3 Xpand

Xpand ist eine Templatesprache des openArchitectureWare-Frameworks, die für die Codegenerierung genutzt wird. Bei der Generierung von Code werden Programme ausgeführt, die Programme als Ausgabe erzeugen. Man spricht bei dieser Art von Programmen auch von Metaprogrammierung (vgl. [SVEH07, S. 143]).

3.3.1 Metaprogrammierung

Es existieren neben Codegeneratoren noch weitere Arten der Metaprogrammierung. In der Programmiersprache *C* wird ein *Präprozessor* eingesetzt, über den z.B. Konstanten am Anfang eines Quelltextes gesetzt werden können. Der Präprozessor wird dabei vor

dem Aufruf des Compilers gestartet und generiert neue Codefragmente innerhalb des Quelltextes.

Als weiteres Beispiel können die Templates aus *C++* und die Generics aus *Java* herangezogen werden. Der Compiler stellt hierbei die Möglichkeit zur Metaprogrammierung bereit und übersetzt die Schablonen jeweils in ein Konstrukt der jeweiligen Programmiersprache.

Es gibt also verschiedene Arten der Metaprogrammierung. Es kann sich um einen festen Bestandteil der Sprache (Templates) handeln oder um eine davon unabhängige Vorgehensweise (Präprozessoren). Der Codegenerator der openArchitectureWare arbeitet hier wie der Präprozessor und ist dem Compiler vorgeschaltet. Allerdings unterscheiden sich Generator und Präprozessor in einigen entscheidenden Punkten (vgl. [SVEH07, S. 145]):

- Es existiert ein austauschbares Metamodell
- Trennung von generiertem und per Hand geschriebenem Quelltext
- Generierung von mehreren, verschiedenen, unabhängigen Artefakten möglich

3.3.2 Gründe für Xpand

Für einen Codegenerator muss nicht zwingend notwendig eine neue Sprache wie Xpand entworfen werden. Es ist durchaus denkbar diesen mit etablierten Methoden und Sprachen zu implementieren.

Generator mit XSLT Ein solcher Generator könnte zum Beispiel deklarativ über *XML Stylesheet Transformation* (XSLT) realisiert werden. XSLT ist Prinzipiell gut geeignet, da es für die Transformation von XML in beliebigen Text genutzt werden kann. Das Modell müsste also als XML-Baum und das Metamodell z.B. als XML Schema vorliegen. Der Vorteil ist hierbei, dass es bereits eine Vielzahl an Editoren und Validierungsmöglichkeiten gibt. Außerdem steht mit *XPath* ein gutes Werkzeug zur Objektnavigation zur Verfügung (vgl. [Kla07, S. 11]).

XSLT hat allerdings 2 entscheidende Nachteile. Auf der einen Seite werden XSLT Dateien, durch die XML Syntax bedingt, sehr schnell unübersichtlich und unleserlich. Zum Anderen kann während des Übersetzungsprozesses mit XSLT nicht ohne weiteres gegen das Metamodell validiert werden. Es muss also entweder vorher mit einen externen

Werkzeug, oder hinterher durch den Compiler die Typsicherheit gewährleistet werden (vgl. [SVEH07, S. 147]).

Generator mit Java Es ist auch denkbar, dass ein Codegenerator in einer imperativen Programmiersprache (z.B. *Java*) geschrieben wird. Gegenüber XSLT bringt Java den Vorteil der Typisierung mit. Diese Typsicherheit kann allerdings nur innerhalb der Sprache gewährleistet werden. Somit muss das Modell in Form von Java-Objekten vorliegen, bzw. erst umgewandelt werden. Die Navigation auf dem Objektgraphen ist allerdings sehr kompliziert und muss oft über Schleifen umgesetzt werden (vgl. [SVEH07, S. 149]).

Ein weiterer Grund, warum sich imperative Programmiersprachen nur selten für Codegeneratoren eignen, ist die Art der Textverarbeitung. In Java muss eine Verkettung zweier Strings explizit mit `+` gekennzeichnet werden und es gibt keine Texte, die über mehrere Zeilen laufen. Hier muss ebenfalls explizit ein Zeilenumbruch (`\n`) eingefügt werden.

Generator in openArchitectureWare Xpand wurde entwickelt, um die Vorteile beider Ansätze zu vereinen, ohne dabei die Nachteile zu übernehmen. In Xpand kann auf das *Typ- und Expressionsystem* der openArchitectureWare zurückgegriffen werden. Hiermit kann komfortable auf den Objektgraphen zugegriffen werden (XPath) und das Typsystem ist nicht abhängig von der Hostsprache (Java). Die Syntax ist nicht so ausführlich wie in XML und somit gut zu warten. Es wird genau wie in XSLT deklarativ programmiert, was die Lesbarkeit des Generators weiter steigert. Außerdem ist die Verarbeitung von Texten sehr einfach, da zum einen Einrückungen und Zeilenumbrüche einfach in den generierten Quelltext übernommen werden. Zum anderen werden die Xpand-Kommandos nicht mit Mitteln der möglichen Zielsprachen (XML, Java, C, etc.) gekennzeichnet. Stattdessen werden französische Anführungszeichen « und » genutzt. Somit müssen Anführungszeichen nicht maskiert werden (vgl. [Kla07, S. 2]).

In Listing 7 ist eine Xpand-Datei zu sehen, in der eine Javaklasse mit einer statischen *main* Methode generiert wird.

```
1 < IMPORT sddsl >
2
3 < DEFINE Root FOR Model >
4   < EXPAND Standalone FOREACH types.typeSelect(Entity) >
5 < ENDDDEFINE >
6
7 < DEFINE Standalone FOR Entity >
8 < FILE name.toLowerCase().toFirstUpper()+"Standalone.java" >
9 package fhw.sd.standalone;
10
11 import fhw.sd.awk.AwkFactory;
12 import fhw.sd.awk.generic.usecase.DialogData;
13 import fhw.sd.client.framework.DialogManager;
14
15 /**
16  * @author oaw generated
17  */
18 public class < name.toLowerCase().toFirstUpper() >Standalone {
19
20   /**
21    * @param args ignored
22    */
23   public static void main(String[] args) {
24     DialogManager.init(new SwingDialogManagerCallback());
25     DialogData data = AwkFactory.getGenericUseCase().getDialogData("<
26     name >UC");
27     DialogManager.openAsEditor("< name >Dialog", data);
28   }
29 }
30 < ENDFILE >
31 < ENDDDEFINE >
```

Listing 7: Beispiel einer Xpand-Datei

3.3.3 Sprachelemente

DEFINE Mit Define gekennzeichnete Blöcke sind die elementaren Einheiten von Xpand und werden *Templates* genannt. Ein Template wird durch seinen Namen identifiziert und kann eine Liste von Parametern erhalten (siehe Listing 8). Nach dem Schlüsselwort **FOR** folgt der Name des Metatypen, für den das Template definiert ist. Der Zugriff auf diesen Typen erfolgt innerhalb des **DEFINE**-Block implizit, oder explizit über **this**.

```
1 < DEFINE templateName(formalParameterList) FOR MetaClass >
2   a sequence of statements
3 < ENDDFINE >
```

Listing 8: DEFINE

EXPAND Mit dem XPAND-Block lassen sich andere Templates aufrufen (siehe Listing 9). `definitionName` entspricht dem aufzurufenden DEFINE-Namen. Sind in dem DEFINE-Block formale Parameter deklariert, müssen die aktuellen Parameter hier in Klammern übergeben werden. Folgt dem Namen kein FOR oder FOREACH, wird das Template für *this* aufgerufen. Ansonsten wird mit FOR und FOREACH das zu übergebende Objekt festgelegt (FOREACH ist für eine Liste von Objekten definiert).

```
1 < EXPAND definitionName [(parameterList)] [FOR expression | FOREACH
   expression ] >
```

Listing 9: XPAND

IF und FOREACH Mit IF- und FOREACH-Blöcken stehen innerhalb von Templates Kontrollstrukturen bereit, die alternative Pfade oder Iterationen über Listen zulassen.

FILE Über den FILE-Block lassen sich aus Xpand heraus Dateien erstellen. Der Dateiname ergibt sich aus der Auswertung des Ausdrucks (siehe Listing 10). Die Statements innerhalb des FILE-Blocks geben den Inhalt der Datei vor. Die Datei wird in einem Verzeichnis gespeichert, welches über den Workflow angegeben wird (siehe auch Abschnitt 3.2).

```
1 < FILE expression >
2   a sequence of statements
3 < ENDFILE >
```

Listing 10: FILE

PROTECT Es ist mit Xpand möglich geschützte Bereiche innerhalb des Quelltextes zu definieren. Im generierten Quelltext wird dieser Bereich über eine Anfangs- und Endmarke festgelegt. In diesem Bereich können die Entwickler manuell Quelltext einfügen, ohne dass bei einem erneuten Generatorkauf dieses Bereich überschrieben wird.

Die Verwendung geschützter Bereiche durchbricht allerdings die gewünschte Trennung von manuellem und generiertem Code, da beide in derselben Datei stehen und sollte nur in Situationen genutzt werden, in denen es gar nicht, oder nur mit erheblichem Mehraufwand zu verhindern ist. In den meisten Fällen können solche Situationen durch Mechanismen der Zielsprache (includes, Vererbung oder Entwurfsmuster) behoben werden (vgl. [SVEH07, S. 159f und S. 145]).

In Listing 11 ist ein Beispiel für einen PROTECTED-Block zu sehen. Über CSTART und CEND werden hier Kommentaranfang und -ende festgelegt. Zwischen CSTART und CEND wird ein über ID festgelegter Identifier geschrieben, der den PROTECTED-Bereich eindeutig kennzeichnet.

```
1 < PROTECT CSTART "/*" CEND "*/" ID ElementsUniqueID >
2   here goes some content
3 < ENDPROTECT >
```

Listing 11: PROTECTED

Eine generierte Quelldatei in Java könnte so aussehen, wie in Listing 12.

```
1 public class Person {
2   /*PROTECTED REGION ID(Person) ENABLED START*/
3   This protected region is enabled, therefore the contents will
4   always be preserved. If you want to get the default contents
5   from the template you must remove the ENABLED keyword (or even
6   remove the whole file :-))
7   /*PROTECTED REGION END*/
8 }
```

Listing 12: Generierte Datei mit geschütztem Bereich

AROUND Die AROUND-Anweisung lässt aspektorientierte Programmierung in Xpand zu. Der nach dem Schlüsselwort AROUND stehende Teil definiert den Join Points des Aspektes. Der Join-Point entspricht einer in einem Template festgelegten Signatur.

Bevor das genannte Template expandiert wird, wird die AROUND-Anweisung ausgewertet. Hier kann über das vorgegebene Objekt *targetDef* auf das überschriebene Template zugegriffen werden. Über die Methode `proceed()` kann das eigentliche Template ausgewertet werden (siehe Listing 13).

```
1 « AROUND qualifiedDefinitionName [(parameterList)]? FOR type »
2 // Before Target processing
3 « targetDef.proceed() »
4 // After Target processing
5 « ENDAROUND »
```

Listing 13: AROUND

Wird auf den Aufruf von `proceed()` verzichtet, wird das Template nicht ausgewertet und somit überschrieben. Die AROUND Anweisung macht es also möglich vorhandene Templates mittels Aspektorientierung zu manipulieren, ohne dass der Quellcode geändert werden muss. Gerade in der Benutzung von Cartridges (siehe Abschnitt 3.2.1) kann dies von großer Bedeutung sein, da so leicht Änderungen vorgenommen werden können.

REM Kommentare lassen sich in Xpand mit dem REM Block kennzeichnen.

3.4 Einheitliches Typ- und Expression-System

Wie in der Einleitung bereits erwähnt, gibt es im openArchitectureWare-Framework ein einheitliches Typ- und Expressions-System. Da die Sprachen *Xtend*, *Xpand* und *Check* mit diesem Typsystem arbeiten ist das Erlernen der verschiedenen Sprachen relativ leicht. Der eigentliche Grund für ein neues Typsystem ergibt sich aber aus den Schwierigkeiten, die bei anderen, etablierten Sprachen existieren. Es soll einfach möglich sein, in einem Objektgraphen zu navigieren. In imperativen Sprachen muss hierfür oft in Schleifen iteriert werden (vgl. [EFH⁺08, S. 57]). Bei der Verwendung von openArchitectureWare muss also nicht gegen eine generische API entwickelt werden. Stattdessen ist es möglich Metametamodelle mittels Adapter in dem Typsystem zu registrieren, wodurch ein direktes Arbeiten auf der Metaebene möglich wird. (vgl. [SVEH07, S. 154f]).

3.4.1 Typsystem

Ein Typ wird durch seinen Namen und einen Namensraum identifiziert. Dieser *Namespace* wird dabei durch `::` getrennt. Ein Typ besteht aus Eigenschaften und Operationen

und kann von anderen Typen abstammen. Das Typsystem besteht aus vorgefertigten *built-in Types*, sowie registrierten Metametamodellen. Es ist also möglich, dass System um weitere Typdefinitionen zu erweitern (vgl. [EFH⁺08, S. 57]).

Built-in Types Einige der vorgegebenen Typen sind in der folgenden Liste aufgezeigt. Es gibt für die meisten dieser Typen mehrere Funktionen. Diese sind in der *openArchitectureWare* Referenz sehr gut erklärt (siehe [EFH⁺08]).

Typ	Erklärung
Object	Ist der Basistyp, von dem alle Typen erben
Void	Ist der Nulltyp. Die einzig erlaubte Instanz von <i>void</i> ist <i>null</i>
String	Der String Datentyp. Wird sowohl durch " als auch ' umschlossen
List, Set	Listen und Mengen in <i>openArchitectureWare</i> . Können durch Angabe eines Typs getypt werden (<i>Set[my::Type]</i>)

Abbildung 6: Built-in Types

3.4.2 Expression-System

Die Syntax der Ausdrücke ist sehr ähnlich zu der in Java gebräuchlichen Notation. Mit `myObject.name` wird das Feld *name* der Instanz *myObject* selektiert. Analog dazu werden Methoden mit der Angabe von (*param1,param2,...*) aufgerufen (`myObject.method()`).

Einfache Operationen Da die meisten Typen so, oder so ähnlich auch in anderen Sprachen vorkommen, gibt es viele bekannte Funktionen für die verschiedenen Typen. Zum Beispiel bringt *Object* die Methode *equals()* mit. Zu diesen trivialen Operationen zählen auch `+`, `-`, `<`, `>`, `!=`, `=`, `&&`, `||`, ... der numerischen und logischen Datentypen.

Funktionen höherer Ordnung Für Listen gibt es einige Funktionen (*Higher order functions*), die den Umgang mit Listen stark vereinfachen und so die geforderte Navigation über den Objektgraphen ermöglichen.

Kontrollstrukturen Auch in der Ausdruckssprache von *openArchitectureWare* gibt es Kontrollstrukturen in Form einer If-Anweisung und eines Switch-Verteilers.

Funktion	Erklärung
<code>l.select(e boolean-expr.-with-e)</code>	Mit <code>select()</code> wird ein Filter auf der Liste definiert. Die Ergebnismenge ist eine Untermenge von <code>l</code> und enthält nur die Elemente, für die <code>boolean-expr.-with-e</code> wahr ergibt.
<code>l.typeSelect(a::Type)</code>	Äquivalent zu <code>l.filter(e e.type=='a::Type')</code>
<code>l.collect(e expression)</code>	Mit <code>collect()</code> kann eine Funktion auf die Elemente ausgeführt werden. Das Ergebnis der Funktionsaufrufe wird zurückgeliefert
<code>l.forAll(e boolean-expr.-with-e)</code>	Prädikat: Ausdruck muss für alle Elemente der Liste gültig sein
<code>l.exists(e boolean-expr.-with-e)</code>	Prädikat: Ausdruck muss für mindestens ein Elemente der Liste gültig sein

Abbildung 7: Higher order functions

3.5 Xtend

Xtend ist eine statisch getypte, funktionale Programmiersprache. *Xtend* eignet sich für die non-invasive Erweiterung von Modelltypen und ist ein effektives Werkzeug im Umgang mit Modellen (vgl. [SVEH07, S. 152]). Mit Hilfe von *Xtend* ist es z.B. möglich die in Abschnitt 2.3.1 vorgestellte Modell-zu-Modell-Transformation zu realisieren (siehe unten)(vgl. [SVEH07, S. 206]).

In Listing 14 ist eine Beispiel-Extension dargestellt. Mit der `import`-Anweisung wird ein Metamodell importiert, das im Klassenpfad liegt, und es werden 3 *Xtend* Funktionen definiert. Für die `aModelFunction` wird der Rückgabotyp auf `String` festgelegt und 2 Parameter definiert. Die `aModelFunction` erwartet also ein Modell und einen `String`. Im Funktionsrumpf werden die beiden Parameter konkateniert und alle „`::`“ aus dem Metatypnamen durch einen „`.`“ ersetzt. Die benutzte Syntax entspricht wieder der, aus dem einheitlichen Typ- und Expression-System. Bei der zweiten Funktion wird auf die Angabe des Rückgabetypen verzichtet. Stattdessen wird das *Xtend*-Framework den Typen selbst ermitteln. Die dritte delegiert die Arbeit an eine externe Java Klasse.

```

1 import sddsl;
2
3 /**
4  * join metaType name with s
5  */

```

```
6 String aModelFunction(Model m, String s) :
7     s+m.metaType.name.replaceAll("::",".");
8
9 /**
10  * trim a string and performs a lowercase
11  */
12 trimToLower(String s) :
13     s.trim().toLowerCase();
14
15 /**
16  * delegates action to java class
17  * trim a string and performs a lowercase
18  */
19 String javaTrimToLower(String s) : JAVA
20     extensions.TrimToLower.format(java.lang.String);
```

Listing 14: Beispiel-Extension

Java Erweiterungen In dem unten stehenden Listing ist die Java-Klasse zu sehen, die aus der dritten Funktion des Extension Beispiels aufgerufen wurde. Die Funktion muss hierbei immer eine öffentliche und statische Funktion sein. Außerdem müssen die Rückgabe- und Parametertypen mit denen in der Xtend-Datei übereinstimmen. Zu beachten ist, dass die Angabe der Klasse und der Java-Typen in der Extension immer als *full qualified class name* erfolgen muss.

```
1 package extensions;
2
3 public class TrimToLower {
4
5     public static String format(String s) {
6         return s.trim().toLowerCase();
7     }
8 }
```

Listing 15: Beispiel Java-Extension

Zwischenspeichern der Funktionswerte Durch Voranstellen des Schlüsselwortes *cached* (siehe Listing 16) vor eine Xtend-Funktion, werden die Ergebnisse der Funktion in einem Cache zwischengespeichert. Der Schlüssel für den Cache ist hierbei erster Para-

meter. Für jeden Parameter wird die Funktion also nur einmal berechnet. Jeder weitere Aufruf erhält seinen Wert aus dem Cache.

```
1 /**
2  * delegates action to java class
3  * trim a string and performs a downcase
4  */
5 cached String javaTrimToLower(String s) : JAVA
6   extensions.TrimToLower.format(java.lang.String);
```

Listing 16: Cached Extension

3.5.1 Aufruf

Wie in obigem Listing zu erkennen ist, können Xtend Funktionen auf verschiedene Art und Weise deklariert werden. Zum einen können Definitionen direkt in der Xtend Datei erfolgen oder aber in externe Java-Klassen ausgelagert werden.

Aufruf in Xpand In Listing 17 ist ein zu dem Beispiel korrespondierender Aufruf innerhalb eines *Xpand* Templates zu sehen. Nachdem die Extension importiert wurde, können deren Funktionen genutzt werden. Für Modelle wird das Root-Template erweitert und eine Datei mit dem Namen "Model"+aModelFunction(".") erstellt. In dieser Datei werden verschiedene Texte an die eben definierten Funktionen übergeben. Die Xtend Funktionen werden jeweils an den ersten Parameter gebunden. Die Aufrufe `s.trimToLower()` und `trimToLower(s)` sind also äquivalent (vgl. [EFH⁺08, S. 71]).

```
1 < IMPORT sddsl >
2 < EXTENSION extensions::MyExtension >
3
4 < DEFINE Root FOR Model >
5 < FILE "Model"+aModelFunction(".") >
6 < " Hello World ".trimToLower() >
7 < " \t\t JAVa Hello World \n ".javaTrimToLower() >
8 < trimToLower(" Hello World ") >
9 < javaTrimToLower(" \t\t JAVa Hello World \n ") >
10 < ENDFILE >
11 < ENDDDEFINE >
```

Listing 17: Beispiel Aufruf der Extension

Wird das Xpand-Template innerhalb eines Workflows aufgerufen, wird eine Datei in das Dateisystem abgelegt. Der Inhalt dieser Datei und damit das Ergebnis des Aufrufs ist in Listing 18 zu sehen.

```
1 hello world
2 java hello world
3 hello world
4 java hello world
```

Listing 18: Beispiel Ausgabedatei *Model.sddsl.Model*

Aufruf aus einem Workflow Eine Xtend-Funktion kann auch aus einem Workflow heraus gestartet werden. In folgendem Listing wird `oaw.xtend.XtendComponent` genutzt, um eine Xtend-Funktion zu starten. Der Komponente werden drei Informationen bereitgestellt. Es wird definiert, welche Extension zu laden ist und welche Funktion daraus aufgerufen werden soll. Außerdem wird der zu übergebende Parameter über einen Slotnamen zugewiesen.

```
1 <!-- transform theModel via extend -->
2 <component class="oaw.xtend.XtendComponent">
3   <metaModel id='mm'
4     class='org.eclipse.m2t.type.emf.EmfRegistryMetaModel' />
5   <invoke value="extensions::MyExtension::toModel(theModel)" />
6   <outputSlot value="transformedModel" />
7 </component>
```

Listing 19: Aufruf über WorkflowComponent

3.5.2 M2M-Transformation

In unten stehendem Listing wird eine Modell-zu-Modell-Transformation in Xtend formuliert. Das Modell wird dabei auf sich selbst abgebildet. Die *create*-Anweisung vor den Funktionsdefinitionen erstellt eine Instanz des darauf folgenden Typen. Auf diese Instanz kann innerhalb der Funktion mit *this* zugegriffen werden. Bei näherer Betrachtung von Listing 20 wird also deutlich, dass die Transformation durch ein *Deep-Copy* des Modells realisiert ist. Wichtig dabei ist, dass die erstellten Instanzen, wie mit der *cached*-Direktive, automatisch in einem Cache abgelegt werden. Somit werden gleiche Instanzen nur einmal erstellt. Soll zum Beispiel über eine Referenz eine Instanz erneut erstellt werden, so wird die bereits existierende Instanz zurückgeliefert.

```
1 create Model toModel(Model m):
2   this.types.addAll(m.types.typeSelect(DataType).toType()) ->
3   this.types.addAll(m.types.typeSelect(Entity).toType());
4
5 create DataType toType(DataType t):
6   this.setName(t.name + "_m2m");
7
8 create Entity toType(Entity e):
9   this.setName(e.name + "_m2m") ->
10  this.fields.addAll(e.fields.typeSelect(Attribute).toField()) ->
11  this.fields.addAll(e.fields.typeSelect(Relationship).toField());
12
13 create Attribute toField(Attribute a) :
14  this.setName(a.name + "_m2m") ->
15  this.setType(a.type);
16
17 create Relationship toField(Relationship r) :
18  this.setName(r.name + "_m2m") ->
19  this.setBackrefname(r.backrefname) ->
20  this.setMultiplicity(r.multiplicity) ->
21  this.setType(r.type);
```

Listing 20: M2M-Transformation mit Xtend

Die Modelltransformation kann dann z.B. über die im vorigen Abschnitt gezeigte *XtendComponent* umsetzen. Es wird also durch den Workflow gesteuert, wann eine M2M-Transformation durchgeführt, welches Modell übergeben und was mit dem Ergebnis passieren wird.

3.6 Check

Die Sprache *Check* dient dazu, notwendige Bedingungen an ein Modell zu formulieren und zu überprüfen. Nach Stahl et al. (vgl. [SVEH07, S. 60]) sind diese *Constraints* Bestandteil des Metamodells und belegen, wann ein Modell konform zu seinem Metamodell ist. Damit möglichst früh erkannt wird, dass eine Bedingung verletzt ist, sollten die benutzten Werkzeuge bereits Kenntnisse über die Constraints haben. So kann direkt bei der Entstehung des Fehlers darauf hingewiesen werden. Das *Check*-Framework arbeitet wie die anderen Sprachen Xtend und Xpand mit dem Typ- und Expression-System von openArchitectureWare. So lassen sich schnell - ohne neuen Lernaufwand - Bedingungen für ein Metamodell formulieren (vgl. [EFH⁺08, S. 69]).

```
1 // import the model
2 import MyDsl;
3
4 // import my extensions
5 extension fhw::sd::Extensions;
6
7 // Check uniqueness of identifiers
8 context Entity ERROR "Name of entity must be unique":
9     allElements().typeSelect(Entity).select(e|e.name == this.name).size
10        == 1;
11
12 // Check nothing
13 context Entity WARNING "I am always valid" :
14     true;
```

Listing 21: Check-Constraint Entity

In Listing 21 ist eine vollständige Check Datei zu sehen. Es werden zunächst das Metamodell, sowie eine Extension (siehe Abschnitt 3.5) geladen. Darauf folgen die eigentlichen Constraints. Nach dem Schlüsselwort `context` steht der Name des Metatypen, für den diese Bedingung gilt. Die Angabe von `ERROR` oder `WARNING` signalisiert, um welchen Fehlergrad es sich bei der Bedingung handelt. Die auszugebende Meldung wird danach in Hochkommata notiert. Die eigentliche Check-Bedingung steht jeweils in der zweiten Zeile. Wird dieser boolesche Ausdruck zu *Falsch* ausgewertet, so wird ein Fehler (respektive Warnung) ausgelöst. Ist der Ausdruck hingegen *Wahr*, so gilt das Modell im Hinblick auf diese Bedingung als valide (siehe auch zweite Regel). In der ersten Bedingung wird die Eindeutigkeit der Entitynamen überprüft. Es werden aus allen Elementen des Modells, diejenigen herausgefiltert, die den Metatypen *Entity* haben

`allElement().typeSelect(Entity)`. Aus dieser Untermenge von Elementen werden anschließend jene herausgefiltert, die den gleichen Namen haben wie die zu validierende Entity *this*. Hat diese Menge mehr als ein Element, so wird ein Fehler ausgelöst. Wird während der Validierung ein Fehler oder eine Warnung erkannt, so werden diese über die aus dem Workflow bekannten *Issues* weitergeleitet, so dass die folgenden Komponenten Zugriff darauf haben.

Überprüfung beim Erstellen Die Check-Constraints können zu verschiedenen Zeitpunkten aufgerufen werden. Wenn ein Metamodell z.B. mit Xtext erstellt wurde, kann damit auch ein Eclipse-Texteditor generiert werden. Dieser kann sofort bei der Eingabe solche Constraints auswerten und dem Entwickler die genaue Fehlerstelle im Modell markieren. Existiert das Metamodell z.B. als EMF Ecore, so kann auch hier der generierte Editor (siehe Abbildung 8) die in Check formulierten Bedingungen auswerten und direkt darauf hinweisen (vgl. [SVEH07, S. 61]).

Überprüfung bei der Generierung Da in einem Projekt nicht immer sichergestellt ist, dass die Editoren die Modelle bereits ausreichend validiert haben, sollte die Validierung vor der Generierung wiederholt und abschließend durchgeführt werden. Bevor in einem Workflow also ein Generator-Aufruf erfolgt, wird eine Validierungs-Komponente eingefügt. Somit ist sichergestellt, dass der Parser immer gültige Modelle erhält (vgl. [SVEH07, S. 61]).

Wie oben gesehen, gibt es verschiedene Zeitpunkte, zu denen ein Modell validiert werden kann und sollte. Um dem Entwicklerteam hier die Arbeit zu erleichtern, ist es sinnvoll die Bedingungen nur an einer Stelle zu formulieren. openArchitectureWare erfüllt mit *Check* genau diese Vorgabe, da sowohl die generierten Editoren, als auch der Workflow auf dieselben Constraints zugreifen. Somit ist eine Validierung bei der Eingabe und im Build-Prozess auf einheitlicher Basis möglich (vgl. [SVEH07, S. 61]).

3.7 Xtext

Xtext ist ein Framework zur Erstellung von textuellen DSLs. Die Beschreibung der DSL erfolgt hierbei in einer verkürzten Notation der erweiterten Backus-Naur-Form. Das Xtext-Framework erstellt aus dieser Grammatik das Metamodell und einen Parser zum Einlesen der Modelle. Dieser Parser kann dann in einem Generator genutzt werden.

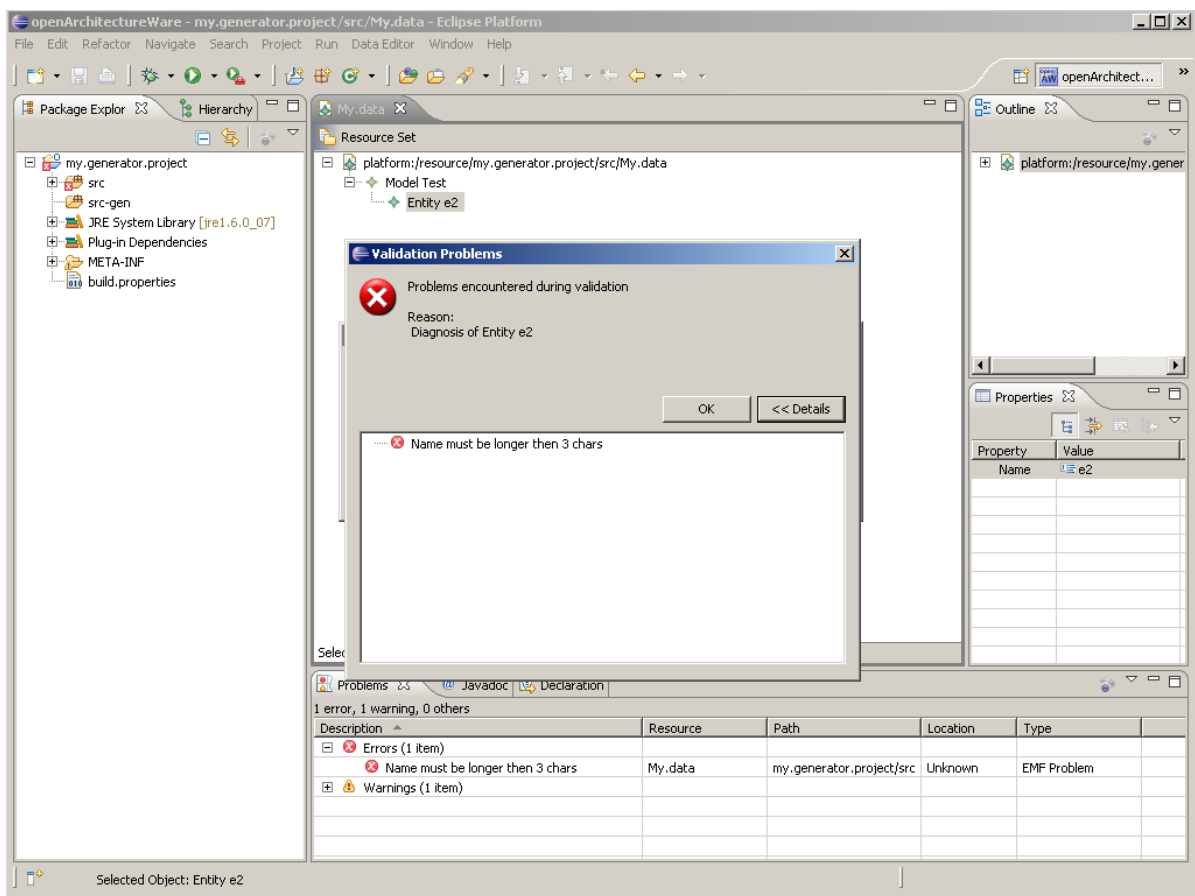


Abbildung 8: Validierung generierter EMF-Editor

Außerdem kann ein Eclipse Texteditor Plugin generiert werden, mit dem sich die Modelle beschreiben lassen (siehe Abbildung 9, (vgl. [EFH⁺08, S. 103])).

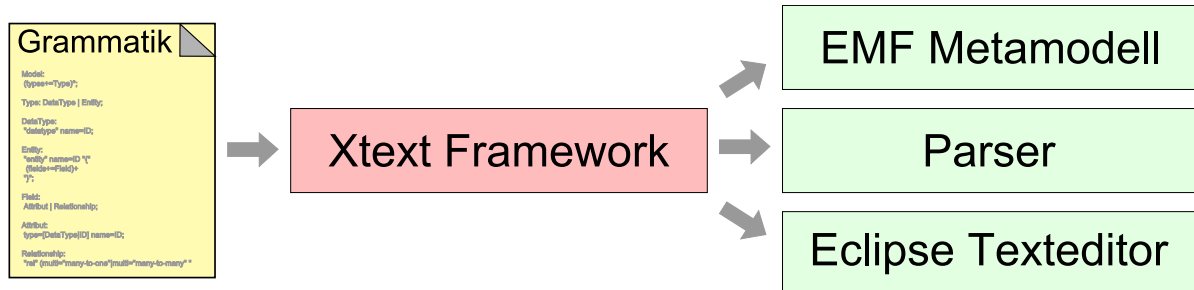


Abbildung 9: Überblick Xtext-Framework

Die Grammatik Wie oben angedeutet wird die Grammatik in einer Backus-Naur-Form beschrieben. Aus diesen Regeln wird aber nicht nur die konkrete Syntax für die Notation des Modells abgeleitet, sondern auch die abstrakte Syntax des Metamodells festgelegt (vgl. [SVEH07, S. 104f]).

```

1 ...
2
3 Entity:
4   "entity" name=ID "{"
5     (fields+=Field)+
6   "}";
7
8 ...

```

Listing 22: Beispiel Entität

In Listing 22 ist ein Ausschnitt aus einer Grammatik zu sehen. Es wird eine Regel mit dem Namen *Entity* definiert. Die Regelbeschreibung folgt dem Namen in Form von *Tokens* und wird mit einem ; abgeschlossen. In der Regelbeschreibung lässt sich die konkrete Syntax erkennen. Es gibt ein Schlüsselwort *entity*, nach dem ein *Identifier* stehen muss. In Klammern eingebettet folgt dann die Definition der Entität. Für das Metamodell bedeutet diese Regel, dass es einen Metatypen mit dem Namen *Entity* gibt und dieser zwei Felder besitzt. Ein einfaches Attribut *name* mit dem Typen *ID* und eine Referenz *fields* auf einen anderen Metatypen *Field*. Das += weist der Referenz den Typen *Liste von Field* zu und das + sorgt dafür, dass die Liste nicht leer sein kann (Vgl.

auch Abbildung 10). In dem Beispiel wird ein von Xtext vorgegebener built-in Token *ID* genutzt. *ID* ist eine vordefinierte Regel der Form `('a-zA-Z_','a-zA-Z_0-9')*` (vgl. [EFH⁺08, S. 104]).

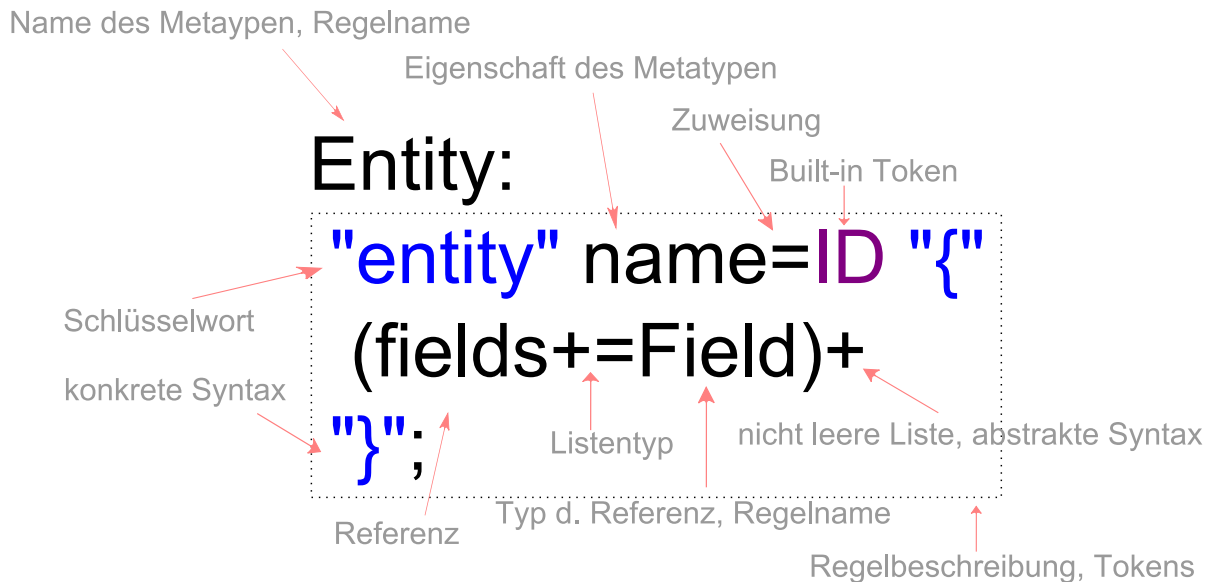


Abbildung 10: Grammatikregel

Die Notation einer *Entity* ist in Listing 23 dargestellt.

```

1 entity Adresse {
2   String strasse
3   String ort
4   String hausnummer
5   Integer plz
6 }

```

Listing 23: Notation einer Entität

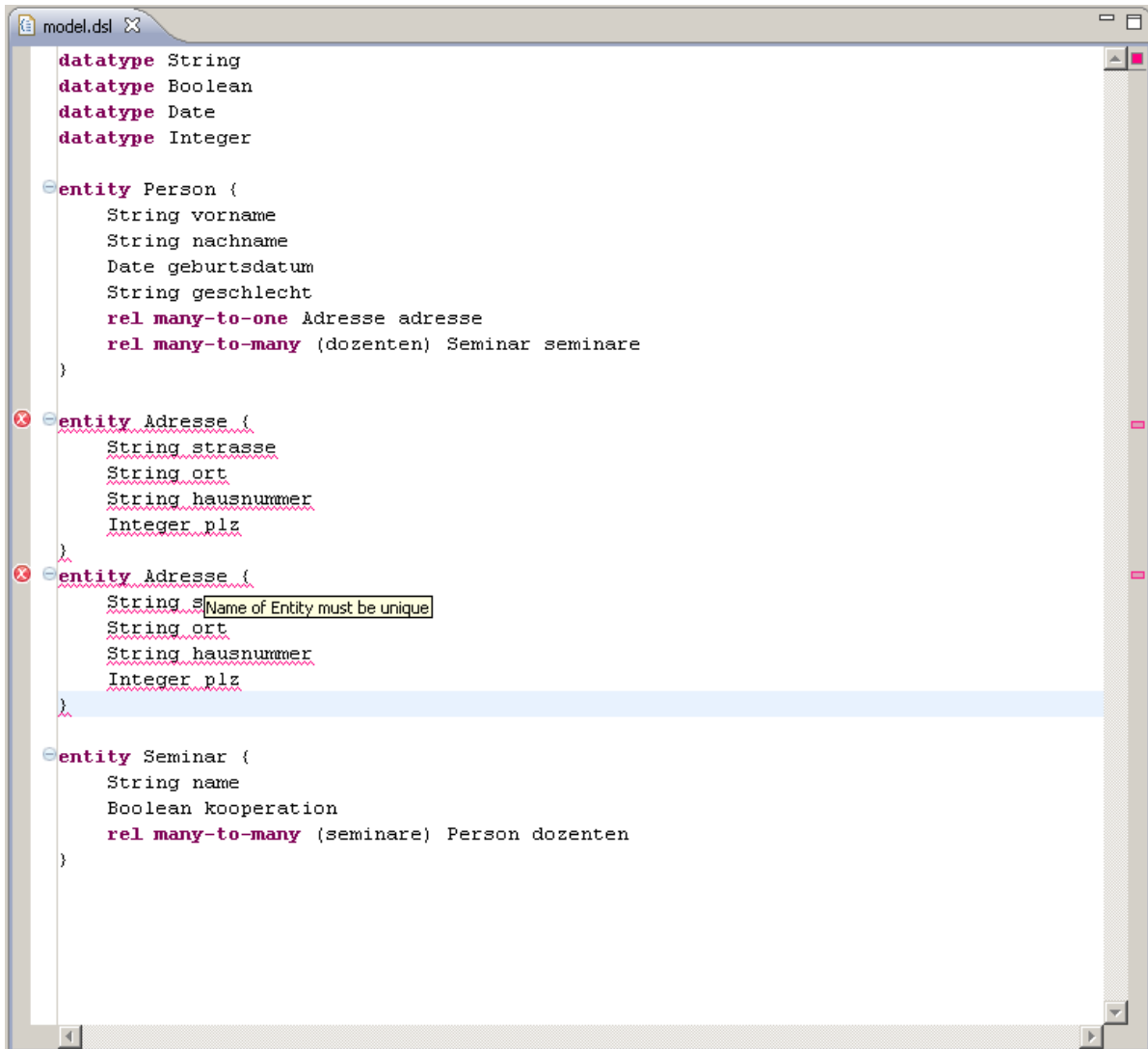
Zuweisungen Bei der Erstellung des Metamodells spielen die Zuweisungen in der Grammatik eine große Rolle. Sowohl die Art der Zuweisung, als auch die rechte Seite haben Einfluss auf den Typen, der letztendlich dem Feld im Metamodell zugewiesen wird. Es gibt folgende Arten von Zuweisungen:

- = der Typ wird durch die rechte Seite bestimmt

- `?=` der Typ ist Boolean
- `+=` der Typ ist eine Liste vom Typ der rechten Seite

Der Editor Das Xtext-Framework kann aus der Grammatik einen Editor für die Eclipse Entwicklungsumgebung generieren (siehe Abbildung 11). Dieser Editor beherrscht dabei schon die Funktionen aus der unten stehenden Liste (vgl. [Völ07, S. 3]). Die Validierung erfolgt hierbei mit den *openArchitectureWare* eigenen Mitteln über die Validierungssprache Check (siehe Abschnitt 3.6).

- Syntax Highlighting
- Code Completion
- Validierung
- Code Folding
- Outline View



```
model.dsl X
datatype String
datatype Boolean
datatype Date
datatype Integer

entity Person {
  String vorname
  String nachname
  Date geburtsdatum
  String geschlecht
  rel many-to-one Adresse adresse
  rel many-to-many (dozenten) Seminar seminare
}

entity Adresse {
  String strasse
  String ort
  String hausnummer
  Integer plz
}

entity Adresse {
  String S[Name of Entity must be unique]
  String ort
  String hausnummer
  Integer plz
}

entity Seminar {
  String name
  Boolean kooperation
  rel many-to-many (seminare) Person dozenten
}
```

Abbildung 11: Beispiel Editor

4 Beispiel – Stammdatenpflege mit openArchitectureWare

Nachdem in den vorhergehenden Kapiteln bereits die modellgetriebene Softwareentwicklung im Allgemeinen sowie die Nutzung des Frameworks openArchitectureWare im Besonderen vorgestellt wurde, soll nun eine Beispielanwendung mit Hilfe von openArchitectureWare umgesetzt werden. Nach der Identifizierung der zu erzeugenden Artefakte in den ersten beiden Abschnitten wird anschließend die Modellierung sowie die Entwicklung von geeigneten Generatoren beschrieben. Abschließend werden mögliche Weiterentwicklungen des Beispiels vorgestellt.

4.1 Zweck und Ziel des Beispiels

Zweck dieses Beispiels ist die Demonstration der prinzipiellen Vorgehensweise und des Zusammenspiels der einzelnen Framework-Komponenten bei der Entwicklung mit openArchitectureWare. Ziel des Beispiels ist die Erstellung einer Stammdatenpflege für eine Seminarverwaltung, deren vereinfachte Datenstruktur in Abbildung 12 angegeben ist. In der Seminarverwaltung sind *Personen* gespeichert, von denen persönliche Angaben wie *Vorname*, *Nachname*, *Geburtsdatum* sowie das Geschlecht erfasst sind. Einer Person ist jeweils eine *Adresse* zugeordnet. Des Weiteren werden *Seminare* verwaltet, die eine *Na-men* haben und optional als *Kooperation* durchgeführt werden. Einem Seminar können beliebig viele *Dozenten* zugeordnet werden.

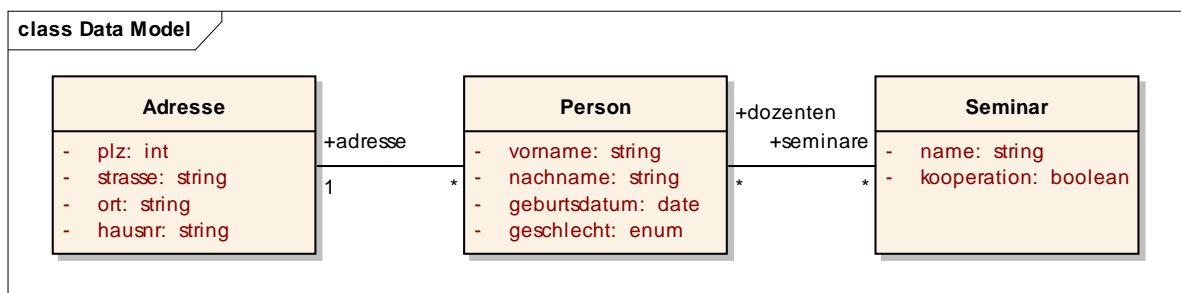


Abbildung 12: Beispieldatenmodell Seminarverwaltung

Eine Stammdatenpflege zeichnet sich in der Regel dadurch aus, dass ihre Struktur weitgehend von den zu bearbeitenden Daten und damit von dem zugrundeliegenden Datenmodell bestimmt werden. Spezielles Verhalten sowie besondere fachliche Logik

werden hingegen nur selten benötigt. Dadurch eignet sich ein modellgetriebenes Vorgehen insbesondere aus den folgenden Gründen:

- Da das zugrundeliegende Datenmodell große Anteile der Anwendung bestimmt, sollte folglich auch der überwiegende Teil der Anwendung (evtl. in mehreren Schritten) aus einem Datenmodell zu generieren sein.
- Durch die klar abgegrenzte Funktionalität wird es möglich, in der Anwendung ein Standardverhalten zu realisieren. Lediglich davon abweichendes Verhalten muss generiert oder programmiert werden.

Der Einsatz von Modellen dient folglich der Abstraktion von der fachlichen Domäne des zugrundeliegenden Datenmodells. Hinsichtlich der aus den Modellen zu erzeugenden Funktionalität werden die folgenden Anforderungen an die Anwendung gestellt:

- Erstellung einer Eingabemaske pro Entitätstyp
- Möglichkeit zum Erzeugen, Verändern und Löschen von Entitäten
- Möglichkeit zum Erzeugen und Löschen von Beziehungen
- Nutzung selbstdefinierter Aufzählungstypen
(z. B. *Geschlecht* = {weiblich, maennlich})

Die Implementierung des Beispiels basiert auf einer Stammdatenpflege von Peemöller [Pee07], die im folgenden Abschnitt beschrieben wird. Diese Anwendung wurde insbesondere deshalb ausgewählt, weil sie das oben genannte Standardverhalten vorgibt. Dies führt dazu, dass der Fokus zunächst auf das Datenmodell gelegt werden kann und die Anwendung ohne manuelle Implementierung lauffähig ist. Die Realisierung von abweichendem Verhalten wird in diesem Beispiel nicht adressiert, jedoch in den möglichen Weiterentwicklungen aufgegriffen.

4.2 Modellgesteuerte Stammdatenpflege

Die bestehende Anwendung zur Stammdatenpflege ist als *modellgesteuerte* Anwendung realisiert, in der von dem fachlich umzusetzenden Datenmodell abstrahiert wird. So sind diejenigen Softwareanteile, die unabhängig von den zu bearbeitenden Daten sind, bereits vollständig implementiert. Dies betrifft beispielsweise die Konfiguration der verwendeten

Frameworks sowie das grundlegende Architekturgerüst. Die fachlichen Anteile hingegen werden durch Modelle beschrieben und zur Laufzeit von der Anwendung interpretiert. Somit wird das Verhalten der Anwendung zur Laufzeit von den interpretierten Modellen gesteuert, was Teurich-Wagner [TW04] folgend als modellgesteuert bezeichnet werden soll.

Architektur Die Anwendung ist als klassische 3-Schichten-Architektur realisiert (vgl. Abbildung 13):

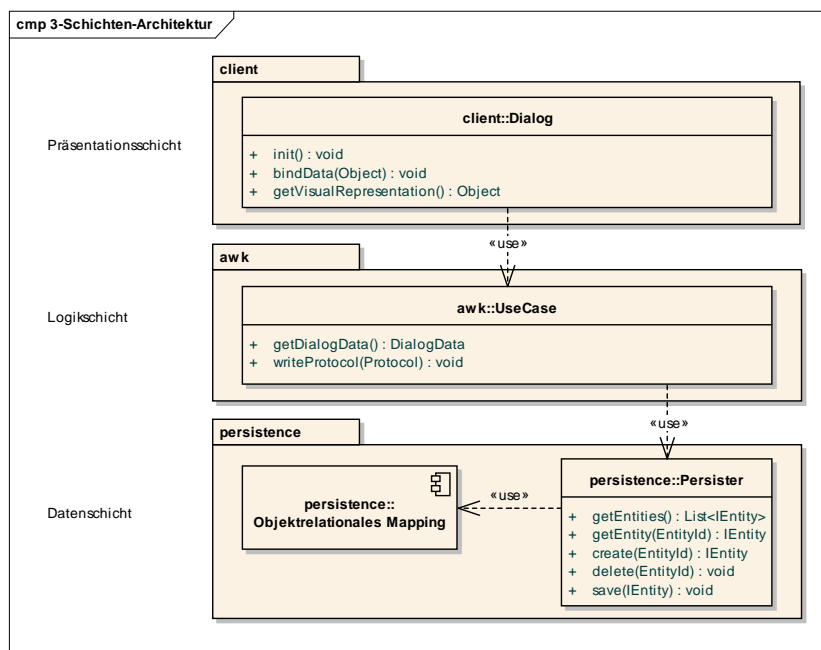


Abbildung 13: 3-Schichten-Architektur der Stammdatenpflege

- Die *Persistenzschicht* ist zuständig für die persistente Datenhaltung und bietet elementare Zugriffsmethoden auf die Entitäten. Da von den konkreten Entitätstypen abstrahiert wird, wird eine generische Schnittstelle angeboten. Für die Persistierung greift die Anwendung auf *Hibernate* [4] als OR-Mapper zurück.
- Die *Logikschicht* enthält die notwendige Geschäftslogik der Anwendung. Aufgrund der überschaubaren Funktionalität der Anwendung besteht die Logik im Wesentlichen aus dem Bereitstellen der Daten zur Anzeige (`getDialogData`) sowie zum Persistieren der getätigten Änderungen (`writeProtocol`).

- Die *Präsentationsschicht* ist zuständig für die Visualisierung und Benutzersteuerung und stellt somit die Schnittstelle zum Benutzer dar. Für die Erzeugung der Oberfläche sowie die Benutzersteuerung wird die *Client Utilities & Framework-Bibliothek* (CUF) [5] verwendet.

Modellsteuerung Um die Fachlichkeit der resultierenden Anwendung zu beschreiben, werden vier unterschiedliche Modelltypen verwendet (vgl. Abbildung 14). Sämtliche Modelle sind als XML-Dokumente realisiert und werden zur Laufzeit eingelesen und verarbeitet.

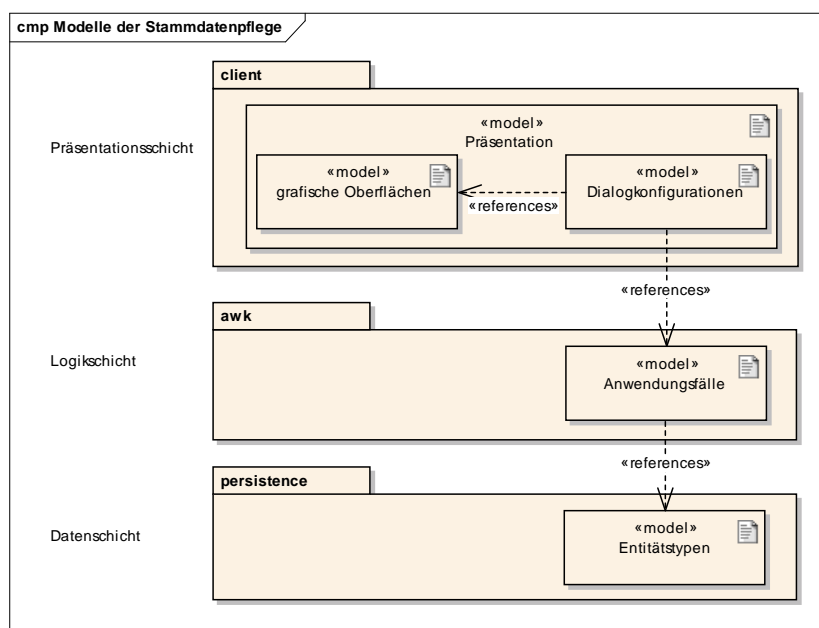


Abbildung 14: Modelle in der Stammdatenpflege

- *Entitätstypen*: Dieses Modell beschreibt die zu bearbeitenden Entitätstypen mittels einer Hibernate-Mappingkonfiguration. Beziehungstypen werden hierbei als Eigenschaften der Entitätstypen angesehen.
- *Anwendungsfälle*: Die Anwendungsfälle stellen einzelne Szenarien der Entitätspflege dar und bündeln die zur Pflege benötigten Informationen (Entitäten und Beziehungen). Listing 24 stellt einen Anwendungsfall für die Personenpflege dar, bei dem die Adressen und Seminare ebenfalls zur Verfügung stehen.

```
1 <usecase name="PersonUC">
2   <query entity-name="Person">
3     <extension path="adresse"/>
4     <extension path="seminare"/>
5   </query>
6 </usecase>
```

Listing 24: Usecase für die Bearbeitung von Personen

- *Präsentation*: Das Präsentationsmodell teilt sich in zwei Teilmodelle auf. Die *Dialogdefinition* modelliert die grafische Oberfläche sowie die Benutzersteuerung und verwendet eine CUF-eigene XML-Sprache. Die *Dialogkonfiguration* verknüpft eine grafische Oberfläche, einen Anwendungsfall und einen entsprechenden Controller miteinander. Listing 25 stellt eine Dialogkonfiguration dar, mit der Personen bearbeitet werden können.

```
1 <dialog name="PersonDialog">
2   <usecase name="PersonUC"/>
3   <presentation path="fhw/sd/client/dialogdefinition/Person.xml"/>
4   <controller
5     class="fhw.sd.client.generic.GenericDialogController"/>
6 </dialog>
```

Listing 25: Dialogkonfiguration für die Bearbeitung von Personen

Somit sind für eine vollständige fachliche Parametrierung der Anwendung die folgenden vier XML-Modelle zu generieren:

- Entitätstypenmodell als Hibernate-Mapping
- Anwendungsfallmodell
- Dialogdefinition als CUF-Modell
- Dialogkonfiguration

Die in den Anforderungen genannte Möglichkeit, eigene Aufzählungstypen zu definieren, ist in der Anwendung nicht über Modelle möglich. Hierzu ist zusätzlich eine Java-Klasse zu generieren, die eine entsprechende Zuordnung auf Hibernate-spezifische Persistenzaspekte realisiert.

4.3 Umsetzung mit openArchitectureWare

4.3.1 Entwicklung des Metamodells

Nachdem in dem vorangegangenen Abschnitt die Artefakte analysiert wurden, die später zu generieren sein werden, folgt nun die Entwicklung des Metamodells. Das Metamodell soll es erlauben, die folgenden Bestandteile zu modellieren:

- *Aufzählungstypen* bestehen aus einer Menge von Werten, denen jeweils ein beschreibender String zugeordnet wird
- *Entitätstypen* bestehen aus einer Menge von Eigenschaften: *Beziehungstypen*, *einfachen Attributen* und *Attributen mit selbstdefinierten Aufzählungstypen*
- *Beziehungstypen* beziehen sich auf einen Entitätstyp, haben eine Kardinalität (*ToOne* oder *ToMany*) und einen inversen Beziehungstyp
- *Einfache Attribute* haben einen primitiven Datentyp (z. B. Integer, String)
- *Attribute mit selbstdefinierten Aufzählungstypen* haben einen Aufzählungstyp als Datentyp

Für die spätere Modellierung soll eine textuelle DSL verwendet werden, da damit eine schnelle Modellerstellung gewährleistet ist, bei der zusätzlich die Möglichkeit des Kopierens ähnlicher Modellteile möglich ist.

In Listing 26 ist ein *Xtext*-Modell der Sprache SDDSL angegeben, die die oben genannten Anforderungen erfüllt. Die Sprachelemente leiten sich direkt aus den Anforderungen ab und sind somit weitgehend selbsterklärend. Für die Vorgabe von Datentypen für einfache Attribute wurde in den Zeilen 23–28 ein Aufzählungstyp definiert. Da einfache und selbstdefinierte Datentypen von Attributen teilweise unterschiedlich behandelt werden müssen, werden diese durch unterschiedliche Klassen repräsentiert.

```
1 // Root-Element
2 Model: (types+=Type)*;
3
4 // Mögliche Typen des Modells
5 Type: EnumType | Entity;
6
7 // Selbstdefinierter Aufzählungstyp
8 EnumType :
```

```
9     "enum" name=ID "{"
10     (values+=Value) ("," values+=Value)*
11     "}";
12
13 // Einzelner Aufzählungswert
14 Value : name=ID "(" description=STRING ")";
15
16 // Entitätstyp
17 Entity:
18     "entity" name=ID "{"
19     (fields+=Field)+
20     "}";
21
22 // Vordefinierte Datentypen
23 Enum PredefinedType :
24     big_decimal="BigDecimal" | boolean="Boolean" |
25     byte="Byte" | character="Character" |
26     date="Date" | double="Double" |
27     float="Float" | integer="Integer" |
28     long="Long" | string="String" ;
29
30 // Feld eines Entitätstyps: Attributtyp oder Relationshiptyp
31 Field: Attribute | Relationship;
32
33 // Attribute
34 Attribute : NormalAttribute | EnumAttribute;
35 NormalAttribute: type=PredefinedType name=ID " ";
36 EnumAttribute : type=[EnumType] name=ID " ";
37
38 // Relationships
39 Relationship: type=[Entity] (toMany?="[ ]"? name=ID "<->"
    backRel=[Relationship] " ");
```

Listing 26: Xtext-Modell für das Datenmetamodell

Somit sind die abstrakte und konkrete Syntax des Metamodells spezifiziert. Durch die Verwendung von Kreuzreferenzen (eckige Klammern in den Zeilen 36 und 39) wird zudem sichergestellt, dass die referenzierten Aufzählungs-, Entitäts- oder Beziehungstypen existieren. Diese Bedingungen bilden bereits einen Teil der zu spezifizierenden Kontextbedingungen. Des Weiteren sind noch folgende Bedingungen zu erfüllen, um ein in der Anwendung valides Modell zu erhalten:

- Der Name eines Aufzählungs- bzw. Entitätstyps muss eindeutig sein
- Der Name eines Aufzählungswertes muss eindeutig sein
- Der Name einer Eigenschaft eines Entitätstypen muss eindeutig sein

- Der inverse Beziehungstyp muss den richtigen Entitätstyp referenzieren
- Der inverse Beziehungstyp muss den ursprünglichen Beziehungstyp selbst als invers definieren

Zur Überprüfung dieser Kontextbedingungen wird die Sprache *Check* verwendet. Listing 27 definiert die oben angegebenen Kontextbedingungen als Check-Constraints.

```
1 import sddsl;
2 extension fhw::sd::Extensions;
3
4 /* Eindeutiger Name von Aufzählungs-/ Entitätstypen */
5 context Type ERROR "Name of Type must be unique : " + this.name :
6   allElements().typeSelect(Type).select(e|e.name == this.name).size ==
7     1;
8
9 /* Eindeutige Elemente eines Aufzählungstyps */
10 context Value ERROR "EnumType values must be unique : " + this.name :
11   allElements().typeSelect(EnumType).select(e|e.values.
12     contains(this)).values.select(e|e.name == this.name).size == 1;
13
14 /* Eindeutiger Name von Eigenschaften */
15 context Field ERROR "Name of Field of an Entity must be unique : " +
16   this.name :
17   allElements().typeSelect(Entity).select(e|e.fields.
18     contains(this)).fields.select(e|e.name == this.name).size == 1;
19
20 /* Inverse Relationships muss richtigen Entitätstyp referenzieren */
21 context Relationship ERROR "Inverse Relationship '" +
22   this.backRel.name + "' refers to a different entity '" +
23   this.backRel.type.name + "'" :
24   this.backRel.type.fields.contains(this);
25
26 /* Inverse Relationship muss this als inverse Relationship haben */
27 context Relationship ERROR "Inverse Relationship '" +
28   this.backRel.name + "' refers to a different inverse Relationship
29   '" + this.backRel.backRel.name + "'" :
30   this.backRel.backRel == this;
```

Listing 27: Check-Constraints für das Datenmetamodell

Nachdem nun das Metamodell als Xtext-Modell mit Check-Constraints spezifiziert ist, kann durch das openArchitectureWare-Framework ein entsprechender Editor generiert werden, der die Sprache SDDSL unterstützt. Abbildung 15 zeigt den erzeugten Editor bei der Bearbeitung eines Beispielmodells. Hervorzuheben ist die zur Verfügung gestellte Funktionalität wie Syntax-Highlighting (Schlüsselwörter wie `enum`), Code Completion

(Vorschlagssystem; hier `entity`, `enum`) und die Überprüfung von Kontextbedingungen (Problems-View). Während der erste Fehler durch die Verwendung von Kreuzreferenzen gefunden wird, entsprechen die restlichen vier Fehler Bedingungen, die soeben mit Hilfe von Check formuliert wurden.

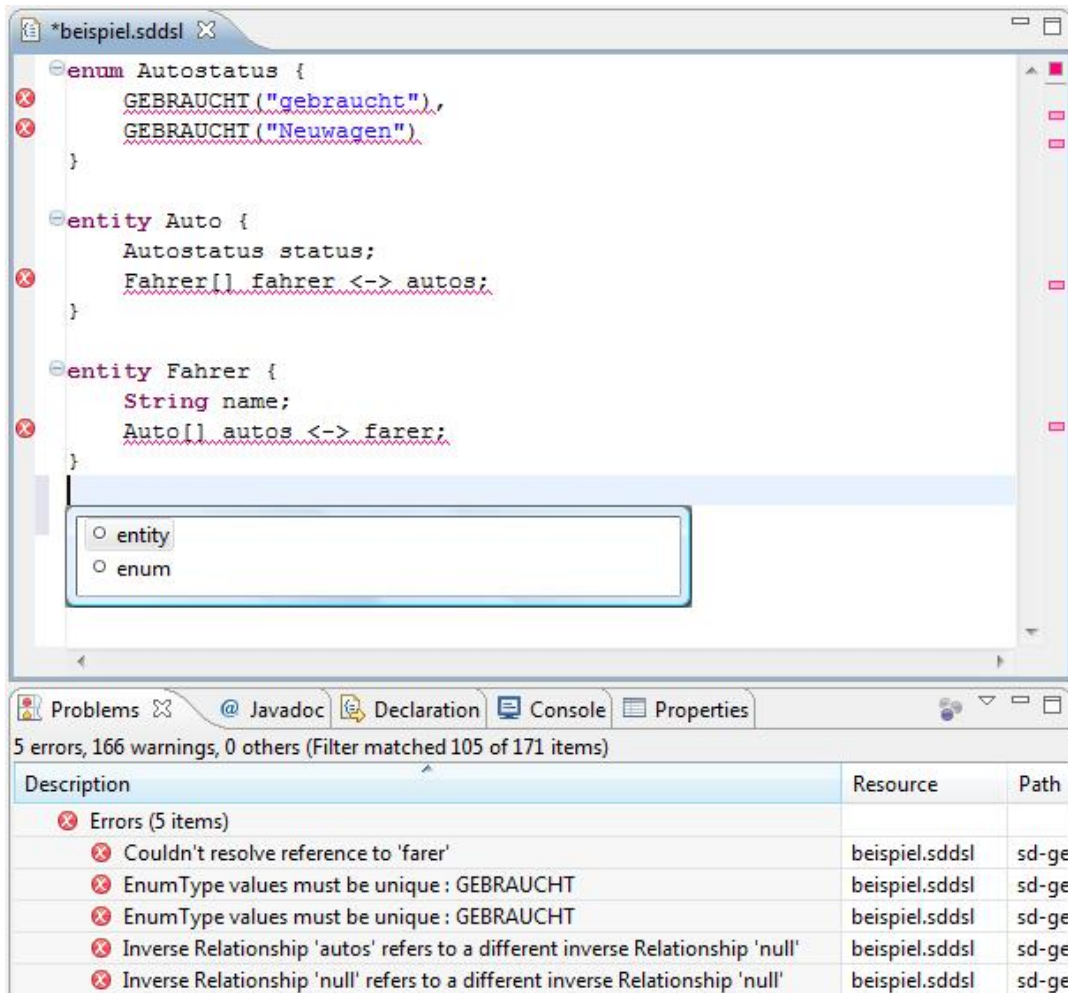


Abbildung 15: Generierter Editor für die Sprache SDDSL

4.3.2 Entwicklung des Generators

Für die Entwicklung des Generators kommen die Template-Sprache Xpand für die Coodeerzeugung (Javaklassen sowie XML-Dokumente) sowie Xtend für eine Modellmodifikation zum Einsatz. Aus dem Modell sind die folgenden Artefakte zu generieren:

- XML-Dokumente
 - Hibernate-Mapping für die Persistenzschicht
 - Anwendungsfallmodell
 - Dialogkonfigurationsmodell
 - Dialogdefinition für jeden Entitätstyp
- Java-Quellcode
 - Java-Klasse für jeden definierten Aufzählungstyp
 - Standalone-Klasse für jeden Entitätstyp, der den entsprechenden Dialog startet

Einen Überblick über die verwendeten Templates bietet Abbildung 16. Die einzelnen Templates traversieren das vom Workflow instanziierte Modell und erzeugen entsprechend der Expansionsregeln spezifischen Code. Der Aufruf der Templates erfolgt dabei

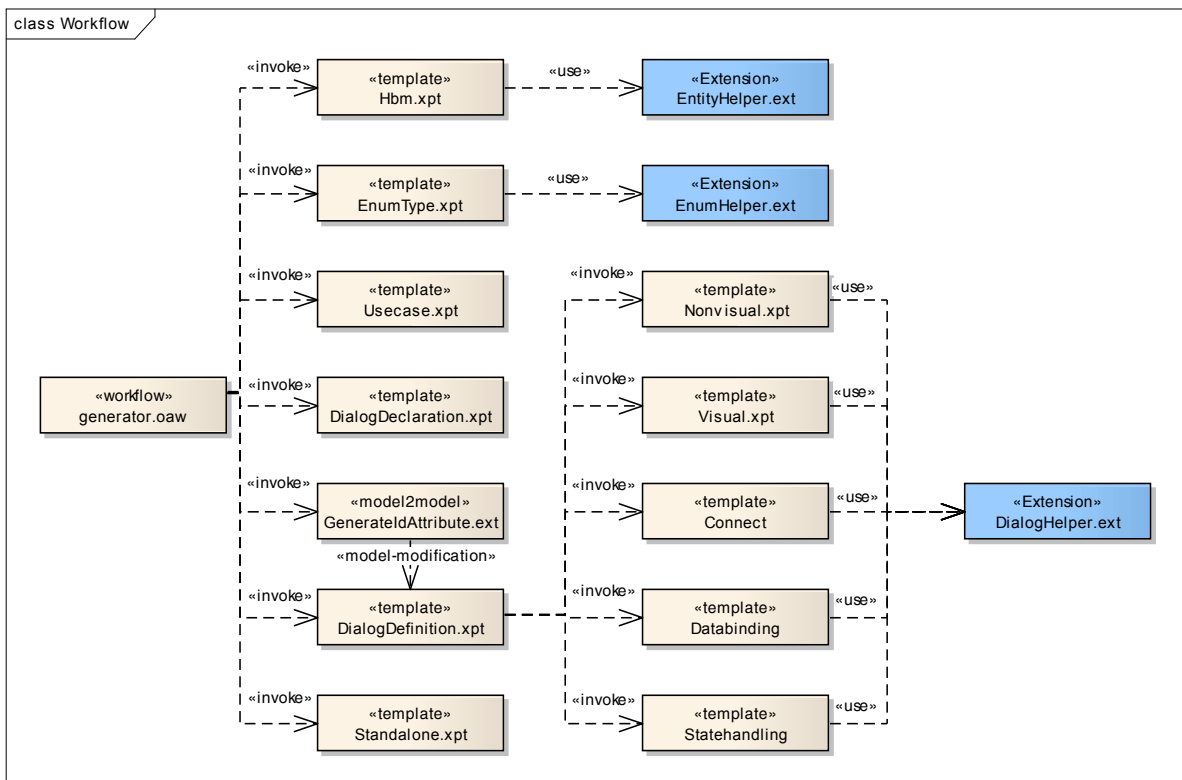


Abbildung 16: Workflow-Struktur

von oben nach unten. Nach dem Aufruf des Templates „DialogDeclaration.xpt“ wird zudem eine Modellmodifikation durchgeführt. An dieser Stelle wird in dem Modell zu jedem Entitätstyp ein technisches Attribut „id“ hinzugefügt (vgl. Listing 28), um dieses in der Oberfläche anzeigen zu können. Dieses Attribut ist in den Hibernate-Mappings bereits automatisch als Schlüsselattribut vorgesehen.

```
1 import sddsl;
2
3 transform(Model this) :
4     types.typeSelect(Entity).addIdAttribute();
5
6 addIdAttribute(Entity this) :
7     fields.add(idAttribute(this));
8
9 create NormalAttribute idAttribute(Entity e) :
10    setName("id") -> setType(PredefinedType::integer);
```

Listing 28: Modell-Modifikation: Hinzufügen eines ID-Attributs

Anschließend wird die Oberfläche über das Template „DialogDefinition.xpt“ erzeugt, das auf fünf Subtemplates aufgeteilt ist. Die Erzeugung von String-Identifikatoren, die von den Subtemplates gemeinsam genutzt werden, ist in die Extension „DialogHelper.ext“ ausgegliedert.

Aus der Generatorkaufteilung wird ersichtlich, wie einzelne Domänen durch das modellgetriebene Vorgehen getrennt werden können:

- Die *fachliche* Domäne der zu bearbeitenden Daten wird durch das Modell der Seminarverwaltung abgebildet. Dieses Modell erfordert keine weiteren Kenntnisse der in der Anwendung verwendeten Frameworks oder anderer technischen Aspekte und kann beispielsweise von einem fachlichen Berater erstellt werden.
- Die in der Anwendung vorhandenen *technischen* Domänen wie die Persistierung mit Hibernate oder die GUI-Definition mit CUF werden durch entsprechende Generatoren gekapselt. Die Generatoren können von entsprechenden technischen Experten erstellt und gewartet werden, ohne dass auf fachliche Belange Rücksicht genommen werden muss.

Als Beispiel für eine technische Domäne ist in Listing 29 ein Teil des Templates für die Erzeugung der Hibernate-Mappings angegeben. Die bei der Metamodellierung einge-

fürhte Unterscheidung von Attributen in *NormalAttribute* und *EnumAttribute* erlaubt durch parametrische Polymorphie eine saubere Trennung der jeweiligen Subtemplates.

```

1 < IMPORT sddsl >
2 < EXTENSION fhw::sd::hbm::entityHelper >
3
4 < DEFINE Root FOR Model >
5   < EXPAND HbmMappingFile FOREACH types.typeSelect(Entity) >
6 < ENDDDEFINE >
7
8 < DEFINE HbmMappingFile FOR Entity >
9 < FILE name+".hbm.xml" ><?xml version="1.0" encoding="UTF-8"?>
10 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
    3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
11 <hibernate-mapping>
12   <class entity-name="« name »" table="« tableName() »">
13     <id name="id" column="« getId() »" type="integer">
14       <generator class="increment" />
15     </id>
16     < EXPAND Attribute FOREACH fields.typeSelect(Attribute) >
17     < EXPAND Relationship(this) FOREACH
        fields.typeSelect(Relationship) >
18   </class>
19 </hibernate-mapping>
20 < ENDFILE >
21 < ENDDDEFINE >
22
23 < DEFINE Attribute FOR NormalAttribute >
24   <property column="« name.toUpperCase() »" name="« name »" type="«
    type.toString() »"/>
25 < ENDDDEFINE >
26
27 < DEFINE Attribute FOR EnumAttribute >
28   <property name="« name »">
29     <column name="« name.toUpperCase() »" default="«
    type.values.first().name.toUpperCase() »" />
30   <type name="fhw.sd.persistence.type.enums.EnumUserType">
31     <param name="class">fhw.sd.common.type.enums.«
    name.toFirstUpper() »Typ</param>
32   </type>
33   </property>
34 < ENDDDEFINE >

```

Listing 29: Template zur Erzeugung der Hibernate-Mappings (Ausschnitt)

Auf eine detaillierte Vorstellung der weiteren Templates soll an dieser Stelle verzichtet werden, da diese jeweils spezifische technische Aspekte behandeln und dies für das

grundsätzliche Verständnis nicht essentiell ist. Der interessierte Leser sei an dieser Stelle auf die entsprechende Literatur ([Pee07], [Red07], [sof07]) verwiesen.

4.3.3 Modellierung der Seminarverwaltung

Mit der Definition der domänenspezifischen Sprache SDDSL und der Erstellung geeigneter Generatoren kann nun eine Stammdatenpflege zur Seminarverwaltung generiert werden. Dazu müssen zunächst im generierten Editor die zu bearbeitenden Daten modelliert werden, wie in Listing 30 angegeben.

```
1  enum Geschlecht {
2      WEIBLICH("weiblich"),
3      MAENNLICH("männlich")
4  }
5
6  entity Person {
7      String vorname;
8      String nachname;
9      Date geburtsdatum;
10     Geschlecht geschlecht;
11 }
12 }
13
14 entity Adresse {
15     String strasse;
16     String ort;
17     String hausnummer;
18     Integer plz;
19 }
20
21 entity Seminar {
22     String name;
23     Boolean kooperation;
24 }
```

Listing 30: Datenmodell der Seminarverwaltung als SDDSL-Modell

Anschließend ist ein Workflow zu erstellen, der lediglich den bereits beim Generator erstellten Workflow „generator.oaw“ aufruft und die gewünschten Ausgabeverzeichnisse angibt. In dem in Listing 31 angegebenen Fall werden die Generierungsergebnisse der Projektstruktur der Anwendung folgend auf einzelne Ordner aufgeteilt.

```
1 <workflow>
```

```

2 <component file="fhw/sd/generator.oaw">
3   <modelFile value="model.sddsl"/>
4   <hbmTarget
5     value="../sd-persistence-impl/res-gen/fhw/sd/persistence/">
6   <ucTarget value="../sd-awk-impl/res-gen/fhw/sd/awk/">
7   <dialogConfigTarget value="../sd-client/res-gen/fhw/sd/client/">
8   <dialogDefTarget
9     value="../sd-client/res-gen/fhw/sd/client/dialogdefinition/">
10  <standaloneTarget value="../sd-test/src-gen/fhw/sd/standalone/">
11  <enumTarget
12    value="../sd-common/src-gen/fhw/sd/common/type/enums/">
13 </component>
14 </workflow>

```

Listing 31: Workflow der Seminarverwaltung

Nun können der Workflow ausgeführt und eine der generierten Standalone-Klassen gestartet werden. Abbildung 17 zeigt den soeben erzeugten Dialog für die Pflege der Personendaten. Neben den hier abgebildeten Attributen ist es ebenso möglich, über die entsprechenden Reiter die Beziehungen zu bearbeiten. Bei den Attributen ist zusätzlich, wie in Abbildung 17 bei dem Geburtsdatum ersichtlich, eine einfache Eingabeüberprüfung generiert worden. Lassen sich die Eingaben nicht in den vorgesehenen Typ konvertieren, so wird die Eingabe farblich hervorgehoben.

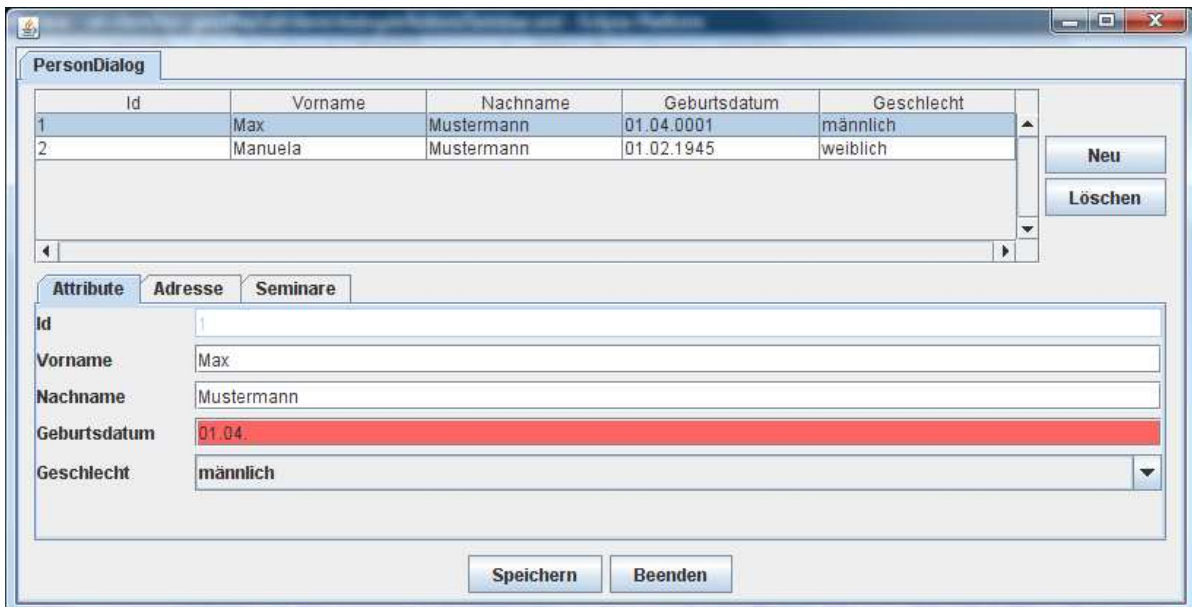


Abbildung 17: Dialog zur Personenpflege

4.4 Mögliche Weiterentwicklungen

Obwohl die in diesem Kapitel erstellte Stammdatenpflege bereits den grundlegenden Funktionsumfang realisiert, sind doch einige stillschweigende Konventionen getroffen worden, die Anlass zu Weiterentwicklungen bieten:

- *Datenbank-Schema:*

Dieses Beispiel basiert auf einer bereits bestehenden Datenbank, deren Struktur der Struktur des Datenmodells entspricht. Diese Konstellation ist in der Realität jedoch sehr unwahrscheinlich. Selbstverständlich ist es möglich, passend zum Datenmodell SQL-Skripte zu generieren, die eine entsprechende Datenbankstruktur realisieren. Sehr viel wahrscheinlicher ist es jedoch, dass die Anwendung auf ein bestehendes Schema aufsetzen soll. Um eine solche Anforderung abdecken zu können, ist das bestehende Modell zu erweitern. Dieses sollte jedoch nicht in einer Erweiterung der Sprache SDDSL resultieren, da die Persistierung einen ganz speziellen Aspekt darstellt. Stattdessen bietet sich die Einführung eines gesonderten Aspektmodells für die Persistenz an. Ein möglicher Ansatz ist die Beschreibung des Mappings von Daten- auf Datenbankstruktur durch Property-Dateien: In der Syntax von Java-Properties könnten von der Konvention abweichende Bezeichner, Längenbeschränkungen oder ähnliche Anforderungen abgebildet werden. Der Erzeugung der Hibernate-Dateien ist dann eine entsprechende Modelltransformation vorzuschalten.

- *Anwendungsfälle und Dialogkonfigurationen:*

Die bestehenden Anwendungsfälle sind stark an der Datenstruktur angelehnt. Hier bietet es sich an, spezifische Anwendungsfälle und dazugehörige Dialogkonfigurationen zu modellieren. Hierfür ließe sich eine separate Sprache modellieren, die diese Aspekte adressiert. Die beiden so entstehenden Modelle könnten dann durch Model-Weaving miteinander verknüpft werden.

- *Vorgegebenes Standard-Verhalten:*

Die erzeugte Anwendung basiert auf einer stark eingeschränkten Menge von Bearbeitungsmöglichkeiten:

- Anlegen und Löschen von Entitäten
- Ändern von Attributwerten

- Anlegen und Löschen von Beziehungen
- Speichern und Beenden

Auch wenn diese Operationen bereits eine große Menge der anfallenden Aufgaben einer Stammdatenpflege abdecken mögen, so werden früher oder später spezifische Funktionen zu realisieren sein. Eine Möglichkeit zur Integration von Verhalten besteht in der Deklaration von Callback-Methoden, die in der Oberfläche an entsprechende Knöpfe gebunden werden. Des Weiteren ist ein entsprechender Controller zu generieren, in dem diese Methoden manuell zu implementieren sind. Dies kann beispielsweise über Protected Regions oder Vererbung geschehen. Eine davon unabhängige Strategie könnte es sein, die oben genannten Basisoperationen als Modellelemente einer neuen Sprache zu realisieren. Damit wäre es möglich, diese Operationen in einem Modell zu komponieren und die Logik erzeugen zu lassen. Beispiele hierfür wäre das Ausfüllen einer neuen Entität mit einer Menge an Vorgabewerten oder kaskadierendes Löschen in der Anwendung. Aufwand und Nutzen dieser Lösung sollten jedoch gewissenhaft gegeneinander abgewogen werden.

- *Komplexität der CUF-Templates:*

Bereits bei der Erstellung der Templates für CUF hat sich gezeigt, dass diese eine hohe Komplexität aufweisen. Der Grund hierfür liegt in der recht geringen Abstraktion von den erzeugten Swing-Elementen und entsprechenden Actions. Hier wäre es sinnvoll, eine entsprechende Cartridge zu realisieren, die die Orchestrierung der Widgets übernimmt.

5 Fazit

5.1 Modellgetriebene Softwareentwicklung

Die MDSD bietet die Möglichkeit, durch die Einführung neuer Abstraktionsebenen und die automatische Erzeugung von Softwareanteilen die Softwareentwicklung zu beschleunigen und die Qualität der Software zu erhöhen (vgl. [HSS05, S. 113]). Durch die Einführung domänenspezifischer Abstraktionen können die Domänen zudem sinnvoll und prägnant beschrieben werden, ohne Implementierungsaspekte berücksichtigen zu müssen. Dies unterstützt insbesondere die Trennung von Fachlichkeit und Technik einer Software: Die Fachlichkeit kann zunächst modelliert werden, bevor sie über Transformationen in die Technik integriert wird. Die Technik wiederum kann unabhängig von der Fachlichkeit entwickelt werden. Des Weiteren kann auch die Technik sinnvoll in einzelne Bestandteile zerlegt werden, was die Trennung von Zuständigkeiten weiter unterstützt. Ferner besteht die Möglichkeit, Spezialwissen dauerhaft in Generatoren abzulegen und somit langfristig zu sichern.

Es darf jedoch nicht vernachlässigt werden, dass die modellgetriebene Softwareentwicklung zunächst Zusatzaufwand bedeutet. In der Praxis steht oftmals eine Referenzimplementierung in Form eines Prototyps am Anfang, von dem ausgehend domänenspezifische Metamodelle identifiziert und entworfen werden. Dabei ist bereits zu Beginn ein sehr umfassendes Problemverständnis notwendig. Somit wird eine Produktivitätssteigerung typischerweise erst in späteren Phasen erreicht werden können. Insbesondere die Wartungs- und Weiterentwicklungsphasen sowie der Entwurf von Produktfamilien werden aber langfristig von der MDSD profitieren.

5.2 openArchitectureWare

openArchitectureWare ist ein flexibler Werkzeugkasten, mit dem es möglich ist, die vorgestellten Konzepte der modellgetriebenen Softwareentwicklung umzusetzen. Die Flexibilität ist bei openArchitectureWare in zweierlei Hinsicht gegeben. Zum einen ist der Input in das Framework nicht durch ein festes Metametamodell vorgegeben, wie es bei den traditionellen CASE-Tools der Fall ist. Zum anderen ist auch der Output aus einem erstellten Generator nicht starr auf Quelltext einer vorgegebenen Sprache festgelegt. Vielmehr ist es mit openArchitectureWare möglich verschiedene Codefragmente und Konfigurati-

onsdateien aus demselben Modell zu berechnen. Die mit diesem Seminar durchgeführte Fallstudie zu der Stammdatenpflege (Abschnitt 4) hat dies bestätigt.

Das Arbeiten mit den Werkzeugen von openArchitectureWare ist nach kurzer Einarbeitungszeit und Dank der Eclipse IDE Integration schnell produktiv und macht Spaß. Die knappe Formulierung des Handbuchs macht ein tieferes Einsteigen in die Materie an manchen Stellen allerdings schwieriger. Weiterhin hat die aktuelle Version 4.3 des Frameworks Probleme mit der Eclipse Integration. Es kam bei der Entwicklung des Generators für die Fallstudie öfters vor, dass Eclipse Fehler meldete die keine waren und die nach diversen Neustarts von Eclipse wieder verschwanden. Diesen Sachverhalt aufzuklären hat mehrfach einige Zeit gekostet und ist gerade für Einsteiger frustrierend.

Ist für ein Projekt eine hohe Flexibilität notwendig, so sollte openArchitectureWare das Tool der Wahl darstellen. Allerdings gilt es abzuwägen, ob für eine konkrete Aufgabenstellung schon Spezialgeneratoren vorliegen, denen unter Umständen der Vorzug zu geben ist. Weiterhin ist abzuwarten, wie stabil sich die API von openArchitectureWare erweist. Alles in allem stellt openArchitectureWare ein gutes Werkzeug dar, um Generatoren systematisch zu entwickeln.

6 Literaturverzeichnis

- [BCT05] BROWN, Alan W. ; CONALLEN, Jim ; TROPEANO, Dave: Introduction: Models, Modeling, and Model-Driven Architecture (MDA). In: BEYDEDA, Sami (Hrsg.) ; BOOK, Matthias (Hrsg.) ; GRUHN, Volker (Hrsg.): *Model-Driven Software Development*. 1. Auflage. Berlin, Heidelberg, New York : Springer, 2005, S. 1–16
- [EFH⁺08] EFFTINGE, Sven ; FRIESE, Peter ; HAASE, Arno ; HÜBNER, Dennis ; KADURA, Clemens ; KOLB, Bernd ; KÖHNLEIN, Jan ; MOROFF, Dieter ; THOMS, Karsten ; VÖLTER, Markus ; SCHÖNBACH, Patrick ; EYSHOLDT, Moritz: *openArchitectureWare User Guide*. <http://www.eclipse.org/gmt/oaw/doc/4.3/openArchitectureWare-4.3-Reference.pdf>. Version: 2008. – Abruf am 15. August 2008
- [Gen08a] GENTLEWARE AG: *Schulungsfolien MDSD*. 2008
- [Gen08b] GENTLEWARE AG: *Schulungsfolien openArchitectureWare*. 2008
- [HR00] HAREL, David ; RUMPE, Bernhard: *Modeling Languages: Syntax, Semantics and All That Stuff - Part I: The Basic Stuff*. <http://wwwbroy.informatik.tu-muenchen.de/~rumpe/ps/Modeling-Languages.pdf>. Version: 2000. – Abruf am 06. August 2007
- [HSS05] HUMM, Bernhard ; SCHREIER, Ulf ; SIEDERSLEBEN, Johannes: Model-Driven Development - Hot Spots in Business Information Systems. In: HARTMAN, Alan (Hrsg.) ; KREISCHE, David (Hrsg.): *ECMDA-FA* Bd. 3748, Springer, 2005 (Lecture Notes in Computer Science), S. 103–114
- [Kla07] KLATT, Benjamin: *A Closer Look at the model2text Transformation Language*. <http://www.bar54.de/benjamin.klatt-Xpand.pdf>. Version: 2007. – Abruf am 1. Dezember 2008
- [Met05] METZGER, Andreas: A Systematic Look at Model Transformations. In: BEYDEDA, Sami (Hrsg.) ; BOOK, Matthias (Hrsg.) ; GRUHN, Volker (Hrsg.): *Model-Driven Software Development*. 1. Auflage. Berlin, Heidelberg, New York : Springer, 2005, S. 19–33

- [Pee07] PEEMÖLLER, Björn: *Konzeption und Implementierung einer modellgesteuerten Stammdatenpflege*. Schenefeld, August 2007
- [Red07] RED HAT MIDDLEWARE LLC: *Hibernate Reference Documentation. Version: 3.2.2*. http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf. Version: 2007. – Abruf am 17. Juli 2007
- [sof07] SOFTWARE DESIGN & MANAGEMENT AG: *Client Utilities & Framework. Version: 1.11, 03. April 2007*. <http://cuf.sourceforge.net/>. Version: 2007. – Abruf am 17. Juli 2007
- [SVEH07] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. 2., aktualisierte und erweiterte Auflage*. Heidelberg : dpunkt.verlag, 2007
- [TW04] TEURICH-WAGNER, Sören: MDA: Weg oder Irrweg. In: RUMPE, Bernhard (Hrsg.) ; HESSE, Wolfgang (Hrsg.): *Modellierung* Bd. 45, GI, 2004 (LNI), S. 223–227
- [VD07] VÖLTER, Markus ; DEMELT, Achim: *Generieren vs. Interpretieren - Die andere Seite von MDSD*. Vortrag auf der OOP Conference, 2007
- [Völ07] VÖLTER, Markus: *openArchitectureWare 4.2 Fact Sheet*. <http://apps.eclipse.org/gmt/oaw/oAWFactSheet.pdf>. Version: 2007. – Abruf am 12. November 2008
- [ZW06] ZEPPENFELD, Klaus ; WOLTERS, Regine: *Generative Software-Entwicklung mit der MDA*. 1. Auflage. München : Elsevier - Spektrum Akademischer Verlag, 2006

7 Weitere Internetquellen

- [1] <http://www.fh-wedel.de/~si/seminare/ws08/Termine/Themen.html>
- [2] <http://www.w3.org/TR/xslt>
- [3] <http://www.fornax-platform.org/>
- [4] <http://www.hibernate.org/>
- [5] <http://cuf.sourceforge.net/>