ulm university universität **u**ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Generierung von grafischen Modellrepräsentationen aus textuellen Domänenmodellen

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Peter Fuchs
peter.fuchs@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Betreuer:
David Knuplesch, Stefan Partsch

2012

Fassung 10. Mai 2012

Satz: PDF-LaTeX 2_ε

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Mit der zunehmenden Komplexität von Computersystemen und den steigenden Anforderungen an Software müssen die Verantwortlichen sich laufend an geänderte Umgebungsbedingungen anpassen. Viele Hersteller sind mit den ständigen wechselnden Implementierungstechnologien überfordert. Einen Lösungsansatz stellt die modellgetriebene Softwareentwicklung dar, bei der die Softwareerstellung in großen Teilen automatisiert abläuft. Durch den Einsatz von Modellen wird eine Trennung zwischen der Beschreibung von fachlichen- und technischen Anforderungen angestrebt. Dies ermöglicht eine leichtere Umstellung auf neue Technologien. Die Modelle beschreiben dabei die Software auf abstrakter Ebene und werden mittels Generatoren in lauffähigen Programmcode überführt.
Ein modellgetriebener Softwareansatz wird bei den Wieland Werken AG in der fertigungsbegleitenden Software WFS umgesetzt. Dabei verspricht man sich in erster Linie verkürzte Entwicklungszeiten zu erreichen, aber auch die Softwarequalität steigern zu können. Weite Teile der Anwendungssoftware werden durch textuelle Modelle beschrieben, dabei werden primär fachliche Aspekte modelliert. Dadurch gewinnen die Modelle nicht nur für den Entwickler an Bedeutung, sondern auch für Projektleiter und Businnes-Consultants. Ein Problem bereitet dabei die textuelle Form der Modelle. Diese ist zum Teil für Mitarbeiter, die keine IT-Experten sind schwer verständlich und oft zu umfangreich.
Ziel dieser Diplomarbeit ist es ein besseres Verständnis von solchen textuellen Modellen im Umfeld der WFS zu ermöglichen. Als Lösungsansatz wird ein Konzept vorgestellt, welches in mehreren Schritten automatisch aus den textuellen Modellen grafische Repräsentationen erzeugt. Dabei werden zunächst die vorhanden Modelle nach relevanten Informationen gefiltert. Anschließend wird eine Applikation zur grafischen Darstellung ausgewählt und die dafür notwendigen Eingangsdaten werden mithilfe eines Generators erzeugt. Für eine gute Übersicht wird ein passendes Layout vorgeschlagen. Schließlich wird eine Implementierung des Lösungskonzeptes vorgestellt, welche in die WFS integriert ist.

1.2 Aufbau der Arbeit

Die vorliegende Diplomarbeit befasst sich mit der Implementierung einer graphischen Anwendungssoftware zur interaktiven Visualisierung von textuellen Domänen-Modellen der Wieland Software-Architektur. Zunächst werden im Kapitel 2 die grundlegenden Begriffe der modellgetriebenen Softwareentwicklung erläutert, die für das Verständnis der Arbeit notwendig sind. Anschließend wird der modellgetriebene Ansatz einer MES-Software anhand der Softwarelösung von den Wieland Werken vorgestellt. Im Kapitel 3 werden Problemfälle aus dem modellgetriebenen Softwareumfeld einer MES-Software analysiert. Dabei werden verschiedene Szenarien in einem Softwareentwicklungsprozess mit den beteiligten Personengruppen dargestellt und mein Lösungsansatz vorgestellt. Dieser wird im Kapitel 4 Schritt für Schritt erläutert. Dabei werden sowohl Konzepte zur visuellen Darstellung von Informationen vorgestellt, als auch technische Softwaretools zur Umsetzung gegenübergestellt und verglichen. Anschließend erfolgt im Kapitel 5 die Implementierung der Anwendung zur grafischen Darstellung von Modellen. Hierbei wird ein Generator entwickelt um Daten aus den textuellen Modellen in grafische Diagramme zu transformieren. Zum Ende werden im Kapitel 6 die Ergebnisse zusammengefasst, evaluiert und abschließend ein Ausblick gewährt.

2 Grundlagen Modellgetriebene Softwareentwicklung

Dieses Kapitel stellt die wichtigsten Grundlagen zu *Modellgetriebenen Softwareentwicklung* vor. Zunächst wird das Konzept der Modellgetriebenen Softwareentwicklung behandelt. Anschließend gehe ich auf Teilaspekte wie Modellierung und Modelltransformationen sowie verschiedene Modellierungssprachen ein. Im Zuge dessen werden die gängingsten Standards zu Modellgetriebenen Softwareentwicklung erläutert und gegenübergestellt. Im Abschluss wird der Modellgetriebene Ansatz zur Softwareentwicklung anhand eines MES-Softwaresystems vorgestellt. Dabei wird die MES-Lösung der Wieland Werke AG als Beispiel verwendet.

2.1 Der modellgetriebene Ansatz

Modellgetriebene Softwareentwicklung ist in den vergangen Jahren insbesondere unter Schlagworten wie *Model Driven Software Development (MDSD*), *Model Driven Architecture (MDA)*, *Model Driven Development (MDD)* oder *Model Driven Engineering (MDE)* zu einem wichtigen Thema für die Software-Branche geworden. Dabei ist eindeutig ein Trend festzustellen: die klassische *Codezentriete Software-Entwicklung* wird immer mehr durch die *Generative Software-Entwicklung* abgelöst.
Im Zuge dieser Arbeit wird die Abkürzung MDSD als Oberbegriff für die Techniken der Modellgetriebenen Softwareentwicklung verwendet. Die Modellgetriebene Softwareentwicklung befasst sich mit der Automatisierung in einem Softwareentwicklungsprozess [24]. Dies äußert sich dadurch, dass die wesentlichen Bestandteile eines Softwaresystems (*Artefakte*) generativ aus formalen Modellen abgeleitet werden. Die erzeugten Artefakte können sowohl Code einer ausführbaren Programmiersprache wie *Java* oder *C++*, als auch Konfigurationdateien sein, die für die Lauffähigkeit eines Systems benötigt werden. Zudem können auch den Entwicklungsprozess unterstützende Artefakte generiert werden, z.B. Softwaretests und Dokumentation. Die möglichen Artefakte eines modellgetriebenen Softwaresystems werden in der folgenden Abbildung 2.1 genauer veranschaulicht.

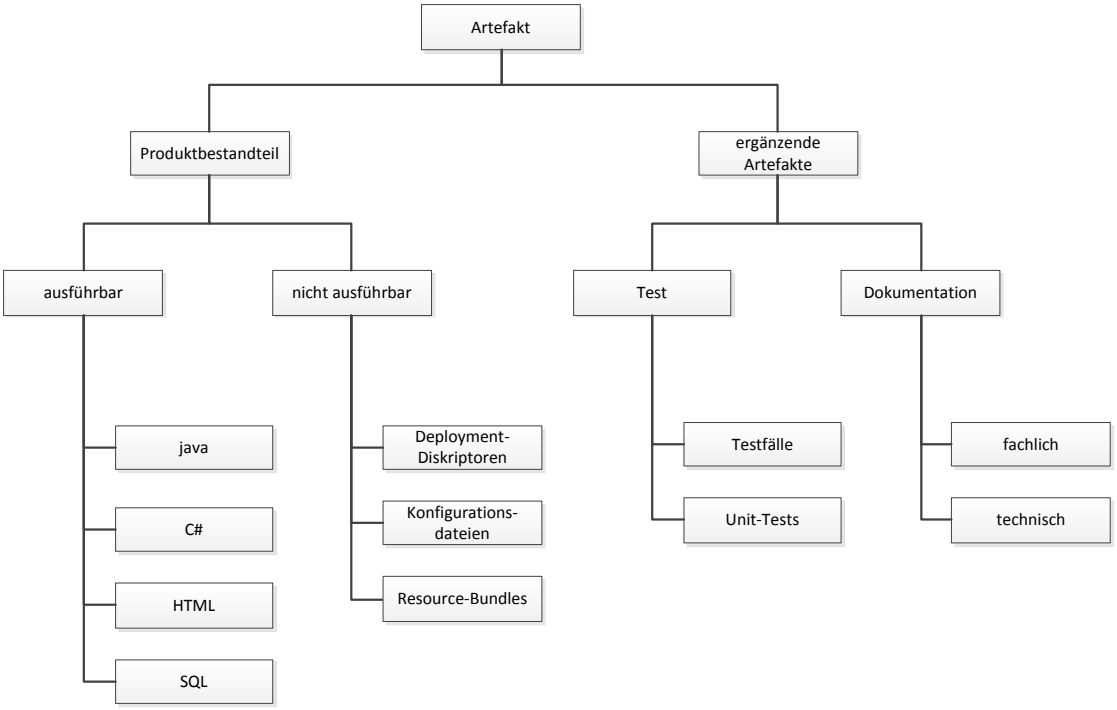

Abbildung 2.1: Generierbare Artefakte [24]

Ein wichtiges Ziel der Modellgetriebenen Softwareentwicklung ist eine Steigerung der Entwicklungsgeschwindigkeit. Das Mittel dazu heißt „Automation durch Formalisierung“ [32]. Im Mittelpunkt des Entwicklungsprozesses steht nicht mehr der Programmcode, sondern Modelle und Transformationen. So lassen sich komplette Software-Architekturen in Form von Modellen beschreiben. Diese Modelle werden durch die Verwendung von formalen Beschreibungssprachen wie *UML* oder domänen-spezifischen Sprachen (*DSL*) definiert. Aus diesen formal eindeutigen Modellen lassen sich mit Hilfe von Generatoren Transformationen durchführen. Diese können ein Modell in ein anderes Modell transformieren (*Model-To-Model*) bzw. eine textuelle Repräsentation (*Model-To-Text*) von diesem Modell erstellen z.B. ausführbarer Javacode. Der Gebrauch solcher Techniken ermöglicht eine Trennung zwischen der Beschreibung von fachlichen und technischen Anforderungen. Somit lassen sich Technologiewechsel in einer Software leichter durchführen, da die Fachlichkeit (langlebig) keinen großen Veränderungen ausgesetzt wäre. Die laufenden Kosten der Wartung und Weiterentwicklung eines Softwareproduktes könnten dadurch gesenkt werden.

Modelle in der MDSD können auf einer höheren Abstraktionsebene definiert werden, unwesentlichen Details werden bewusst weggelassen, was die Problembeschreibungen erheblich vereinfacht und eine fachbezogene Basis aufbauen lässt. Dies führt dazu, die stetig wachsende

Komplexität in heutigen Systemen leichter beherrschen zu können. Im weiteren verspricht man sich durch den Einsatz von MDSD eine Steigerung der Softwarequalität, da sich eine einmal definierte Softwarearchitektur gleichförmig in der Implementierung wiederfindet [29]. Natürlich sind diese Ziele nichts Neues, in der Vergangenheit (der IT-Branche) versuchten auch schon andere Konzepte, wie Hochsprachen, Objektorientierung oder Komponentensystem diese zu erreichen. Die modellgetriebene Softwareentwicklung stellt einen weiteren IT-Ansatz dar, welcher versucht den wachsenden Anforderungen gerecht zu werden.

2.2 Modellierung und Modelltransformationen

Bei modellgetriebenen Softwareentwicklung spielen die Modelle eine zentrale Rolle. Diese beschreiben nicht nur die Architekur eines Systems, sondern auch das Verhalten der ausführbaren Anwendung. Im folgenden werden Modelle und ihre Verwendung, im Kontext der modellgetriebener Softwareentwicklung behandelt. Zur Beschreibung von Modellen werden Domänenspezifische Sprachen eingeführt und in ihrem Aufbau über die Domäne und der Metamodelle beschrieben. Zur Verdeutlichung werden am Ende des Kapitels einige gängige Modellierungssprachen vorgestellt.

2.2.1 Modelle

In der MDSD ist ein Modell eine abstrakte Abbildung eines Softwaresystems. Unter dem Begriff der Abstraktheit wird ein kompaktes und vereinfacht dargestelltes Modell verstanden, bei dem auf unwesentliche Details bewusst verzichtet wird. Das ermöglicht die gleiche Sicht auf den Prozess sowohl für den Kunden als auch für den Entwickler, was die Kommunikation erheblich erleichtert [24]. So können z.B. im Laufe der Modellierung die technischen Informationen bewusst weggelassen werden um den Fokus ausschließlich auf den fachlichen Teil zu legen. Genauso kann die Architektur eines Softwaresystems modelliert werden. Man unterscheidet zudem zwischen den dynamischen Modellen, die Verhalten darstellen und statischen Modellen, die eine feste Struktur wie z.B. Daten und deren Beziehungen darstellen. Modelle werden durch eine *Modellierungssprache* beschrieben. Eine Modellierungssprache ist jede formal beschreibbare Sprache die von einem Generator verstanden werden kann [24]. Diese kann sowohl grafisch oder tabellarisch definiert werden aber auch in strukturierter Textform. Eine wohl der bekanntesten Modellierungssprachen in der Praxis ist die *Unified Modeling Language (UML)*. Durch ihre unterschiedlichen Diagrammtypen und die einheitlichen Notation, lassen sich sowohl

statische als auch dynamische Modelle konstruieren. *Entity Relationship* (ER) ist eine weitere Modellierungssprache mit der die persistente Datenstrukturen modelliert werden können. Auf Modellierungssprachen wird im Unterkapitel 2.2.4 ausführlicher eingegangen. Zusammengefasst lassen sich Modelle nach [24] folgendermaßen charakteresieren:

- Modelle helfen komplexe Konzepte besser zu verstehen und zu kommunizieren. Die Gefahr von Missverständnissen ist geringer.
- Es können verschiedene Merkmale je nach Kontext modelliert werden. Dies ermöglicht eine Eingrenzung des Problemraums auf einen bestimmten Bereich
- Modelle ermöglichen eine einheitliche Architektur da die Umsetzung systemweit gleich ist
- Modelle lassen sich häufig wiederverwenden. Die führt zu einer beschleunigten Softwareentwicklung
- allerdings ist der Initialaufwand meistens höher.

2.2.2 Domäne

Ausgangspunkt der Modellierung bei MDSD ist stets eine *Domäne*. Sie bezeichnet im Allgemeinen ein begrenztes Interessens- oder Wissensgebiet [29]. Um dieses Wissen verarbeiten zu können, bedarf es einer Erfassung aller elementaren Eigenschaften der Domäne. Domänen können sich sowohl in fachliche als auch in technische Richtung ausprägen. Beispieldomänen findet man im Bereich Web-basierten Business-Anwendungen, im Finanz- und Versicherungswesen oder in Eingebetteten Systemen. Ist eine Domäne eindeutig erfasst, so kann man dazu übergehen eine Struktur in Form eines Metamodells aufzubauen.

2.2.3 Metamodellierung

Nachdem man sich über die Struktur der Domäne im Klaren ist, ist der nächste Schritt, diese zu formalisieren. Die Formalisierung findet durch das *Metamodell* statt. Meta stammt aus dem griechischen und Bedeutet soviel wie „über, nach, neben oder zwischen“. Somit ist ein Metatamodell ein „Übermodell“, welches sich auf einer höheren Ebene befindet. Dies bedeutet soviel, dass Metamodelle, andere Modelle definieren also eine Modellierungssprache beschreiben. Ein Metamodell sagt demnach etwas über die Struktur, Beziehungen und Einschränkungen eines konkreten Modells aus [24]. Das Metamodell definiert die *abstrakte Syntax* und die *statische Semantik* einer Sprache. Die *abstrakte Syntax* beschreibt die Regeln zur korrekten Konstruktion von Modellen auf Basis der Elemente des Metamodells. Die *statische Semantik* umfasst

Constraints die von den Modellelementen und Beziehungen der abstrakten Syntax eingehalten werden müssen, siehe Abbildung 2.2. Beispiel dafür wäre ein Deklarationszwang bei Variablen.

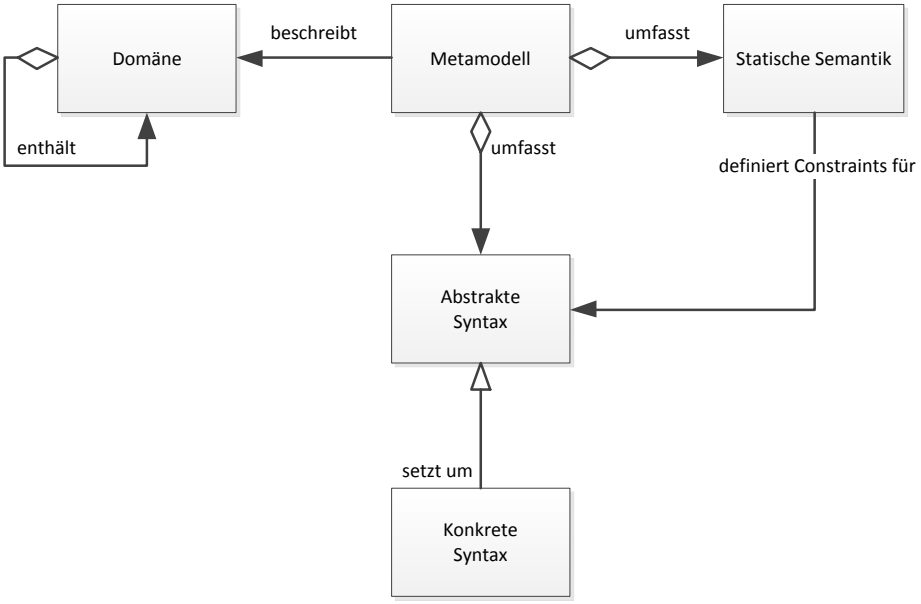

Abbildung 2.2: Modellierung - Metamodell [29]

Dadurch werden Modellierungsfehler schon frühzeitig erkannt und tauchen nicht erst als Fehler in der Generierung oder in dem Anwendungscode auf.
Metamodell und Modell stehen in einer Klasse-Instanz-Beziehung zueinander: Jedes Modell ist die Instanz eines Metamodells [29]. Beim definieren eines Metamodells benötigt man ebenfalls Sprachmittel um dieses beschreiben zu können. Die Beschreibung der Abstrakten Syntax und Statischen Semantik des Metamodells findet im *Meta-Metamodell* statt. Ein Metamodell ist demnach eine Instanz des *Meta-Metamodells*. Theoretisch lässt sich dieses Prinzip weiter fortsetzen, so dass durch weitere Abstraktionsschichten weitere Meta-Metamodelle erstellt werden können. Der Zusammenhang zwischen Modell, Metamodell und Meta-Metamodell wird anhand der Modellierungssprache *UML* verdeutlicht. Die obereste Ebene M3(Meta-Metamodell) wird durch die Metasprache *Meta Object Facility (MOF)* beschrieben. Die *MOF* Spezifikation ist eine Entwicklung der OMG(Object Management Group) und dient in vielen MDSD-Werkzeugen als formale Basis für die Definition von Metamodellen. Danach folgen die UML-Metamodelle, diese beschreiben die wichtisten Konstrukte die im M1 Modell Verwendung finden. Die Elemente des

M1 Modells sind damit Instanzen der Elemente des M2 Metamodells. In diesem Beispiel wird auf der M1 Ebene eine Klasse Stadt definiert. Diese bekommt eine Reihe von Attributen wie Name, Einwohner. Diese Klasse wird auf der M0 Ebene instanziert und beschreibt zur Laufzeit das Objekt mit dem Namen Ulm und dem Attribut Einwohner, siehe Abbildung 2.3.

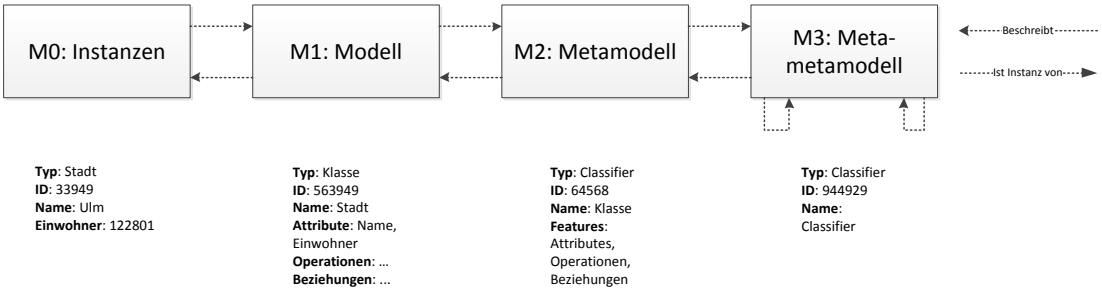

Abbildung 2.3: UML Modell in vier Metaschichten[29]

2.2.4 Domänenspezifische Sprache

Wie bereits im Kapitel 2.2.1 angedeutet wurde, existieren unterschiedliche Modellierungssprachen von denen einige in diesem Abschnitt kurz erläutert werden. Da in der MDSD unterschiedliche Problemaspekte behandelt werden, bedarf es heutzutage Sprachen, die sich in einer Domäne genau und effizient ausdrücken können. Einige Modellierungsprachen sind spezialisiert, Prozesse zu modellieren, andere wiederum beschreiben Datenbankstrukturen. Da die MDSD diese Sprachen verwendet um Sachverhalte innerhalb von bestimmten Domänen zu formulieren, spricht man von *domänenspezifischen Sprachen*(*D*omain *S*pecific *L*anguage, *DSL*). Van Deursen [13] schlägt folgende Definition vor: „A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.“ Mit einer *DSL* lassen sich also Modelle einer Domäne erzeugen. Neben dem Metamodell, welches die abstrakte Syntax und die statische Semantik für die DSL definiert hat, bedarf es noch einer *konkreten Syntax* sowie der *dynamischen Semantik*. Eine *konkrete Syntax* ist die genaue Ausdrucksweise einer Sprache. Das bedeutet genauer, dass eine konkrete Syntax die Realisierung einer abstrakten Syntax ist. Würde man die Grammatik der deutschen Sprache als Beispiel nehmen und die Regel des Satzbaus *Subjekt, Prädikat, Objekt* als Teil der abstrakten Syntax definieren, so wäre der Satz „Ich habe Durst“ ein Teil der konkreten Syntax.
Die *dynamische Semantik* ist im Allgemeinen an den Modellierer gerichtet, da dieser die jeweilige Bedeutung der einzelnen Modellelemente kennen und verstehen muss. Das Zusammenspiel

zwischen den einzelnen Elementen wird in der Abbildung 2.4 verdeutlicht. Im weiteren taucht die Semantik im Zuge der automatischen Transformationen auf, welche die Semantik umsetzen, mehr dazu im Kapitel 2.2.5.

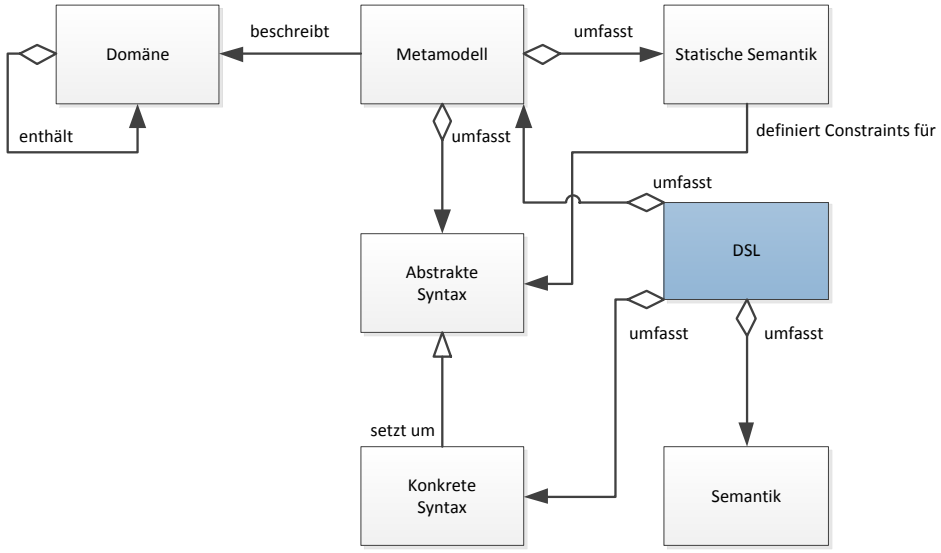

Abbildung 2.4: Modellierung - Domain Specific Language (DSL)[29]

Beispiele für Modellierungssprachen (DSLs)

In der MDSD werden verschiedene Modellierungssprachen für die unterschiedlichsten Anwendungsfälle eingesetzt. Es existieren heute buchstäblich hunderte solcher DSLs. Unter den grafischen Modellierungssprachen hat sich die Sprache *Unified Modeling Language* (UML) durchgesetzt. Durch die OMG wurde dieser Standard geschaffen, um abstrakte Abbildungen eines realen Systems erstellen zu können. Besonders objektorientierte Softwarearchitekturen können damit gut abgebildet werden. Durch die weite Verbreitung dieser Sprache ist sichergestellt, dass viele Softwareentwickler die Sprache beherrschen werden und somit der Einarbeitungsaufwand relativ gering ausfällt [7]. Mit Hilfe von UML-Profilen lässt sich die Sprache auf die relevante Konzepte einer Domäne spezialisieren. Zur Realisierung von Profilen werden existierende Erweiterungsmechanismen der UML benutzt, die auch kompatibel mit den Modelling-Tools sind. UML-Profile sind in ihrer Semantik detaillierter, aber weiterhin auch konsistent zur UML-Semantik. In der Abbildung 2.5 sieht man wie mit Hilfe von Stereotypen ein UML-Konstrukt erweitert werden

kann. Gesamtbetachet bedeutet es, dass sich das Metamodell der UML anhand vorgegebener Konstrukte erweitern oder einschränken lässt [20].

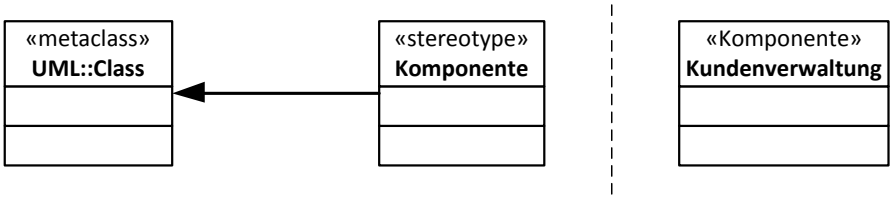

Abbildung 2.5: Stereotypisierung in UML [29]

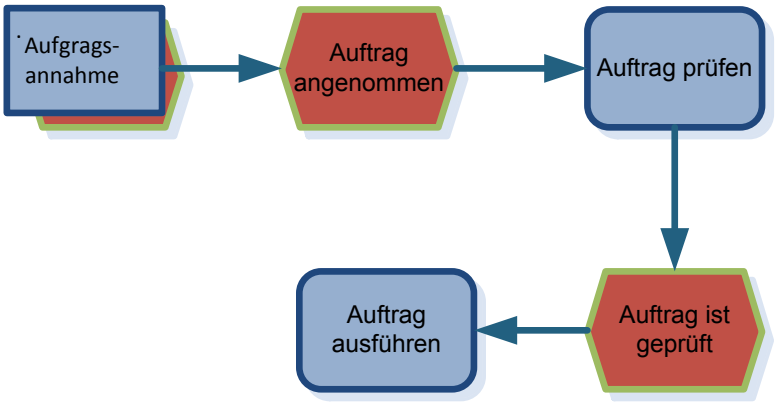

Abbildung 2.6: EPK-Beispiel

Ereignisgesteuerte Prozessketten (EPK) können Geschäftsprozesse in ihren wesentlichen Elementen darstellen. Dadurch, dass EPKs einerseits eine wichtige Komponente des *SAP R/3*-Systems und andererseits im ARIS(Architektur integrierter Informationssysteme)-Framework enthalten sind, stellen sie eine verbreitete Methode der Geschäftsmodellierung dar. In den Wieland Werken AG wo ich meine Diplomarbeit schreibe, werden EPKs zur Modellierung von Geschäftsprozessen in der Softwareentwicklung verwendet. Die Darstellung der Arbeitsprozesse erfolgt grafisch. Im wesentlichen wird ein gerichteter Graph mit Verknüpfungslinien und -pfeilen zwischen den Objekten aufgebaut. In den Prozessketen wechseln sich Objekte zwischen Ereignis (Raute) oder Funktion(abgerundetes Rechteck) ab, dadurch entsteht eine alternierende

Folge. Zudem existieren logische Verknüpfungen wie *or, and, xor*, die Ereignisse mit den Funktionen verknüpfen können [24]. In der Abbildung 2.6 wird ein einfacher Prozess der Auftragsabwicklung modelliert.

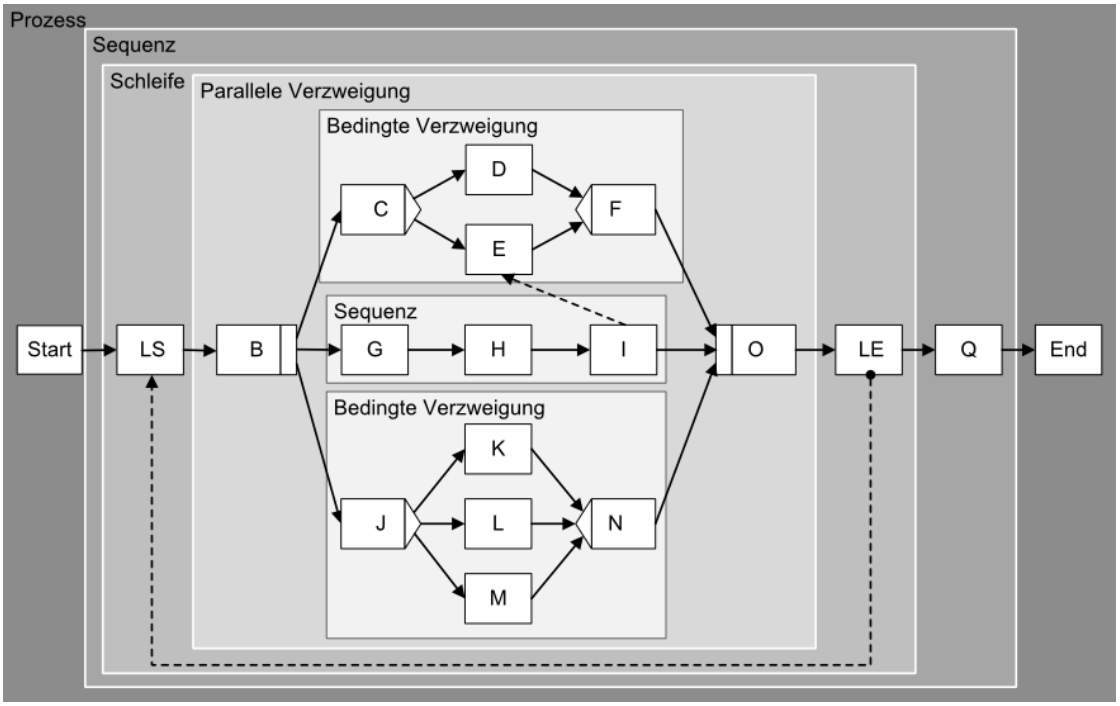

Abbildung 2.7: Darstellung aller Kontrollfluss-Konstrukte des ADEPT-Basismodells [14]

ADEPT ist eine an der Universität Ulm entwickelte Sprache zur Beschreibung von Gesschäftsprozessen, welche direkt von Softwaresystemen ausgeführt werden. Ein ADEPT-Prozessmodell ist ein gerichteter Graph mit eine symmetrischer Blockstruktur wie die Abbildung 2.7 demonstriert. ADEPT$_{flex}$ bietet dem Anwender dynamische WF-Änderungen vorzunehmen. Hiermit lassen sich Struktur, Attribute oder Status einer ausführbaren WF-Instanz abändern. Dadurch lassen sich Änderungen realisieren wie z.B. das dynamische Einfügen einer Aktivität zwischen zwei Knotenmengen oder für die Umsetzung von Ad-hoc-Vorwärtssprügen [26].

Eine weitere Geschäftsmodellerierungssprache ist die *Business Process Model and Notation (BPMN)*. Sie stellt nach [6], einige grafische Notationen zur Verfügung womit sich sowohl fachliche Modelle als auch technisch ausgerichtete Diagramme erstellen lassen, die als Grundlage für die Ausführung in einem Workflow- oder Business Process Management-System dienen. Ein einfaches Beispiel zum Loginprozess wird in der Abbildung 2.8 verdeutlicht.

Entity Relationship Modelle (ERM) werden verwendet um persistente Datenstrukturen und deren Beziehungen graphisch abzubilden. Das modellieren erfolgt grafisch, wobei Entities (Ab-

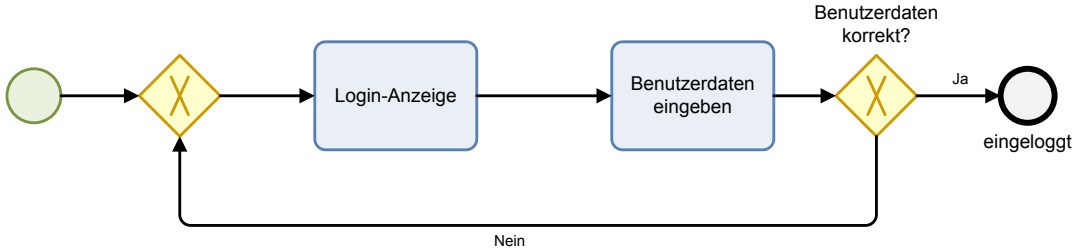

Abbildung 2.8: Einfaches BPMN-Diagramm

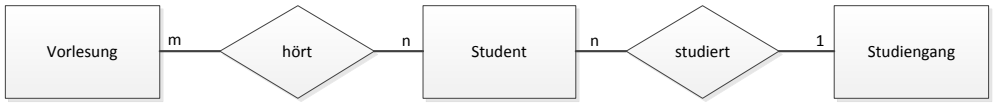

Abbildung 2.9: Einfaches ER-Diagramm

bildung realer Dinge) mit den dazugehörigen Relationships (Beziehungen zwischen Entities) verknüpft werden und mit Kardinalitäten versehen werden. Ein einfaches Beispiel sieht man in der Abbildung 2.9.

Zu all diesen Modellierungssprachen werden in der Praxis häufig eigene spezialentwickelte DSLs konstruiert. Häufig wird man textuelle DSLs in der MDSD wiederfinden. Textuell beschriebene Modelle werden auch bei den Wieland Werken AG eingesetzt um die Softwarearchitektur zu beschreiben. Wie so etwas in der Praxis umgesetzt wird, wird im Kapitel 2.4 beschrieben.

2.2.5 Platform und Transformationen

Nachdem wir mit Hilfe einer DSL einen Problemraum eindeutig erfassen können, ist der nächste Schritt die Transformation der erzeugten Modelle. Diese soll unter anderem bei der Erzeugung eines ausführbaren Programmcodes helfen.

Formales Modell

Formale Modelle bilden die Basis für automatisierte Transformation, sie sind das Produkt einer DSL mit der dazugehörigen konkreten Syntax. Folglich ist ein Modell in der DSL formulierter «Satz » [29].

Ein Beispiel könnte man sich die Programmiersprache Java herannehmen. Ein Java-Programm wäre eine Instanz der Programmiersprache Java bzw. des entsprechenden Metamodells. Die Domäne von Java ist sehr umfassend, denn alles was man mit einem Rechner mit Hilfe der Programmiersprache Java machen kann, ist eine Domäne. Das folgende schaubild 2.10 veranschaulicht die Rolle eines formalen Modells im Kontex der Modellierung einer MDSD-Architektur.

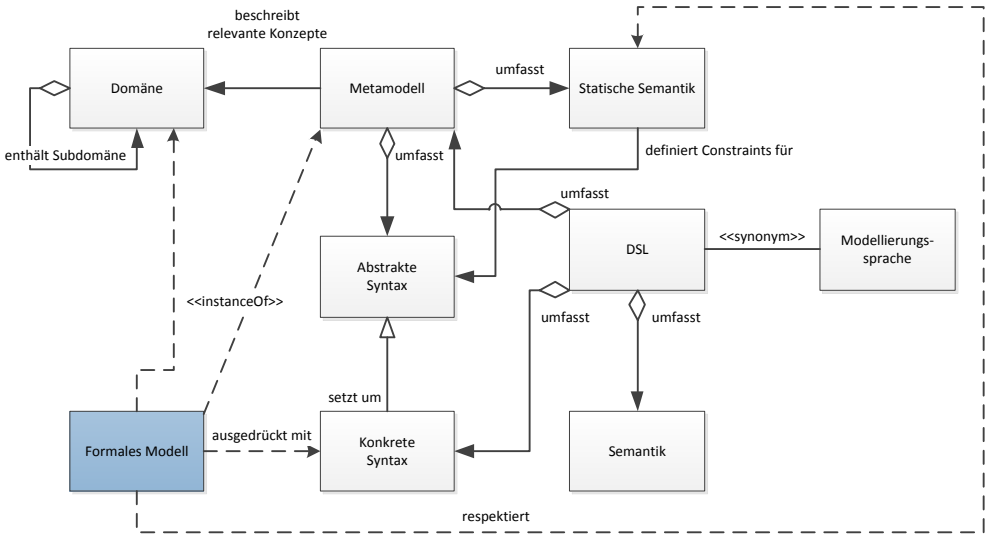

Abbildung 2.10: Formales Modell in der MDSD

Plattform

Unter einer Plattform versteht man in der MDSD eine Umgebung in der die Anwendung ausgeführt wird und für die Programmcode generiert wird. Während man mit einer DSL den Problemraum abbildet, so beschreibt eine Plattform den Lösungsarum und bietet Unterstützung für Transformationen an. Sie besteht neben der Zielprogrammiersprache aus Bausteinen wie Frameworks, Komponenten und Programmbibiliotheken. Ziel ist es nicht diese mit in den Generator aufzunehmen, sondern Code zu generieren, der diese Bausteine benutzt. Eine mächtige Plattform verhilft zu einer einfachen Generierung von Programmcode [29].

Transformationen

Modelltransformationen sind berechenbare Funktionen die als Eingabe ein Quellmodell erwarten und die Ausgabe ein Zielmodell liefert. Diese verwenden die Metamodelle der jeweiligen Modelle um Transformationsvorschriften ausführen zu können. Dabei unterscheidet man in de MDSD zwischen den Modell-zu-Modell-Transfoormationen (*ModelToModel, M2M*) und Modell-zu-Code-Transformationen(*ModelToCode, M2C*). Eine *M2M*-Transformation erzeugt wiederum ein Modell, welches auf einem anderen Metamodell formuliert sein kann.
Eine *M2C*-Transformation erzeugt entweder Programmcode oder andere textuellen Repräsentationen z.B. Frameworkkonstrukte, Datenbankskripte etc.. Da der erzeugte Code für eine bestimmte Plattform bestimmt ist, sind die erzeugten Artefakte plattformspezifisch.
Die Transformationen werden von einem Generator durchgeführt, der die Semantik der Modelle abbilden soll 2.11. Der grobe Ablauf einer M2C-Transformation wird im folgenden Schaubild verdeutlicht. Dabei werden für die Mappings sogenannte *Templates* verwendet, diese enthalten die entsprechenden Vorlagen für die Codekonstrukte. *Extensions* sind nichts anderes als Hilfsfunktionen die von den Templates verwendet werden können um beispielsweise den Pfad für die Ausgabedatei zu ermitteln.

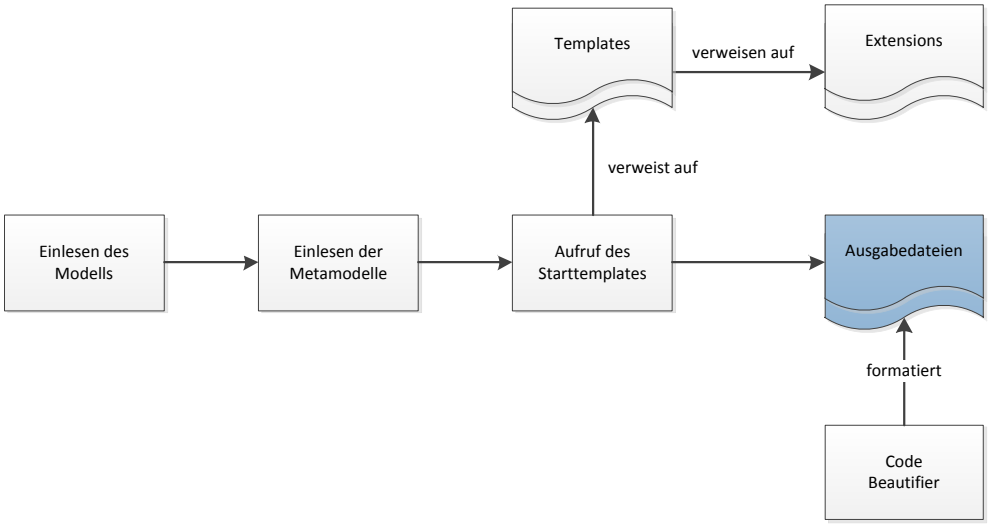

Abbildung 2.11: Ablauf einer M2C-Transformation [17]

2.3 Ansätze in der Modellgetriebene Softwareentwicklung

Bisher wurden in dieser Arbeit die Grundkonzepte der modellgetriebenen Softwareentwicklung vorgestellt. Im Laufe der vergangenen Jahren enstanden mehrere Ansätze der MDSD die unterschiedliche Ziele verfolgen. Auf zwei dieser Ansätze, die im späteren Verlauf wichtig erscheinen können, gehe ich folgenden näher ein.

2.3.1 Architekturzentrierte MDSD

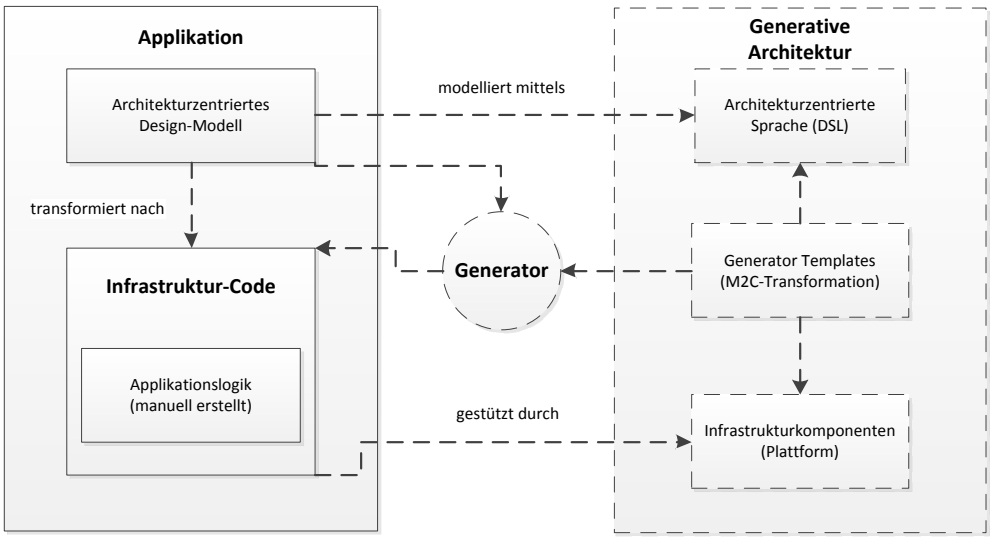

Abbildung 2.12: Prinzip archtitekurzentrierter MDSD nach [29]

Eine der Formen der modellgetriebenen Softwareentwicklung stellt die *architekturzentrierte MDSD* (*AC-MDSD* engl. Architecture Centric MDSD) dar. Dieser Ansatz stellt die Softwarearchitektur in den Vordergrund. Die Domäne wird entweder mit einem UML-Profil oder einer textuellen DSL modelliert. Das Metamodell womit die Domäne abgebildet werden soll, enthält möglichst abstrakte Architekturkonzepte. Aus den formalen Modellen findet in der Transformationsphase die automatische Generierung von Artefakten statt. Dabei wird versucht nicht die komplette Software abzubilden, sondern ein Grundgerüst zu schaffen, der den Infrastrukturcode beinhaltet und einen Rahmen schafft um die Implementierung von Hand vorzunehmen. Demnach wird bei der Generierung nur auf die M2C-Transformation zugegriffen, siehe Abbildung

2.12. Das Ziel der AC-MDSD ist es den immer größer werdenen Teil von Routinearbeiten zu automatisieren und somit den entstehenden Aufwand und die Fehleranfälligkeit zu reduzieren. Dadurch verspricht man sich eine höhere Effizienz und eine bessre Qualität in der Softwareentwicklung. Weitere Vorteile wären beispielsweise, die Wiederverwendung bereits entwickelter Softwarearchitekturen, sogar über das aktuelle Projekt hinaus. Die AC-MDSD eignet sich demzufolge besonders für Anwendungen die eine ähnliche Softwarearchitektur aufweisen, diese werden als *Software-Systemfamilien* bezeichnet [29].

2.3.2 Model Driven Architecture

Unter dem Begriff *Model Driven Architecture (MDA)* versteht man eine Spezifikation, die für die MDSD durch die Objectiv Management Group (OMG) entwickelt wurde. Dabei stellt die MDA kein vorgeschriebenes Vorgehensmodell dar, sondern lediglich Vorschläge zur Standardisierung allgemeiner MDSD-Konzepte. Modelle werden in der MDA auf unterschiedlichen Abstraktionsebenen beschrieben. Am Anfang der Implementierungskette steht das *Computation Independet Model* (*CIM*), das die fachlichen Dinge einer Domäne in der Umgangssprache beschreibt. Dieses stellt die Basis für die Kommunikation zwischen den Entwicklern und Produktverantwortlichen, da es frei von technischen Details ist.
Die Implementierung findet bei der MDA in dem *Platform Independent Model* (*PIM*) statt. Das PIM bildet die Geschäftslogik auf eine technische Weise ab, ohne dabei eine spezifische Plattform anzusprechen.

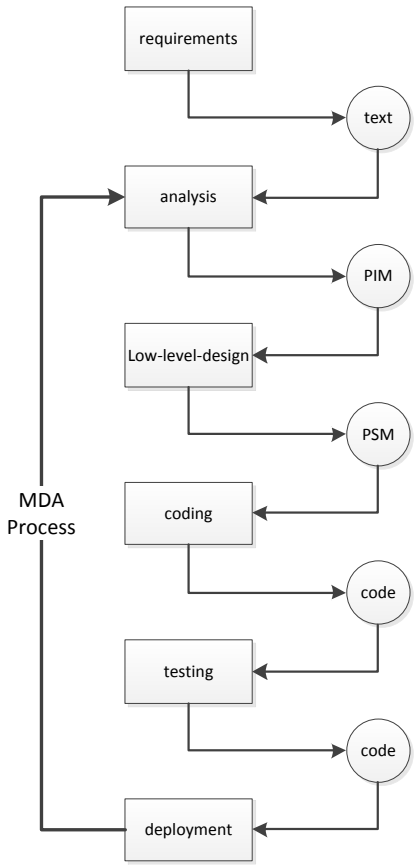

Abbildung 2.13: MDA Softwareprozess

Erst im nächsten Schritt wird das PIM in *Platform Specific Model* (*PSM*) transformiert. Ab dieser Ebene können weitere M2M-Transformationen stattfinden, bis man zum Programmcode der Anwendung gelangt. Durch diese Aufteilung der Modelle in verschiedene Abstractionsebenen wird es sichergestellt einen Technologiewechsel ohne größeren Aufwand vollziehen zu können. Dadurch bleibt die Fachlogik unverändert, man passt nur den Plattformspezifischen Teil an.
Bei der Modellierung schreibt die OMG für das MDA-Konzept die Sprache UML vor. Als Metasprache für die UML wird die Metamodellierungssprache *Meta Object Facility* (*MOF*) vorgechrieben. Ein weiterer Standard ist das *XML Metadata Interchange* (*XMI*), diese wird für die Kommunikation zwischen den verschiedenen MDA-Tools eingesetzt. Für die Modelltransformationen schreibt der Standard die *Query View Trarnsformation* (*QVT*)-Sprache vor.
Durch diese Standards versucht die OMG dem Ziel der Portabilität und einer besseren Interoperabilität näher zu kommen [22].

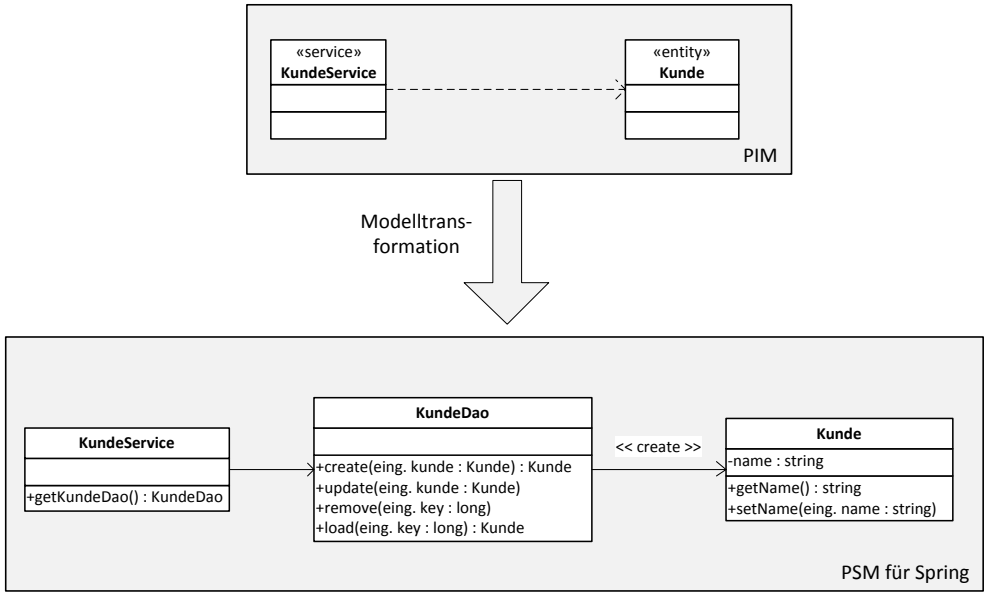

Abbildung 2.14: MDA bei Verwendung eines PIM nach: [31]

2.3.3 Vergleich von MDA und AC-MDSD

Nachdem in den vorangegangenen Abschnitten die wesentlichen Merkmale von MDA und AC-MDSD erläutert wurden, folgt nun eine Gegenüberstellung der beiden Konzepte. Die Unterschiede der beiden Arten der Modellgetriebenen Softwareentwicklung sind an den verschiedenen Primärzielen fest zu stellen. MDA verfolgt das oberste Ziel der Interoperabilität und Portabilität während man bei der AC-MDSD das Ziel der Verbesserung des Entwicklungsprozesses als wichtig ansieht. Dabei spielt bei der AC-MDSD die Effizienz, Qualität und die Wiederverwendbarkeit eine große Rolle. Die MDA ist sehr an Standards der OMG gebunden, dadurch ist die Flexibilität in der Auswahl der Tools nicht gegeben. In größeren Softwareunternehmen würde der MDA-Ansatz größeren Zuspruch finden, da hier die Metamodelle zwischen den verschiedenen Domänen leichter geteilt werden können [22]. Durch seinen praxisnahen Ursprung ist AC-MDSD wesentlich pragmatischer in seinen Ansätzen und verzichtet bewusst auf die Punkte Interoperabilität und Portabilität, da hier kaum Standards existieren. Des Weiteren spielen M2M-Transformationen in AC-MDSD eine untergeordnete Rolle, im speziellen erzeugt AC-MDSD den Sourcecode direkt aus dem PIM und verzichtet auf ein PSM.

Weitere Merkmale der MDA und AC-MDSD sind der folgenden Auflistung zu entnehmen.

MDA	AC-MDSD
Standard für Modellierungssprache (UML)	Metamodellierung durch DSLs frei wählbar
Interoperabilität als Antreibungsziel	praktische Umsetzung mit Toolunterstützung
Abstraktion der Modelle in PIM, PSM etc.	Speziell für die Domäne angefertigten Modelle
Theoretische Konzepte, keine praktikablen Vorlagen in der Umsetzung	viele praktische Frameworks, da weniger eingeschränkt
Strikte Standards lassen wenig Spielraum	offener Ansatz, begünstigt Open Source Infrastruktur
Bieten dafür verlässliche Umgebung	Beständigkeit der Tools in Frage zu stellen

2.4 Anwendung von MDSD bei der MES-Entwicklung

Eins der größten Hemmnisse bei der Entwicklung und Wartung von komplexen Softwaresystemen ist, dass an ein solches System eine große Anzahl verschiedener fachlicher wie technischer Anforderungen gestellt werden, welche letztendlich in mittels Quellcode umgesetzt werden. In diesem Kapitel wird das modellgetriebene Vorgehen zur Umsetzung von Softwareanforderungen anhand eines MES-Systems vorgestellt. Dabei wird beispielhaft am MES-System der Wieland Werke AG, der Aufbau einer modellgetriebenen Softwarearchitektur erläutert.

2.4.1 Manufactoring Execution System

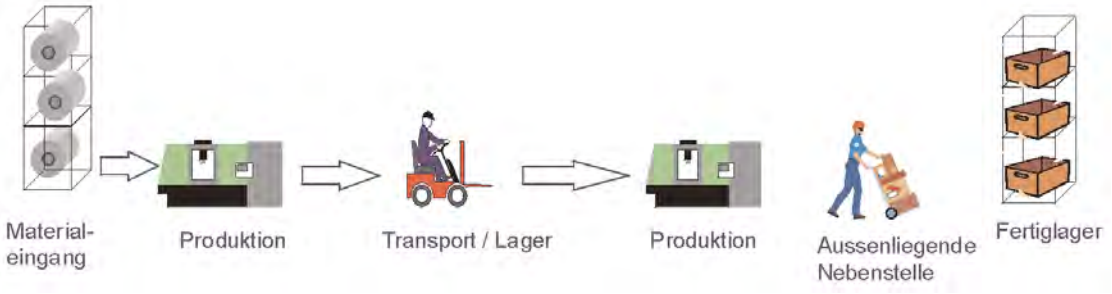

Abbildung 2.15: Ablauf in der Produktion
nach [27]

Der Schwerpunkt der Wieland Werke AG liegt in der Produktion von Fabrikaten aus Kupfer und Kupferlegierungen. Um auf dem Markt konkurrenzfähig agieren zu können, bedarf es einer Optimierung der Produktionsabläufe. Um die Produkton besser planen und steuern zu können, setzt man in der Praxis auf moderne *MES* (*Manufactoring Execution System*) Software.
Ein Manufactoring Execution System ist für die Arbeitsdurchläufe (Abbildung 2.15) in der Fertigung eines Unternehmens zuständig. Es überwacht das Fertigungskontrollsystem, entscheidet über die Folge der Produktionssschritte und wann bzw. wo die Fertigung erfolgen soll. Das System kennt jegliche Konfigurationen der Fertigung und ermöglicht eine Verwaltung der Ressourcen. [30].

Das MES wird in der Automatisierungspyramide unterhalb der produktionsplanenenden Schicht *ERP - Enterprise Resource Planning* angesiedelt, siehe Abbildung 2.16. Es steuert die Durchsetzung einer bestehenden Planung aus dem ERP-System und leitet Rückmeldungen aus dem Fertigungsprozess weiter um somit künftige Planungen verbessern zu können. Dies ermöglicht eine bessere Transparenz und bietet Möglichkeiten Kosten einsparen zu können.

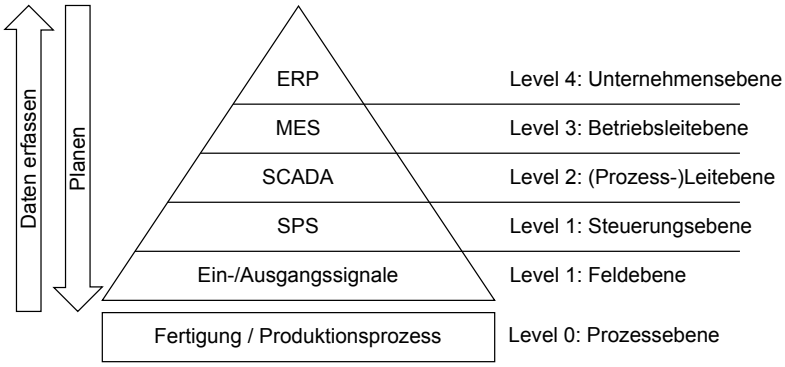

Abbildung 2.16: Vertikale Integration eines MES zwischen ERP System und Prozessleitebene nach [12]

2.4.2 Softwareprodukt „Wieland Factory Suite (WFS)“

In diesem Kapitel wird eine MES-Software aus der Praxis näher vorgestellt. Die Wieland Werke AG haben im laufe der letzten Jahre eine eigene MES-Lösung entwickelt, welche an die Bedürfnisse der Produktionsbereiche angepasst wurde. Dieses Produkt wurde unter dem Namen *Wieland Factory Suite (WFS)* eingeführt. Zum Produkt WFS gehört die WFS Software (Standard- und Sonderlösungen) d.h. alles was in Solutions, Features und Units definiert ist und die Dokumentation.

Die WFS unterliegt nach [19] einer Produktorganisation und ist in 10 *Factory Solutions* aufgeteilt. Diese wiederum setzen sich aus *Factory Features* zusammen. Jede Factory Feature besteht aus *Factory Units*. Diese 3-schichtige Hierachie dient der Kategorisierung und Klassifizierung der MES-Komponenten.

Folgende Factorysolutions (Abbildung 2.17) sind im Prozessmodell der WFS definiert. Jede Solution besitzt einen Solution-Manager (fachliche Verantwortung) und einen Application-Manager (technische Verantwortung). Diese Mitarbeiter sind für die Qualität der Software in ihren Solutions zuständig und beeinflußen jegliche neuen Softwareentwicklungen in diesen Bereichen.

Eine der am meisten genutzen Solutions in der Fertigung ist die Betriebsdatenerfassung BDE. Hier werden die Istdaten über Zustände in der Fertigung direkt am Entstehungsort erfasst und auftragsbezogen weitergeleitet. Erfasste Betriebsdaten für das BDE System können sein:

- Produktionsdaten wie Zeiten, Mengen, Stückzahlen
- Rückmeldung von Arbeitsfortschritt, Auftragsstatus
- Personaldaten der Werker die am Auftrag beteiligt sind

Abbildung 2.17: Factorysolutions in der WFS nach [19]

BDE Komponente: Arbeitsverteilerliste

Um sich ein Bild von der bei Wieland eingesetzten MES-Software machen zu können, stelle ich im Folgenden die zentrale Komponente der BDE vor die *Arbeitsverteilerliste (AVL)*. Bevor der Werker einen Fertigungsauftrag starten kann, werden einige Zwischenschritte vorher ausgeführt. Der Kunde bestellt Waren über das Wielandportal. Anschließen werden diese Informationen an das ERP System (SAP) weitergeleitet. Hier findet die Planung der Aufträge statt. Sind die Aufträge soweit vorbereitet, werden sie von der Softwareintegrationsschnittstelle EAI übernommen. Hier werden die Daten in die Datenbanken der Produktion eingespeist und vom MES-System registriert. Ab jetzt taucht der Auftrag im Softwaredialog Arbeitsverteilerliste

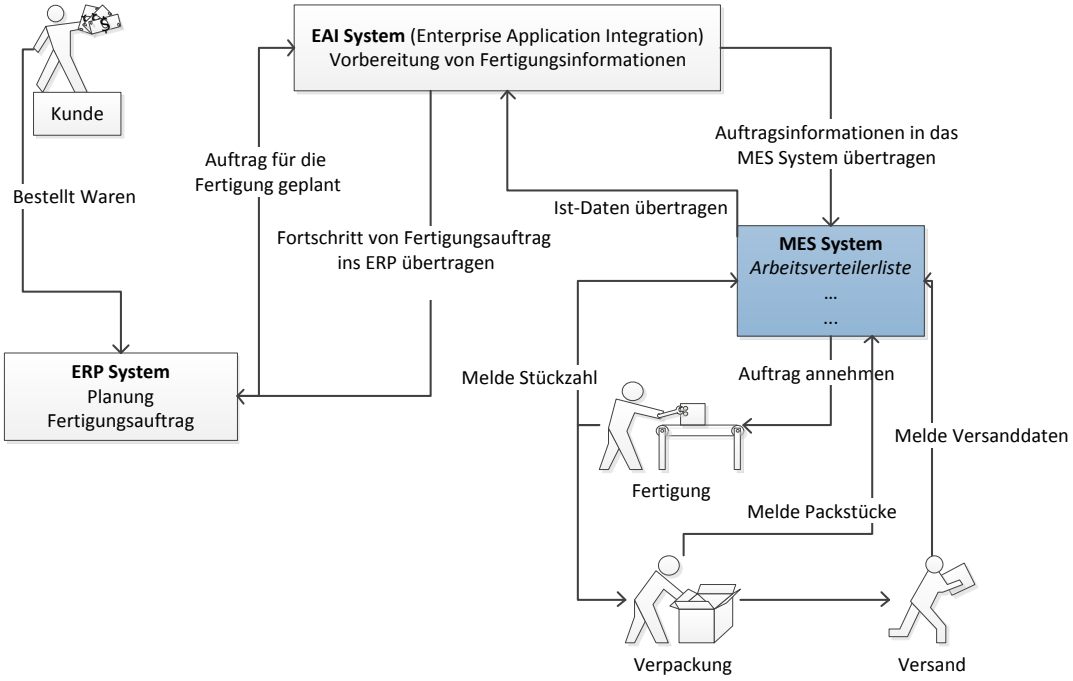

Abbildung 2.18: Ablauf eines Fertigungsprozesses bei Wieland

auf und kann verarbeitet werden. Ist der Auftrag abgeschlossen, wird dies vom Werker in das MES-System eingegeben, danach kann der Auftrag abgeschlossen werden und die Informationen zum Status des Auftrags können in das ERP-System übertragen werden. Der Ablauf eines Fertigungsprozesses bei Wieland könnte etwa wie im Schaubild 2.18 dargestellt, erfolgen.

Der Dialog Arbeitsverteilerliste zeigt je nach eingestellten Filterkriterien, verfügbare Aufträge mit den dazugehörigen Parametern. In dieser Übersicht kann der Werker einen bestimmten Auftrage auswählen und zum Bearbeiten starten. Ab dann werden die nötigen Arbeitsgänge abgearbeitet und die entstandenen Stückzahlen und Mengen anschließen zurückgemeldet. Wie so ein Dialog in der WFS aussieht, kann man dem Schaubild 2.19 entnehmen.

2.4.3 Systemarchitektur einer MES am Beispiel von WFS

Die WFS ist als ein Client-Server-System aufgebaut. Um den Client starten zu können, bedarf es einer virtuellen Umgebung, welche über einen Citrix-Dienst realisiert wird. Der WFS-Client übernimmt die Rolle eines View-Containers während der WFS-Server die fachliche Logik der

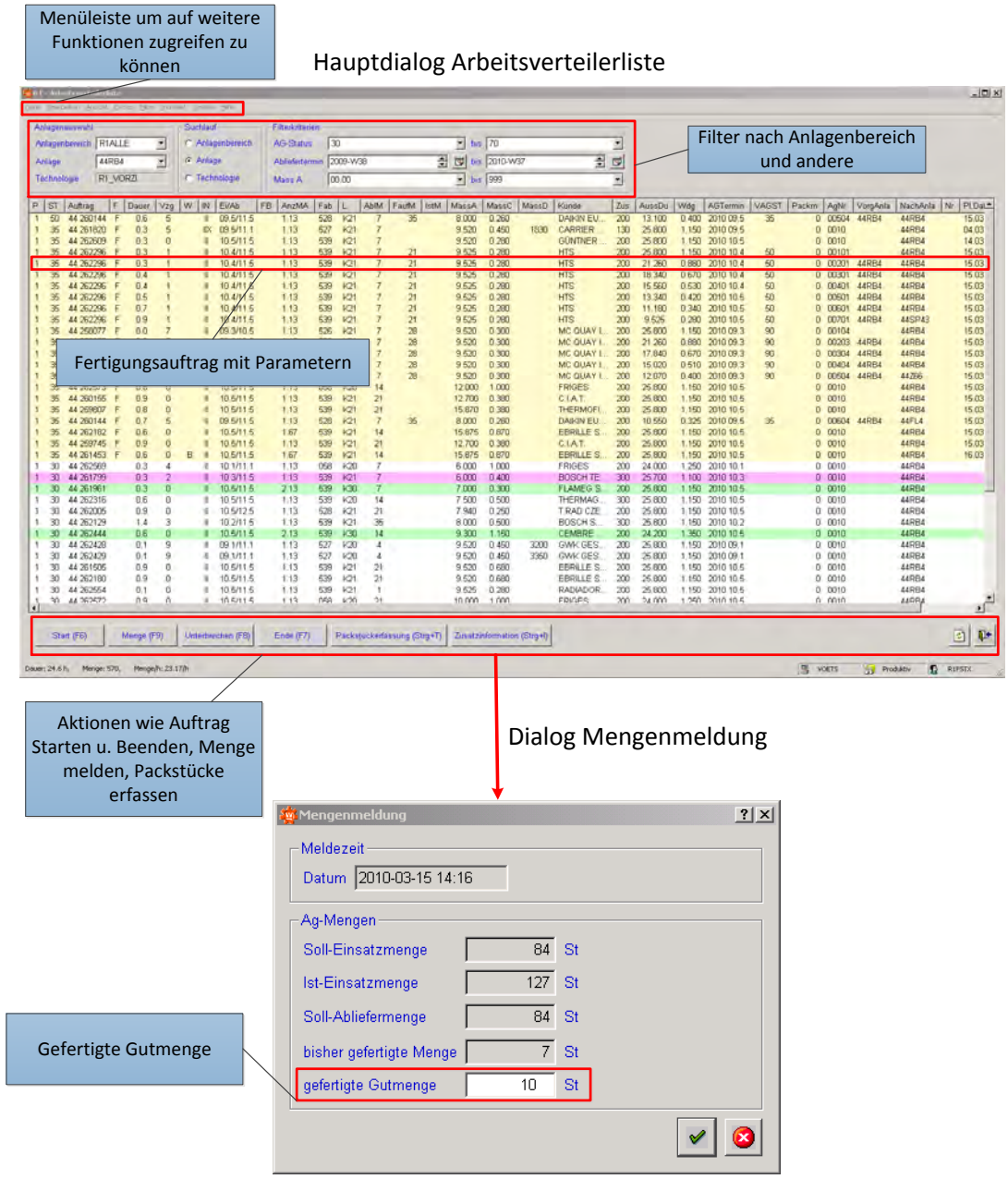

Abbildung 2.19: Hauptdialog Arbeitsverteilerliste

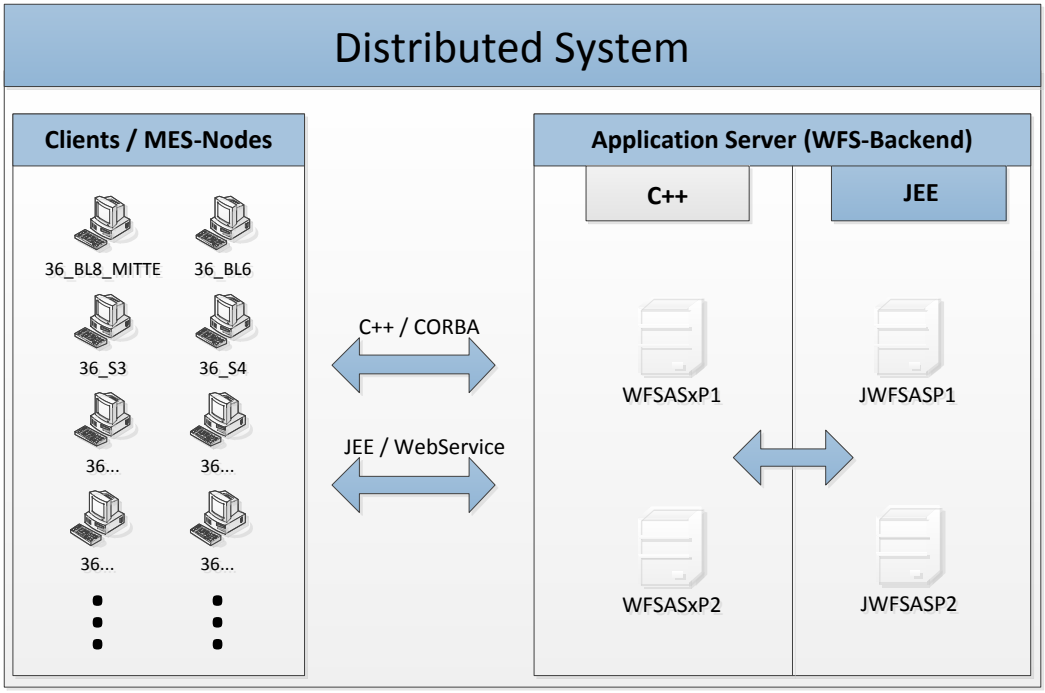

Abbildung 2.20: Wieland Distributed System

Anwendung auführt. Die Serverarchitektur in der WFS basiert auf die *Java Enterprise Edition (JEE)*-Technologie. Zusätzlich existieren noch ältere Komponente auf Corba/C++ Basis. Im weiteren Verlauf der Arbeit wird nur das JAVA-WFS-Backend von Bedeutung sein. Die Clients kommunizieren über einen Webserviceaufruf mit den JBoss-Application-Servern, diese setzen die fachliche Logik einer Anwendung um. Im Schaubild 2.20 wird die Client-Server-Architektur der WFS nochmals genauer dargestellt. Jeder Client der WFS wird einer eindeutigen Benutzergruppe (*MES-Node*) zugeordnet um daraus Berechtigungen ableiten zu können und Konfigurationen vorzunehmen.

Java Architektur der WFS

Java Enterprise Edition definiert eine Softwarearchitektur-Spezifikation für die transaktionsbasierte Ausführung von Java-Anwendungen. Dabei existieren zahlreiche APIs die standardisiert sind und unabhängig voneinander eingesetzt werden können. Stark vereinfacht lässt sich die JEE in drei Schichten unterteilen:

1. Presentation Tier

2. Business Tier
3. Integration Tier

Die Presentation Tier der JEE-Spezifikation besteht lediglich aus dem Webanteil. JSF, JSPs und Servlets kümmern sich um die Veröffentlichung der Geschäftslogik im XML- oder HTML-Format. Die Geschäftslogik wird in der Business Tier durch *Enterprise Java Beans (EJB)* umgesetzt. Diese Schicht ist von besenderer Bedeutung für uns, da damit die Wieland-Businesslogik in der WFS umgesetzt wird [8]. Durch diese Mehrschichtenarchitektur ist es möglich eine klare Aufgabentrennung zu realisieren. Zudem lassen sich die entwickelten Komponenten einfacher wiederverwenden und erzielen eine höhere Schnittstellen-Stabilität. Dabei werden die technischen Aspekte einer Anwendung in einen Container verlagert, während die fachlichen Aspekte in Form einer Komponente realisiert werden. Wurde eine Komponente implementiert, ist der nächste Schritt das Deployment auf einen Application Server. Bei Wieland wird als Application Server JBoss AS eingesetzt. Die relevantesten Technologien aus dem JEE Standard die im WFS-Umfeld eingesetzt werden, sind:

- Enterprise JavaBeans (EJB)
- Java WebServices
- Java Persistence API
- Java Connector Architecture

Auf ausführliche Beschreibung dieser Technologien wird hier bewusst verzichtet, da die technische Umsetzung durch die MDSD in den Hintergrund gestellt wird.

Drei Schichten der WFS-Serverarchitektur

Wie soeben skizziert, lassen sich Java Server Anwendungen durch eine drei Schichten Architektur realisieren. Dieser Ansatzt kommt bei der WFS ebenfalls zum Einsatz. Im Folgenden werden die Einzelnen Komponenten dieser Architektur erläutert, anhand des Schaubilds 2.21 erkennt man die wichtigsten Architekturbestandteile der WFS. Die serverseitige Architektur besteht aus drei Schichten, deren Interaktionen, Funktionen und Bestandteile im Folgenden erläutert werden:

Serverseitige Präsentationsschicht ist nach [18] zuständig für die Aufbereitung der Daten für den Client bzw. Rückumwandlung in Java-Objekte. Realisiert wird diese durch eine *Client-Gateway*-Komponente in einem EJB-Container mit einem WebService-Interface. Der *Dispat-*

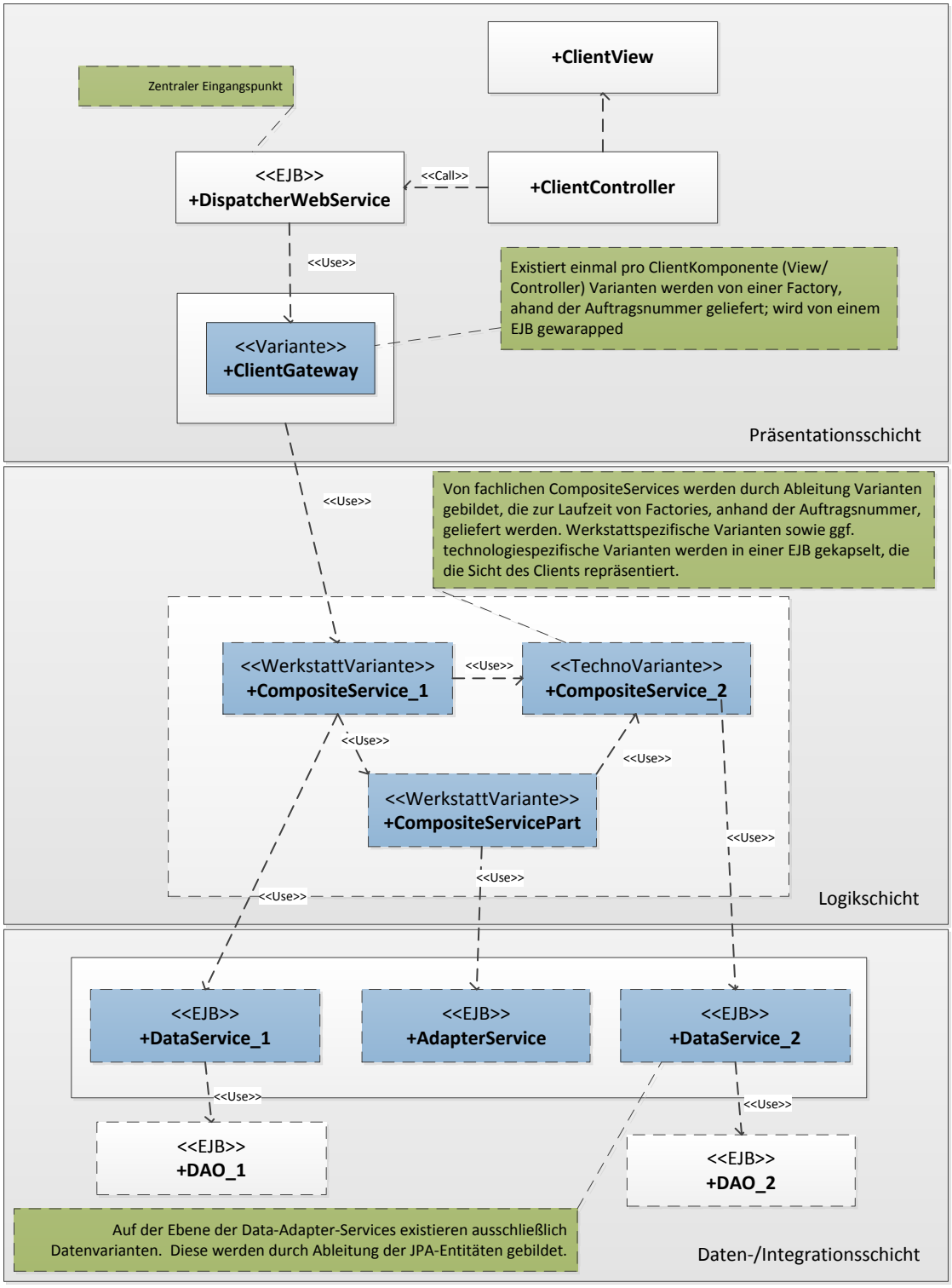

Abbildung 2.21: WFS-Serverarchitektur

cherWebservice (Schnittstelle zum Client) nimmt die Aufrufe des Clients entgegen und ermittelt das zuständige ClientGateway. Dieses Gateway ermittelt die zuständige Variante, beispielsweise anhand der Auftragsnummer und konvertiert die erhaltene XML-Struktur in Java-Objekte. Zur Abarbeitung der Anfrage werden diese Objekte im Rahmen eines Methodenaufrufs an das zuständige CompositeService-Objekt weitergereicht. Das Ergebnis wird wieder in XML umgewandelt und über den DispatcherWebService an den Client zurückgegeben.

Logikschicht vereint mehrere Service-Aufrufe im Rahmen eines fachlichen Prozesses, innerhalb einer Transaktion. Realisiert wird diese durch eine *Composite-Service*-Komponente in einem EJB-Container mit einem Local-Business-Interface. Im Gegensatz zu Data- bzw. Adapter-Services, die eine wiederverwendbare Funktionalität realisieren, bilden Composite-Services nach [18], innerhalb der Prozess-Schicht einen konkreten Anwendungsfall eines fachlichen Prozesses ab. Sie nutzen dazu die vorhandenen Services, wobei ein Aufruf einer Composite-Methode in mehrere Service-Aufrufe münden kann. Zudem können von den Composite-Services unterschiedliche Varianten zur Laufzeit instanziert werden, um dabei die Implementierung austauschen zu können.

Daten- und Integrationsschicht ist zuständig für die Realisierung einer fachlich klar definierten und abgeschlossenen Zuständigkeit (z.B. Verwaltung von Aufträgen) Dies erfolgt mittels einer *Data-Service / Adapter-Service*-Komponente in einem EJB-Container. Ein *Data-Service* umfasst eine fachlich abgeschlossene, klar definierte Zuständigkeit. Er referenziert keine anderen Services und kennt nur das Datenmodell, für das er selbst verantwortlich ist. Der Service realisiert den rein fachlichen Anteil seiner Domäne. Die gesamte Persistenzlogik ist in DataAccessObjekte (DAOs) ausgelagert. Dadurch wird der Service „persistenztechnologiefrei“. Der Service arbeitet auf Domänenobjekten (genauer: Datentransferobjekten), die er über seinen DAO erhält.

Ein DAO nutzt JPA (Java Persistence API)-Entitäten um Daten in die Datenbank zu persistieren bzw. von dort zu laden. Er führt ein Mapping zwischen Entitäten und Data Transfer Objects (DTOs) durch. Der Zugriff auf die Datenbank (Persistierung, Abfragen, etc.) erfolgt über JPA (genauer: über einen EntityManager). Die Abbildung der Objekte in die Datenbank erfolgt innerhalb der Entitäten, über die Angabe der entsprechenden JPA-Annotationen.

Ein Adapter-Service bietet eine ähnliche funktionalität wie ein Data-Service, hinzu kommt die Eigenschaft, andere Services referenzieren zu können um aus Fremdsystemen Daten auslesen zu können und per Data-Service-Aufruf zu persistieren, nach[18].

Ablauf und Interaktion der Schichten

Der C++-Client kommuniziert auf Basis von XML-Dokumenten über eine Webservice-Schnittstelle mit dem JEE-Backend. Dieses XML-Modell wird in der serverseitigen Präsentationsschicht in Datenmodell-Objekte umgewandelt. Anschliessend wird die jeweilige Methode an der zuständigen CompositeService-Implementierung aufgerufen. Innerhalb dieses Methodenaufrufs wird die Abarbeitung an den beteiligten Services (Data- oder AdapterServices) angestossen. DataServices schreiben über DataAccessObjects (DAOs) Daten in die Datenbank bzw. lesen sie aus dieser. Innerhalb dieser erfolgt das Mapping von Datentransferobjekten in JPA-Entitäten. Die Abbildung auf die Datenbanktabellen erfolgt innerhalb der Entitäten. AdapterServices integrieren fremde Systeme, wie beispielsweise Waagen und andere Messgeräte [18].

2.4.4 Modellgetriebene Softwarearchitektur in der WFS

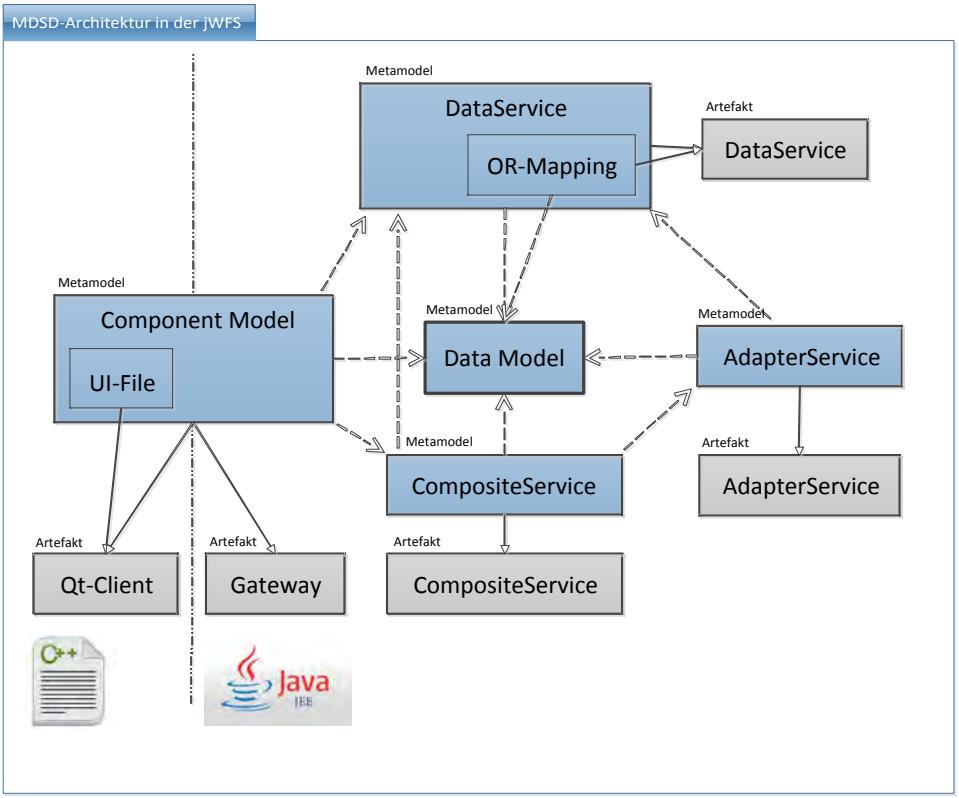

Abbildung 2.22: modellgetriebene WFS Architektur

Im Rahmen der Modellgetriebenen Entwicklung in der WFS, steht am Anfang die Modellierung des gewünschten Dienstes bzw. Artefakts. Die Anforderung ist, die im Kapitel 2.4.3 eingeführten Komponenten in formalen Modellen abbilden zu können. Hierbei wird mittels textueller DSL das gewünschte Artefakt fachlich beschrieben. Die DSLs unterliegen bestimmten Metamodellen wie in der Abbildung 2.22 gezeigt wird. Jedes erzeugte Artefakt (Gateway, Composite-Service, Data-Service) instanziert ein solches Metamodell. Das textuelle Modell dient als Input für den Generator, der aus dem Modell, den entsprechenden Java-Code (M2T-Transformation) erstellt und somit die Implementierung der technischen Infrastruktur (Plattform) übernimmt.

MDSD-Plattform OpenArchitectureWare

Um formale Modelle erstellen und verarbeiten zu können, verwendet man in der WFS das Generator Framework *OpenArchitectureWare (OAW)*. Es ist Teil der Eclipse Foundation und gehört dem *Generative Modeling Technologies (GMT)* Projekt an. OAW verwendet existierende Eclipse Projekte wieder und integriert sich nahtlos in die Eclipse-IDE. Das Framework bietet einem Modellprüfungen, Transformationen und Generierung von Programmcode an. Das Erzeugen einer DSL wurde in unserem Fall durch das Framework *Xtext* gelöst. Mit Xtext können nicht nur Modellierungssprachen erzeugt werden, sondern auch spezifische Texteditoren für Eclipse, die u.a. eine Syntaxhervorhebung sowie eine Überprüfung von definierten Constraints implementieren [15]. Im folgenden Code-Ausschnitt sieht man ein textuelles Modell aus der WFS, zu sehen ist ein Composite-Service-Modell das für das Bedrucken von Chargenetiketten zuständig ist.

```
1  import 'platform:/resource/wfs-model/src/main/resources/model.wfs';
2
3  import 'platform:/resource/wfs-bde-chargenetikett-ds-api/src/main/resources/
       Chargenetikett.wfsdm';
4  import 'platform:/resource/wfs-bde-chargenetikett-ds-api/src/main/resources/
       Chargenetikett.wfssv';
5
6  import 'platform:/resource/wfs-framework-print-as-api/src/main/resources/Print.wfssv';
7
8  import 'platform:/resource/wfs-framework-print-ds-api/src/main/resources/Print.wfssv';
9
10 compositeservice Chargenetikett {
11   solution BDE;
12
13     dependencies {
14        data Chargenetikett;
15        data Print;
```

```
16
17        adapter Print;
18      }
19
20    services {
21      void druckeChargenetikett(enum ChargenetikettenTyp typ, ChargenetikettDaten daten,
            long anzahl, long anzahlAusdrucke) throws ChargenetikettCS;
22    }
23
24    // Fehlernummern: 22050 - 22069
25    exception ChargenetikettCS {
26      id 22050 PRINT(exceptionId, exceptionMessage);
27       id 22051 PRINTER_ASSIGNMENT(exceptionId, message);
28    }
29  }
```

Die verwendete Template-Engine in OAW ist das Framework *Xpand*. Diese Engine dient der Generierung von Ausgaben auf Basis von Templates. Templates sind Schablonen, die den Text enthalten, welcher während der M2C-Transformation in die Ausgabedateien geschrieben wird. Für die Beschreibung der Transformationen und zur Metamodellierung wird im OAW die Komponente *Xtend* verwendet. Hier werden sogenannte *Extentions* definiert, die einen bestimmten Zweck erfüllen wie z.B. Verwaltung von Datenstrukturen. Diese können dann z.B. aus den Xpand-Templates aufgerufen werden. Die letzte Komponente die im WFS-Generator-Framework zum Einsatz kommt, ist die *Check*-Komponente. Diese wird im Rahmen der Modellvalidierung eingesetzt. Mit ihr kann die statische Semantik umgesetzt werden. Dazu werden die Elemente ausgewählt, die einem Constraint der statischen Semantik unterliegen. Anschließend folgt die Formulierung des Constraints. Ein solches könnte beispielsweise die Eindeutigkeit eines Attributs einer bestimmten Klasse von Elementen verlangen. Im Schaubild 3.6 wird der Weg von der Modellierung bis hin zum Programmcode veranschaulicht.

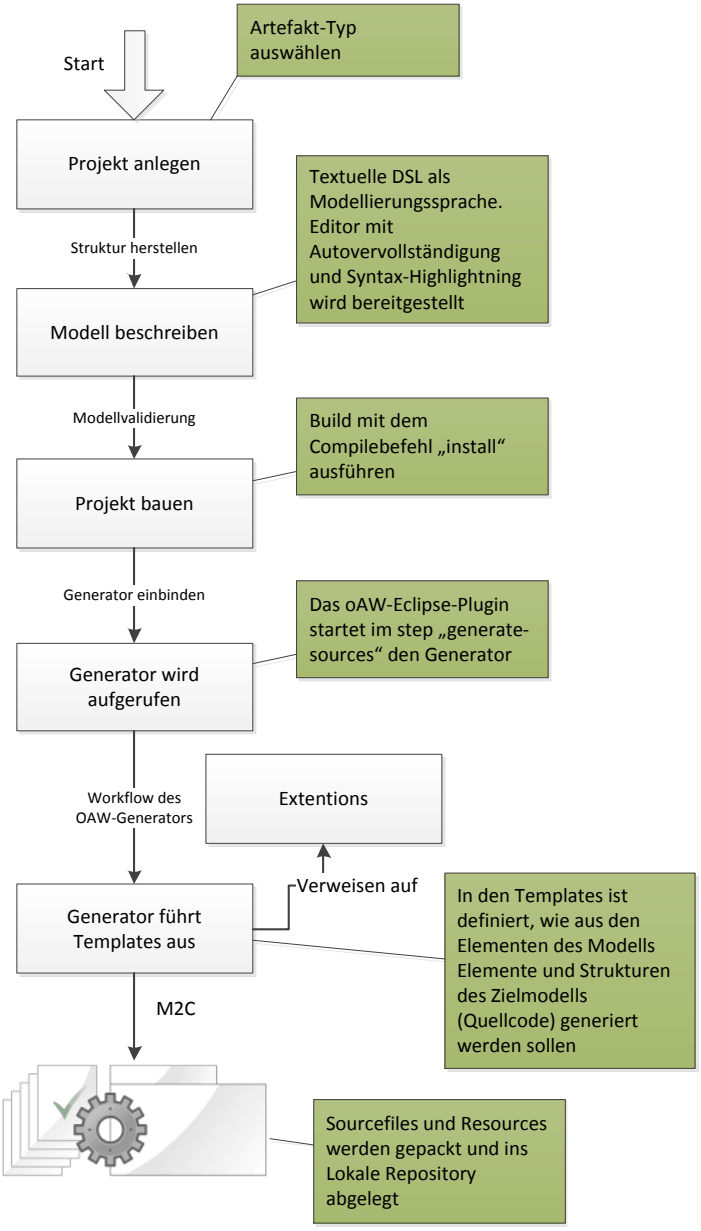

Abbildung 2.23: Softwareentwicklungsprozess im Generatorumfeld der WFS

3 Problembeschreibung und Zielsetzung

MES-Softwaresysteme sind mehrschichtige Gesamtsysteme welche die Fertigungsprozesse eines Unternehmens abdecken. Hierbei können die einzelnen Softwarelösungen in ihrer Funktionalität erweitert bzw. angepasst werden. Dabei entstehen sowohl fachliche- als auch technische Anforderungen an das System [27]. Je nach Umfang des MES-Systems, ergeben sich größere Softwareprojekte die umgesetzt werden müssen. Hierbei ergeben sich unterschiedliche Projektbeteiligte, die sich für den Verlauf oder das Ergebnis des Projektes interessieren. Diese werden in der betriebswirtschaftlicher Sicht als *Stakeholder* bezeichnet.
In den modellgetriebenen MES-Systemen werden fachliche Abläufe mit Hilfe von abstrakten Modellen beschrieben. Dies führt dazu, dass immer unterschiedliche Mitarbeitergruppen mit formalen Modellen in Berührung kommen. Diese Modelle werden in der Praxis häufig in textueller Form mit Hilfe einer DSL beschrieben. Dabei wird der gewünschte Abstraktionsgrad häufig nicht erreicht. Je nach Kenntnisstand eines Mitarbeiters, entstehen Probleme im Umgang mit diesen Modellen. Das führt dazu, dass die Kommunikation zwischen den Projektbeteiligten erheblich beeinträchtigt wird. Um dieses Problem zu lösen, bedarf es einer passenden Abstraktion zur Beschreibung verschiedener Sachverhalte. Im folgenden werden Probleme im Umgang mit den MDSD-Technologien am Beispiel von der WFS-MES-Software veranschaulicht und ein Lösungsansatz präsentiert.

3.1 Problemanalyse im MDSD-Umfeld der WFS

Die Wieland Factory Suite ist ein Softwareprodukt mit dem mehr als 1400 Benutzer täglich in Berührung stehen. Einer der Hauptaufgaben der Wieland IT-Abteilung ist die Entwicklung von Software, die Organisation und Modellierung der zugehörigen formalen Modelle und der Betrieb von WFS-Softwaresystemen. Aufgrund des hohen Aufwandes zur Erstellung und Pflege komplexer MES-Software erfolgt die Entwicklung bei Wieland anhand eines Prozessmodells 3.1.

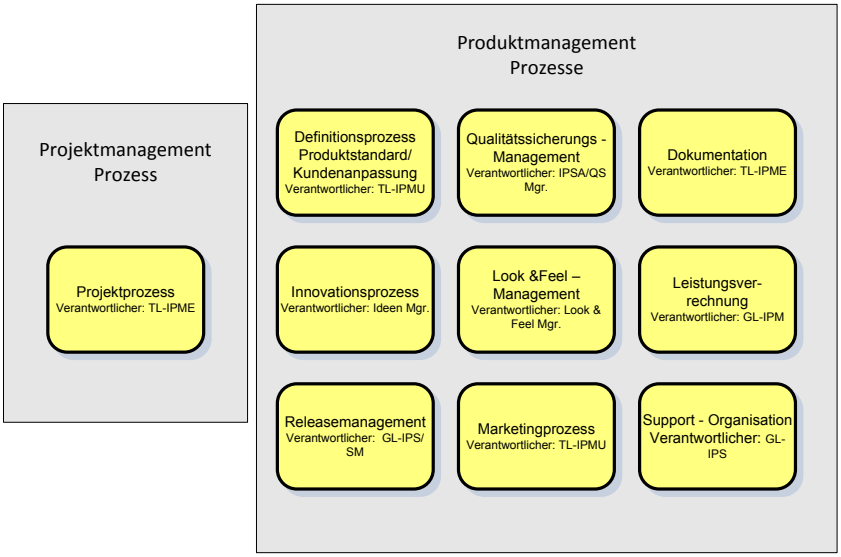

Abbildung 3.1: WFS-Prozessübersicht

Das WFS-Prozessmodell ist nach [19] als Leitfaden zum Planen und Durchführen von Entwicklungsprojekten unter Berücksichtigung des gesamten Systemlebenszyklus konzipiert. Dabei definiert es die in einem Projekt zu erstellenden Ergebnisse und beschreibt die konkreten Vorgehensweisen mit denen diese Ergebnisse erarbeitet werden. Darüber hinaus legt das es die Verantwortlichkeiten der Projektbeteiligten fest. Das WFS-Prozessmodell regelt also detailliert, „Wer“ „Wann“ „Was“ in einem Projekt zu tun hat. Aus dem Prozessmodell lassen sich die verschiedenen Stakeholder eines Softwareentwicklungsprozesses entnehmen, siehe Abbildung 3.1.1. Die für uns relevantesten Beteiligten werden im folgenden Kapitel beschrieben, wobei Anzahl und der Einfluss der jeweiligen Gruppen unterschiedlich ausfällt. Verbunden werden sie durch eine gemeinsame Domäne. Die WFS-Software-Domäne wird in vielen Teilen durch textuelle Modelle abgebildet, sowohl fachlich als auch technisch. So werden beispielsweise fachliche Domänenobjekte wie Datenstrukturen aus der Fertigung, mittels Data-Services modelliert. Dieses Vorgehen bietet viele Vorteile da man nicht mehr ausschließlich auf den Datenbanktabellen agieren muss. Trotzdem bereitet diese textuelle Form der Darstellung einigen Mitarbeitern Verständnisschwierigkeiten. Der Detailgrad dieser Modelle verhindert eine übersichtliche Abbildung der fachlichen Struktur. Um das Problem zu beheben, wurden in der Vergangenheit UML-Struktrudiagramme angefertigt (siehe Abbildung 3.3), diese weisen aber an vielen Stellen Mängel auf. Die größte Schwierigkeit besteht in der Pflege dieser Diagramme.

Abbildung 3.2: Stakeholder im WFS-Entwicklungsprozess

Zudem bilden diese Diagramme nur einen geringen Teil der Domäne und sind nicht aus den textuellen Modellen abgeleitet.

Deshalb haben die beteiligten Gruppen in der Praxis häufig mit Kommunikationsschwierigkeiten zu kämpfen, worauf im Abschnitt 3.1.2 anahand von Praxisbeispielen näher eingegangen wird.

3.1.1 Stakeholder

An einem Softwareentwicklungsprojekt sind zahlreiche Menschen beteiligt die unterschiedliche Rollen mit ganz unterschiedlicher Handlungsmotivation einnehmen. Neben Entwicklern und Projektleitern existieren noch weitere Gruppen. Dabei verfügen die beteiligten Mitarbeiter über unterschiedliche Qualifikationen im Bereich der MDSD. So verfügen Mitarbeiter des Frameworks, Softwareentwickler und die Solution-Designer über eine IT-Ausbildung. Die Mitarbeiter welche die Rolle der Projektleiter, Application-Manager, Businnes-Consultant, Solution-Manager oder des Kunden einnehmen, verfügen über eine kaufmännische Ausbildung im Bereich Produktionstechnik. Das bedeutet, diese Mitarbeiter sind oft nicht in der Lage die softwaretechnische Aspekte einer Anwendung zu verstehen.

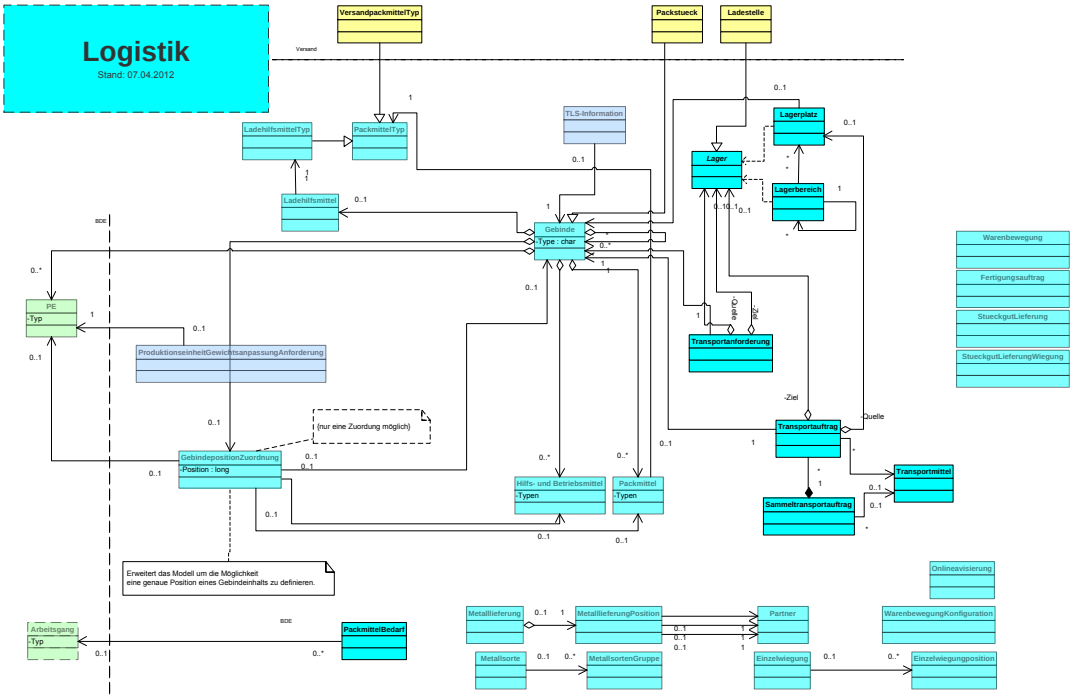

Abbildung 3.3: Domänenmodell grafisch [19]

Im folgenden werden die verschiedenen Stakeholder in ihrer Rolle näher beschrieben:

Solution-Designer sind Experten für eine Solution. Sie verfügen über den Softwaretechnischen Überblick ihrer Solution und sind für die Qualität des Softwarebestands zuständig. In Projekten setzten sie sich für Standardisierungslösungen in den betrfeffenden Solutions ein.

Solution-Manager ist der Verantwortliche für die Factory Solution und damit zuständig für den Aufbau und die Struktur der Module. Weitere Aufgaben liegen in der strategischen Ausrichtung der Solution und Marktanalyse zu möglichen Softwarelösungen. In den Projekten trägt der Solution Manager die Verantwortung für die Dokumentation neuer Factory Units / Factory Features und deren korrekten Zuordnung in der Solution.

Framework-Mitarbeiter erarbeiten und betreuen die Basis-Bibliotheken/Tools zum Thema Datenverarbeitung, Programmierschnittstellen für Anwendungsprogrammierer und Anbindung

an Applikationslogik. In Projekten unterstützen Sie die Software-Entwickler in Themen wie Strukturierung, Schnittstellen und Testbarket der Software.

Auftraggeber legt Abnahmekriterien für die Requierement-Spezifikation und den Projektabschluss fest. Eine weitere Aufgabe besteht darin, für ausreichende Bereitstellung, der für die Projektabwicklung benötigten Mitarbeiter seines Bereiches zu sorgen. Der Auftraggeber kann diese Aufgaben an einen Mitarbeiter seines Bereiches delegieren. Die Mitarbeiter die diese Rolle einnehmen sind aus der Produktionabteilung und kennen sich besonders gut mit den technischen Abläufen aus.

Application Manager ist verantwortlich für alle MES-Anforderungen und Beratungsleistungen eines Bereiches. Sie definieren strategische Ausrichtungen in ihren Bereichen und organisieren die Releases der WFS. In Projekten können sie bei fachlichen Beschreibungen Hilfestellung leisten.

Projektleiter werden ausschließlich projektbezogen in ihrer Aufgabenbeschreibung definiert. Demnach hat der Projektleiter die Hauptverantwortung für die Abwicklung von IT-Projekten und damit auch für die Budget– und Terminplanung. Er sorgt im weiteren dafür, dass die erforderlichen Prozesschritte zur Softwareerstellung ausgeführt werden und dass alle betroffenden Mitarbeiter integriert und informiert werden. Im weiteren ist der Projektleiter für die Beauftragung der Reviews in den verschiedenen Softwareentwicklungsphasen zuständig.

Business Consultant trägt fachliche Projekt-Verantwortung für die zu erarbeitende Projektlösung. In der Requirement-Phase organisiert er die externe Abnahme des Requierement-Dokuments in Abstimmung mit dem Projektleiter. Im weiteren ist er für die Erfassung von Nachforderungen im Projekt zuständig. Am Ende der Implementierungsphase veranlasst er den Deploy auf das Development-System, Quality-System und Rollout-System.

Softwareentwickler arbeiten an dem Design und der Implementierung der MES-Software. Ihr Hauptaugenmerk liegt in der technischen Umsetzung von Anforderungen in einem Softwareprojekt. Sie verfügen über die nötigen Kenntnisse in der Softwaretechnik.

3.1.2 Kommunikationprobleme im MES-Softwareentwicklungsprozess

Im folgenden werden einige Anwendungsszenarien aus dem täglichen Softwarebetrieb der Wieland IT-Abteilung beschrieben, die auf Probleme im Umgang mit den textuellen Modellen hinweisen. Diese Szenarien leiten sich aus dem WFS-Entwicklungsprozessmodell ab, siehe Schaubild 3.4.

Requirementsphase Möchte ein Wieland-Kunde neue Anforderungen an die Software umsetzten, wird ein Softwareprojekt geplant. Dabei wird der Kunde(Werksleitung) mit einem Produktmanager (Business Consultant) in Kontakt treten und die fachlichen Aspekte der Anforderungen durchsprechen. Anschließend wird für das bevorstehende Projekt eine Abschätzung gemacht, um die Kosten zu kalkulieren. Ein Buisness Consultant entwirft einen Anforderungskatalog mit Funktionalen- und nicht-funktionalen Anforderungen. Hierbei ist es wichtig die bestehende Software zu analysieren um die betreffendenden Factory-Units identifizieren zu können. Die Ist-Analyse beinhaltet eine Betrachtung der bestehenden Modelle aus der Domäne. An dieser Stelle haben unsere projektverantwortlichen Mitarbeiter wie Business Consultants und Projektleiter Probleme die Auswertung durchzuführen, da ihnen das technische Wissen im Bereich textueller Modellierung fehlt.

Analysephase Im laufe eines Softwareentwicklungsprozessen finden nach dem Prozessmodell Analysesessions statt um die Umsetzung der Anforderungen vorzubereiten. In der Abbildung 3.5 wird es anhand eines Ausschnittes aus dem Prozessmodell deutlich gemacht. In den Analyse-Sessions müssen bei mindestens einer Veranstaltung der Solution Designer, ein Framework Mitarbeiter und der Business Consultant/Projektleiter beteiligt sein. Bei Bedarf kann auch der Solution Manager oder der Kunde an einer der Analyse-Sessions teilnehmen. Dabei werden die fachlichen Anforderungen identifiziert und auf Umsetzbarkeit geprüft. Hierbei wird geprüft welche Modelle aus der Domäne man wiederverwenden kann und welche Modelle hinzugefügt werden müssen. Da auch Mitarbeiter an den Analysesessions mitwirken, die wenig technisches Know-How mitbrigen, entstehen Kommunikationsprobleme zwischen den Beteiligten.

Inbetriebnahmephase Bei Wieland ist es üblich, dass zwei bis drei mal im Jahr neue Softwarestände in der Produktion aufgespielt (ausgerollt) werden. Dies bedeutet dass kurz vorher ein lauffähiger Stand vom Head-Repository abgezogen werden muss und ggf. die vorhanden Fehler

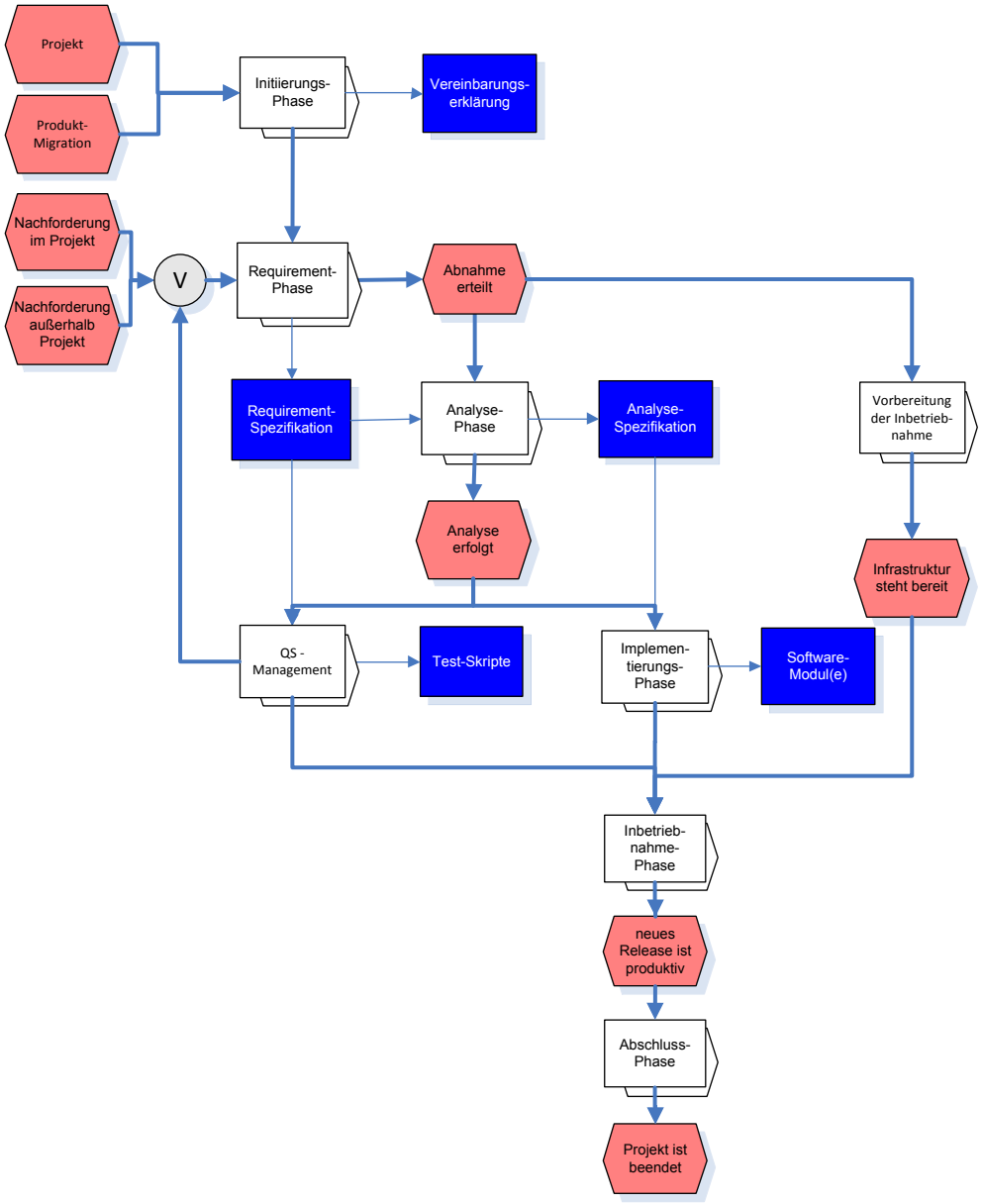

Abbildung 3.4: WFS-Entwicklungsprozessmodell [19]

beseitigt werden müsen. Dieser gesamte Vorgang wird durch einen Business Consultant initiiert. Dabei arbeiten sowohl Frameworkmitarbeiter, Softwareentwickler als auch die Projektleiter zusammen. Der Projektleiter entscheidet darüber, wann der Softwarestand sicher genug ist, um ausgerollt werden zu können. Wird die Software an die Kunden verteilt, wird die Stabilität des

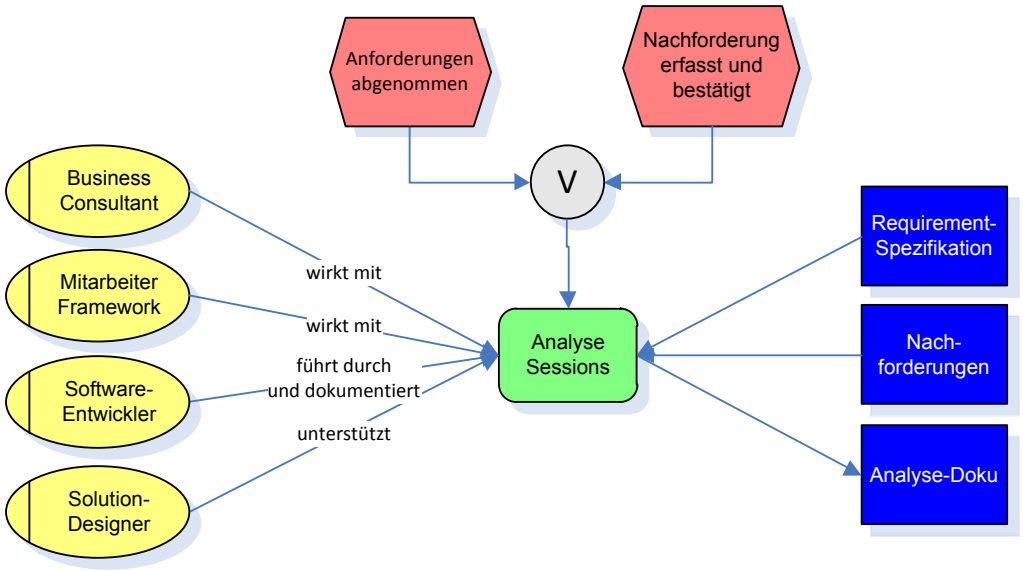

Abbildung 3.5: Ausschnitt aus dem Prozessmodell Analysephase [19]

Systems geprüft um ggf. eingreifen zu können. Für die Softwarequalität sind die für die Inbetriebnahme eingeteilten Projektleiter zuständig. Werden während dieser Phase Softwarefehler gefunden, so werden diese unmittelbar behoben. Fachliche Fehler leiten sich sehr häuftig aus den textuellen Modellen ab. Um diese Fehler schnell identifizieren zu können, bedarf es einer Erfassung und Analyse relevanter Artefakte der betreffenden Komponenten, womit man wieder vor dem bestehenden Problem steht, die Modelle verstehen zu können.

3.2 Lösungsansatz

Die im Rahmen von Softwareprojekten enstandenden textuellen Modelle sind wichtige Bestandteile eines modellgetriebenen MES-Systems. In diesen Modellen werden sowohl fachliche-, als auch technische Abläufe der Software beschrieben. Während eines Softwareentwicklungsprozesses arbeiten unterschiedliche Mitarbeitergruppen mit diesen Modellen. Die textuelle Struktur der Modelle sowie der Detailgrad dieser Beschreibungungen bereiten Mitarbeitern die über wenig IT-Wissen verfügen Verständnisprobleme. Die Lösung für dieses Problem stellt eine grafische Repräsentation der formalen Modelle dar. Diese werden in einem Repository verwaltet und müssen zur Verarbeitung zuerst lokal verfügbar gemacht werden. Aus den textuellen Mo-

dellen werden die relevanten Informationen extrahiert und in grafischer Form abgelegt. Dabei sollen mögliche Darstellungsformen analysiert werden und ein Mapping von Modellelementen auf ein grafisches Diagramm entwickelt werden. Um die grafischen Modelle anzeigen zu können, werden unterschiedliche Visualisierungstools unter die Lupe genommen, aus denen sich für eine Lösung entschieden wird. Anschließend werden Beschreibungssprachen untersucht um Transformationen in einem Generator erstellen zu können.
Die Applikation zur grafischen Darstellung von textuellen Modellen wird unter den Aspekten der Softwareergonomie entwickelt, um dem Benutzer Komfort bieten zu können. Eine wichtige Anforderung an die Applikation ist die automatische Aktualisierung der grafischen Modelle und die Eingliederung in den WFS-Softwarelifecycle. Die einzelnen Schritte zur Generierung graphischer Modellrepräsentationen werden in der Abbildung 3.6 verdeutlicht und im nächsten Kapitel im Einzelnen beschrieben.

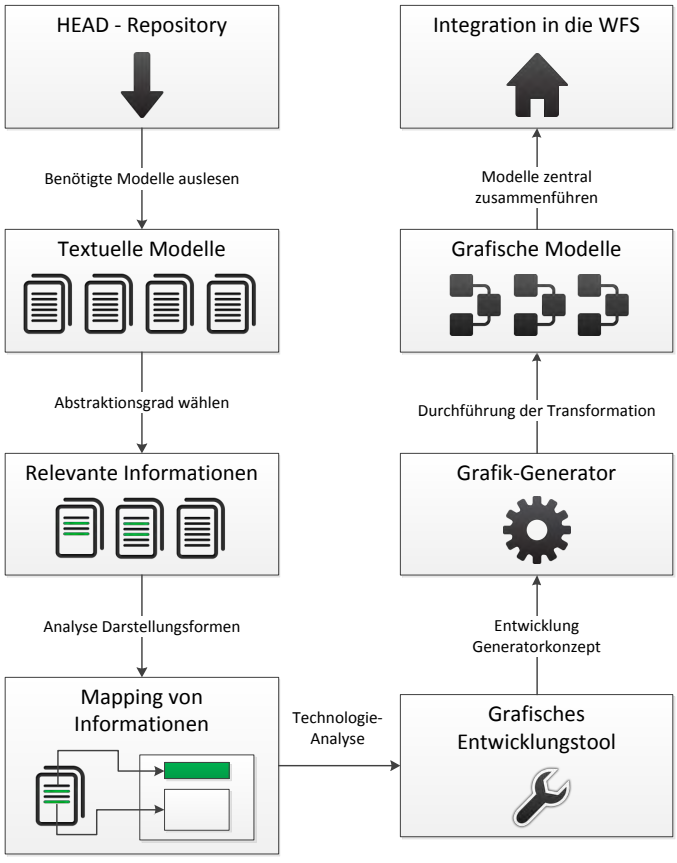

Abbildung 3.6: Lösungsansatz zur Generierung grafischer Modellrepräsentationen

4 Lösungskonzepte und –technologien

In diesem Kapitel werden Lösungsansätze zur Generierung grafischer Modellrepräsentationen aus textuellen Modellen vorgestellt. Aus der Problembeschreibung im Kapitel 3 geht hervor, dass viele textuelle Modelle in der MES-Softwareentwicklung zu umfangreich beschrieben werden. Um ein besseres Verständnis von den Modellen zu bekommen, bedarf es oftmals einer weiteren Abstraktionsschicht. Diese ermöglicht einen höheren Abstraktionssgrad indem ein weiteres Domänenmodell aus dem Urspungsmodell abgeleitet wird. In diesem Kapitel wird der Vorgang zu einer abstrakten Darstellung am Beispiel von textuellen Modellen der WFS erläutert. Hierbei werden die einzelnen Informationselemente der Metamodelle auf Relevanz untersucht und gefiltert. Anschließend werden mögliche Darstellungsformen untersucht, um textuelle Modelle grafisch abbilden zu können. Im Anschluss wird eine mögliche grafische Darstellung der unterschiedlichen Modelle in der WFS anhand von Mocks vorgestellt. Dabei werden die gefilterten Elemente in einer grafischen Form visualisiert. Um Grafiken erstellen zu können, werden spezielle Tools benötigt, diese werden im Kapitel Technologieanalyse im direkten Vergleich dargestellt. Schließlich wird ein Entwurf für den Generator gezeigt, welcher die textuellen Modelle verarbeitet und mit Hilfe von Transformationen die gewünschten Grafiken erzeugt. Das Ganze wird unter der Berücksichtigung der MDSD-Archtektur des WFS-Umfeldes entworfen, um eine Integration in den Software-Lifecycle zu ermöglichen.

4.1 Bestimmung der Abstraktionschicht anhand von Metamodellen

Da textuelle Modelle häufig zur direkten Programmcodeerzeugung verwendet werden, enthalten sie viele irrelavante Informationen die der Generator zum Erzeugen von Code benötigt. Dabei werden fachliche Informationen vom Anwender häufig schlechter wahrgenommen. Dies führt dazu, dass sich IT-Laien mit dem Verstehen von Modellen schwer tun. Die MDA beschreibt in ihrem Standard mehrere Modellebenen, um dieses Problem zu lösen. Dieser Ansatz wird im Folgenden realisiert, indem zusätzliche Modelle abgeleitet werden, die nur die benötigten Infor-

mationen beinhalten. Zudem wird eine grafische Darstellungsform gewählt um den Zugang zu den Informationen zu erleichtern.

Um einen höheren Abstraktionsgrad aus den vorhandenen textuellen Modellen zu erzielen, bedarf es bestimmter Modelltransformationen. Eine Modelltransformation erzeugt mithilfe eines Generators, aus einem oder mehreren Quellmodellen, das erforderliche Zielmodell. Dabei werden auf der Meta-Ebene Transformationsregeln definiert. Wie im Kapitel 2.2.3 beschrieben, charakteresieren Metamodelle die Struktur einer Domäne und werden in Modelltransformationen als Parameter übergeben. Um Transformationsregeln definieren zu können, werden Quell-Metamodelle analysiert und die gewünschten Elemente für die grafischen Modelle gesammelt. Diese Informationsfilterung wird im Folgenden an textuellen Modellen aus der WFS durchgeführt. Die Filterkriterien wurden in Zusammenarbeit mit der IT-Abteilung Wieland festgelegt. Die relevanten Artefakte mit den dazugehörigen Metamodellen werden in der Abbildung 4.1 veranschaulicht.

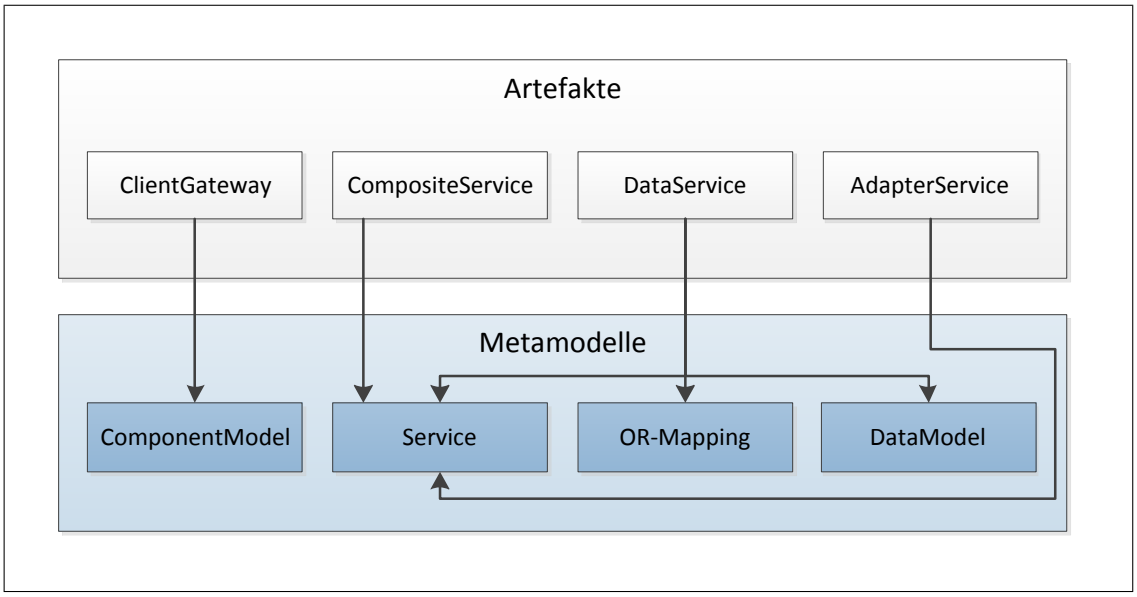

Abbildung 4.1: Relevante Metamodelle zu den WFS-Artefakten

4.1.1 ClientGateway–Modellanalyse

ClientGateways dienen als serverseitige Präsentationsschicht in der WFS-Dreischichten Architektur (siehe Abschnitt 2.4.3). Typischerweise werden Schnittstellen definiert über die der Client mit dem Server kommunizieren kann. Dabei werden vom Client ausgelösten GUI-Events mit

den dazugehörigen modellierten Servicemethoden gekoppelt. Für die grafische Darstellung der ClientGateways ist die Beziehung zwischen den Events und den Servicemethoden von besonderer Wichtigkeit. ClientGateways werden durch das Component-Metamodell beschrieben. Für eine bessere Übersicht des Metamodells, wird dieses im folgenden Schaubild 4.2 als UML-Strukturdiagramm dargestellt.[1] Durch die hervorgehobenen Elemente soll deutlich werden, welche Strukturteile für die grafische Darstellung relevant sind. Da das Komponentenmodell gleichzeitig als Eingabe für die Generierung des Clients verwendet wird, können viele Informationen gleich am Anfang gefiltert werden.

[1] Die Diminesion der Grafik verhindert eine detaillierte Darstellung in gedruckter Form. Es wird emfohlen einen PDF-Reader zu verwenden um das Schaubild vergrößert betrachten zu können

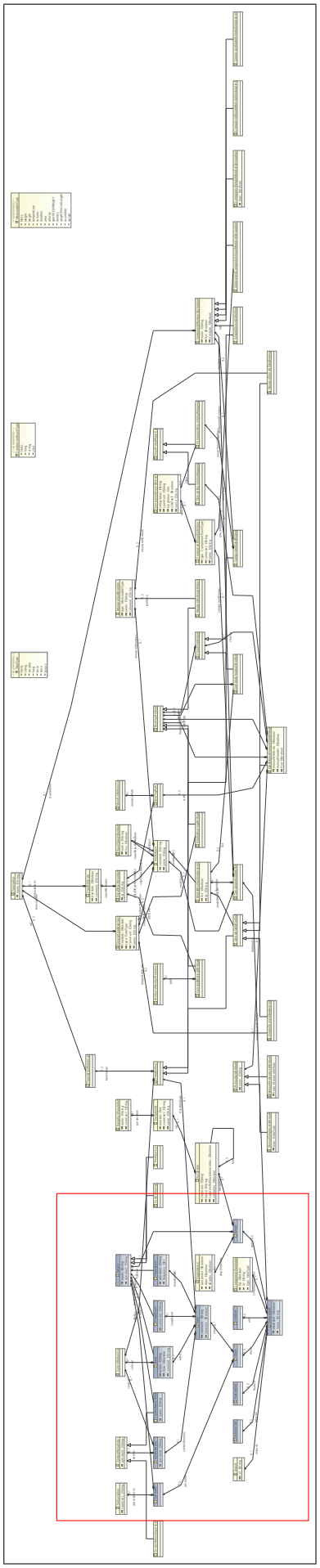

Abbildung 4.2: UML-Strukturdiagramm zum Komponenten-Metamodell

Gefilterte Modellinformation

Wird das Komponenten-Metamodell von überflüssigen Strukturelementen bereinigt, entsteht ein neues Metamodell, das für die grafische Darstellung verwendet werden kann. Das Schaubild 4.3 zeigt das neu enstandene Metamodell in einem UML-Struktur-Diagramm. Die relevanten Elemente ist einerseits der *Name* des Komponentenmodells welcher eindeutig sein muss und die dazugehörigen Solution, Feature und Unit. Jedes Modell besitzt ein Kommentarelement welches das Modell in fachlicher Funktionssicht beschreibt. Weitere Elemente sind zum einen der Client, der die verschidenen Views mit ihren namen enthält, zum anderen die Events die jeweils in einer View untergebracht sind. Ein Event besitzt ein Control-Element welches durch eine Action ausgelöst kann und den bestimmten Service aufruft. Die Servicemethoden werden im Serverobjekt eingeordnet. Diese besitzen ebenfalls einen Namen und einen Kommentar. Zudem werden im Serverobjekt die ahängigen Services modelliert. Dabei wird der Typ und der Name des Services spezifiziert. Eine strukturierte Auflistung der einzelnen Elemente die zur grafischen Darstellung benötigt werden, kann der Abbildung 4.4 entnommen werden. Diese dient im späteren Verlauf als Basisvorlage für die Transformationsregeln im Generator und für das Mapping von Modellinformationen in ein grafisches Diagramm.

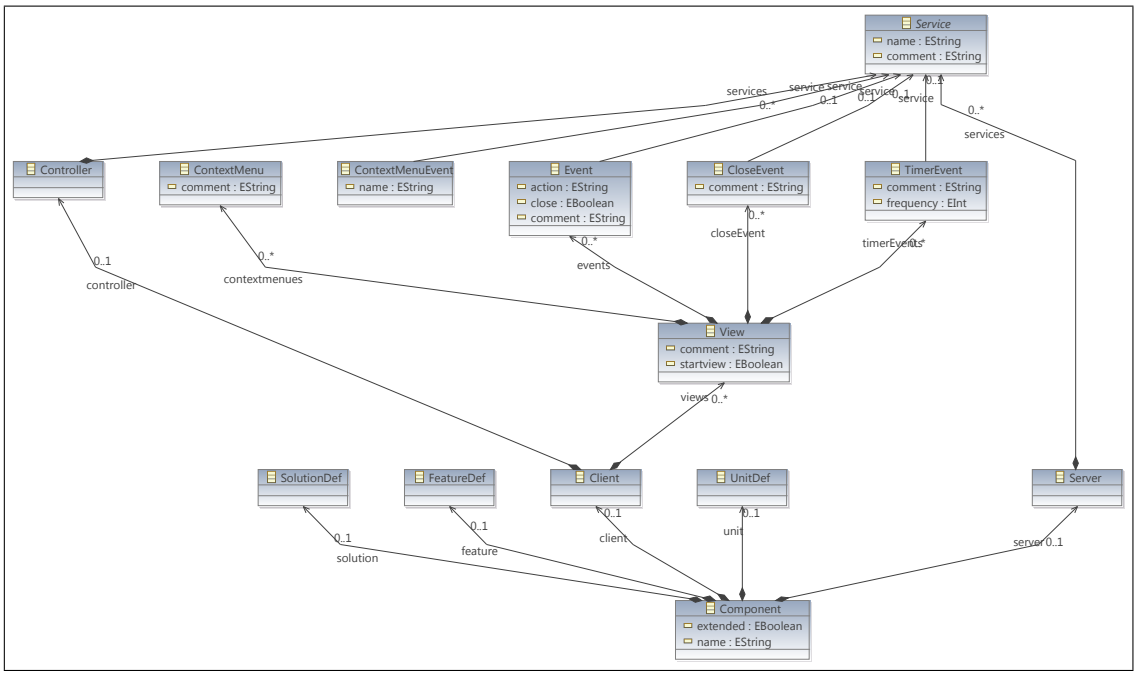

Abbildung 4.3: Gefiltertes ComponentModel-Metamodell

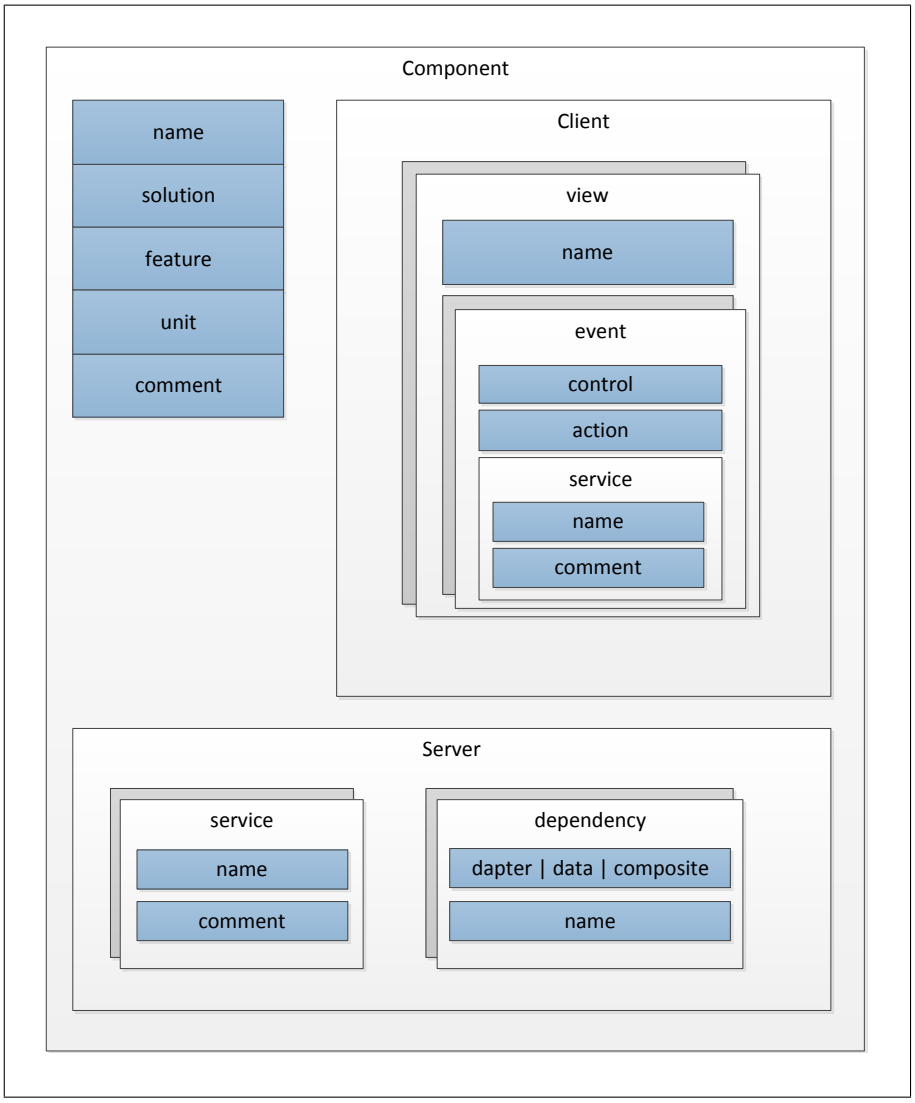

Abbildung 4.4: Gefilterte Strukturübersicht ClientGateway

4.1.2 CompositeService–Modellanalyse

Composite-Services bilden in der WFS die Logikschicht einer Anwendung ab, siehe Abschnitt 2.4.3. Sie werden als eine Ansammlung von Servicemethoden bereitgestellt. Dabei können Composite-Services andere Data–und Composite-Services referenzieren. Composite-Services werden durch das *Service*-Metamodell beschrieben, siehe Abbildung 4.5. Für die grafische Darstellung relevanten Elemente, wurden in dieser Abbildung farbig hervorgehoben.

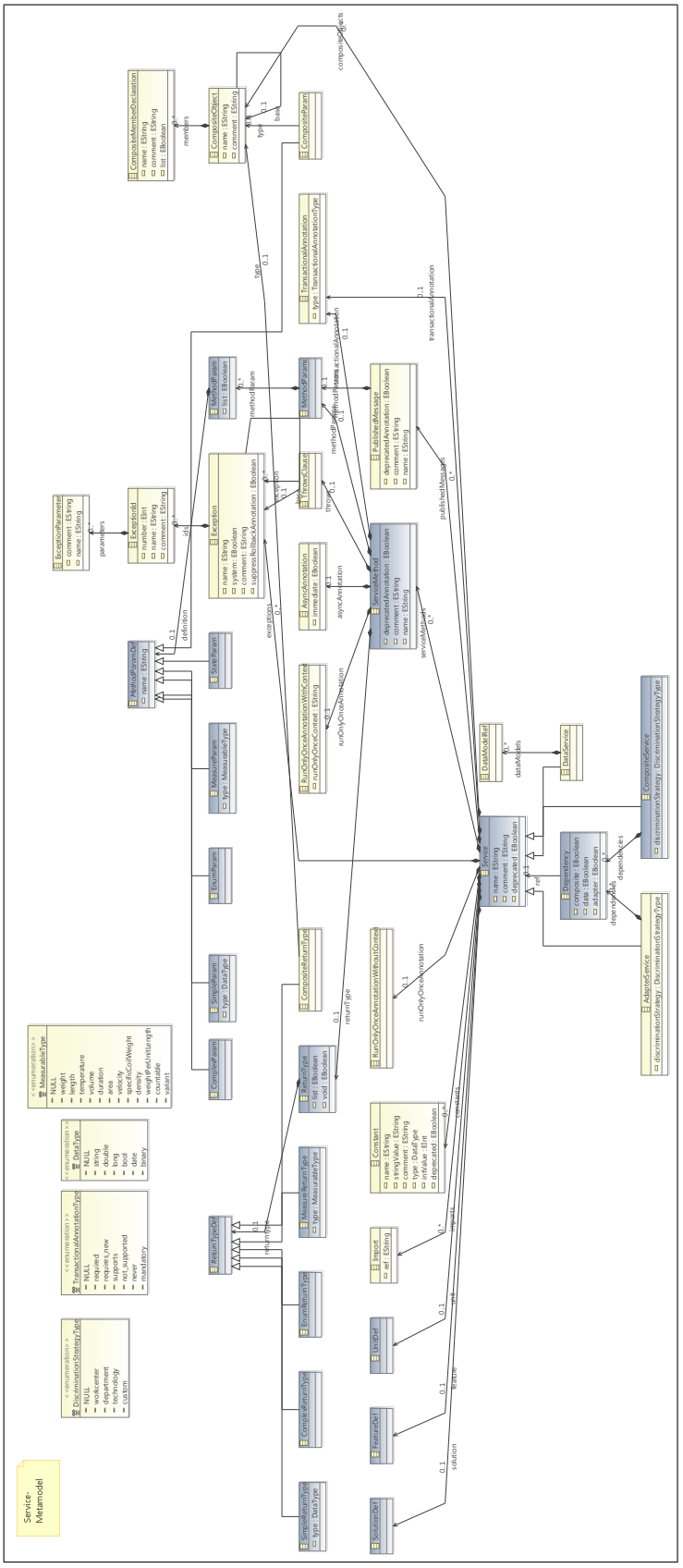

Abbildung 4.5: UML-Strukturdiagramm zum CompositeService-Metamodell

Gefilterte Modellinformation

Nach einer Bereinigung des CompositeService-Metamodells, unter der Berücksichtigung der vordefinierten Filterkriterien, entsteht ein neues Metamodell das zur grafischen Darstellung verwendet werden kann. Das Schaubild 4.6 zeigt das neu enstandene Metamodell in einem UML-Struktur-Diagramm. Besonderen Wert wird auf die Anzeige der Servicemethoden (Name, Rückgabetyp, Übergabeparameter) mit den Kommentaren gelegt, da diese die fachliche Logik repräsentieren. Zusätzlich sollen die abhängigen Services zum Ausdruck gebracht werden, was die Zugriffslogik auf die Datenschicht wiederspiegelt. In der Abbildung 4.7 werden die wesentlichen Strukturelemente zusammengefasst. Diese sind wie im Komponentenmodell ebenfalls enthalten, der Name des CompositeServices, die Solutionzugehörigkeit, das Feature, und die Unit. Anschließend folgen die Servicemethoden mit den jeweiligen Attributen Name, Kommentar, DeprecationAnnotation, Rückgabetyp und den Methodenparametern. Zusätzlich können CompositeServices abhängige Services wie DataServices oder andere CompositeServices enthalten. Diese werden im Datenobjekt dependency definiert. Dabei wird der Typ und der Name der Dependency spezifiziert. Diese Auflistung der einzelnen Elemente ist für den späteren Generatorentwurf und für das Mapping auf grafische Diagramme von Bedeutung.

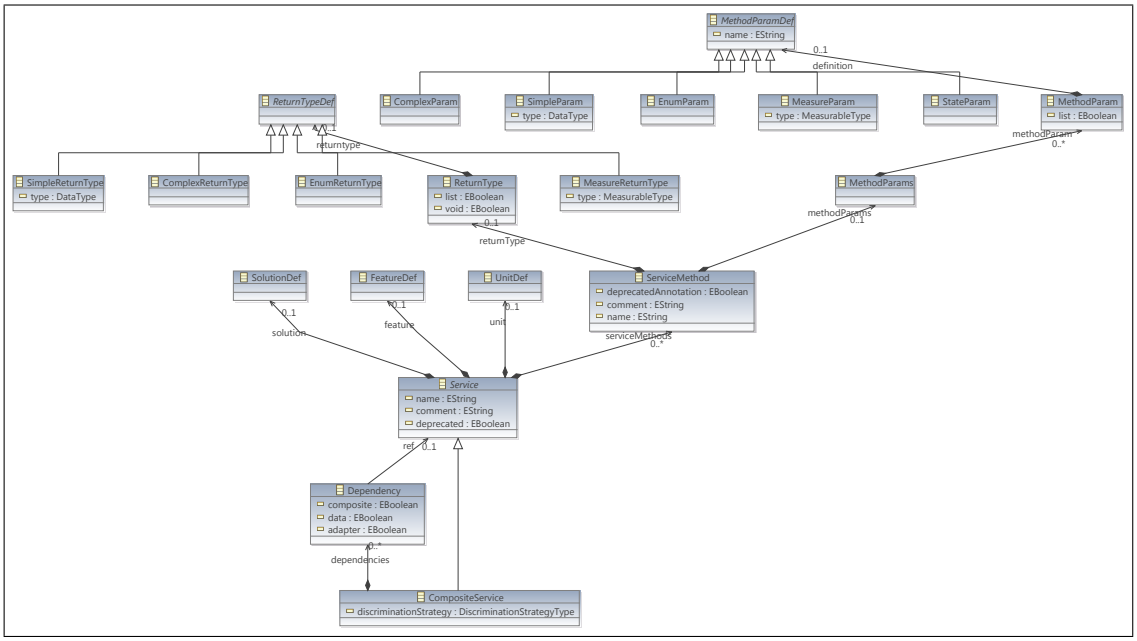

Abbildung 4.6: Gefiltertes CompositeService-Metamodell

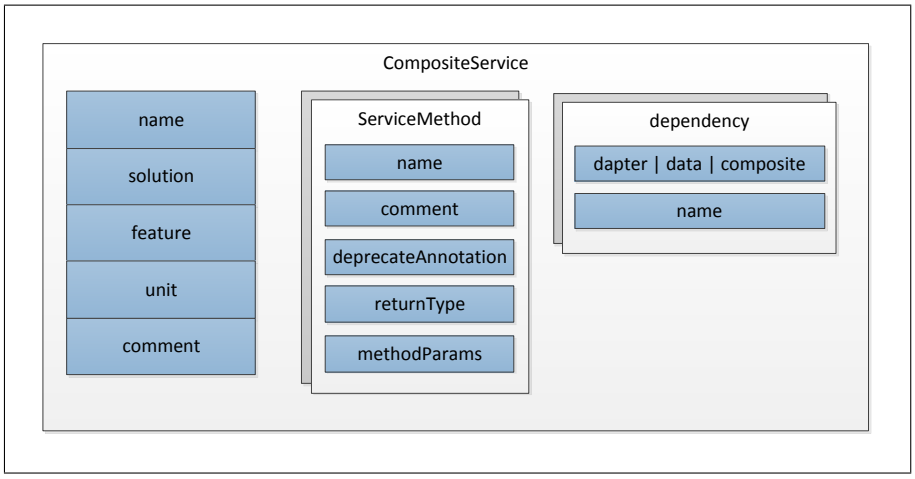

Abbildung 4.7: Gefilterte Strukturübersicht CompositeService

4.1.3 AdapterService/DataService–Modellanalyse

Ein AdapterService bietet die Möglichkeit in der Datenschicht auf Fremdsysteme lesend zuzugreifen. Um diese Informationen zu persistieren, werden Dataservices instanziert und Create-Servicemethoden aufgerufen. Der strukturelle Aufbau des Metamodells unterscheidet sich im wesentlichen nicht vom Metamodell des CompositeServices. Da die Filterkriterien aus dem CompositeService übernommen werden können, wird auf eine detaillierte Darstellung der Elemente verzichtet und analog zum CompositeService weiter vorgegangen. Mittels eines Data-Services wird der Zugriff auf die Produktionsdatenbank realisiert. Der DataService wird mit den Metamodellen Service, *OR-Mapping* und *DataModel* beschrieben, siehe Abbildungen 4.8, 4.9, 4.10.

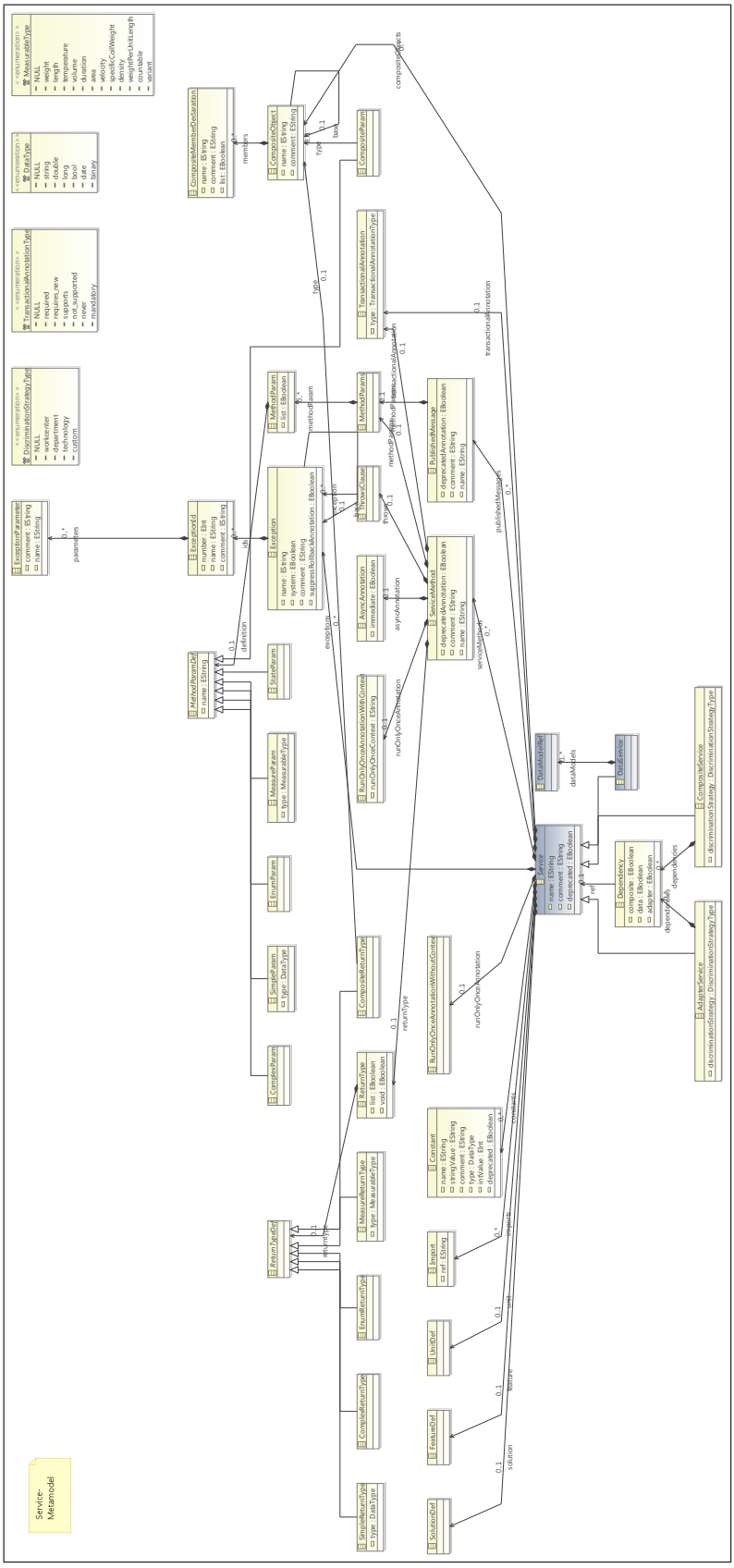

Abbildung 4.8: UML-Strukturdiagramm zum DataService-Metamodell

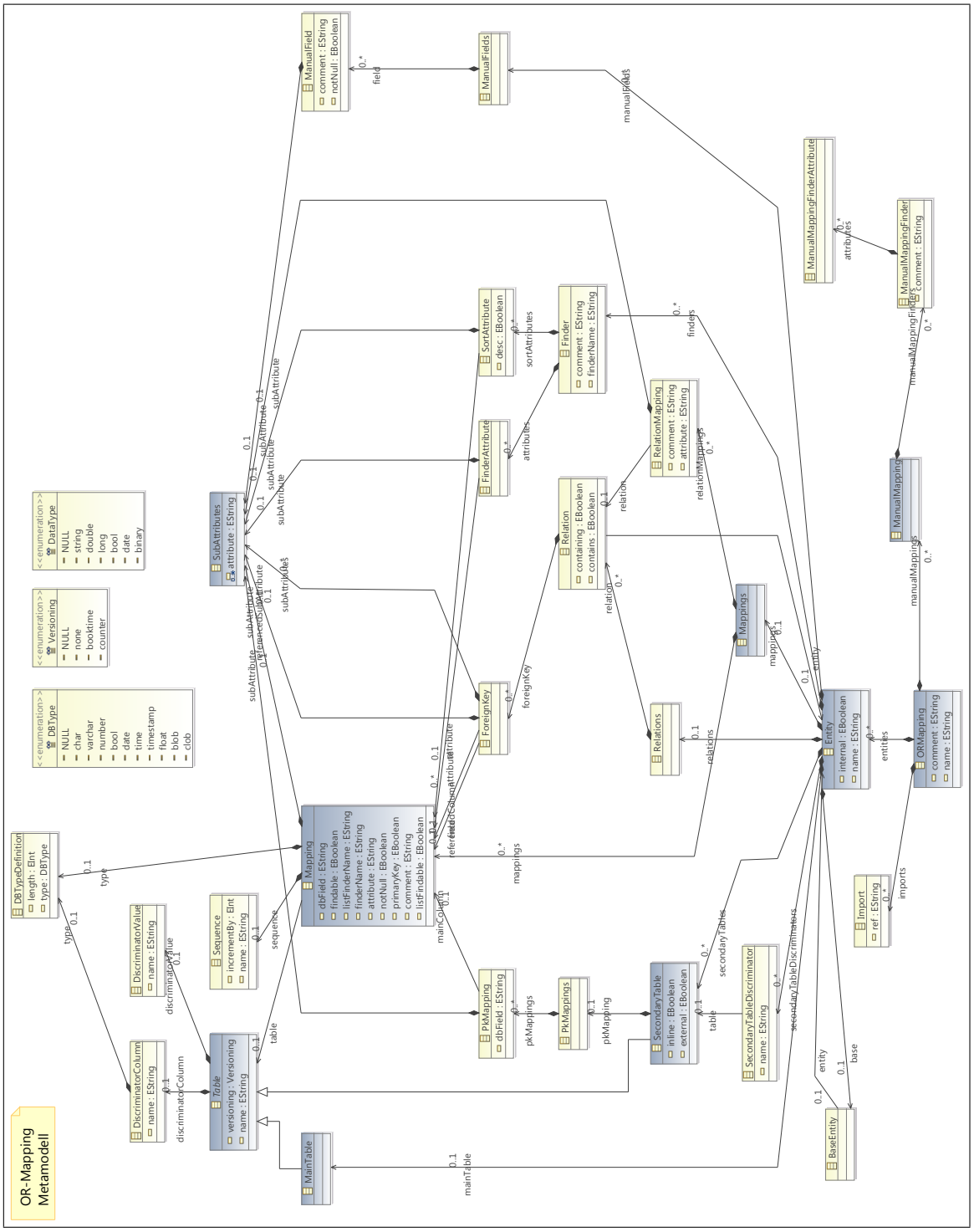

Abbildung 4.9: UML-Strukturdiagramm zum OR-Mapping-Metamodell

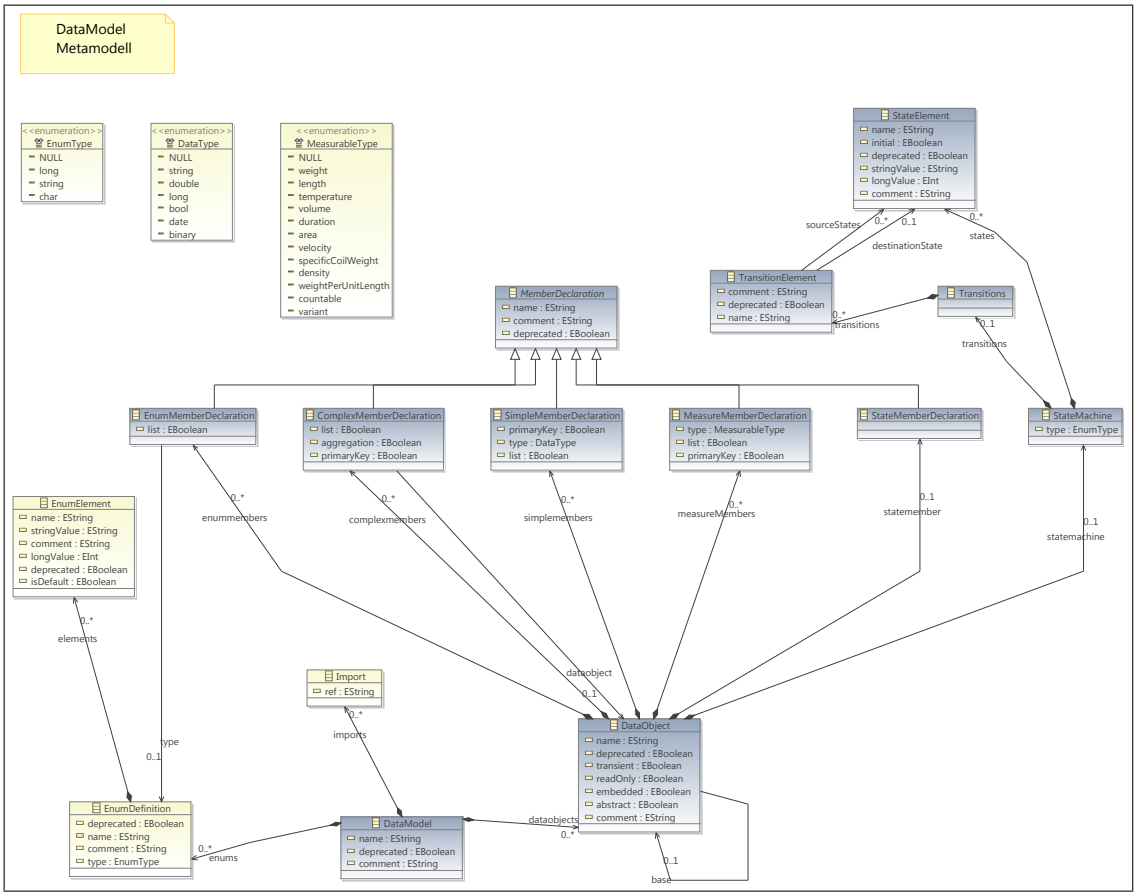

Abbildung 4.10: UML-Strukturdiagramm zum DataModel-Metamodell

Gefilterte Modellinformation

Bei der Strukturanalyse des Dataservices wurde ziemlich schnell klar, dass die Darstellung der DataModels als das wichtigstes Ziel bei der Visualisierung darstellt. Der Grund dafür ist, dass die Abteilung im Vorfeld eine Aktion durchgeführt hatte, mit dem Ziel, jegliche Domänenobjekte die auf verschiedenen Datenbanktabellen verteilt waren zu identifizieren. Anschließend sollte diese Auswertung als Vorlage für die Modellierung von DataServices dienen. In der grafischen Darstellung sollen Definition aus den textuellen Datenmodellen strukturiert und übersichtlich in grafischer Form dargestellt werden. Dabei soll eine Möglichkeit gefunden werden die Attribute aus dem Datenmodell und die passenden Datenbankmappings zu visualisieren. Als Folge dessen wurden die relevanten Elemente identifiziert und ein Ziel-Metamodell aufgebaut, wie im Schaubild 4.11 zu erkennen ist. In der Abbildung 4.12 werden die wesentlichen Strukturelemente, als Vorbereitung für den Generatorentwurf und des Mappings der Informationen auf ein

Diagramm, zusammengefasst. Die einzelnen Elemente sind wie schon in den vorherigen Modellen vorkommend, der Name des DataServices, die Solution, das Feature und die Unit. Zudem können DataServices mehrere DataModels enthalten, die wiederum einen Namen haben und eine Anzahl von Dataobject. DataObjects spiegeln Datenstrukturen wieder und können demnach verschiedene Variablendeklarationen enthalten. Außerdem können DataObjects sich von anderen DataObjects ableiten, wie eben Klassen in objektorientierten Programmiersprachen. Ein weiteres Konstrukt eines DataServices ist das OR-Mapping. Dieses definiert die Abbildung von Dataobjects auf Entities. So können Attribute aus den DataObjects in Mappingskonstrukten auf Datenbankfelder abgebildet werden. Dabei werden Informationen wie Attributname, Tabelle und Tabellenspalte benötigt. All diese Informationen werden im späteren Verlauf durch einen Generator in grafische Modelle überführt.

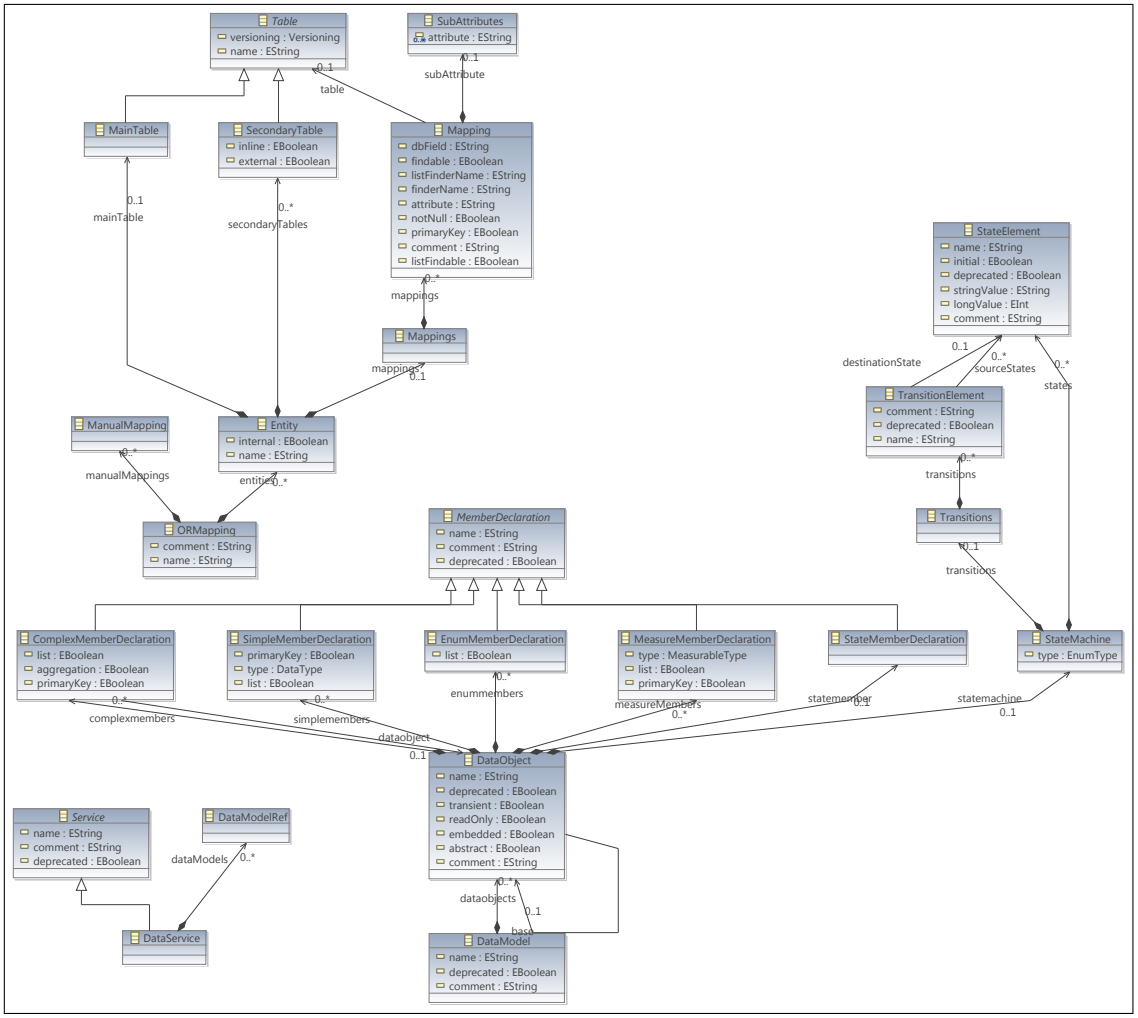

Abbildung 4.11: Gefiltertes DataService-Metamodell

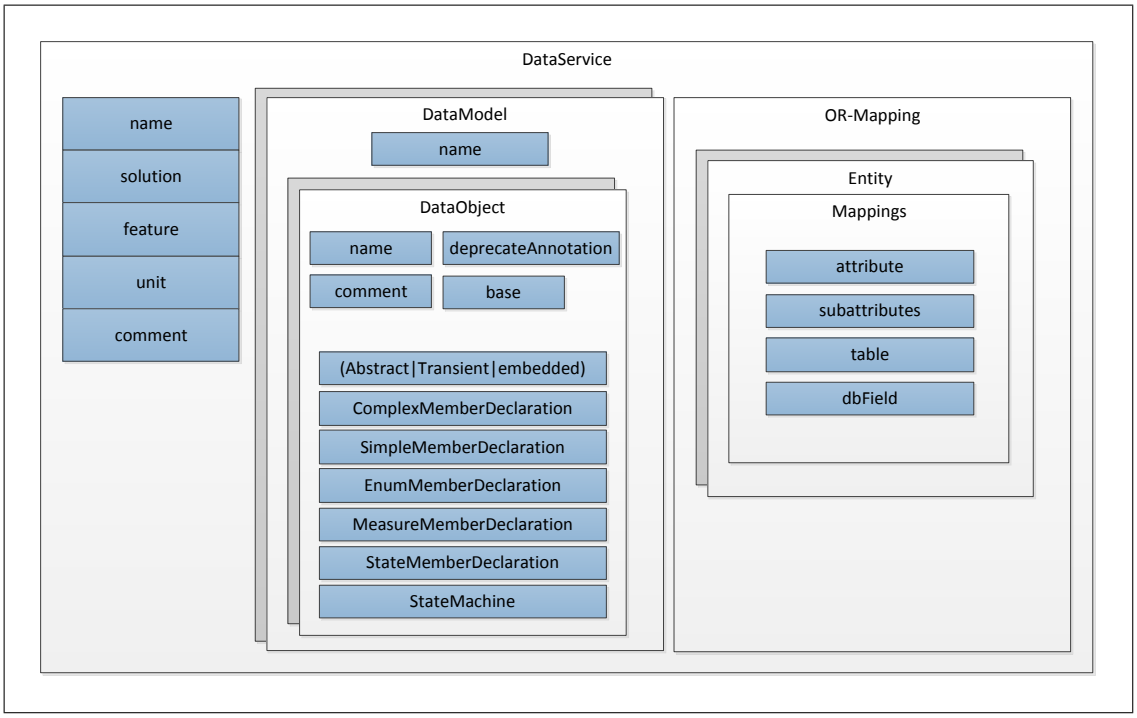

Abbildung 4.12: Strukturübersicht DataService für grafischen Abbildung

4.2 Konzepte für grafische Darstellung von textuellen Modellen

Nachdem im letzten Kapitel, die zur grafischen Darstellung erfoderlichen Informationen und Sachverhalte analysiert und aufbereitet wurden, findet in diesem Kapitel die Analyse zu möglichen Darstellungsformen unter der Berücksichtigung von ergonomischen Aspekten statt. Dabei soll eine interaktive Anwendung auf generischer Basis entwickelt werden, um Primäraufgaben wie Anzeige von Daten, Navigation, Selektion und Manipulation zu relisieren.

4.2.1 Softwareergonomische Aspekte

In diesem Abschnitt werden im Rahmen der Konzeptionsphase, Grundsätze für den Aufbau der Benutzeroberfläche und der Benutzerinteraktion entwickelt. Dabei werden die verschiedenen Aspekte der Software-Ergonomie während der Entwicklung einer Applikation untersucht.
Um eine intuitiv zu bedienende Benutzerschnittstelle realisieren zu können, ist es wichtig zu er-

kennen, welche Kriterien ein solches System erfüllen muss. Unter der Bezeichnung der *Software-Ergonomie* werden die Benutzeranforderungen an das System in Bezug auf Bedienbarkeit und Ausführung der Aufgaben beschrieben. Die Software-Ergonomie kann nach [11] folgerndermaßen beschrieben werden:

„Die Software-Ergonomie entwickelt Kriterien und Methoden zur Gestaltung interaktiver Programmsysteme, die den menschlichen Bedürfnissen der Benutzer nach einer ausgeglichenen Verteilung der geistigen, körperlichen und sozialen Belastung weitgehend entgegenkommt."

Einfach ausgedrückt zeichnet sich eine ergonomische Software dadurch aus, dass sie an die Bedürfnisse der Benutzer anpepasst ist. Selbstverständlich ist diese Forderung nicht immer zu erfüllen, da unterschiedliche Benutzer verschiedene Wahrnehmungen besitzen können. Trotzdem kann man Software nach bestimmten Kriterien entwickeln um den Umgang für den Benutzer zu erleichtern. Viele Erkenntnisse aus der Software-Ergonomie wurden im laufe der letzten 20 Jahren in bestimmten Normen und Richtlinien definiert. Im folgenden werden Richtlinien für die Entwicklung von Benutzeroberflächen anhand der Norm ISO 9241-110 behandelt. Bevor man mit der Gestaltung der Oberfläche und der grafischen Modelle beginnt, sollte man sich im Klaren sein, welche Benutzer das Programm bedienen sollen und wie ihre Anforderungen an das System sind. Um eine intuitiv zu bedienende Software entwickeln zu können, hilft uns die international anerkannten Norm ISO 9241-110. Dort werden Prinzipien für Interaktive Systeme definiert die den Umgang mit Mensch und Maschine erleichtern. Für uns relevante Grundsätze sind nach [16]:

Aufgabenangemessenheit

Ein interaktives System muss seinen Benutzer dabei unterstützen, seine Aufgabenziele vollständig, korrekt und mit einem vertretbaren Aufwand und zur Arbeitsaufgabe passenden Dialogschritten zu erledigen. Dieser Grundsatz wird kurz als Aufgabenangemessenheit bezeichnet.

Selbstbeschreibungsfähigkeit

Ein interaktives System muss so gestaltet sein, dass sein Benutzer jeder Zeit weiß, wo er sich im Dialog befindet, wie er da hingekommen ist und wie er von dort aus wieder weg kommt.

Erwartungskonformität

Ein interaktives System ist erwartungskonform, wenn es die Sprache und die „Arbeitsgebräuche" der Benutzer im Dialog berücksichtigt. Die Erwartungskonformität kann oft schon durch die Einhaltung von Konventionen und einer konsistenten Systemgestaltung erheblich verbessert werden.

Schlussfolgerungen für Gestaltung der Grafischen Modelle und Oberfläche
Aus diesen software-ergonomischen Aspekten lassen sich für die Entwicklung der Visualisierungsanwendung folgende Punkte festhalten:

- Da die WFS hierarchisch in zehn Solutions unterteilt ist, wird eine Navigationsstruktur angestrebt, welche diese Struktur widerspiegelt. Diese Maßnahme soll den Grundsatz Aufgabenangemessenheit erfüllen, indem der Benutzer schnell zu seinem Ziel navigiert wird.
- Um die Navigation zu den Zielmodellen so kurz wie möglich zu halten, sollen die unterschiedlichen Artefakte anhand einer Auswahl eingegrenzt werden, dieses Vorgehen erleichtert die Suche von Grafischen Modellen.
- Damit der Benutzer sich schnell zurechtfindet, soll bei jedem Modell an der selben Stelle die Information zu der Solution, Feature, Unit hinterlegt werden.
- Um die verschiedenen Solutionbereiche besser abzutrennen, soll eine Farbkodierung zum Einsatz kommen. Diese wird aus den bestehenden Dokumenten übernommen, damit wird das Prinzip der Erwartungskonformität berücksichtigt.
- Für die grafische Notationform soll beachtet werden, bestehende Diagramme zu den Domänenobjekten, in die Entscheidungsfindung einfließen zu lassen. Dadurch soll das Verständnis der grafischen Modellen erleichtert werden.

Eine ausführliche Auseinandersetzung mit der Software-Ergonomie würde den Rahmen dieser Diplomarbeit sprengen. Deshalb habe ich mich hier auf die wichtigsten Aspekte der Ergonomie beschränkt und so für das weitere Vorgehen eine Basis geschaffen um den Bedürfnissen der Nutzer gerecht zu werden.

4.2.2 Grafische Benutzeroberfläche

Nach Abschluss der Analyse in den letzten Kapiteln sollen erste konzeptionelle Entwürfe der Benutzerschnittstelle entwickelt werden. Um die grafische Oberfläche leichter in die bestehende Dokumenatationsplattform der WFS einbinden zu können, wurde der Entschluss gefasst, die Oberfläche in einem HTML-Container unterzubringen. Dies bedeutet, dass dynamische HTML-Seiten erzeugt werden, welche die grafischen Modelle verlinken. Im Folgenden beschäftige ich mich zuerst mit dem Navigationskonzept und Darstellung der Elemente. Im späteren Abschnitt wird auf die einzelnen Modellrepräsentationen eingegangen. Bevor einzelne Elemente am Bildschirm modelliert wurden, wurden Bleistiftzeichnung angefertigt und mögliche Darstellungsfor-

men für die Navigation analysiert. Als Einstiegspunkt auf der Bedienoberfläche soll eine Startseite erscheinen mit einer Überschrift „WFS-Domänenmodell“ und der Auswahl der möglichen Solutions. In der Mitte soll ein Bild, den Inhalt dieser Web-Seite dem Benutzer näher bringen. An der oberen linken Ecke erscheint das Wieland-Logo um das Produkt zu identifizieren und als Startseiten-Link zu dienen. Die Abbildung 4.13 zeigt das Konzept für die Einstiegsseite. Die Navigationsleiste soll an die Navigation aus der WIKI-Seite der WFS angelehnt sein. Die Darstellung ist horizontal angeordnet und nach Solutions geordnet. Die einzelnen Solutions sollen die vorhandene Farbkodierung aus der Dokumentation übernehmen und wie in der Abbildung 4.14 aussehen.

Abbildung 4.13: Konzept Startseite der Anwendung

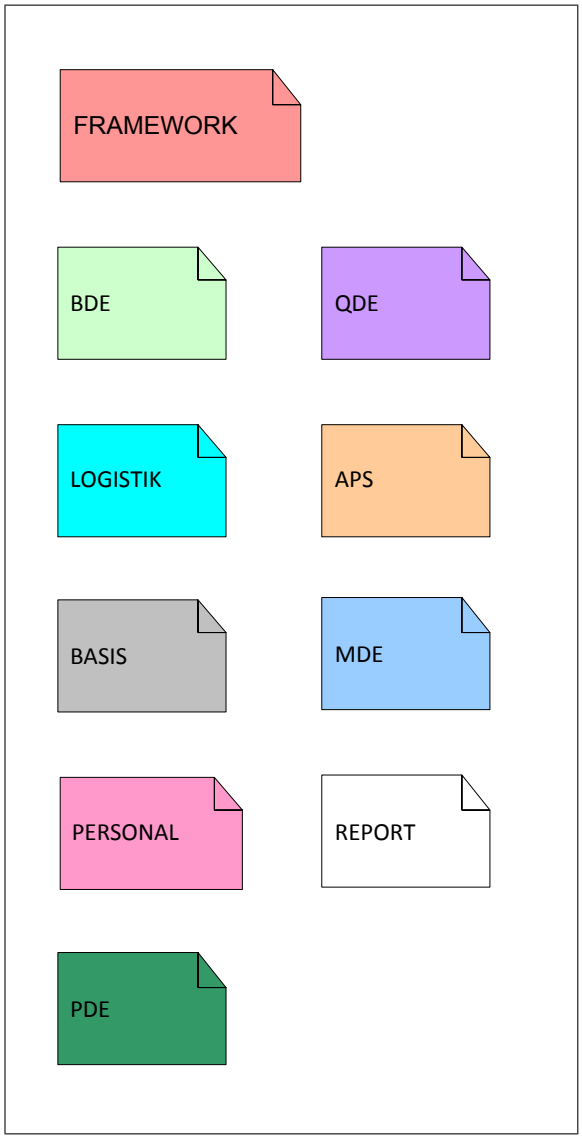

Abbildung 4.14: Farbkonzept der Solutions

Die Anzeige der grafischen Modelle aus verschiedenen Artefakten soll zentriert Angeordnet sein. Hinzukommt die Möglichkeit nach bestimmten Typen Einschränkungen vornehmen zu können, siehe Abbildung 4.15. Dabei sollen mithilfe von Checkboxen eine Auswahl getroffen werden um die gewünschte Ansicht zu realisieren.

Da die Oberfläche mehrere grafische Modelle von verschiedenen Artefakten darstellen soll, empfiehlt sich zuerst eine Liste von vorhanden Modellen anzuzeigen. Dabei soll sowohl eine

Abbildung 4.15: Darstellung des Hauptfensters mit Navigationselementen

Verlinkung auf die Grafik als auch die Druckansicht in Form von PDF angeboten werden, siehe Abbildung 4.16

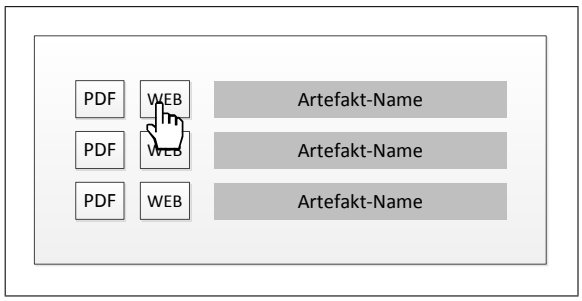

Abbildung 4.16: Darstellung von Artefakten in einer Liste

Wenn die Modelle in der Bildschirmdarstellung einen größeren Bereich für die Anzeige benötigen, soll eine Scrollfunktionalität das Problem beheben. Im weiteren erfolgt die Darstellung immer in einem Fenster, außer bei den PDF-Dokumenten, hier erscheint ein neues Fenster mit der PDF-Datei. Um zum Startbildschirm navigieren zu können, klickt man auf das Logo oben-

links. Zudem können die Vor- und Zurück-Buttons des Browsers zur Navigation sinnvoll verwendet werden. Die farbliche Gestaltung der Benutzerschnittstelle wurde so gewählt, dass ein Benutzer diesen Dialog als angenehm empfindet. Dabei wurden sehr dezente Grau- und Brauntöne aus dem Wieland-Web-Styleguide für die Hintergründe und die Widgets gewählt und eine Farbkodierung für die Solutionlinks vorgenommen. Dadurch soll gewährleistet werden, dass der Benutzer sich in der Anwendung „heimisch“ fühlt und nicht abgeschreckt wird.

4.2.3 Grafische Darstellung von Informationen

Aus der Problembeschreibung im Kapitel 3 folgt ein Lösungsansatz, der die textuellen Modelle aus einem MES-System in grafischer Form visualisiert und dadurch eine weitere Möglichkeit schafft, von Implementierungsdetails zu abstrahieren. In der ersten Stufe der Visualisierung wurde im Abschnitt 4.2.2 ein Lösungskonzept für die Bedienoberfläche entwickelt. Im Folgenden werden grafische Darstellungsmöglichkeiten für die textuellen Modelle untersucht.
„Ein Bild sagt mehr als tausend Worte“ ist ein geläufiges Sprichwort und bezieht sich darauf, dass komplexe Sachverhalte oft mit einem Bild einfacher erklärt werden können. Auf den Punkt gebracht, wollen wir abstrakte Informationen grafisch darstellen. Card, Mackinlay und Shneiderman [10] definieren den Begriff *Informationsvisualisierung* als „The Use of computer-supported, interactive, visual representations of abstract data to amplify cognitition.“ Demnach wird Informationsvisualisierung als ein externes Hilfsmittel zur Steigerung der kognitiven Leistung angesehen. Abstrakte Daten können in vielerlei Form vorkommen, wie beispielsweise als Tabellen mit numerischen Werten, hierarchische Strukturen, Datenrelationen und Verknüpfungen in Form von Graphen oder als Programmcode aus dem Softwareentwicklungsprozess [25]. In unserem Fall sollen Informationen die in textueller Form vorliegen, in eine sinnvolle visuelle Darstellung transformiert werden, die es den Mitarbeitern aus der MES-Softwareentwicklung erlaubt, Softwaremodelle leichter verstehen zu können. Damit richtet sich die Informationsvisualisierung an eine breitere Nutzergruppe mit unterschiedlichen Wissenshintergrund, wie im Kapitel 3.1.1 näher erörtert wurde. So ist es ein wesentliches Kriterium, dass z.B. die textuellen Datenmodelle auch für Business-Consultants zugänglich und verständlich sein müssen.
Um die Datenmenge zu reduzieren und den Fokus auf wichtige Daten zu legen, wurden im Abschnitt 4.1 die Informationen entsprechend aufbereitet. Die Resultate dieser Datentransformationen sollen nun in geeignete grafische Modelle überführt werden. Schließlich erfolgt die grafische Anzeige auf einem Ausgabemedium wie dem Bildschirm oder Papier. Durch die Betrachtung dieser Grafiken wird der Erkenntnisgewinn für die beteiligten Mitarbeiter untersützt. Card, Mackinlay und Shneiderman [10] beschrieben diese Visualisierungspipeline als Refe-

renzmodell der Visualisierung mit den Phasen Datentransformation (Vorverarbeitung), Visuelle Abbildung (Transformation in ein grafisches Modell) und Bildgenerierung (Rendering), siehe Schaubild 4.17. Zudem definieren die Autoren die wesentliche Ziele des Erkenntnisgewinns durch die Visualisierung:

- Entdecken von Zusammenhängen oder Besonderheiten
- Zuverlässiges Treffen von Entscheidungen
- Auffinden von Erklärungen für Muster und Gruppen von Datenobjekten

Diese Ziele sollen im Informationsmappingprozess berücksichtigt werden um eine bestmögliche Darstellungsform finden zu können.

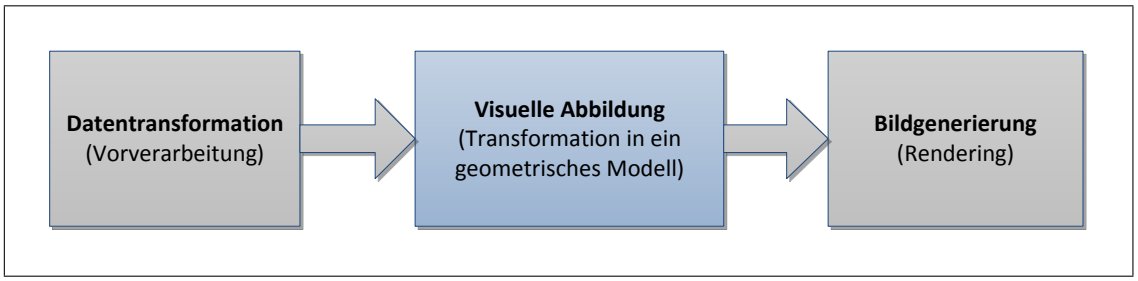

Abbildung 4.17: Visualisierungspipeline als Referenzmodell der Visualisierung nach [10]

Aufgaben der Visualisierung

Der Fokus der grafischen Darstellung soll auf Klassifikation–und Strukturierung der Modellinformationen liegen. Die erste Klassifizierung findet in der Aufteilung der Modelle nach Artefakttyp statt. Dadurch soll ein Überblick über den gesamten Informationsraum gewonnen werden. Die Bedienoberfläche grenzt die Modelle wiederum nach Solutiontyp ein und bietet dadurch eine hierarchische Anordnung auf der Navigationsebene. Bei größeren Datenmengen soll eine Zoom-Funktion eingesetzt werden um interessante Informationsobjekte deutlich sichtbar zu machen. Um Details für ein bestimmtes Datenobjekt oder einer Gruppe von Daten anzeigen zu können, soll eine Tooltip-Unterstützung eingebunden werden. Zusammengefasst lassen sich die Visualisierungsaufgaben aus dem Schaubild 4.18 ablesen. Diese Aufgaben leiten sich aus dem viel zitierten *Mantra visueller Informationssuche* [28] ab:

> Overview first, zoom and filter, then details on demand.

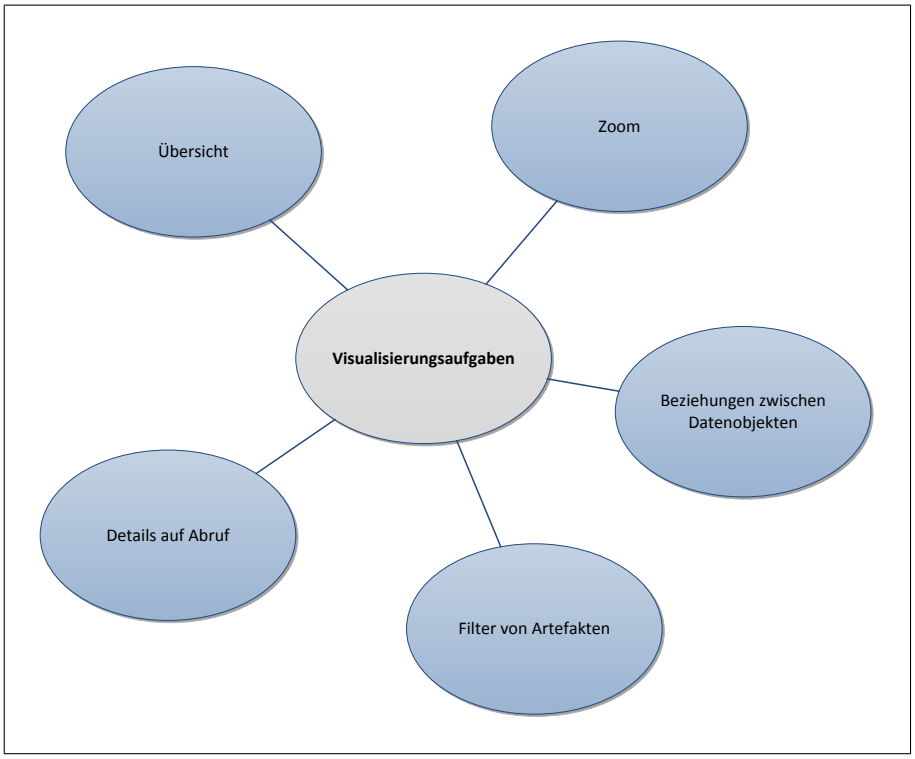

Abbildung 4.18: Visualisierungsaufgaben bei Umsetzung der grafischen Modelle nach [28]

Geometrische Visualisierungsformen

Nachdem die wesentlichen Informationsvisualisierungsmerkmale in den vorausgegangenen Abschnitten erläutert wurden, soll hier näher auf das Mapping von Informationsdaten auf konkrete geometrische Objekte eingegangen werden.

Die Basis für sämtliche Informationsvisualisierungen ist ein *Datenmodell*, das typischerweise einen Ausschnitt der realen Welt repräsentiert. Es besteht aus Informationsobjekten mit ihren Attributen und Relationen untereinander [25]. *Informationsobjekte* können in unserer Domäne die verschiedenen Artefakttypen wie CompositeServices, DataServices usw. sein. *Attribute* nennt man die Eigenschaften eines Objekts, im Beispiel von CompositeServices wären die einzelnen Servicemethoden Merkmale dieses Informationsobjektes. *Relationen* sind Beziehungen zwischen zwei oder mehreren Datenobjekten bzw. Attributen der gleichen oder unterschiedlicher Domäne. Ein DataService kann beispielsweise mehrere Datenmodelle enthalten. In einem Datenmodell existieren Dataobjects die wiederum von anderen Dataobjects abgeleitet werden können. So können Relationen sowohl hierarchischer Natur sein, aber auch beliebige andere

Relationen darstellen.
Durch den Fokus auf Strukturierung und der Berücksichtigung von Relationen in den Datenmodellen, kommen für unsere Datenmengen folgende Abbildungstechniken in die nähere Auswahl:

Hierarchien(Bäume). Hierarchische Strukturen in Datenmengen sind verbreitet, weil sie Informationsräume sinnvoll strukturieren, vereinfachte Zuordnungen zu Kategorien erlauben und generell der Wissensstrukturierung dienen. Der Relationstyp bei Bäumen ist fast immer ein Enthalten-Sein-In. Hierarchien von Datenobjekten bzw. Attributen werden in der Regel als Baumstrukturen repräsentiert [25]. Im Falle von CompositeServices könnte diese Visualisierungstechnik sinnvoll eingesetzt werden. Dabei werden die einzelnen Artefakte als ein Kästchen repräsentiert und die dazugehörigen Servicemethoden können in einer Enthalten-Sein-In-Beziehung aufgelistet werden. Im Abschnitt 4.2.3 wird diese 4 näher untersucht und ein Konzept für betreffende Artefakte entwickelt.

Netzwerke(Graphen). Verknüpfte Datenobjekte bilden häufig Netzwerke, die sich als Graphen beschreiben lassen, wobei Datenobjekte bzw. Attribute die Knoten bilden. Die Relationen zwischen den Datenobjekten werden als Kanten repräsentiert [25]. Dabei können die Kanten, nach dem Vorbild von UML, verschiedene Assoziationsbeziehungen abbilden. Diese Visualisierungstechnik eignet sich besonders für die Darstellung von Datenmodellen aus dem DataService. Dabei repräsentieren die Knoten einzelne Dataobjects mit den dazugehörigen Attributen, durch gerichtete Kanten können Assoziationen und Aggregationen zu anderen Dataobjects abgebildet werden. Im Abschnitt 4.2.3 wird auf einige der Visualisierungslösungen in diesem Bereich eingegangen.

Hierarchievisualisierungen

Wie im letzten Abschnitt schon deutlich wurde, steht bei der hierarchischen Visualisierungsform die Abbildung von Datenobjekten, Attributen und der Relationen im Vordergrund. Häufig besteht die Relation von Datenobjekten jedoch in ihrer Verschachtelung bzw. ihrem hierarchischen Enthaltensein. Hier bieten sich Baumvisualisierungen an, um z.B. in unserem Fall, die Struktur der CompositeServices und AdapterServices abzubilden. Je nach Visualisierungstyp und Raumausnutzung teilen Preim, B. und Dachselt, R. [25] die Hierarchievisualisierungen folgendermaßen ein:

- Einfache Einrückungen
- Node-Link-Diagramme
- Flächenfüllende Verschachtelung
- Geschichtete Ansätze

Im folgenden werden die verschiedenen Ansätze kurz umrissen und im Anschluß ein Konzept für den CompositeService und AdapterService vorgelegt.

Einfache Einrückungen Diese Ansammlung von einfachen Visualisierungen verzichtet auf die Darstellung von Kanten für Bäume, da mit Kanten hier keine besondere Semantik assoziiert ist. Die bekanntesten Beispiele dafür sind Inhaltsverzeichnisse von Büchern oder hierchische Dateiansichten in einem Dateibrowser, siehe Abbildung 4.19.

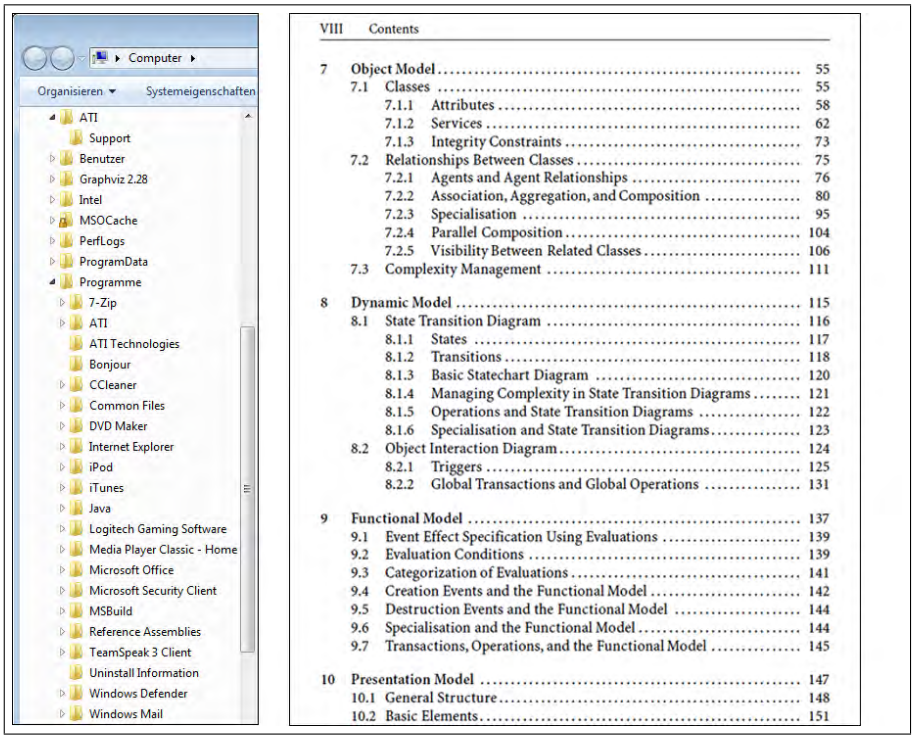

Abbildung 4.19: Baumvisualisierung durch Einrückung

Eine Einrückung symbolisiert den Beginn einer Hierarchiestufe unterhalb des übergeordneten Knotens. Damit wird das Prinzip der Unterordnung betont. Diese Visualisierungsform stellt ausschließlich Elternknoten dar, also keine Blattknoten. Im Fall eines Textdokuments könnte man die Absätze unterhalb einer Überschrift als Blattknoten auffassen, bei Dateisystemen sind es

die Dateien [25]. Eine einfache Konzeptzeichnung die ein CompositeService mit den enthaltenen Servicemethoden und Abhängigkeiten angezeigt wird in der Abbildung 4.20 dargestellt. Zusätzlich könnte man Hilfslinien einfügen um eine bessere Abschätzung der Hierarchietiefe

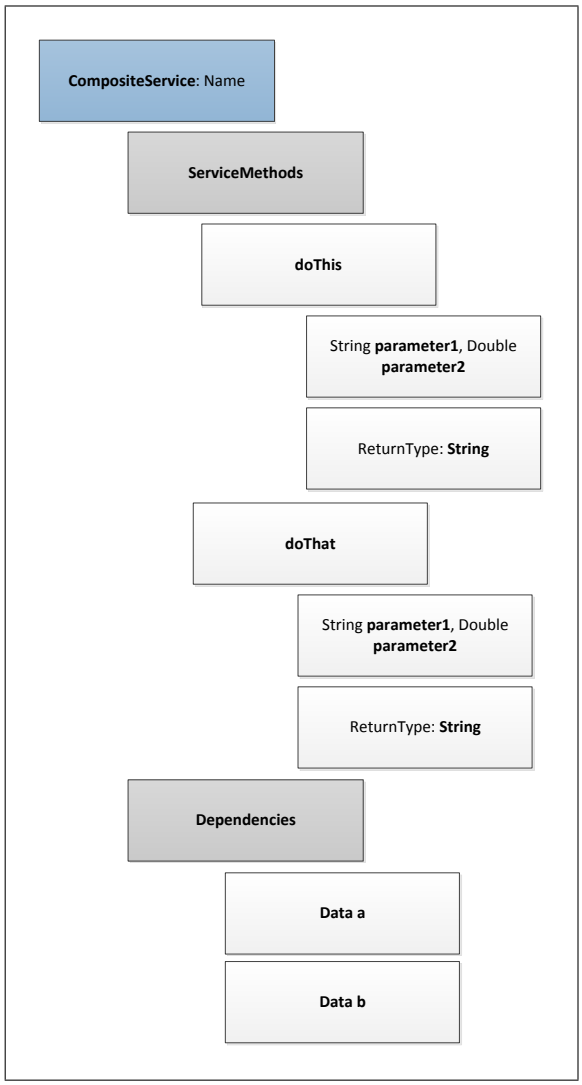

Abbildung 4.20: Konzept CompositeService Baumvisualisierung

zu bekommen. Die Einrückungsvisualisierung ist besonders für Knoten mit textueller Auszeichnung geeignet, da die Lesbarkeit begünstigt wird. Andererseits besteht die Gefahr, durch die hochformate Ausrichtung, bei größeren Datenmengen die Übersicht zu verlieren.

Node-Link-Diagramme Node-Link-Diagramme visualisieren Knoten und Kanten explizit und sind wohl die häufigste Visualisierungsform für hierarchische Strukturen und Netzwerke. Dabei

werden die Knoten durch verschiedene geometrische Formen dargestellt, z.B. durch Kreise, Rechtecke oder Icons. Die zentrale Herausforderung dabei ist die Platzierung der Knoten im Raum. Dazu wurden verschiedene Layoutvarianten entwickelt, einige solcher Layouts sind in dem Schaubild 4.21 verdeutlicht. Relationen werden realisiert, indem man die Knoten durch einfache Linien verbindet [25]. Node-Link-Diagramme sind gut geeignet, um strukturelle Zusammenhänge zu erkennen. Zudem ist die klassische Baumdarstellung für viele Benutzer sehr vertraut. Im Falle der Visualisierung eines CompositeServices könnte man sich eine Darstellungsform vorstellen, wo die Attribute durch Knoten repräsentiert werden und der Baum von oben nach unten aufgebaut wird. Dadurch können Datenbeschriftungen einfacher abgelesen werden. Als nachteilig erweist sich die mangelnde Kompaktheit bei der Darstellung von Servicemethoden.

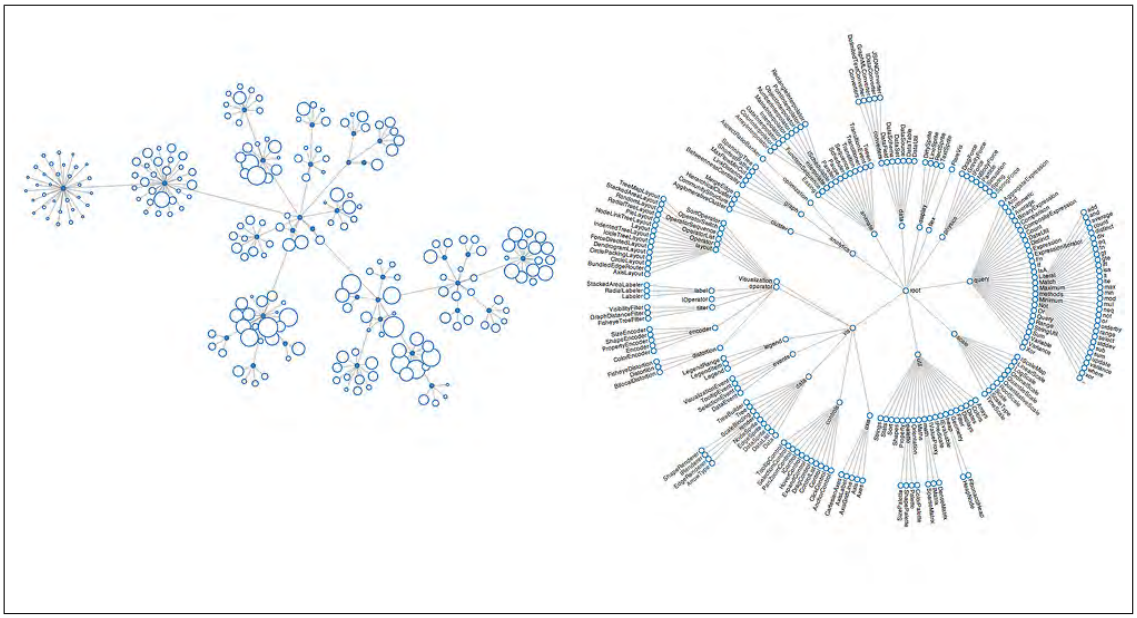

Abbildung 4.21: Klassische Baumvisualisierungen der gleichen Hierarchie: Radialer Baum (rechts) und Kräftebasierter Baum (links)[9]

Flächenfüllende Verschachtelung Die im vorherigen Abschnitt vorgestellten Node-Link-Visualisierungen erzeugen je nach Baumstruktur wenig platzoptimierte Darstellungen. Diesen Nachteil versuchen flächenfüllende Ansätze auszugleichen. Dabei wird der Platz einer rechteckigen–oder kreisförmigen Form in Abschnitte aufgeteilt. Die hierarchische Struktur wird durch räumliches Enthaltensein in geometrischen Formen abgebildet. Dieses Visualisierungskonzept wird in der Literatur oft als *Treemaps* bezeichnet. Der Aufbau einer flächenfüllenden Verschachtelung wird im Schaubild 4.22 verdeutlicht. Dabei repräsentiert die gesamte zur Verfügung stehende Fläche

den Wurzelknoten. Für jeden Teilbaum wird das entsprechende Rechteck wiederum unterteilt. So kann rekursiv der gesamte Baum abgearbeitet werden. Bestimmte Attribute können bei dieser Darstellungsform farbig kodiert werden. Der große Vorteil dieser Darstellungsform besteht darin, gesamte Bäume platzsparend visualisieren zu können. Probleme tauchen bei dieser Art der Darstellung bei der Erkennung von Elternknoten, da diese nicht explizit gezeichnet werden. Beispiele für Anwendungen in der Praxis können News-Maps sein, wie im Schaubild 4.23 zu sehen ist. Dabei werden News-Kategorien farbig kodiert und die Häufigkeit der Meldungen durch Größe repräsentiert. Zusätzlich wird mit der Helligkeit das Alter einer Nachricht kodiert [25]. Bei der Darstellung von CompositeServices oder AdapterServices ist die Darstellungsform nur bedingt geeignet da die Struktur der Modelle nicht mehr erkennbar wird.

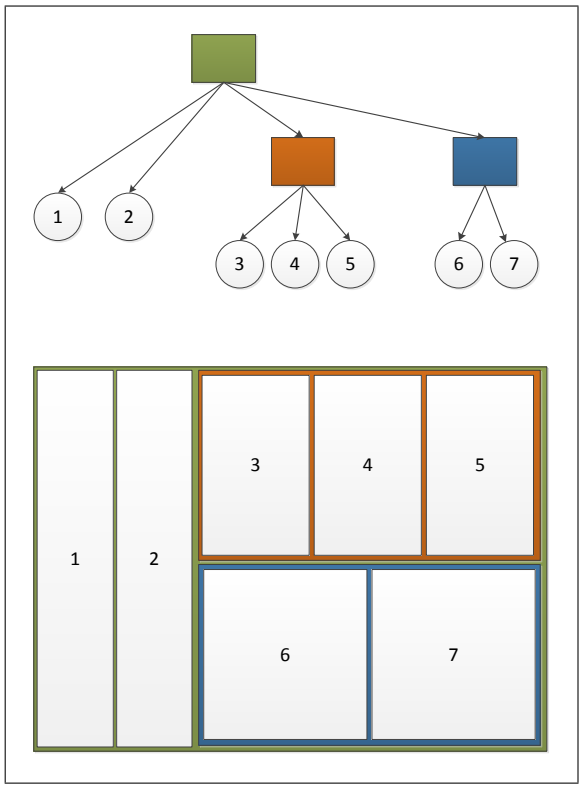

Abbildung 4.22: Konstruktion einer Treemap (nach [25])

Geschichtete Ansätze Während der flächenfüllende Ansatz meistens den kompletten Bereich eines Bildschirms ausnutzt, stellt die Baumvisualisierung auf Basis von Schichtung einen Kompromiss in der Flächennutzung dar. Die Ausrichtung der Knoten wird als Mittel zur Strukturierung der Daten eingesetzt. Die von Kruskal und Landwehr [23] beschriebenen *Icicle Plots* erlauben die Darstellung einer Hierarchie, indem sie den Knoten jeder Hierarchieebene genau

Abbildung 4.23: Eine Treemap-Darstellung von Google-News-Meldungen (Screenshot von http://newsmap.jp/

eine Zeile mit konstanter Höhre zuordnet. Die Abbildung 4.24 zeigt, wie dem Wurzelknoten 1 eine komplette Zeile zugeordnet ist. Seine beiden Kinder 2 und 3 werden genau darunter angeordnet, wobei ihnen prozentual so viel Platz zusteht wie die ermittelte Größe des jeweiligen Teilbaumes. Der Baum wächst dabei weiter nach unten bis schließlich die Blattknoten gezeichnet werden. Der wesentliche Unterschied zu den Treemaps ist die Sichtbarmachung der Zwischenknoten als Rechtecke, womit die Baumstruktur leichter zu erkennen ist. Die Aufteilung der Knoten kann sowohl vertikal erfolgen, wie in der Abbildung 4.24 gezeigt wird, aber auch kreisförmig als Ring angeordnet [25]. Wie bei dem verschachtelten Ansatz können hier ebenfalls Farbkodierungen eingebaut werden. Zusammengefasst lässt sich sagen, dass für die Darstellung von CompositeServices und AdapterServices, eine kombinierte Form der Darstellung aus dem geschichteten–und verschachtelten Ansatz, als erstrebenswert angesehen wird. Ein konkretes Visualisierungskonzept wird im späteren Verlauf vorgestellt.

Netzwerkvisualisierungen

Ein Netzwerk lässt sich als Graph repräsentieren und besteht aus einer Menge von Knoten und Kanten, wobei jede Kante eine Verbindung zweier Knoten dieser Menge darstellt. Dabei lassen sich Datenobjekte oder Attribute auf Knoten abbilden, während Kanten die Beziehungen zu anderen Objekten repräsentieren. Während bei Hierarchien die durch Kanten beschriebenen Relationen häufig nur eine Enthaltensein Abähngigkeit darstellen, können Kanten in Graphen

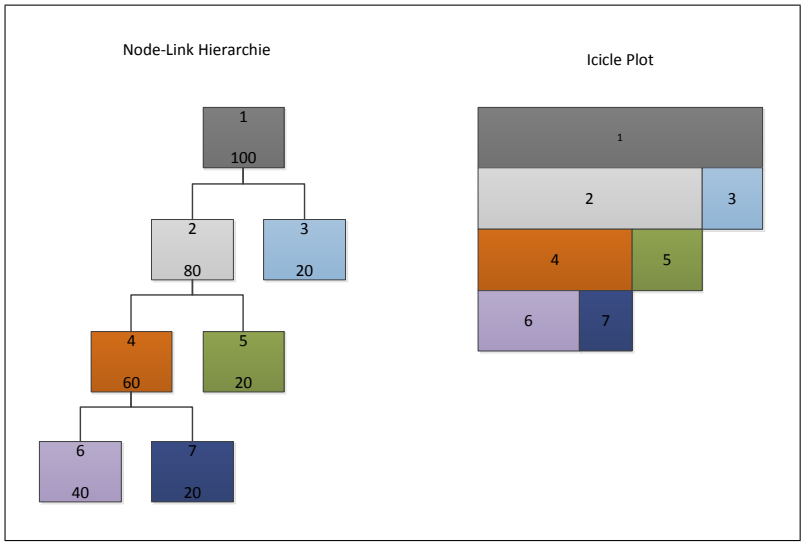

Abbildung 4.24: Der als Node-Link-Hierarchie dargestellte Baum als Icicle Plot (nach [25])

gerichtet und gewichtet sein und damit eine vielzahl von Beziehungen abbilden [25]. Damit wird die Darstellung von Datenmodellen in den DataServices möglich gemacht. Es können nämlich jegliche Vererbungsbeziehungen, Aggregationen und Assoziationen abgebildet werden. In den ClientGateways lässt durch diese Visualisierungstechnik das Eventhandling grafisch darstellen. Im vorausgegangenen Abschnitt wurden Visualisierungskonzepte für Bäume als Spezialform von Graphen vorgestellt. Bei der Graphenvisualisierung kommen weitere Aspekte hinzu, die in Hinsicht auf Relevanz im Folgenden untersucht werden.

Layout Ein wesentlicher Aspekt beim Zeichnen eines Graphs ist die räumliche Positionierung von Knoten und Kanten. Dafür wurden zahlreiche Algorithmen entwickelt, die einerseits das Ziel haben eine visuelle Klarheit zu gewährleisten, andererseits aber auch eine ästhetische Qualität sicherzustellen. Für die Darstellung von textuellen Modellen entstanden folgende Anforderungen an das Layout:

- Darstellung von Hierarchien begünstigen
- Horizontale oder Vertikale Darstellung der Baumstruktur
- Subgraphendarstellung zwecks Strukturierung
- Möglichst keine Überschneidung von Kanten
- Gleichmäßige Knotenverteilung
- Symmetrische Darstellung

Knotenreprärsenation Bei der Darstellung von Knoten ergeben sich unterschiedliche Möglichkeiten. So können alle den gleichen Typ besitzen oder es werden unterschiedliche geometrische Objekte für die Knotendarstellung verwendet. Für die Darstellung von Datenobjekten und Attributen aus den textuellen Modellen ergeben sich folgende Designkriterien bei der Knotendarstellung:

- Rechtecke als primäre Form für Datenobjektdarstellung
- Trapezform als Darstellungsform für Abhängigkeiten
- Knotennamen als Textfelder
- Attribute von Datenobjekten im Inneren des Knotens darstellbar
- Textlabels als Strukturierungsmittel
- Knoten sollen visuell gruppiert werden z.B. durch Farbkodierung der Fläche
- Größe der Knoten soll modifizierbar sein um Objekte hervorheben zu können

Kantenrepräsentation Um unterschiedliche Arten von Beziehungen zwischen Knoten darstellen zu können, wird zwischen gerichteten–und ungerichteten Kanten unterschieden. In der Visualisierung werden zur Darstellung von gerichteten Kanten Pfeile und zur Darstellung von ungerichteten Kanten Linien verwendet. Zusätzlich können diese Kanten visuelle Eingenschaften wie Dicke oder Farbe zur Kodierung von Attributen einsetzen. Weitere Varianten entstehen durch gekrümmte oder gerade Linien zwischen den Knoten, aber auch durch das Beschriften von Kanten mit Attributwerten. Für die Visualisierung von DataServices ist die Abbildung von Assoziationen, Aggregationen und Vererbung von großer Wichtigkeit. Durch die weite Verbreitung der UML-Sprache in der Softwareentwicklung, soll für die Beziehungen zwischen den Knoten, eine an UML angelehnte Notation eingesetzt werden. Damit soll das Ergonomiekriterium Erwartungskonformität erfüllt werden. In den ClientGateways sollen durch gerichtete Kanten die View-Events mit den auszulösenden Aktionen verbunden werden.

4.2.4 Visualisierungsentwurf für grafische Modellrepräsentation

Für die Visualisierung von Software-Artefakten der WFS-Plattform wurden im Kapitel 4.2.3 verschiedene Darstellungsformen untersucht. Als Ergebnis lässt sich festhalten, dass für die Visualisierung von CompositeServices und AdapterServices, eine hierarchische Darstellungsform in Betracht gezogen wird. Um die Struktur der Modelle hervorzuheben und eine kompakte Form

der Darstellung zu erhalten, wird ein Entwurf auf Basis eines geschichteten Visualisierungsansatzes entwickelt. Für die grafische Darstellung von DataServices und ClientGateways eignet sich eine graphenbasierte Darstellungsform. Dadurch können unterschiedliche Relationen zwischen Datenobjekten oder Attributen dargestellt werden. Unter Berücksichtigung der Erkenntnisse aus dem vorherigen Kapitel und der ermittelten Anforderungen, wird im Folgenden ein Visualisierungskonzept vorgelegt.
Bei der Entwicklung von grafischen Diagrammen wird ein prototypenbasierter Ansatz verfolgt, nach dem internationalen Standard *EN ISO 9241*, siehe Schaubild 4.25. So werden mehrere Durchläufe mit Prototypen in Form von Mocks durchgeführt und jeweils mit den Mitarbeitern aus der Abteilung abgestimmt. Erfüllt ein Entwurf die wichtigsten Anforderungen, wird dieser zur Impelementierung freigegeben. Dabei wird für jeden Artefakttyp ein grafisches Modell entwickelt, das als Basis für die technische Umsetzung im Generator dient.

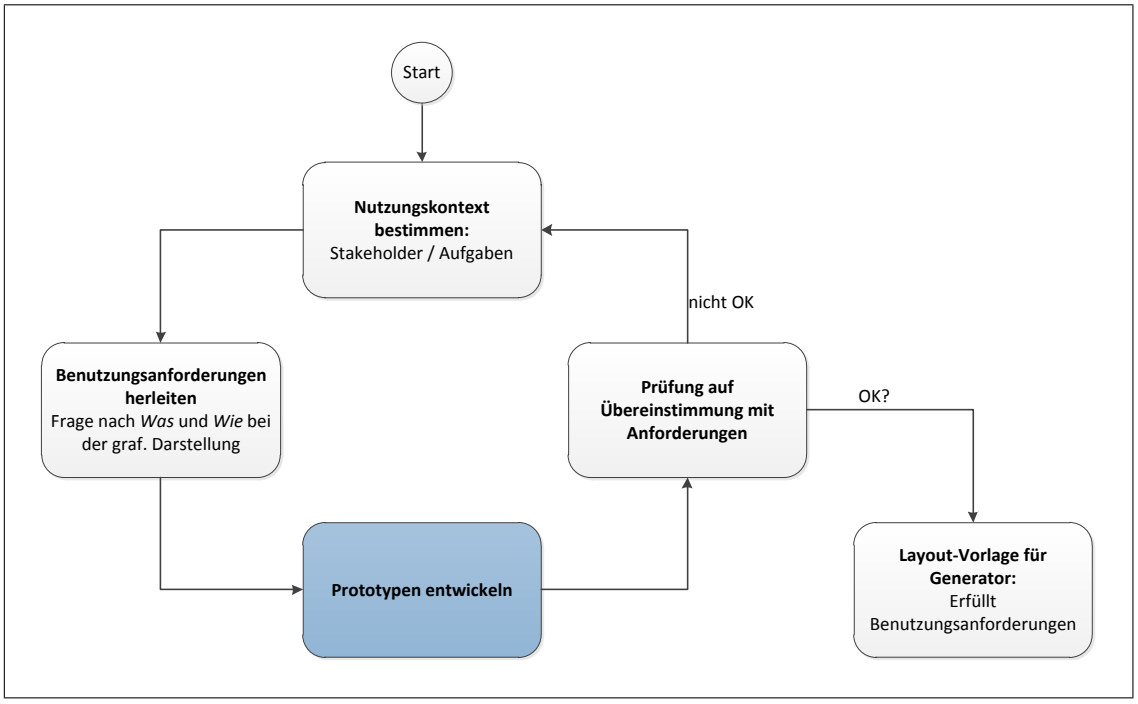

Abbildung 4.25: Prototypenbasierter Entwicklungsprozess (nach EN ISO 9241)

Entwurf CompositeService und AdapterService

Die Artefakte CompositeService und AdapterService sind von der Modellstruktur sehr ähnlich, weswegen hier nur ein Entwurf des CompositiServices vorgestellt wird.
Ein wesentlicher Punkt bei der Abbildung von CompositeServices liegt darin, die enthaltenen

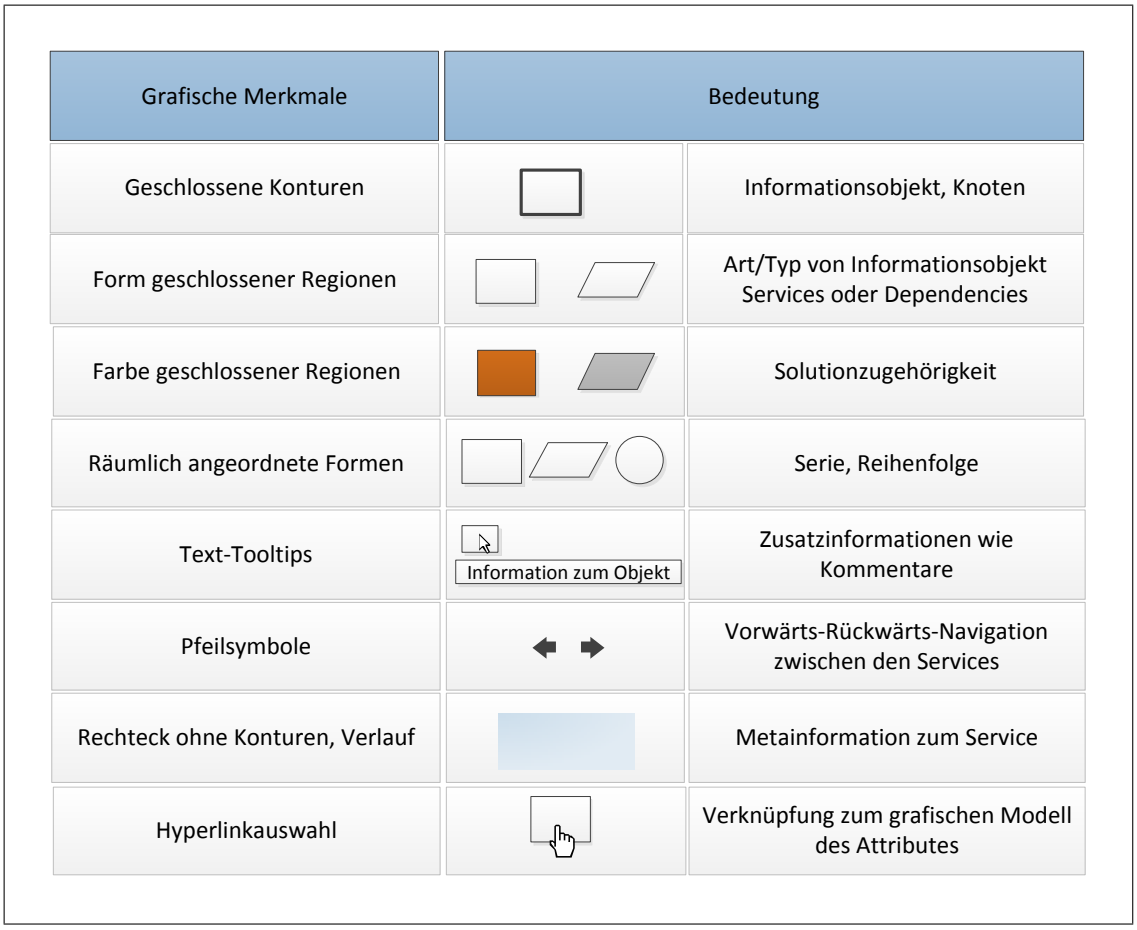

Abbildung 4.26: Diagrammelemente CompositeService, AdapterService

Servicemethoden grafisch sichtbar zu machen und besonders hervorzuheben. Bei der Visualisierungsform wird ein geschichteter Ansatz verfolgt. Die Abbildung 4.26 zeigt einzelne Diagrammelemente die zur Darstellung verwendet werden. Dabei repräsentiert das äußere Rechteck den Wurzelknoten des Services. Als Relation kommt nur die Enthaltenseinbeziehung zum Einsatz. Deswegen werden die Elemente die ein CompositeService enthält, im inneren des Rechteckes angeordnet. Als erstes Informationsobjekt wird der Name des CompositeServices samt der dazugehörigen Solution, Feature und Unit, zentral oben angeordnet. Der Verlauf von hell nach dunkel des Inhaltes soll die Information dezent hervorheben und als eine Einheit wahrnehmbar machen. Als nächstes werden im helleren Rechteck die Servicemethoden visualisiert. Durch das Rechteck wird ebenfalls eine Enthaltenseinbeziehung symbolisiert. Die einzelnen Servicemethoden sind untereinander angeordnet und werden als eine Liste wahrgenommen. Zudem werden die Methodennamen durch eine schwarze Schrift hervorgehoben. Ein

Mouseover-Effekt macht die Kommentare der jeweiligen Servicemethoden sichtbar, dadurch soll nicht der Blick vom Wesentlichen abgelenkt werden. Überschreitet die Methodenliste eine bestimmte Größe, soll ein Scrollmechanismus die Navigation unterstützen. Als nächstes Informationsobjekt werden die vom CompositeService abhängigen Artefakte in einer Raute visualisiert. Diese Geometrieform macht den wechselnden Kontext zu den Services sichtbar. Äquivalent zu den Servicemethoden, wird bei größerer Elementenanzahl ein Scrollbalken implementiert. Um weitere Informationen zu den einzelnen Artefakten zu bekommen, werden die grafischen Modelle durch Hyperlinks miteinander verbunden. Für die Navigation zwischen den verschiedenen CompositeServices werden im oberen Bereich Pfeilsymbole als Hyperlinks eingebunden. Die Hintergrundfarbe der einzelnen Informationsobjekten soll zusätzlich die Solutionzugehörigkeit veranschaulichen.

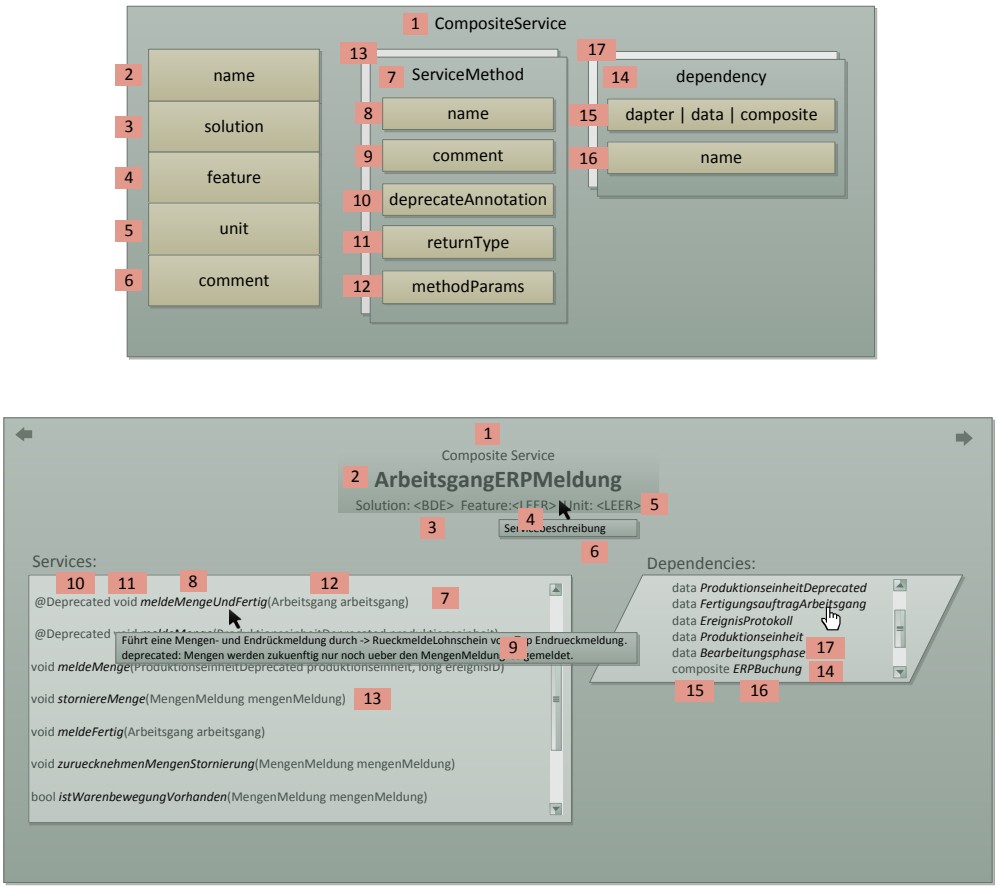

Abbildung 4.27: Grafische Repräsentation des CompositeServices (unten) mit Mapping von Informationen (oben)

Mapping aus dem Metamodell Im Kapitel 4.1 wurden die Metamodelle der Artefakte auf Relevanz geprüft und daraus die zu visualisierenden Informationen gewonnen. Diese werden im folgenden Schaubild 4.27 auf das neu entstandene Diagramm gemappt. Dabei repräsentieren die roten Zahlen die Mappingvorschrift vom oberen zum unteren Diagramm. Das hier vorliegende Ergebnis dient als Vorstufe für das Rendering und als Vorlage für die notwendigen Transformationsregeln im Generator.

Entwurf ClientGateway

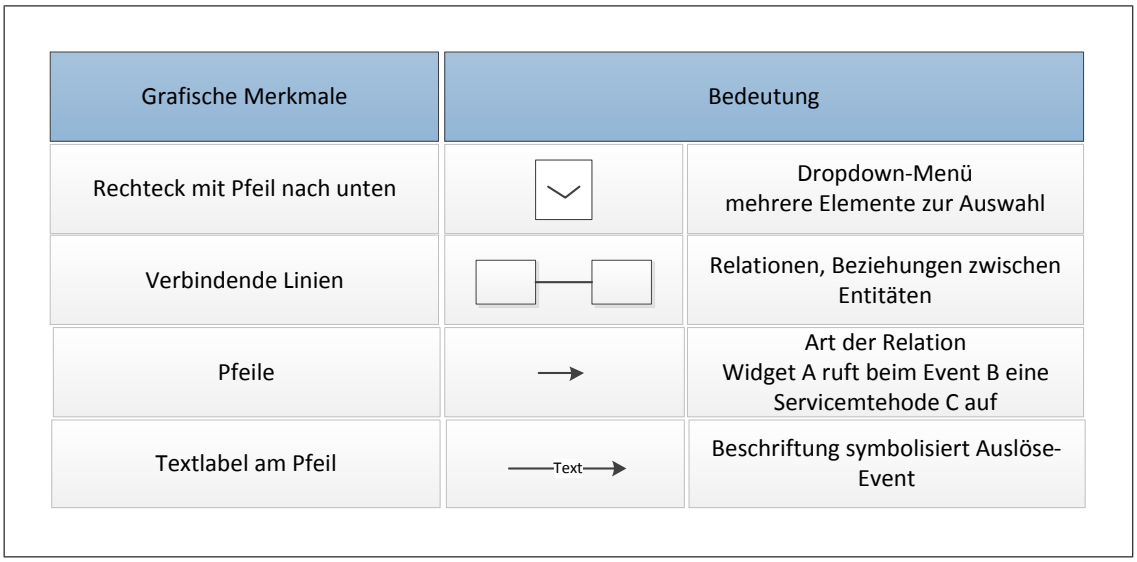

Abbildung 4.28: Hinzukommenden Diagrammelemente des CientGateways

Die Visualisierung des ClientGateways soll die drei wichtigen Informationselemente des Komponentenmodells (Views, Services und Dependencies) hervorheben. Dabei soll das Eventhandling zwischen den Views und den Services grafisch veranschaulicht werden. Da in diesem Fall verschiedene Datenobjekte in einer Beziehung zueinander stehen, die nicht nur das Enthaltensein widerspiegelt, langt der hierarchische Ansatz zur Visualisierung nicht aus. Aus diesem Grunde wird als Repräsentationsform eine Graphendarstellung angewandt. Der geschichtete Ansatz aus der CompositeService-Visualisierung wird um weitere Darstellungsmerkmale erweitert um Beziehungenstypen abbilden zu können, siehe Schaubild 4.28. Das Eventhandling wird durch eine gerichtete Verbindung zwischen einer Kontrollstruktur z.B. einem Widget und einer Servicemethode grafisch dargestellt. Werden in einem ClientGateway mehrere Views definiert, so sollen diese in der Visualisierung in einer Combo-Box zur Auswahl gestellt werden. Die Startview des ClientGateways soll dabei immer an erster Stelle abgebildet werden. Zusatzinfor-

mationen zu den verschiedenen Informationselementen sollen durch einen Mouseover-Effekt dargestellt werden. Die Darstellung von Knoten, die Farbkodierung und die Navigation zwischen den Artefakten soll dabei analog zur CompositeService-Visualisierung übernommen werden.

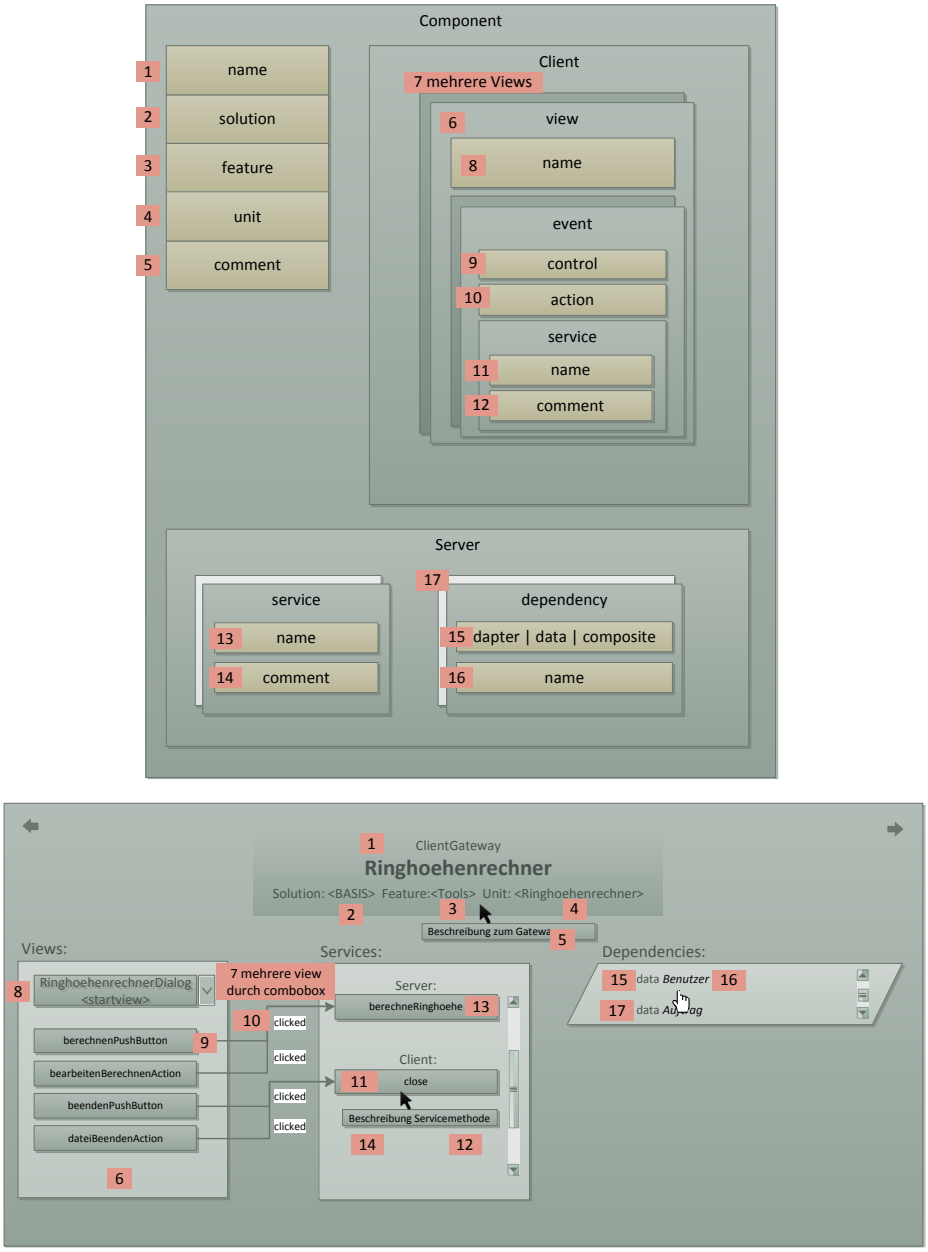

Abbildung 4.29: Grafische Repräsentation des ClientGateways (unten) mit Mapping von Informationen (oben)

Mapping aus dem Metamodell Das Mapping der aufbereiteten Informationen aus dem Kapitel 4.1 erfolgt konzeptionell im Schaubild 4.29. Wie das CompositeService-Diagramm dient auch dieses Ergebnis im weiteren Verlauf als Grundlage für Transformationsregeln.

Entwurf DataService

Ein Dataservice beinhaltet neben Servicemethoden genauso Datenmodelle die eine bestimmte Domäne abbilden. Diese Datenmodelle bilden die Möglichkeit die wachsende Datenbankstruktur der MES-Software zu strukturieren. Um die Datenstrukturen aus den Modellen visualisieren zu können, wird ein graphenbasierter Ansatz gewählt. Dabei soll eine symmetrische Darstellung von Knoten und Kanten bevorzugt werden. Die Servicemethoden werden bei der Visualisierung komplett vernachlässigt, da diese in der Anforderungsanalyse als unwichtig erachtet wurden. Aufgrund der breiten Akzeptanz der UML, soll die Darstellung der Datenmodelle sich am Klassendiagramm der UML orientieren. Mögliche grafische Elemente die in dem Diagramm zum Einsatz kommen, werden in der Abbildung 4.30 aufgeführt.
Dabei wird jedes Dataobject aus dem DataModel als ein Knoten in Form eines Kästchens dargestellt. Handelt es sich um Dataobjects vom Typ transient, abstract oder embedded, wird dieses in die Namensbeschriftung als Schlüsselwort hinzugefügt. Bei der Abbildung von Beziehungen wird zwischen verschiedenen Assiziationstypen unterschieden. Einfache Assoziationen werden als gerichtete Linien mit Pfeilen dargestellt und stellen eine Referenz zwischen zwei Dataobjects dar. Eine Generalisierung ist eine gerichtete Beziehung zwischen einem generellen und einem speziellen Dataobject, diese wird durch eine durchgezogene Linie mit einer nicht ausgefüllten Pfeilspitze gezeichnet. Eine weitere Beziehung die im Datamodel vorkommt, ist die Aggregation. Diese wird durch eine durchgezogene Linie mit einer Pfeilspitze und einer Raute symbolisiert. Durch eine Aggregation können z.B. Embedded Dataobjects innerhalb eines Datenmodells instanziert werden, dabei wird die Instanzvariable zur Beschriftung der Linien verwendet. Werden Dataobjects aus anderen Datamodels referenziert, so soll diese Beziehung besonders sichtbar gemacht werden. Deswegen fallen diese Knoten aus der Hierarchie des Wurzelknotens heraus und werden alleinestehend dargestellt. Desweiteren sollen alle Knoten die nicht zum Datamodel dazugehören, eine Hyperlinkfunktion realisieren, um zum dazugehörigen Datamodel navigieren zu können.

Grafische Merkmale	Bedeutung	
Geschlossene Konturen, rechteckige Form		Datenobjekt, Teil eines Datamodels
Farbe geschlossener Regionen		Solutionzugehörigkeit
Text-Tooltips		Mapping von Attributen auf Datenbankspalten
Räumlich angeordnete Formen		Serie, Reihenfolge
Verbindende Linie mit Pfeilspitze		Referenzbeziehung zwischen zwei Dataobjects
Verbindende Linie mit Pfeilspitze und Raute		Aggregationsbeziehung zwischen zwei Dataobjects
Verbindende Linie mit unausgefüllter Pfeilspitze		Generalisierungsbeziehung zwischen zwei Dataobjects
Beschriftete Linien		Bezeichnung der Instanzvariablen
Pfeilsymbole		Vorwärts-Rückwärts-Navigation zwischen den Datenmodellen
Rechteck ohne Konturen, Verlauf		Metainformation zum Datenmodell
Hyperlinkauswahl		Referenzverknüpfung zum Datamodel eines Attributes außerhalb des eigenen Dataservices

Abbildung 4.30: Diagrammelemente zur Visualisierung eines DataServices

Alle Attribute eines Dataobjects vom Typ Simple, Enum, Measure oder State sollen innerhalb des Knotens untereinander aufgeführt werden. Sollten die Attribute eines Dataobjects über ein Mapping auf eine Datenbankspalte verfügen, so soll dieses durch ein Tooltip sichtbar gemacht werden. Weiterhin soll die Farbkodierung der Solutions auch in diesem Diagramm zum Ausdruck kommen. Falls ein Dataobject ein Statuskonzept realisiert, soll dieser als Zustandsautomatgraph mit gerichteten Kanten gezeichnet werden.

Mapping aus dem Metamodell Ein mögliches Visualisierungskonzept wird mithilfe eines Mocks in der Abbildung 4.31 vorgestellt. Dabei werden die gewünschten Informationen des gefilterten Metamodells aus dem Kapitel 4.1 in dem Diagramm abgebildet. Alle aufgeführten Attribute in der oberen Abbildung werden durch Zahlenverknüpfungen den unteren Diagrammelementen zugeordnet. Referenze auf Elemente außerhalb des eigenen Modells werden aus dem Hauptknoten herausgenommen um die Zugehörigkeit zu symbolisieren. Das Statuskonzept wird von der Statemachine auf den Automatengraphen direkt gemappt (Markierung 17). Weiterhin ist zu erkennen, dass unterschiedliche DataObjects ihren Typen im Namen kodiert haben. Die einzelnen Attribute eines DataObjects werden innerhalb des Rechtecks als eine Liste aufgeführt. Nebenbei wird die an UML angelehnte Darstellung des Diagramms sichtbar.

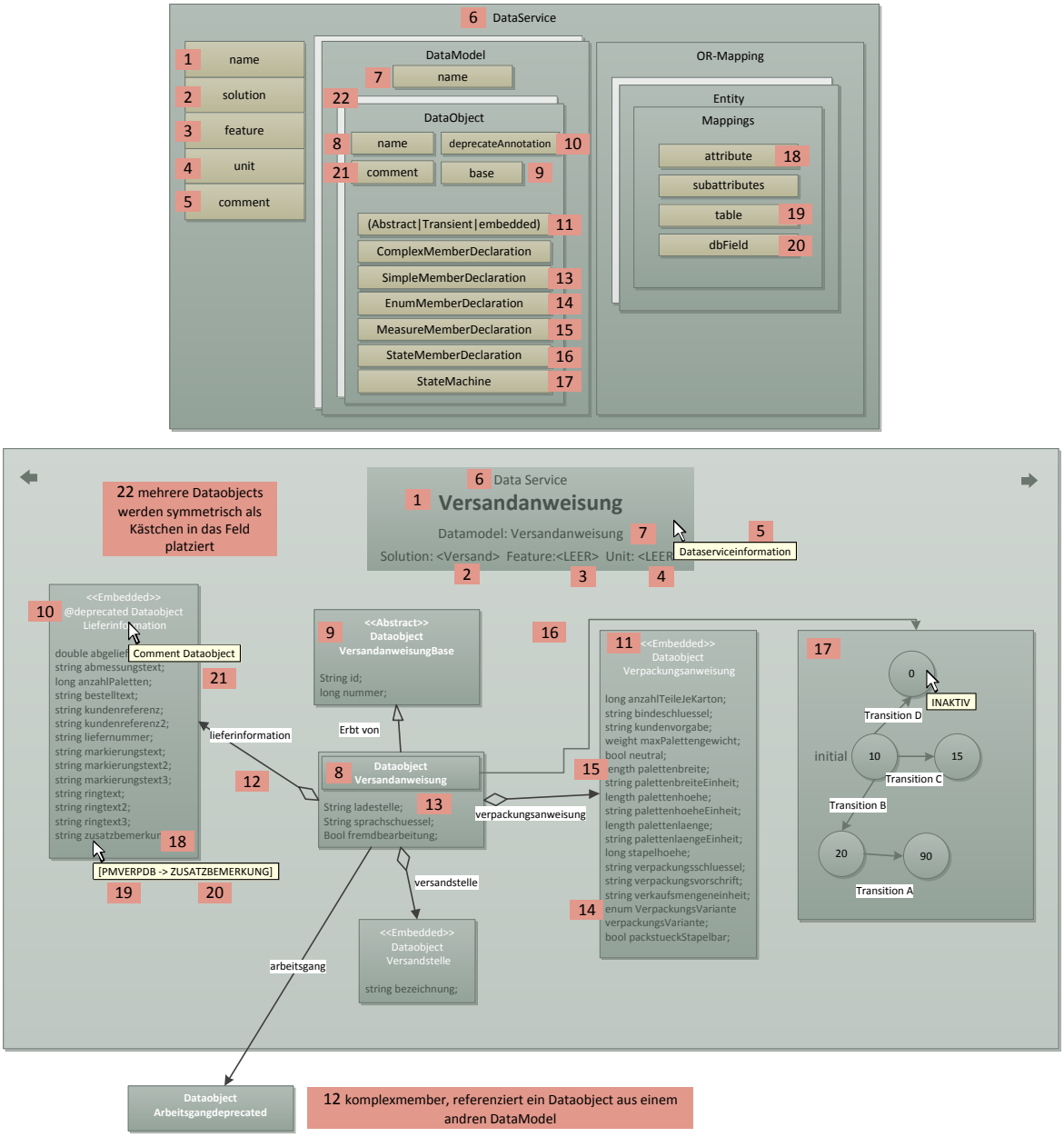

Abbildung 4.31: Grafische Repräsentation des DataServices (unten) mit Mapping von Informationen (oben)

4.3 Technologieanalyse

Das vorherige Unterkapitel beschreibt Visualisierungskonzepte für die grafische Darstellung von textuellen Softwaremodellen. Dabei wird ein Transformationsmodell entwickelt, das Informationsdaten in ein geometrisches Modell überführt. Die nächste Phase in dem Referenzmodell für Visualisierung ist die Bildgenerierung (Rendering), siehe Abbildung 4.32. Dieses Kapitel gibt einen Überblick über bestehende Werkzeuge, die zur Visualisierung von textuellen Modellen eingesetzt werden können. Dabei handelt es sich um Werkzeuge, die eine Graphenvisualisierung unterstützen.
Als nächsten Schritt sollen mögliche MDSD-Tools zur Erzeugung von Transformationen untersucht werden. Ziel ist es, mithilfe eines Generators, aus den textuellen Modellen, die gewünschten Graphendiagramme anzufertigen. Dabei soll die Integration in die MDSD-Architektur der WFS angestrebt werden. Als Ergebnis wird ein technisches Basiskonzept vorgestellt, das als Basis für die Implementierung dienen soll.

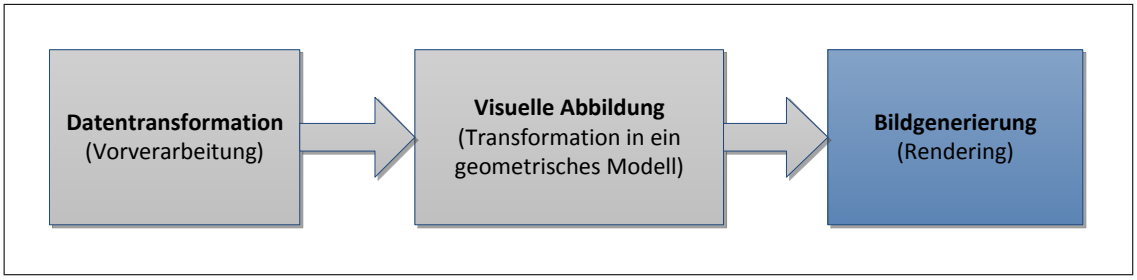

Abbildung 4.32: Bildgenerierung nach dem Referenzmodell (nach [10])

4.3.1 Visualisierungsverkzeuge

Es existieren eine Reihe von Programmen und Bibliotheken zur Analyse, Visualisierung und Editierung von Graphen. Dabei werden unterschiedliche Funktionalitäten je nach Anforderungen bereitgestellt. Weiterhin werden hier einige solcher Werkzeuge vorgestellt und anhand folgender Anforderungen evaluiert.

Anforderungen

Die Anforderungen an das Renderingtool zur Umsetzung der Diagrammen, welche im Abschnitt 4.2.4 vorgestellt wurden lauten:

Betriebssystemunabhängigkeit Die Nutzbarkeit von Software soll unter gängigen Betriebssystemen wie Microsoft Windows oder Linux möglich sein. Damit soll das Ziel der IT-Abteilung, Zukunftssichere Software zu entwickeln unterstützt werden.

Benutzerfreundlichkeit Die Softwaretools zur Graphendarstellung sollen möglichst übersichtlich gestaltet sein und dabei intuitiv zu bedienen sein. Die erstellten Diagramme sollen dabei automatisch angeordnet werden und mögliche Layoutveränderungen unterstützen.

Kompatibilität und Flexibilität Die Applikation soll eine Möglichkeit bereitstellen Graphen aus einer Datenstruktur einlesen zu können und anzuzeigen. Die Informationen aus den textuellen Modellen sollen in eine geeignete Datenstruktur transformiert werden und anschließend an das Renderingtool übergeben werden.

Exportmöglichkeiten Das Graphtool soll eine Exportfunktion bereitstellen, um aus den Graphen Formate wie SVG, PNG oder PDF exportieren zu können. Damit soll es möglich werden die Diagramme in anderen Systemen zu visualisieren.

Knoten- und Kantendarstellung modifizierbar Um Informationsdaten strukturiert darstellen zu können, bedarf es einer Anpassung des Layouts von Knoten und Kanten. Dabei sollen Farben, Größe und Beschriftungsmöglichkeiten angeboten werden. Das Visualisierungswerkzeug soll zudem das Rendern von Subgraphen unterstützen.

Lizenzierung Ziel ist es für die Entwicklung der grafischen Repräsentation von textuellen Domänenmodellen kommerzielle Linzenzen zu vermeiden. Damit soll die Entwicklung keine Zusatzkosten produzieren, wodurch man unabhängiger agieren kann.

Im Folgenden werden nun bekannte Applikationen *aiSee*, *yEd/yFiles* und *GraphViz* unter der Berücksichtigung dieser Kriterien näher untersucht.

aiSee - Graph Visualization

aiSee ist eine kommerzielle Anwendung zur Graphenvisualisierung, entwickelt von der Firma AbsInt [1]. Die Visualisierung basiert auf einem schichtenbasierten Verfahren für gerichtete

Graphen mit Gruppenknoten (siehe Beispielgraphen in Abbildung 4.33). Die Gruppenknoten können gefaltet und wieder entfaltet werden. Zudem besteht die Möglichkeit, Kanten in Klassen einzuteilen und bei Bedarf ein–oder auszublenden. Das Tool bietet bis zu 15 verschiedene Layoutalgorithmen zur Darstellung von Graphen. Die Diagramme können zudem in Bildformate wie PNG, SVG oder PostScript exportiert werden. Der Hersteller bewirbt die Anwendung, die weltweit von Tausenden Benutzern in vielen verschiedenen Bereichen eingesetzt wird, z.B. in Bereichen wie Ahnenforschung, Projektmanagement, Softwareentwicklung, Datenbankverwalung oder Informatik. aiSee kann textuell beschriebene Graphen (GDL-Graphformat) einlesen und visualisieren. Die Software ist für Windows, Linux und MacOS erhältlich.

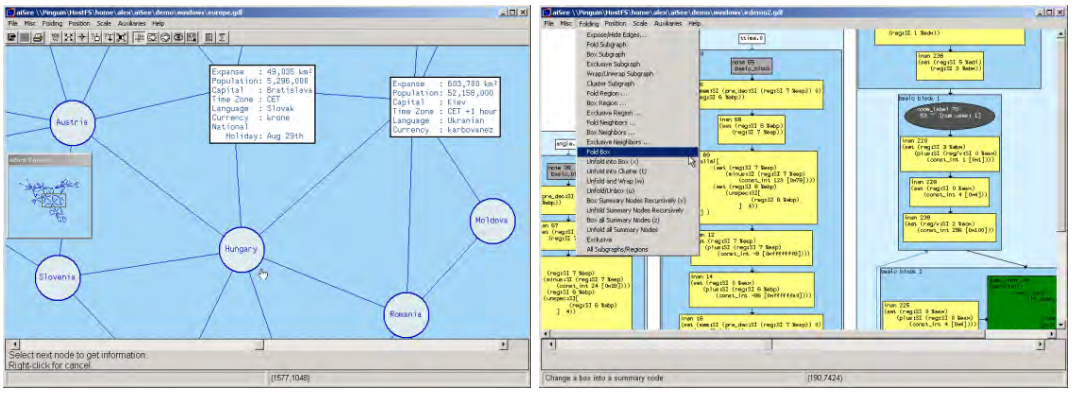

Abbildung 4.33: aiSee-Features: zusätzliche Informationen über Knoten (links), Verschachtelung von Graphen (rechts) [1]

yEd/yFiles

Die Anwendung *yEd* ist eine frei erhältliche Visualisierungssoftware und ist für Darstellung jeglicher Graphen konzepiert. yEd ist ein Editor mit umfassenden Funktionen zum Erzeugen und Bearbeiten von Graphen. Die Oberfläche basiert auf Java Swing und bietet eine übersichtliche Anordnung der Funktionen. Ebenso können externe Daten importiert werden, es werden Formate wie GraphML, Excel, GML oder XML unterstützt. Für den Export von Diagrammen bietet yEd Unterstützung für folgende Bild- und Vektorgrafikdateiformate: PDF, SVG, JPG, GIF, BMP oder SWF. Der Editor yEd läuft auf jeder Windows, Linux und Mac-Plattform. Der Anwendung yEd liegt die Java Graphen-Bibliothek *yFiles* zugrunde. Diese unterstützt erweiterbare Einbettungsverfahren, mit verschiedenen Parametern. So wird das schichtungsbasierte Verfahren realisiert, das Gruppenknoten berücksichtigt. Die Nutzung der Java-Bibliothek ist mit Lizenzkosten

verbunden [5]. Beim manuellen Erstellen von Graphen bietet yEd die vielfältigsten Möglichkeiten Diagramme darzustellen, allerdings ist die Benutzung von yFiles ziemlich kostenaufwändig da zahlreiche Addons dazugekauft werden müssen um die Featurevielfalt nutzen zu können.

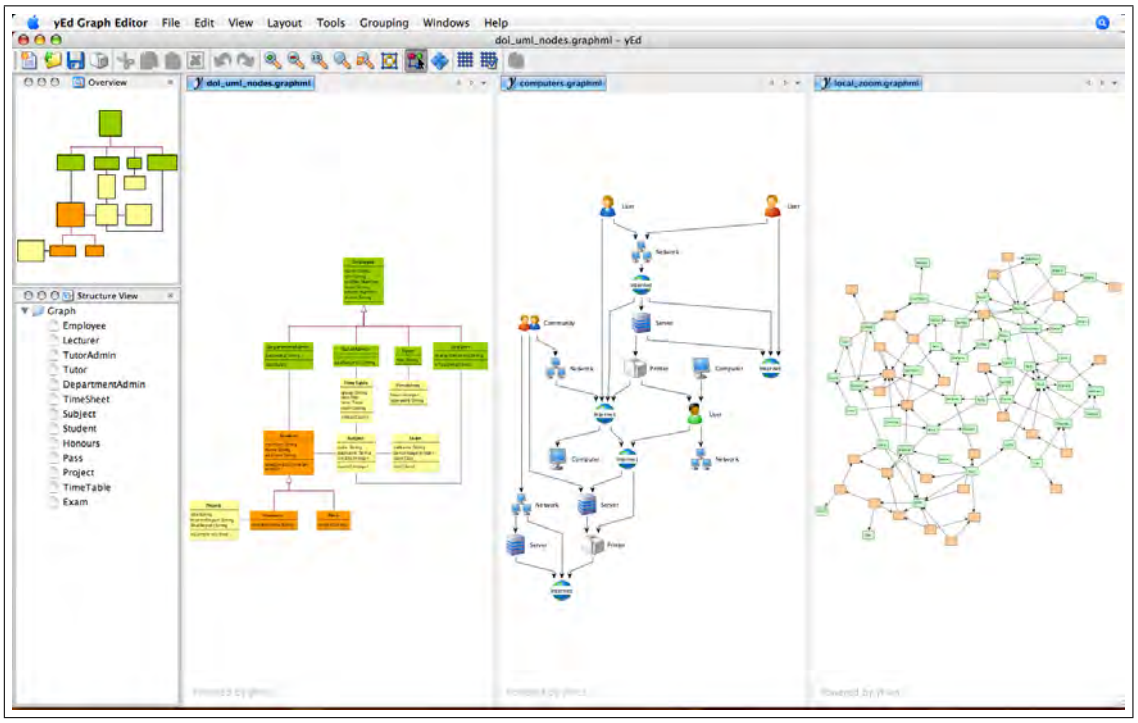

Abbildung 4.34: Verschiedene Darstellungstechniken im yEd-Editor [5]

GraphViz

Graphviz ist ein Open-Source-Werkzeug zur Visualisierung von gerichteten und ungerichteten Graphen und unterstützt ebenfalls den geschichteten Visualisierungsansatz. Es verfügt über mehrere Werkzeuge, unteranderem Kommandozeilentools wie *dot*, *neato* und *twopi* [4]. Diese bestimmen das Verfahren zur Visualisierung der Graphen, siehe Abbildung 4.35:

1. **dot**: Darstellung hierarchischer Strukturen. Alle Kanten verlaufen dabei in etwa in dieselbe Richtung, von oben nach untern oder von links nach rechts. Überschneidungen der Kanten werden möglichst vermieden und die Kantenlänge wird so kurz wie möglich gehalten.
2. **neato** und **fdp**: Visualisiert Graphen im so genannten „spring model“ Layout. Der Startknoten wird mittig angelegt. Neato benutzt dabei den Kamada-Kawai-Algorithmus. Fdp implementiert die Fruchterman-Reingold-Heuristik für größere Graphen.

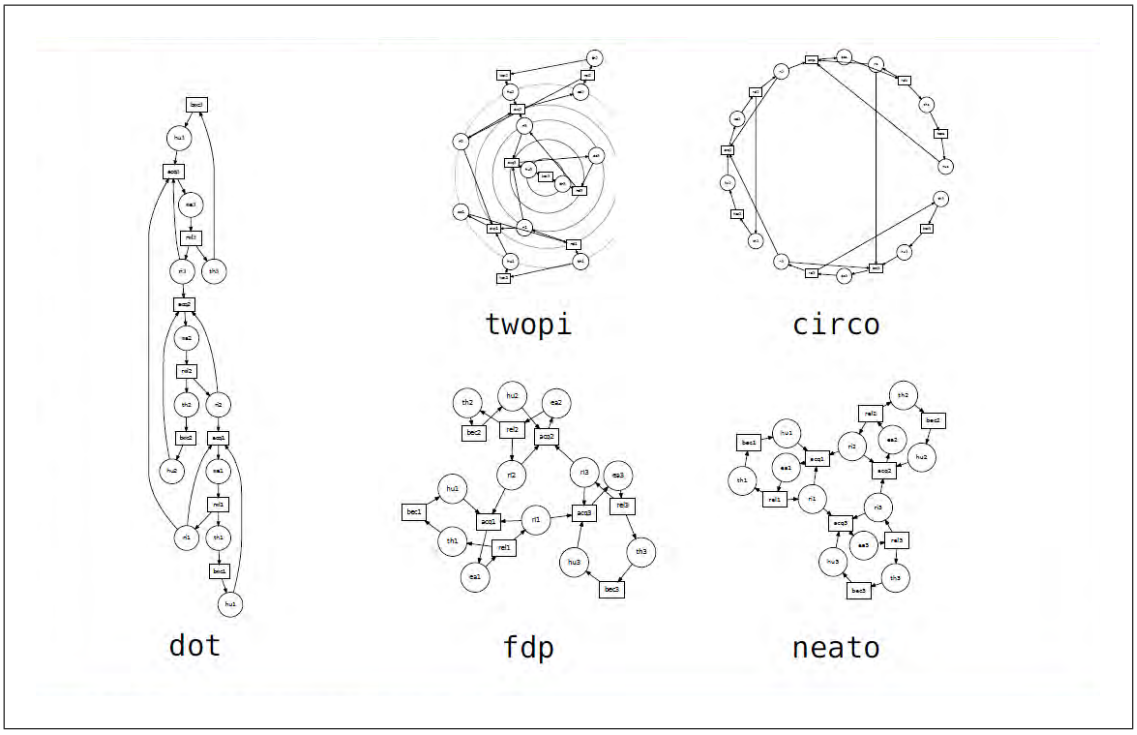

Abbildung 4.35: Vergleich Layout-Algorithmen von GraphViz[4]

3. **twopi**: Radiales Layout, nach Graham Wills.

4. **circo**: Circuläres Layout, nach Six and Tollis. [3]

Zudem bietet GraphViz einen Texteditor *lefty* und einen Graph-Viewer und -Editor *dotty*. Dotty beschränkt sich dabei primär auf die Visualisierung der Graphen und bietet keine besonderen Möglichkeiten zur Navigation. Zur Erzeugung von Diagrammen, verwendet GraphViz Anweisungen aus einer Textdatei, die eine Beschreibung der Knoten und Kanten enthält. Diese liegt in Form einer *DOT*-Sprache vor welche syntaktisch an die Programmiersprache C angelehnt ist, ein Beispiel dazu wird in der Abbildung 4.36 veranschaulicht. GraphViz bietet zudem verschiedene Layoutformen sowie eine Veränderung der Knotentypen und der Farbgebung. Im Allgemeinen genügt die Definition eines Graphen in Form einer DOT-Datei um akzetable Ausgaben zu erzeugen. Daher können besonders automatisch ablaufende Prozesse die Schnittstelle zur Visualisierung von Informationsdaten nutzen. Dazu wird ein DOT-Datei über einen Kommandozeilenbefehl an das Programm übergeben, mit einem Ausgabeparameter kann man das gewünschte Format festlegen. Es werden Dateiformate wie Postscript, SVG, JPEG, PNG und PDF unterstützt.

```
digraph G {

        subgraph cluster_0 {
                style=filled;
                color=lightgrey;
                node [style=filled,color=white];
                a0 -> a1 -> a2 -> a3;
                label = "process #1";
        }

        subgraph cluster_1 {
                node [style=filled];
                b0 -> b1 -> b2 -> b3;
                label = "process #2";
                color=blue
        }
        start -> a0;
        start -> b0;
        a1 -> b3;
        b2 -> a3;
        a3 -> a0;
        a3 -> end;
        b3 -> end;

        start [shape=Mdiamond];
        end [shape=Msquare];
}
```

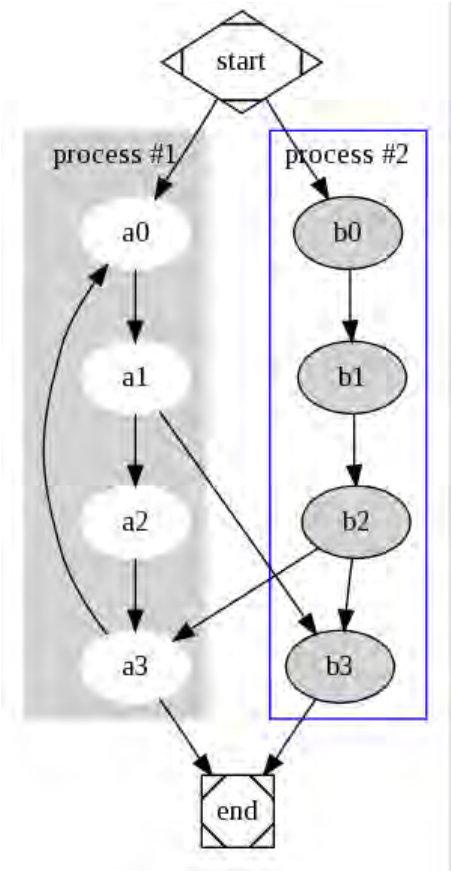

Abbildung 4.36: GraphViz unter der Verwendung der hierarchischer Darstellung der dot - Funktion [4]

Fazit

Die drei Visualisierungswerkzeuge die hier vorgestellt wurden, sind alle sehr mächtige Werkzeuge, manche zeigen allerdings geringfügige Schwächen hinsichtlich der aufgestellten Bewertungskriterien. Insgesamt lässt sich feststellen, dass das Werkzeug yEd/yFiles sehr innovativ ist und besonders viele Möglichkeit bei der Darstellung der Diagramme bietet. Allerdings ist man bei der Verwendung der yFiles-API hinsichtlich Exportmöglichkeiten eingeschränkt. So müssen zusätzliche Addons lizenziert werde, um den vollen Funktionsumfang genießen zu können. Der Editor yEd kann vollstens im Umfang und Bedienkomfort überzeugen, bietet aber keine automatisierte Exportmöglichkeit für den Modellgetriebenen Einsatz. Alle Werkzeuge lassen sich unter den gängigen Betriebssystemen einsetzen und erfüllen somit das Kriterium der Betriebssyste-

	Betriebssystem-unabhängigkeit	Benutzer-freundlichkeit	Kompatibilität und Flexibilität	Exportmöglich-keiten	Knoten- und Kantendarstellung modifizierbar	Lizenzierung
aiSee	+	0	0	+	+	-
yEd/yFiles	+	++	+	0	++	-
GraphViz	+	+	++	+	+	++

Abbildung 4.37: Bewertung der Visualisierungsprogramme *GraphViz, yEd/yFiles, GraphViz*

munabhängigkeit. Als klarer Favorit für den Einsatz im Modellgetriebenen Umfeld lässt sich das Visualisierungstool GraphViz ausmachen. Es bietet eine sehr einfach zu benutzende Schnittstelle um automatisch Diagramme in Form von SVG-Dokumenten oder PDF-Dokumenten zu generieren. Zudem ist es die einzige Lösung die im vollen Umfang kostenlos einsetzbar ist. Dieser Umstand bekräftigt die Entscheidung, GraphViz für die Visualisierung der textuellen Modelle einzusetzen.

4.3.2 MDSD Werkzeugunterstützung

Im vorherigen Unterkapitel werden verschiedene Applikationen untersucht, die das Rendering von Daten aus den textuellen Modellen ermöglichen sollen. Dabei wird das Graphenvisualisierungsprogramm GraphViz zur Umsetzung von grafischen Modellrepräsentationen gewählt. Um Diagramme herstellen zu können, benötigt das Programm ein Graphenmodell das in textueller Form vorliegen muss. Dieses soll mithilfe eines Generators erzeugt werden und an das Programm übergeben werden. Das Mittel dazu heißt Modelltransformationen. Die aufbereiteten formalen Modelle werden eingelesen und über Transformationsregeln in Zielmodelle transformiert, siehe Abbildung 4.39 (mehr zum Thema Transformationen wird im Kapitel 2.2.5 behandelt).
Da die Zielmodelle in textueller Form vorliegen, kann sowohl eine Model-To-Model-Transformationen, als auch eine Model-To-Text-Transformation durchgeführt werden.

Die Technologien, die in der MDSD-Umgebung der WFS eingesetzt werden, bauen auf das Eclipse Modeling Project auf, einige dieser Technologien werden im Schaubild veranschaulicht. Dabei wird das Framework *OpenArchitectureWare* eingesetzt, das wiederum eine Ansammlung von Frameworks darstellt. Bedeutende Frameworks wären z.B. Atlas(ATL), Xpand, Jet, Xtext/Xtend und EMF(Core). Durch die Verwendung des Ecore-Metamodells (EMF) in der WFS-

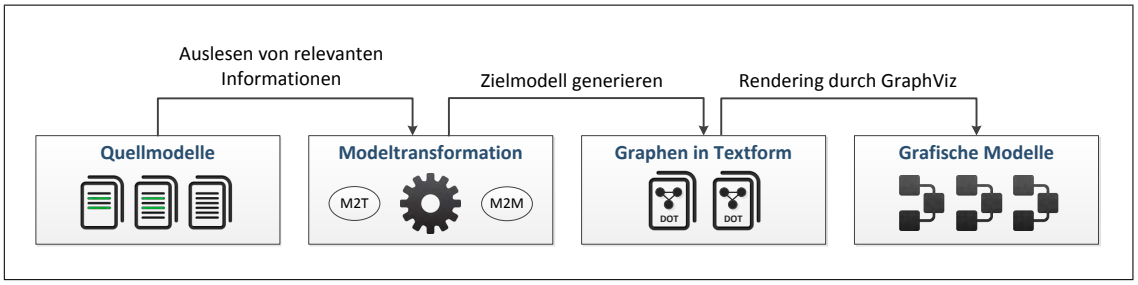

Abbildung 4.38: Generierung von grafischen Modellen durch Modelltransformation

MDSD-Architektur, soll die Technologieanalyse sich auf solche Tools beschränken, die dieses Metamodell unterstützen, da sonst der Aufwand für eine Anbindung zu hoch wäre.

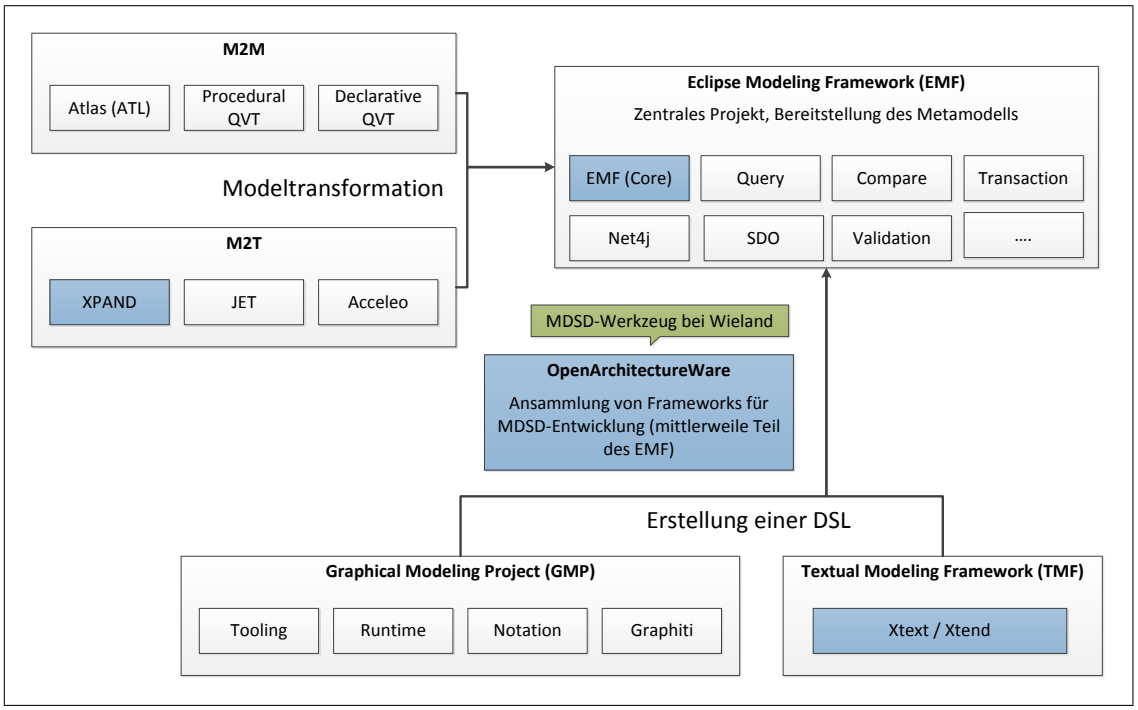

Abbildung 4.39: Überblick über Eclipse Modeling Project

Im Folgenden werden Werkzeuge zur Umsetzung von Modelltransformationen untersucht:

Xtend *Xtend* ist eine funktionale Transformationssprache die im oAW-Framework eingesetzt wird. Dabei kann die Sprache vielfältig eingesetzt werden, für unseren Fall soll die Unterstützung einer Model-To-Model-Transformation untersucht werden. Xtend Modelltransformationen

werden in Xtend-Dateien definiert, dabei ermöglicht es einen flexiblen Aufruf von Funktionen, indem zwei verschiedene Syntaxen unterstützt werden (siehe Quelltextausschnitt A.5):

- Funktionale Syntax
  Die Funktion a wird auf b angewendet: a(b)
- Objekt-Syntax
  Die Funktion a wird auf b angewendet: b.a()

```
1 private cached String getBindingPath(List[AttributeMemberReference] list, int i):
2    i >= list.size ?
3       ""
4    :
5       "."+list.get(i).getMember().name+getBindingPath(list, i+1);
6 ComponentMemberDeclaration getMember(ComponentDataBinding this):
7    member;
```

Listing 4.1: Definition von Funktionen in Xtend [19]

Zusätzlich unterstützt Xtend die Einbindung von Java-Methoden, so kann eine Java extension definiert werden, z.B.

```
1 Void myJavaExtension(String param) :
2 JAVA my.Type.staticMethod(java.lang.String);
```

Die Implementierung wird an eine statischen Java-Methode weitergereicht, die Syntax sieht wie folgt aus:

```
1 JAVA fully.qualified.Type.staticMethod(my.ParamType1, my.ParamType2, ...);
```

Durch die Verwendung von Java-Extensions ergeben sich zahlreiche Möglichkeiten der Anwendung von Xtend in unterschiedlichen Anwendungsszenarien.
Modelltransformationen werden durch Create-Extensions umgesetzt mit folgender Syntax [2]:

```
1 create Package toPackage(EPackage x) :
2   this.classifiers.addAll(x.eClassifiers.toClass());
3 create Class toClass(EClass x) :
4   this.attributes.addAll(x.eReferences.toReference());
5 create Reference toReference(EReference x) :
6   this.setType(x.eType.toClass());
```

Xpand *Xpand* ist eine Sprache zur Definition von Templates, um Model-To-Text-Transformationen umzusetzten. Durch die Verwendung eines Typsystems das in Xtend vorzufinden ist, erlaubt

es Xpand Extensions aus Xtend innerhalb von Templates aufzurufen. Darüberhinaus verfügt Xpand über eigene Sprachkonstrukte die zur Verbesserung der Lesbarkeit der Templates führen. Ein Beispieltemplate kann im folgenden Quelltextausschnitt aus dem Component-Generator begutachtet werden.

```
1  «FOREACH services AS service-»
2  «REM»TODO Comment«ENDREM»
3  public String «service.name»(String komponentenDaten, int validierungsstufe)
4           throws ApplicationException;
5  «ENDFOREACH-»
```

Xpand wird insbesondere für Programmcodegenerierung eingesetzt und kann in diesem Umfeld seine Stärken ausspielen. Die Integration in die Eclipse-IDE erlaubt es moderne Editoren mit Codevervollständigung und Refactoring-Tools aufzubauen um komfortabler entwickeln zu können. Im folgenden Quellcodeausschnitt lässt sich eine einfach Java-Klase erzeugen und in eine Datei ablegen.

```
1  «IMPORT meta::model»
2  «EXTENSION my::ExtensionFile»
3  «DEFINE javaClass FOR Entity»
4  «FILE fileName()»
5  package «javaPackage()»;
6  public class «name» {
7  // implementation
8  }
9  «ENDFILE»
10 «ENDDEFINE»
```

ATL Die ATLAS Transformation Language (*ATL*) ist eine Model-To-Model-Transformationssprache [21]. ATL ist ein Teil der ATLAS Model Management Architecture (AMMA) und wurde ursprünglich von den Forschungsgruppen INRIA (Institut National de Recherche en Informatique et en Automatique) an der Universität Nantes entwickelt. ATL wird als eine hybride Transformationssprache bezeichnet, weil deren Transformationsregeln sowohl im imperativen als auch im deklarativen Stil formuliert werden können. ATL definiert sowohl ein Metamodell als auch eine konkrete Syntax, zudem werden die Transformationen selbst als Modelle angesehen. Dadurch sind bei einer Transformation drei Metamodelle wie das Quell-Metamodell, das Ziel-Metamodell und letztendlich das Transformations-Metamodell beteiligt. Das zentrale Element einer Transformationsdefinition in ATL ist das sogenannte Modul, das einen Pflicht-Header, Import-Anweisungen, Helper-Operationen und die Regeln umfasst. Der Header definiert den Namen der Transforma-

tion und spezifiziert deren Quell- und Ziel-Metamodelle. Die Import-Anweisung kann Bibliotheken oder auch andere Module importieren. Helper-Operationen entsprechen Hilfsmethoden die man auch unter Xpand als Extensions aufrufen kann. Die Regeln beschreiben wie aus dem Quellmodell das Zielmodell erzeugt werden soll. Als Werkzeug für ATL wird eine Eclipse-basiert Umgebung angeboten die dem Benutzer komfortable Editoren und einen Compiler bietet. Auf dem Schaubild wird die Abhängigkeit zu den verschiedenen Metamodellen deutlich gemacht, dabei haben alle das gleiche Meta-Metamodell.

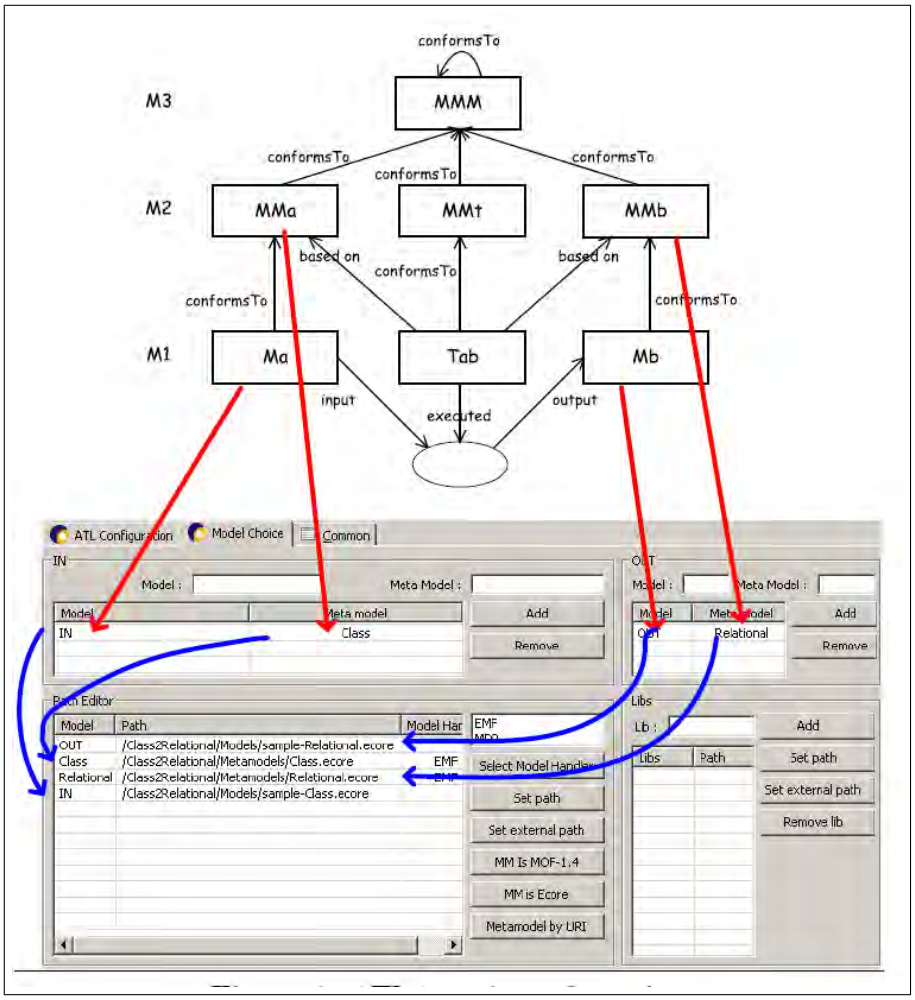

Abbildung 4.40: ATL Transformation-Konfiguration [21]

Features	ATL	XPAND	XTEND
Ansatz	deklarativ/imperativ	templateorientiert	imperativ
Typsystem	statisch (Laufzeit)	statisch	statisch
Constraints	OCL	Check (OCL ähnlich)	Check (OCL ähnlich)
Codecompletion Syntax	ja (unkomfortabel)	ja (erst mit neueren Versionen)	ja (erst mit neueren Versionen)
Outline	ja	nein	nein
Fehlermarkierung Syntax	ja	Bedingt, häufig Fehler obwohl keine vorhanden	ja
Erlernbarkeit	schwer	leicht	leicht
Verbreitung	wissenschaftliches Umfeld	praxisnahe Lösungen vorhanden	praxisnahe Lösungen vorhanden
Paradigma	Model-To-Model	Model-To-Text	Model-To-Model

Abbildung 4.41: Vergleich Transformationstools

Fazit

Die hier vorgestellten Transformationswerkzeuge können relativ schwer miteinander verglichen werden, da sie zum Teil unterschiedliche Ziele verfolgen. Deswegen wird versucht nach bestimmten Eigenschaften die Werkzeuge zu untersuchen, die Ergebnisse können im Schaubild 4.41 betrachtet werden. Die Anforderungen ein Graphenmodell in der *Dot*-Syntax zu erzeugen, erfüllen alle vorgestellten Sprachen. Betrachtet man den Faktor Anwendbarkeit, so fiel es mir schwer, mich in die ATL-Entwicklungsumgebung einzuarbeiten. Zudem benötigt das Werkzeug insgesamt drei Metamodell die immer vorliegen müssen. Xtend und Xpand sind zwei Werkzeuge die sehr nah aneinander geknüpft sind, der Hauptunterschied liegt darin, ob man eine Model-To-Text- oder ein Model-To-Model-Transformationslösung anstrebt. Durch das harmonische Zusammenspiel zwischen den beiden Sprachen, empfiehlt es sich beide Werkzeuge einzusetzen. Da das Dot-Modell für die Graphenvisualisierung in einer textueller Form vorliegen muss, ist die Entscheidung einen Model-To-Text-Transformationsansatz zu wählen naheliegend. Es verringert auch den Aufwand, man kein Ziel-Metamodell mit den dazugehörigen Transformationensregeln definiert werden muss. Es werden lediglich die im Kapitel 4.1 gefilterten Informationen mithilfe von Templates auf einen Graphen abgebildet. Um Informationen leichter aus den Quellmodellen aufbereiten zu können, kann auf die Sprache Xtend zurückgegriffen werden. Wie die Implementierung unter Verwendung dieser Sprachen abläuft, wird im nächsten Kapitel behandelt.

5 Implementierter Lösungsansatz

In den vorigen Kapiteln wurden die Grundlagen und Anforderungen für das zu entwickelnde Visualisierungsystem ermittelt. Anschließend wurde ein Lösungskonzept vorgestellt, das sowohl auf Technologien aus dem Bereich der MDSD, als auch aus der Graphenvisualisierung aufsetzt. In diesem Kapitel wird die Umsetzbarkeit dieser Konzepte anhand einer Implementierung demonstriert. Schwerpunkt war dabei die Implementierung des Visualisierungs-Systems. Dabei wird unter Berücksichtigung der MDSD-Systemarchitektur von Wieland WFS, eine integrierbare Lösung im strukturellen Aufbau beschrieben. Dazu wird im ersten Unterkapitel ein Gesamt-Systementwurf vorgestellt, der in den folgenden Unterkapiteln in einzelnen Teilen ausführlich erläutert wird. Abschließend werden im Anhang die wichtigsten Teile des Programmcodes zusammengefasst.

5.1 Systemübersicht

Die für die verschiednen Systemartefakte vorgestellten Lösungsansätze, werden im Folgenden am Beispiel der Visualisierung der DataService-Artefakte realisiert. Den grundlegenden Programmablauf mit den verschiedenen Phasen, verdeutlich die Abbildung 5.1.

5.1.1 Anforderungen an das System

Die Verarbeitung der Informationsdaten wird in den MDSD-Software-Lifecycle integriert. Für die Realisierung der Anwendung ergeben sich aus der Analysephase folgende Anforderungen die umgesetzt werden müssen:

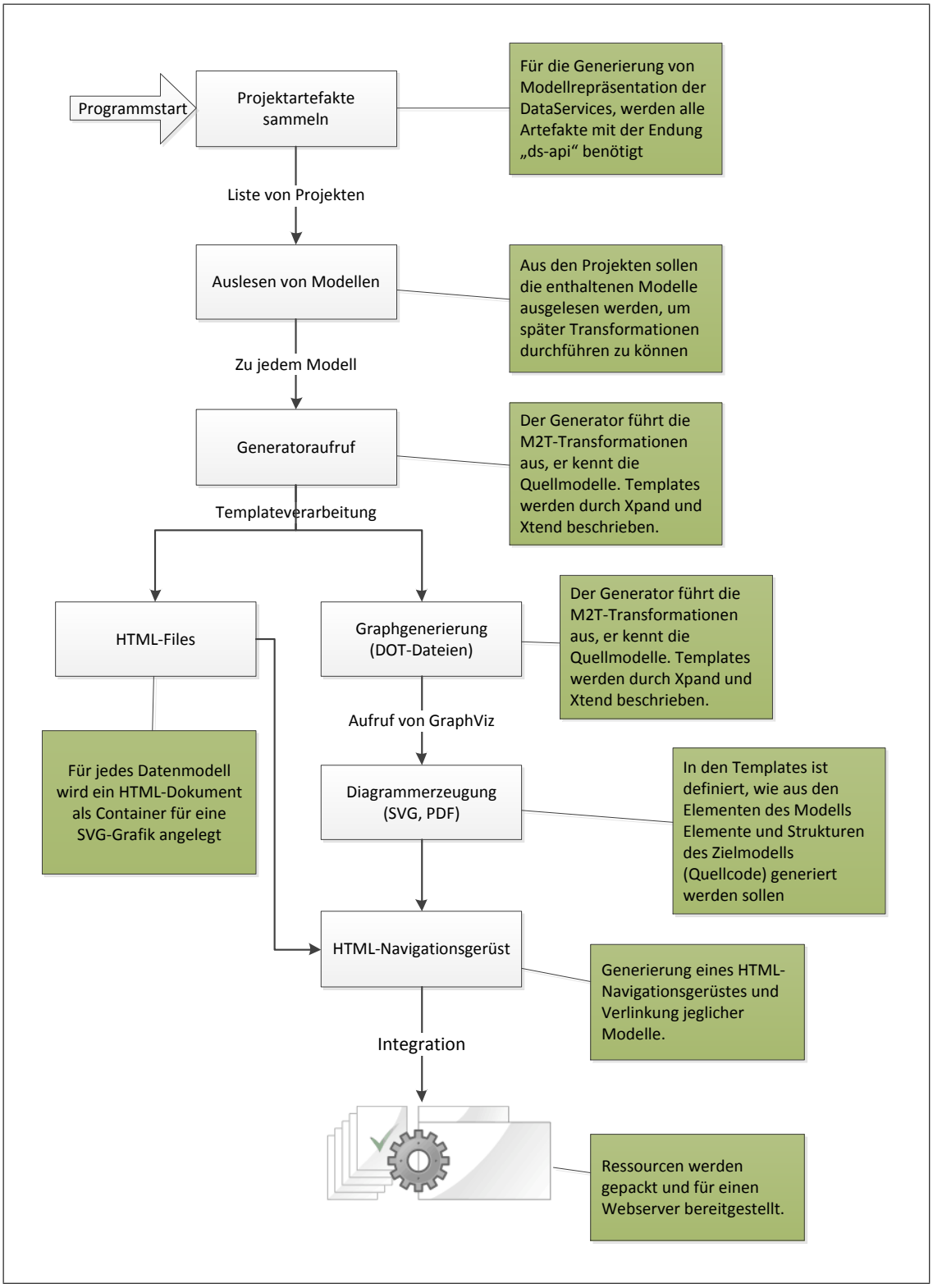

Abbildung 5.1: Systemablauf der Visualisierungsanwendung

Datenbeschaffung Die zu visualisierenden DataService-Artefakte werden in einem SVN-Repository verwaltet. Der erste Schritt ist, die Modelle automatisiert auschecken zu können und lokal für die Anwendung verfügbar zu machen. Anschließend werden nur die relevanten Artefakte ausgewählt, welche die benötigten Modelle beinhalten. Die Umsetzung erfolgt dabei über ein Eclipse-Maven-Plugin, das eine Schnittstelle zum Build-Tool Maven bietet und eine Itegration in das WFS-Buildsystem ermöglicht.

Generierung einer Benutzeroberfläche Die Navigation zwischen den grafischen Modellen wird in Form einer HTML-Seite realisiert. Dazu wird ein HTML-Container benötigt, der die aktuellen Modelle verwaltet kann. Wie im Lösungskonzept vorgestellt, werden die Artefakte nach Solutionzugehörigkeit angeordnet. Das HTML-Grundgerüst mit den einzelnen HTML-Files wird hierbei zusammen mit den Modellen generiert.

Generierung von grafischen Modellrepräsentationen Der erste Schritt in der Generierung besteht darin, einen Generator zu entwickeln, welcher die Quellmodelle einlesen kann und über Aufruf von Templates ein dot-Graphenmodell erzeugt. Anschließend werden die Dot-Dateien lokal abgelegt. Dieser Vorgang erfolgt über alle Modelle die relevant sind. Der zweite Schritt besteht darin, die Dot-Modelle über einen *Systemcall* an GraphViz zu übergeben um daraus das Grafikformat SVG für die Anzeige im HTML-Browser zu generieren. Zusätzlich werden die Modelle für die Druckausgabe im PDF-Format abgelegt.

Automatische Aktualisierung Weil sich die textuellen Modelle ständig verändern oder neu hinzugefügt werden, benötigt man einen automatisierten Mechanismus, der täglich die Modelle neu generiert. Dazu wird das *Continuous Integration*-Tool *Hudson* eingesetzt, welches den Generatorprozess zeitgesteuert starten kann.

5.1.2 Verwendete Technologien

Der Prozess zur Visualisierung textueller Domänenmodelle ist in verschiedene Teilbereiche unterteilt, die auf unterschiedliche Technologien aufbauen.
Für die Realisierung der Modelltransformationen aus textuellen Modellen in grafische Modelldiagramme, wird das *openArchitectureWare*-Framework mit den enthalteten Sprachen Xtend, Xpand und Xtext eingesetzt. Für die Generierung der HTML-Oberfläche wird zum einen Xpand verwendet, wo man auf Modellinformationen angewiesen ist, für Grundkonstrukte wie Navigation und CSS wird Java verwendet. Die eigentliche Applikation wird als ein Maven-Eclipse-Plugin

realisiert und wird über einen Kommandozeilenaufruf auf eine bestimmte Projektstruktur aufgerufen. Dabei ist es möglich, auch einzelne Artefakte zu generieren, um gegebenfalls aktuelle Modelldaten erzeugen zu können. Für die tägliche Aktualisierung der grafischen Modelle wird ein Continuous Integration Tool namens *HUDSON* eingesetzt. Dabei kann es konfiguriert werden, zu welchen Zeiten der Generierungsprozess auszuführen ist. Eine grafische Übersicht über die verwendeten Technologien wird im Schaubild 5.2 gezeigt.

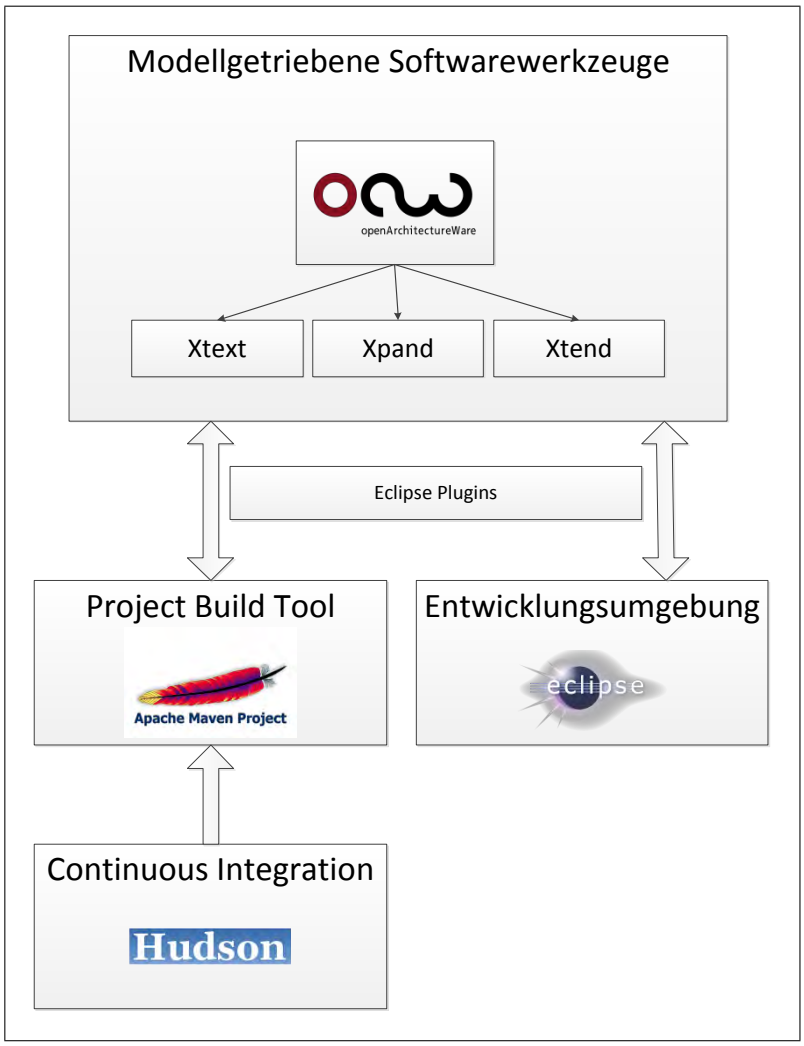

Abbildung 5.2: Verwendeten Technologien für Generierung grafischer Modellrepräsentationen

5.2 Systemstart

Das Hauptprogramm ist über einen Maven-Plugin realisiert (Quellcode ist im Anhang unter A.1 hinterlegt). Dabei wird das bestehende WFS-Plugin *wfsbuild* um eine Klasse *MakeGraphica-ModelMojo* erweitert. Der Aufruf erfolgt wie folgt über die Konsole:

```
1 mvn_call wfs-build:make-graph pom.xml
```

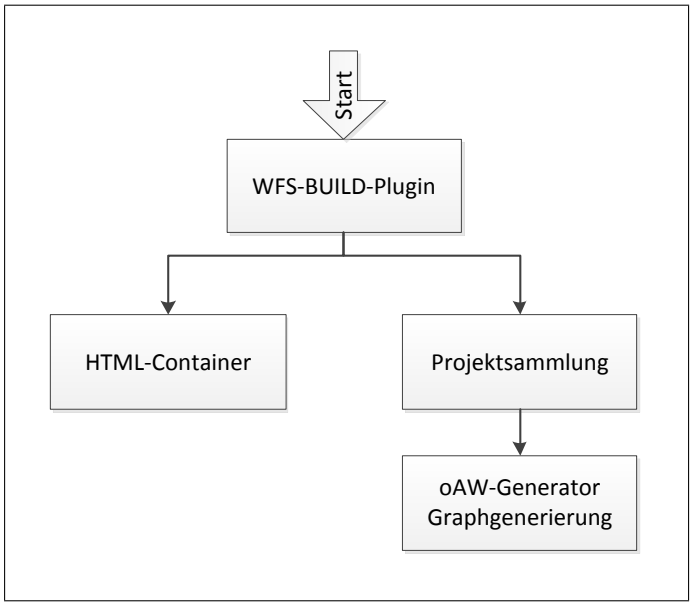

Abbildung 5.3: Aufgaben des WFS-BUILD-Plugins

Dabei wird die oberste Projektdatei in Form einer xml-Beschreibung als pom.xml übergeben. Hier werden alle Artefakte als Module beschrieben die wiederum eigene Projektdateien enthalten.
Die Aufgabe dieses Plugins besteht daraus, die notwendigen Artefakte zu sammeln und dann jeweils den GraphGenerator aufzurufen. Anschließen werden zusätzlich die HTML-Container erzeugt. Für die Filterung der Projekte ist die Methode *determineProjectsToConsider()* zuständig. Der eigentliche oAW-Generator wird über die Methode *execute()* aufgerufen. Die Abbildung 5.3 veranschaulicht die Aufgaben dieses Plugins. Für detaillierte Informationen, kann der Quelltext A.1 betrachtet werden.

5.3 OpenArchitectureWare Generator-Implementierung

Der Generator wird während eines Mavenbuild-Prozesses in der Phase *generete-sources* über ein Plugin aufgerufen. Das oAW-Generatorplugin ruft wiederum das DataModelGraphGenerator-Plugin mit dem Goal *generate-dm-graph* auf. Ab hier wird einerseits das MainTemplate des Generators aufgerufen, das die Erzeugung von DOT-Modellen durchführt, andererseits wird anschließend über einen SystemCall der Graphenvisualisierungs-Werkzeug GraphViz mit den entsprechenden Parametern aufgerufen, siehe Abbildung 5.4. Der Quellcode dazu befindet sich im Anhang A.2.

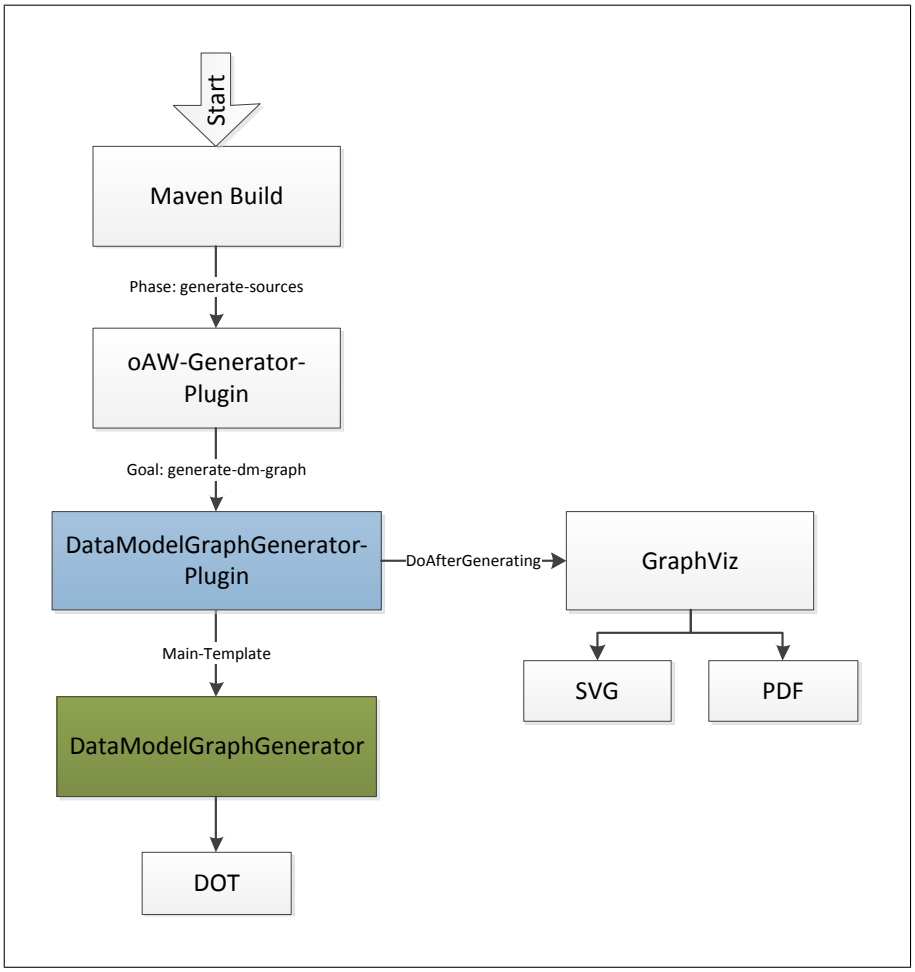

Abbildung 5.4: Generatoraufbau im oAW-Umfeld

5.4 Templategenerierung von DOT-Modellen

Zur Generierung von DOT-Modellen wird die templatebasierte Sprache Xpand verwendet. Zusätzlich wird diese durch Extensions aus der Xtend-Sprache unterstützt. Die zwei Hauptaufgaben der Templates sind die Generierung von DOT-Dateien aus den Quellmodellen und die Erzeugung von HTML-Container für die SVG- und PDF-Dateien. Den Haupteinstiegspunkt stellt das Template *Main*. Hier werden wiederum die andere zwei Templates *Dot::dotModel* und *htmlContainer* aufgerufen.

```
1  «DEFINE main FOR DataService»
2     «EXPAND Dot::dotModel(this) FOREACH dataModels.datamodel-»
3     «EXPAND htmlContainer FOREACH dataModels.datamodel-»
4  «ENDDEFINE»
```

Ein kleiner Ausschnitt aus dem dotModel-Template wird im folgenden Quelltext gezeigt. Hier werden alle Datenmodelle eines DataServices durchlaufen und die entsprechenden Graphmodelle erzeugt und eine Datei geschrieben.

```
1  «DEFINE dotModel(DataService service) FOR DataModel»
2     «LET getSolution().name+"/"+name+".dot" AS dotPath-»
3     «FILE dotPath GRAPH_DOT-»
4     «REM»«setDotFileRelativePath(dotPath)-»«ENDREM»
5     «addDotFileRelativePath(dotPath)-»
6     «EXPAND beforeDotModel FOR this-»
7     «EXPAND dotNodes(service) FOREACH dataobjects-»
8     «EXPAND dotEdges FOR this-»
9     «EXPAND afterDotModel FOR this-»
10    «ENDFILE-»
11    «ENDLET-»
12 «ENDDEFINE»
```

Die ausführliche Implementierung der Templates und den dazugehörigen Extensions kann im Quelltext-Anhang A.3, A.4, A.5 eingesehen werden.

5.5 Ergebnisse

In der Phase der Implementierung entstand ein modellgetriebenes Werkzeug, das in der Lage ist, aus einem SVN-Repository die Verzeichnisstruktur zu durchlaufen und die Artefakte vom Typ DataService einzusammeln. Anschleßend erzeugt es mithilfe eines templatebasierten

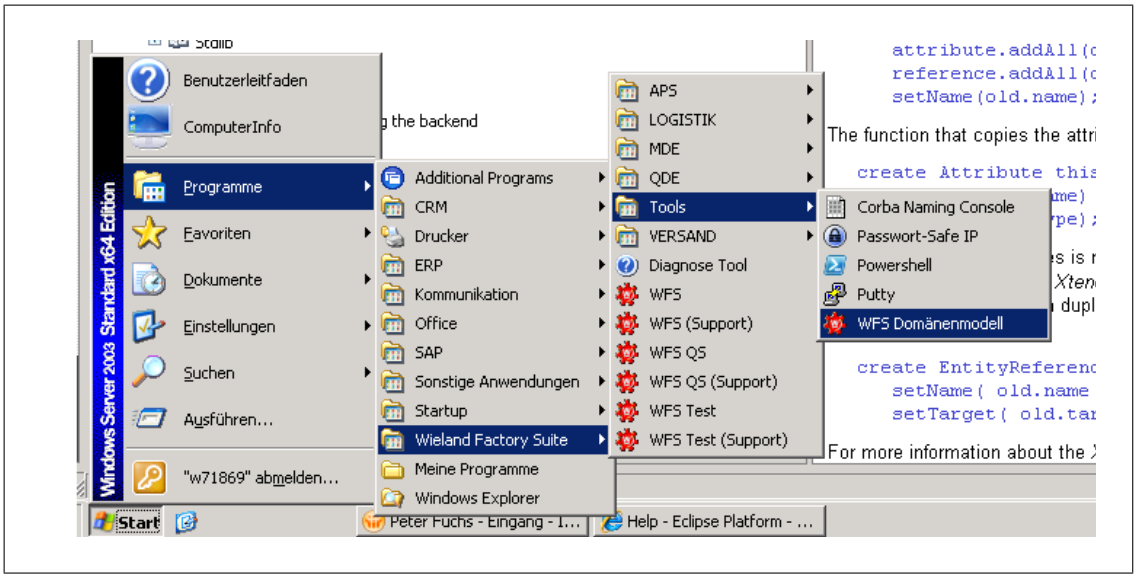

Abbildung 5.5: Integration in die Wieland Factory Suite im Startmenü

Generators DOT-Textmodelle. Diese werden an die Schnittstelle von GraphViz übergeben und daraus Grafiken im SVG-Format oder Dukumente im PDF-Format erzeugt. Als Navigationoberfläche wird ein HTML-Container verwendet, der sich an die Modelle angepasst. Ebenso wurde die Unterstützung von MouserOver-Effekten implementiert. Das System ist für die Verwendung unter dem Browser Firefox optimiert, da die vorhande Version von Internet Explorer die Anzeige von SVG-Dokumenten nur geringfügig abdeckt. Über die Gesten *Ctrl+MouseUp-Down* wird das Zoomen von SVG-Dokumenten im Browser möglich gemacht.
Die Ergebnisse werden im Folgenden in Form von Screenshot präsentiert:

```
1  digraph "Abzweig" {
2  rankdir = "LR"
3  overlap = "false"
4  node [shape=none, margin=0, color="#333333"]
5  node [fontname=Univers, fontcolor="#333333", fontsize=10]
6  edge [color="#333333" arrowsize="0.6" fontname=Univers, fontcolor="#333333",
       fontsize=10]
7
8  Abzweig [label=<
9  <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
10   <TR>
11     <TD COLSPAN="2" border="1" ALIGN="CENTER" height="32" BGCOLOR="#ccffcc"><FONT
         POINT-SIZE="12" FACE="Univers">Abzweig</FONT></TD>
```

```
12      </TR>
13
14    <TR>
15      <TD href="#" TOOLTIP="SCHNEIDEINTEILUNG.ANZAHLPE" ALIGN="LEFT">anzahlPE</TD><TD
            ALIGN="RIGHT"><FONT COLOR="#888888">long</FONT></TD>
16    </TR>
17    <TR>
18      <TD href="#" TOOLTIP="SCHNEIDEINTEILUNG.FOLGEAUFTRAG" ALIGN="LEFT">folgeAuftrag</
            TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">string</FONT></TD>
19    </TR>
20    <TR>
21      <TD href="#" TOOLTIP="SCHNEIDEINTEILUNG.ART" ALIGN="LEFT">art</TD><TD ALIGN="RIGHT
            "><FONT COLOR="#888888">string</FONT></TD>
22    </TR>
23  </TABLE>>];
24
25    Arbeitsgang [label=<
26    <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
27      <TR>
28        <TD href="../BDE/Arbeitsgang.html" TITLE="Arbeitsgang" TARGET="_top" COLSPAN="2"
              border="1" ALIGN="CENTER" height="32" BGCOLOR="#ccffcc"><FONT POINT-SIZE="12
              " FACE="Univers">Arbeitsgang</FONT></TD>
29      </TR>
30    </TABLE>>];
31
32    subgraph cluster0 {
33    fontname="Univers";
34    fontcolor="#cccccc";
35    label="Solution: BDE - Datenmodell: Abzweig";
36    color="#333333";
37
38    Abzweig
39    }
40    Abzweig -> Arbeitsgang [label="arbeitsgang"];
41    }
```

Listing 5.1: Ein einfaches DOT-Graphenmodell in Textform

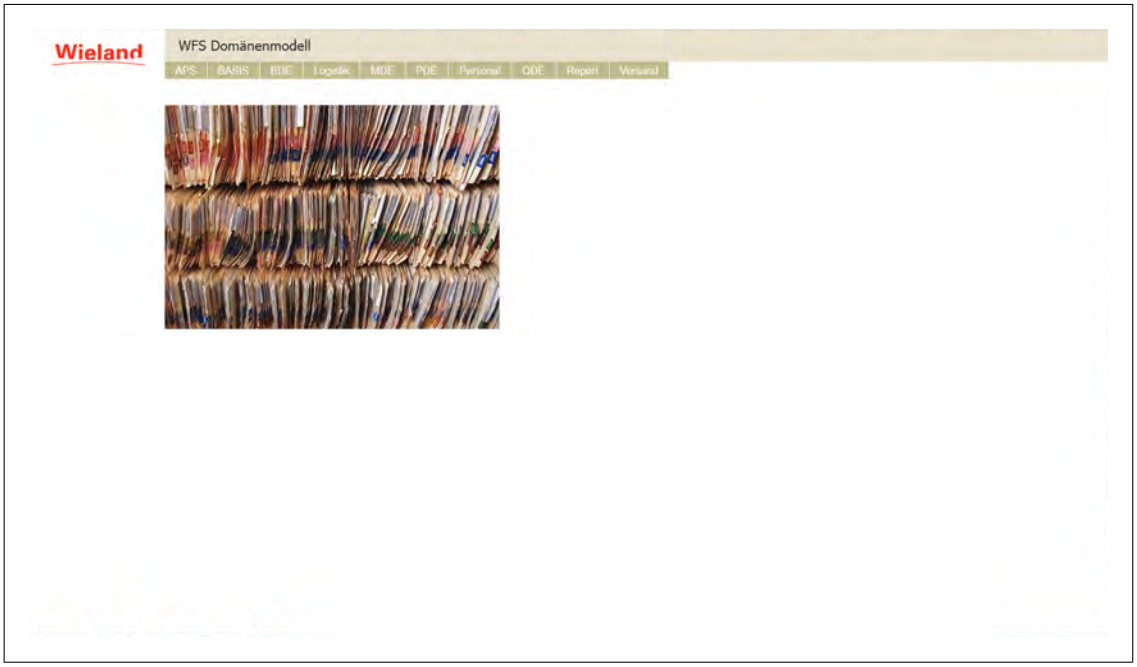

Abbildung 5.6: Startseite der grafischen Modellrepräsentation, Navigation über Solutions

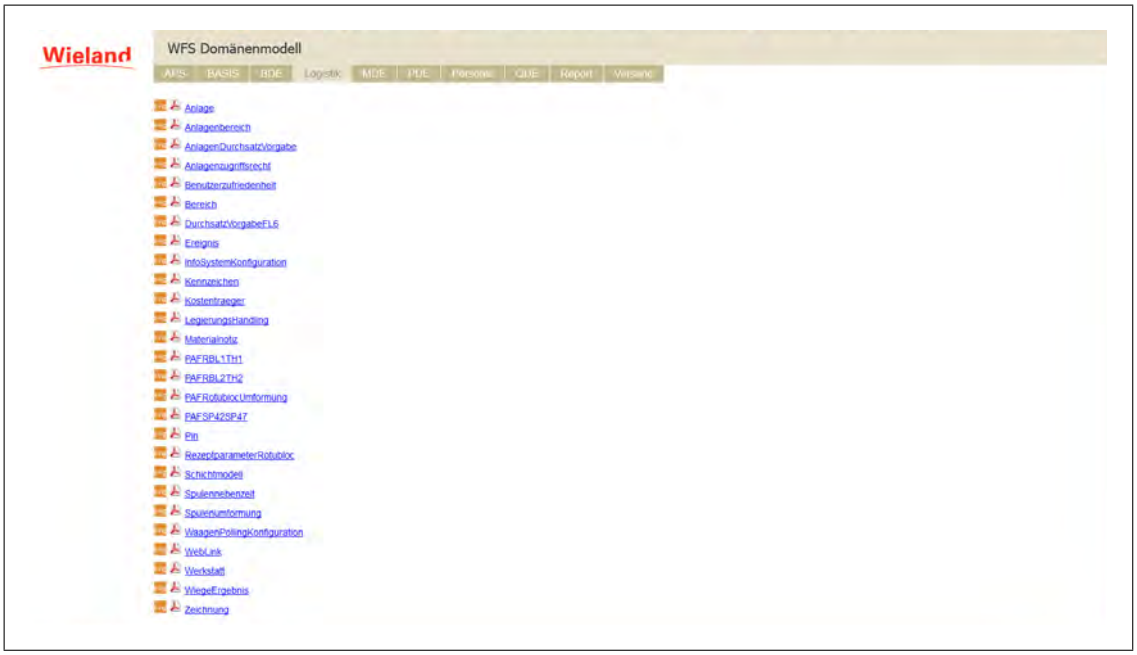

Abbildung 5.7: Auflistung von grafischen Modellen aus der Solution Logistik, Hyperlinkverknüpfungen zu den SVG- und PDF-Dokumenten

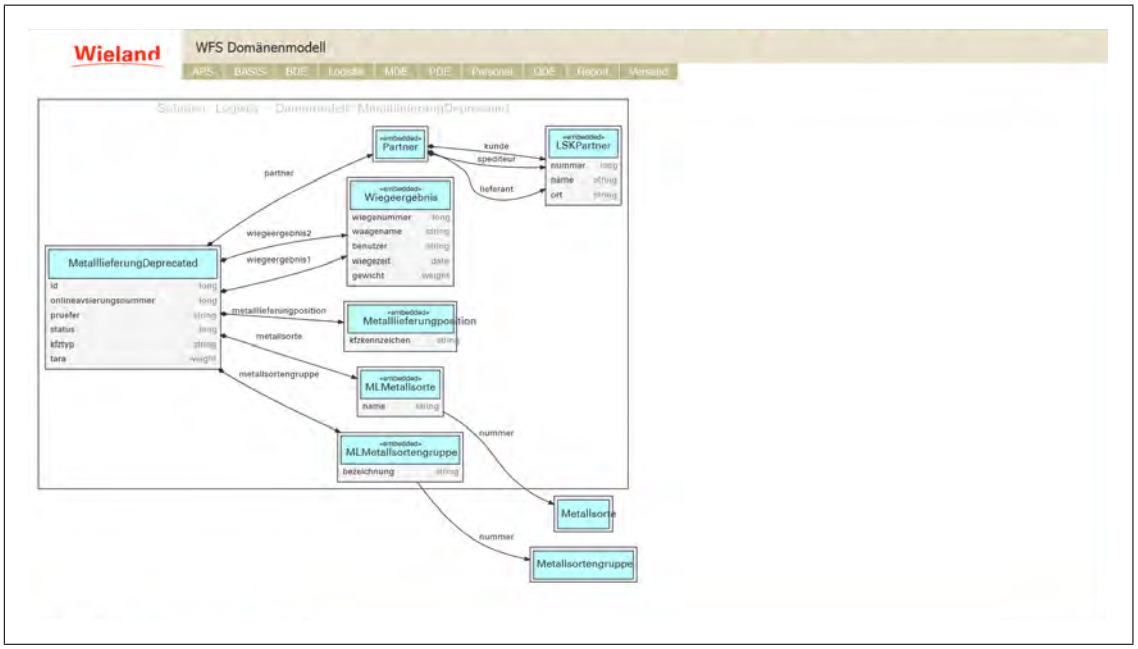

Abbildung 5.8: Ein grafisches Datenmodell

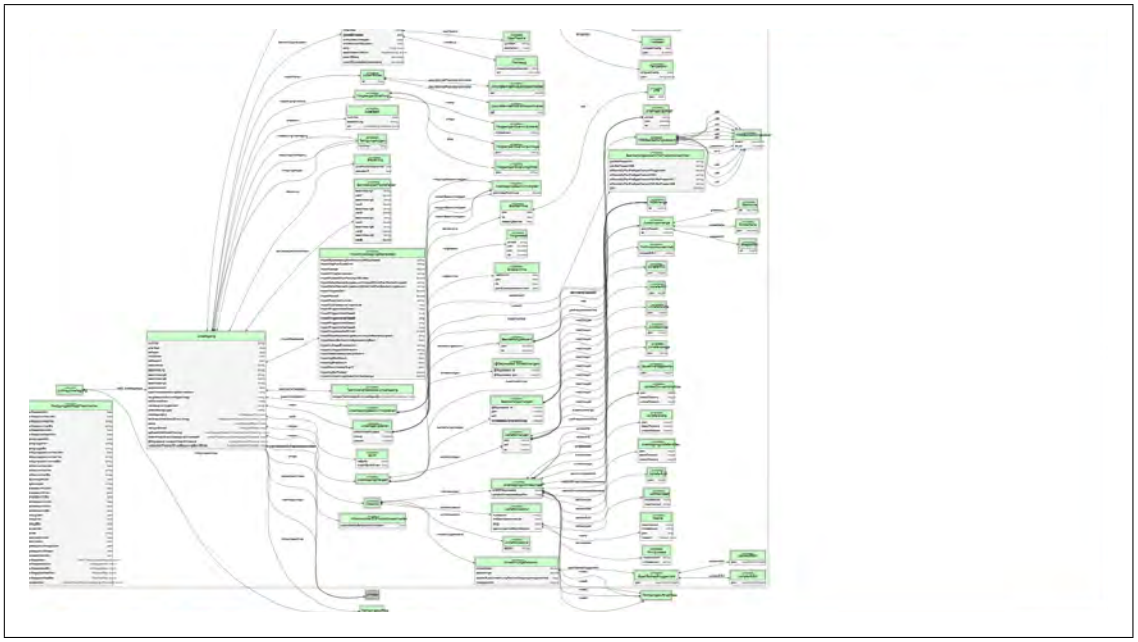

Abbildung 5.9: Ein größeres Datenmodell, herausgezoomt

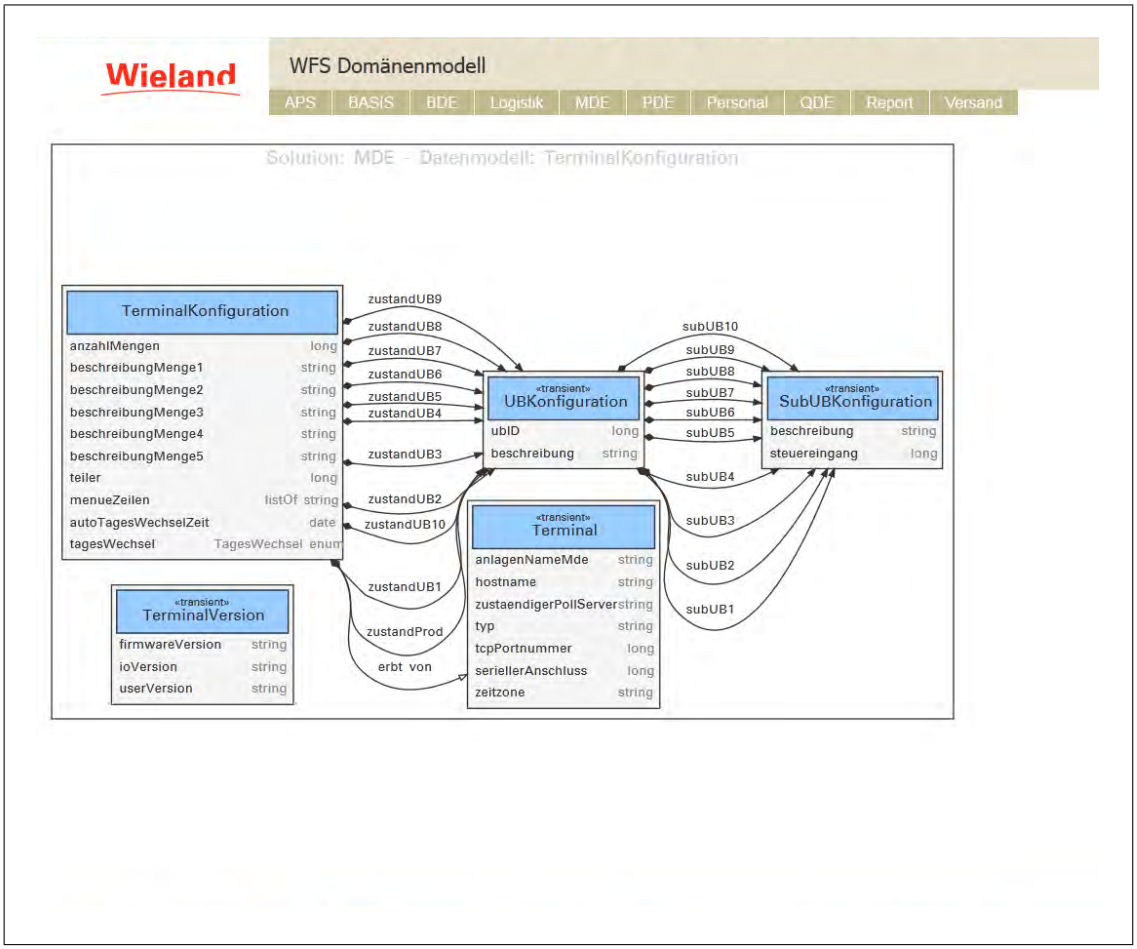

Abbildung 5.10: Farkodierung der Soulutions in den Knoten

6 Zusammenfassung, Evaluation und Ausblick

Dieses Kapitel beschreibt die Schlussphase in der Entwicklung des Visualisierungssystems. In dieser Phase wird das System aufgrund der in Kapitel 4.2 vorgestellten Eigenschaften evaluiert. Außerdem werden die Benutzer in die entwickelte Anwendung eingewiesen und können Änderungs- und Erweiterungsvorschläge melden. Zum Schluß wird die Integration in das modellgetriebene Softwaresystem WFS betrachtet und ein Ausblick auf weiterführende Entwicklungen gegeben.

6.1 Zusammenfassung

Das Ziel dieser Arbeit lag in einer wissenschaftlichen Analyse der Problemstellung, die sich aus einem modellgetriebenen Software-Entwicklungsumfeld ergab und der Entwicklung eines Lösungsansatzes. Anhand eines Software-Artefaktes wurde eine praktische Umsetzung der Lösungsidee durchgeführt. Die Hauptanforderung dieser Arbeit war die Umsetzung einer grafischen Repräsentation von textuellen Softwaremodellen. Damit soll das Verständnis von Software aus modellgetriebener Sicht verstärkt werden und die Kommunikationsprobleme zwischen den beteiligten Projektmitarbeitern beseitigt werden.
Gesamtbetrachtet lässt sich feststellen, dass die grafische Modellrepräsentation aufgrund von deutlicher Reduktion der Informationsdichte, die fachliche Struktur besser hervohebt. Zudem lassen sich die Zusammenhänge zwischen verschiedenen Informationsobjekten durch eine grafische Modellierurung einfacher ableiten. Bei der Implementierung wurde ein hoher Wert auf Softwareergonomie gelegt, wodurch die Grundsätze Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit und Erwartungskonformität bei der Umsetzung als Qualitätsmaßstab dienten. Bei dem Konzept der Navigationsoberfläche wurde viel Wert auf kurze und eindeutige Navigationswege gelegt, um das Kriterium Aufgabenangemessenheit zu erfüllen. Bei der Wahl der Farben wurden Richtlinien aus den Wieland-Styleguides umgesetzt um dem Benutzer eine vertraute Umgebung zu bieten. Die im Kapitel 4.2 definierten Visualisierungsaufgaben konnten im Konzept berücksichtigt werden, bei der Implementierung bereitete die Aufgabe *Details auf Abruf* Schwierigkeiten. Um unnötige Information verbergen zu können, wäre eine dynamische

Graphenanzeige mit einem Aufklappmechanismus von Vorteil gewesen. Leider konnte diese Anforderung aufgrund der Wahl des Renderingwerkzeuges GraphViz nicht umgesetzt werden. Bei der Darstellung von Beziehungen in den DataServices ist zudem bei größerer Menge an Datenobjekten die Beschriftung der Kanten schwer zuzuordnen. Bei Modellen von kleiner bis mittlerer Größe werden die grafischen Diagramme optimal gerendert und können problemlos auf Papier gedruckt werden.
Gesamtbetrachtet lässt sich festhalten, dass eine modellgetriebene Vorgehensweise bei der Entwicklung der Visualisierungsanwendung sehr aufwändig umzusetzen ist. Die Gründe dafür sind zum einen die komplexere Systemarchitektur, zum anderen ist durch den Einsatz verschiedener Modellierungssprachen und Frameworks der Arbeitsaufwand um einiges höher als bei der herkömmlichen Softwareentwicklung.

6.2 Evaluation

Bei einer Nutzerbefragung wurde die Visualisierungslösung positiv angenommen. Als wünschenswerte Erweiterung wurde eine Suchfunktion für Datenattribute in den Diagrammen genannt. Desweiteren wurde die etwas unübersichtliche Darstellung bei größeren Datenmengen kritisiert. Dieses Problem tritt in der Praxis allerdings nur selten auf, da die meisten Artefakte auf einer Bildschirmseite darstellbar sind. Insgesamt bewerteten die Nutzer die Visualisierungslösung als wichtigen Bestandteil für projektbeteiligte Mitarbeiter, besonders in den Analysesessions. Dabei können Mitarbeiter mit wenig IT-Fachwissen softwaretechnische Domänenmodelle ohne viel Aufwand aus den grafischen Modellen auslesen. In neuen Projekten können zudem die grafischen Modelle als Hilfsmittel in der Requiermentserstellung genutzt werden. Für die Softwareentwickler ermöglicht sich ein Weg, leichter mit ihren Projektmanagern kommunizieren zu können, da beide über das selbe Basis-Fachwissen verfügen. Weiterhin dient die strukturierte Web-Darstellung aller Domänenmodelle als wichtiges Dokumentationshilfsmittel in der WFS. Die im Kapitel 5 vorgestellte Implementierung zeigt, dass das Konzept zur grafischen Repräsentation von textuellen Modellen umsetzbar ist. Zum heutigen Zeitpunkt ist die entwickelte Lösung als Teil der MES-Entwicklungs-Software verfügbar. Die grafischen Modelle werden täglich automatisch auf den aktuellen Stand gebracht und stehen den Abteilungsmitarbeitern zur Verfügung.

6.3 Ausblick

Die für diese Arbeit entworfene Anwendung befindet sich im Einsatz und durchläuft gerade eine Erprobungsphase. In dieser sollen weitere Verbesserungsvorschläge der Mitarbeiter gesammelt werden und daraus Anforderungen definiert werden, die zu einem späteren Zeitpunkt bei Bedarf umgesetzt werden können. Es bestehen mehrere Möglichkeiten, wie die Anwendung zur grafischen Repräsenation von Modellen erweitert werden kann. Eine Weiterentwicklung wäre die vollständige Implementierung der restlichen bislang unberücksichtigten Softwareartefakte aus der MDSD-Systemarchitektur der WFS. Zusätzlich wäre die Unterstützung von Änderungen der grafischen Modelle ein weiterer Schritt um die Stärken der MDSD auszunutzen. Demnach würden die grafischen Modelle über die Dokumentation hinausreichen und Teil der Implementierung werden.
Eine weitere Möglichkeit den Funktionsumfang der Software sinnvoll zu erweitern, stellt eine Suchfunktion dar. Sie ermöglicht bei umfangreichen Graphen mit vielen Knoten eine Verbesserung der Übersicht. Dadurch muss der Nutzer nicht lange nach einem bestimmten Datenelement suchen. Um die Darstellung großer Datenmengen weiterhin verbessern zu können, ist angedacht eine interaktive Graphendarstellung mit unterschiedlichen Layoutformen zu untersuchen. Schlußendlich muss die hier entwickelte Visualisierungssoftware parallel zum eigentlichen MDSD-Framework weitergepflegt werden, da diese die aktuell verwendete Architektur wiederspiegelt.

A Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
1  /**
2   *
3   */
4  package com.wieland.mavenplugins.wfsbuild;
5
6  import java.io.File;
7  import java.io.FileOutputStream;
8  import java.io.IOException;
9  import java.io.InputStream;
10 import java.text.MessageFormat;
11 import java.util.ArrayList;
12 import java.util.Arrays;
13 import java.util.List;
14
15 import org.apache.maven.plugin.MojoExecutionException;
16 import org.apache.maven.plugin.MojoFailureException;
17 import org.apache.maven.project.MavenProject;
18 import org.apache.maven.shared.invoker.Invoker;
19
20 import com.wieland.mavenplugins.wfsbuild.utils.InvokerExecutionException;
21 import com.wieland.mavenplugins.wfsbuild.utils.SimpleInvoker;
22
23 /**
24  *
25  * @goal make-graph
26  * @aggregator
27  * @phase process-sources
28  */
29 public class MakeGraphicalModelMojo extends CreateMultiPomMojo {
30
31    /**
32     * Dir the graphs are generated to.
33     *
34     * @parameter expression="${graph.dir}"
35     */
```

```
36    protected String graphDir;
37
38    /**
39     * Provided by Maven
40     *
41     * @component
42     */
43    protected Invoker invoker;
44
45    /**
46     * @see com.wieland.mavenplugins.wfsbuild.AbstractWfsBuildMojo#execute()
47     */
48    @Override
49    public void execute() throws MojoExecutionException, MojoFailureException {
50       if (multiPomName == null) {
51          multiPomName = "graph";
52       }
53
54       super.execute();
55
56       generateGraphs();
57    }
58
59    /**
60     * @throws MojoExecutionException
61     *
62     */
63    private void generateGraphs() throws MojoExecutionException {
64       if (graphDir == null) {
65          graphDir = currentProject.getBasedir().getAbsolutePath() + "/target/graph";
66       }
67
68       String[] goals = new String[] { "oaw-generator:generate-dm-graph", "-DgraphDir="
             + graphDir };
69
70       try {
71          SimpleInvoker.getSimpleInvoker().runMaven(getJEEMultiPomFile(), Arrays.asList(
                goals), invoker,
72                                                createInvokeSettings(), getLog());
73       } catch (InvokerExecutionException e) {
74          throw new MojoExecutionException(e.getMessage(), e);
75       }
76
77       createMainHtmlContainer();
78       createSolutionHtmlContainer();
```

```
79
80      }
81
82      private void createMainHtmlContainer() {
83         try {
84            String outputPath = graphDir + "/output";
85            FileOutputStream fis = new FileOutputStream(new File(outputPath + "/index.html
                  "));
86
87            InputStream stream = MakeGraphicalModelMojo.class.getResourceAsStream("/graph/
                  GraphIndexTemplateMain.html");
88
89            StringBuilder template = new StringBuilder();
90            int read;
91            while ((read = stream.read()) != -1) {
92               template.append((char) read);
93            }
94
95            fis.write(template.toString().getBytes());
96            fis.flush();
97            fis.close();
98         } catch (IOException e) {
99            e.printStackTrace();
100
101        }
102     }
103
104     private void createSolutionHtmlContainer() throws MojoExecutionException {
105        String outputPath = graphDir + "/output";
106
107        ArrayList<String> solutions = listDir(new File(outputPath));
108
109        for (String solution : solutions) {
110           String solutionPath = outputPath + "/" + solution;
111           ArrayList<String> datamodelNamesForSolution = listDatamodelNamesForSolution(
                  new File(solutionPath));
112
113           ArrayList<String> links = new ArrayList<String>();
114           for (String string : datamodelNamesForSolution) {
115              links.add(getLink(string));
116           }
117
118           try {
119              writeIndexFile(links, solutionPath);
120           } catch (IOException e) {
```

```
121             throw new MojoExecutionException(e.getMessage(), e);
122          }
123       }
124    }
125
126    private ArrayList<String> listDatamodelNamesForSolution(final File dir) {
127
128       File[] files = dir.listFiles();
129       ArrayList<String> datamodelNames = new ArrayList<String>();
130       if (files != null) {
131          for (int i = 0; i < files.length; i++) {
132             //System.out.print(files[i].getAbsolutePath());
133             if (files[i].isFile() && files[i].getName().endsWith(".html")
134                   && !files[i].getName().equals("index.html")) {
135                datamodelNames.add(files[i].getName().substring(0, files[i].getName().
                       lastIndexOf(".")));
136             }
137          }
138       }
139
140       return datamodelNames;
141    }
142
143    private ArrayList<String> listDir(final File dir) {
144
145       File[] files = dir.listFiles();
146       ArrayList<String> solutions = new ArrayList<String>();
147       if (files != null) {
148          for (int i = 0; i < files.length; i++) {
149             if (files[i].isDirectory() && !files[i].getName().equals("pics")) {
150                solutions.add(files[i].getName());
151             }
152          }
153       }
154       return solutions;
155    }
156
157    private String getLink(final String name) {
158
159       return "<li><a href=\"" + name + ".html\"><img src=\"../pics/svg.bmp\"></a> 
           ;<a href=\"" + name
160          + ".pdf\"><img src=\"../pics/pdf.bmp\"></a> <a href=\"" + name + ".
               html\">" + name + "</a></li>";
161    }
162
```

```
163     private String generateTemplate(final List<String> links) throws IOException {
164
165        StringBuilder stringBuilder = new StringBuilder();
166        for (String link : links) {
167           stringBuilder.append(link);
168        }
169
170        return MessageFormat.format(getTemplate(), stringBuilder.toString());
171     }
172
173     static String getTemplate() throws IOException {
174        InputStream stream = MakeGraphicalModelMojo.class.getResourceAsStream("/graph/
              GraphIndexTemplate.html");
175
176        StringBuilder template = new StringBuilder();
177        int read;
178        while ((read = stream.read()) != -1) {
179           template.append((char) read);
180        }
181
182        return template.toString();
183
184     }
185
186     private void writeIndexFile(final List<String> links, final String path) throws
           IOException {
187
188        FileOutputStream fis = new FileOutputStream(new File(path + "/index.html"));
189
190        fis.write(generateTemplate(links).getBytes());
191        fis.flush();
192        fis.close();
193     }
194
195     /**
196      * @see com.wieland.mavenplugins.wfsbuild.CreateMultiPomMojo#determineArtifactList()
197      */
198     @Override
199     protected void determineArtifactList() throws MojoFailureException {
200        // Nichts zu tun, wir bestimmen nachher mit determineProjectsToConsider ohnehin
201        // alle ds-apis.
202     }
203
204     /**
```

```
205      * @see com.wieland.mavenplugins.wfsbuild.AbstractWfsBuildMojo#
             determineProjectsToConsider()
206      */
207     @Override
208     protected void determineProjectsToConsider() throws MojoExecutionException {
209        projectsToConsider = new ArrayList<MavenProject>(collectedProjects.size());
210        for (MavenProject project : collectedProjects) {
211           if (project.getParent() != null) {
212              // Wir erkennen interessante Artefakte anhand des Parents (= ds-api)
213              if ("ds-api".equals(project.getParent().getArtifactId())) {
214                 projectsToConsider.add(project);
215              }
216           }
217        }
218     }
219
220  }
```

Listing A.1: Haupteinstiegspunkt ist das wfsbuild-plugin

```
1  package com.wieland.mavenplugins.oawgenerator;
2
3  import java.io.File;
4  import java.io.InputStream;
5  import java.io.InputStreamReader;
6  import java.util.Set;
7
8  import org.apache.maven.plugin.MojoExecutionException;
9  import org.apache.maven.plugin.MojoFailureException;
10 import org.openarchitectureware.workflow.WorkflowContextDefaultImpl;
11 import org.openarchitectureware.workflow.issues.Issues;
12 import org.openarchitectureware.workflow.monitor.NullProgressMonitor;
13 import org.openarchitectureware.xpand2.Generator;
14
15 import com.wieland.wfs.mdsd.datamodel.graph.extensions.GraphHelper;
16
17 /**
18  * @author w68908
19  *
20  * @goal generate-dm-graph
21  * @phase generate-sources
22  */
23 public class G extends ServiceApiGeneratorMojo {
24 //public class DataModelGraphGeneratorMojo extends AbstractDataModelGeneratorMojo {
25    /**
```

```
26      * GRAPH_GEN: Das src-gen Verzeichnis, in das alle immer generierten Source-Dateien
            generiert werden sollen.
27      *
28      * @parameter expression="${graphDir}" default-value="target/generated-graph/wfs-
            mdsd" *
29      * @optional
30      */
31     protected File graphGenDir;
32
33     @Override
34     protected String getGeneratorPath() {
35        return "com::wieland::wfs::mdsd::datamodel::graph::Main::main";
36     }
37
38     /**
39      * @see com.wieland.mavenplugins.oawgenerator.OawGeneratorMojo#addOutlets(org.
            openarchitectureware.xpand2.Generator)
40      */
41     @Override
42     protected void addOutlets(final Generator generator) {
43        super.addOutlets(generator);
44
45        //generator.setPrSrcPaths(graphGenDir.getAbsolutePath() + "/output");
46
47        addOutlet(generator, "GRAPH_DOT", graphGenDir.getAbsolutePath() + "/dot", true);
48        addOutlet(generator, "GRAPH_OUTPUT", graphGenDir.getAbsolutePath() + "/output",
              true);
49
50     }
51
52     /**
53      * @see com.wieland.mavenplugins.oawgenerator.OawGeneratorMojo#doBeforeGenerating(
            org.openarchitectureware.workflow.WorkflowContextDefaultImpl,
54      * org.openarchitectureware.workflow.issues.Issues,
55      * org.openarchitectureware.workflow.monitor.NullProgressMonitor)
56      */
57     @Override
58     protected void doBeforeGenerating(final WorkflowContextDefaultImpl ctx, final Issues
            issues,
59           final NullProgressMonitor monitor) throws MojoFailureException,
                 MojoExecutionException {
60        super.doBeforeGenerating(ctx, issues, monitor);
61
62        GraphHelper.clearDotFileRelativePath();
63     }
```

```
64
65    /**
66     * @see com.wieland.mavenplugins.oawgenerator.OawGeneratorMojo#doAfterGenerating(org
          .openarchitectureware.workflow.WorkflowContextDefaultImpl,
67     * org.openarchitectureware.workflow.issues.Issues,
68     * org.openarchitectureware.workflow.monitor.NullProgressMonitor)
69     */
70    @Override
71    protected void doAfterGenerating(final WorkflowContextDefaultImpl ctx, final Issues
          issues,
72            final NullProgressMonitor monitor) throws MojoFailureException,
                MojoExecutionException {
73
74       Set<String> dotFileRelativePaths = GraphHelper.getDotFileRelativePaths();
75       if (dotFileRelativePaths == null) {
76          return;
77       }
78
79       for (String dotFileRelativePath : dotFileRelativePaths) {
80          dotFileRelativePath = dotFileRelativePath.replaceAll("\\/", "\\\\");
81          String dotFilePath = graphGenDir.getAbsolutePath() + "\\dot\\" +
                dotFileRelativePath;
82
83          String svgOutputFile = graphGenDir.getAbsolutePath() + "\\output\\"
84                + dotFileRelativePath.replaceAll("\\.dot", "\\.svg");
85          String pdfOutputFile = graphGenDir.getAbsolutePath() + "\\output\\"
86                + dotFileRelativePath.replaceAll("\\.dot", "\\.pdf");
87
88          createDotOutput(dotFilePath, svgOutputFile, "svg");
89          createDotOutput(dotFilePath, pdfOutputFile, "pdf");
90
91          getLog().info("Generated model output for " + dotFileRelativePath);
92       }
93
94    }
95
96    /**
97     * @param dotFile
98     * @param outputFile
99     * @throws MojoExecutionException
100    */
101   private void createDotOutput(final String dotFile, final String outputFile, final
          String outputType)
102         throws MojoExecutionException {
103
```

```
104         File outputDir = new File(outputFile).getParentFile();
105         outputDir.mkdirs();
106 
107         if (dotFile != null) {
108            ProcessBuilder builder = new ProcessBuilder("dot.exe", "-T" + outputType,
                dotFile, "-o", outputFile);
109            builder.redirectErrorStream(true);
110            try {
111               Process process = builder.start();
112               int returnCode = process.waitFor();
113               if (returnCode != 0) {
114                  InputStream inputStream = process.getInputStream();
115                  InputStreamReader reader = new InputStreamReader(inputStream);
116                  char[] buffer = new char[1024];
117                  int readCount = -1;
118                  StringBuilder error = new StringBuilder();
119                  while ((readCount = reader.read(buffer)) > 0) {
120                     error.append(new String(buffer, 0, readCount));
121                  }
122 
123                  throw new MojoExecutionException("Error occured calling dot: " + error.
                       toString());
124               }
125            } catch (Exception e) {
126               throw new MojoExecutionException(e.getMessage(), e);
127            }
128         }
129      }
130 
131      /**
132       * @see com.wieland.mavenplugins.oawgenerator.OawGeneratorMojo#checkIfNecessary()
133       */
134      @Override
135      protected boolean checkIfNecessary() {
136         return true;
137      }
138 
139   }
```

Listing A.2: oAW-Generator-Plugin

```
1  «IMPORT wfsdatamodel»
2  «IMPORT wfsormapping»
3  «IMPORT wfsservicemodel»
4 
```

```
5  «EXTENSION com::wieland::wfs::mdsd::datamodel::extensions::java::datamodel»
6  «EXTENSION com::wieland::wfs::mdsd::datamodel::Extensions»
7
8  «EXTENSION com::wieland::wfs::mdsd::datamodel::graph::extensions::GraphHelper»
9
10 «EXTENSION org::openarchitectureware::util::stdlib::io»
11
12
13
14
15 «EXTENSION com::wieland::wfs::mdsd::servicemodel::Extensions»
16 «EXTENSION com::wieland::wfs::mdsd::service::extensions::java::ServiceCommonHelper»
17
18 «REM»Erzeugung von DOT-Modell und HTML Container«ENDREM»
19 «DEFINE main FOR DataService»
20    «EXPAND Dot::dotModel(this) FOREACH dataModels.datamodel-»
21    «EXPAND htmlContainer FOREACH dataModels.datamodel-»
22 «ENDDEFINE»
23
24
25 «DEFINE htmlContainer FOR DataModel»
26    «LET getSolution().name+"/"+name+".html" AS htmlPath-»
27    «FILE htmlPath GRAPH_OUTPUT»
28
29    «ENDFILE-»
30    «ENDLET-»
31 «ENDDEFINE»
```

Listing A.3: Template Main und htmlContainer

```
1  «IMPORT wfsdatamodel»
2  «IMPORT wfsservicemodel»
3  «IMPORT wfsormapping»
4
5  «EXTENSION com::wieland::wfs::mdsd::datamodel::extensions::java::datamodel»
6  «EXTENSION com::wieland::wfs::mdsd::datamodel::Extensions»
7
8  «EXTENSION com::wieland::wfs::mdsd::servicemodel::Extensions»
9  «EXTENSION com::wieland::wfs::mdsd::ormappingmodel::Extensions»
10 «EXTENSION com::wieland::wfs::mdsd::ormapping::extensions::java::ormappingJava»
11 «EXTENSION com::wieland::wfs::mdsd::ormapping::extensions::ormappingCommon»
12
13 «EXTENSION com::wieland::wfs::mdsd::datamodel::graph::extensions::GraphHelper»
14
15 «EXTENSION org::openarchitectureware::util::stdlib::io»
```

```
«REM»WFSDM Umwandlung in DOT Modell«ENDREM»
«DEFINE dotModel(DataService service) FOR DataModel»
   «LET getSolution().name+"/"+name+".dot" AS dotPath-»
   «FILE dotPath GRAPH_DOT-»
   «REM»«setDotFileRelativePath(dotPath)-»«ENDREM»
   «addDotFileRelativePath(dotPath)-»
   «EXPAND beforeDotModel FOR this-»
   «EXPAND dotNodes(service) FOREACH dataobjects-»
   «EXPAND dotEdges FOR this-»
   «EXPAND afterDotModel FOR this-»
   «ENDFILE-»
   «ENDLET-»
«ENDDEFINE»

«REM»Kopfdaten vom Graphen«ENDREM»
«DEFINE beforeDotModel FOR DataModel»
   digraph "«name»" {
   rankdir = "LR"
   overlap = "false"
   node [shape=none, margin=0, color="#333333"]
   node [fontname=Univers, fontcolor="#333333", fontsize=10]
   edge [color="#333333" arrowsize="0.6" fontname=Univers, fontcolor="#333333",
       fontsize=10]
«ENDDEFINE»

«REM»Graph schliessen«ENDREM»
«DEFINE afterDotModel FOR DataModel»
   }
«ENDDEFINE»

«REM»Kantengenerierung fuer alle dataobjects im Modell«ENDREM»
«DEFINE dotEdges FOR DataModel»
   subgraph cluster0 {
   fontname="Univers";
   fontcolor="#cccccc";
   label="Solution: «this.getSolution().name» - Datenmodell: «this.name»";
   color="#333333";
   «REM»Kanten fuer DatenObjects im selben Modell«ENDREM»
   «FOREACH dataobjects AS object-»
   «REM»alle Dataobjects in das jeweilige Datenmodell aufnehmen um Cluster bilden zu
       koennen«ENDREM»
   «object.name»
      «FOREACH object.complexmembers AS member-»
        «IF member.dataobject.dataModel().name == object.dataModel().name-»
```

```
58    «object.name» -> «member.dataobject.name» [«IF member.aggregation»arrowtail =
          diamond dir = both «ENDIF»label="«IF member.list-»listOf «ENDIF-»«member.name»"
          ];
59          «ENDIF-»
60       «ENDFOREACH-»
61    «ENDFOREACH-»
62    }
63    «REM»Kanten fuer DataObjects ausserhalb vom eigenen Modell«ENDREM»
64    «FOREACH dataobjects AS object-»
65       «FOREACH object.complexmembers AS member-»
66          «IF member.dataobject.dataModel().name != object.dataModel().name-»
67    «object.name» -> «member.dataobject.name» [«IF member.aggregation»arrowtail =
          diamond dir = both «ENDIF»label="«IF member.list-»listOf «ENDIF-»«member.name»"
          ];
68          «ENDIF-»
69       «ENDFOREACH-»
70    «ENDFOREACH-»
71    «REM»Kanten fuer vererbte DataObjects«ENDREM»
72    «FOREACH dataobjects AS object-»
73       «IF object.base != null-»
74    «object.name» -> «object.base.name» [arrowhead = empty label="erbt von"];
75       «ENDIF-»
76    «ENDFOREACH-»
77
78 «ENDDEFINE»
79
80 «REM»Erzeugung von Knoten mit Labels«ENDREM»
81 «DEFINE dotNodes(DataService service) FOR DataObject»
82 «REM»Abbildung von Embedded Objects«ENDREM»
83 «IF this.embedded-»
84    «EXPAND labelDataobject FOR this-»
85    «FOREACH getMembers().select(e|!e.isComplex()) AS member-»
86       «LET getMappingsForEmbeddedAttribute(service, this, member) AS mappings-»
87          «LET getMappingTooltip(mappings) AS tooltip -»
88            «IF member.list-»
89            <TR>
90              <TD href="#" TOOLTIP="«tooltip»" ALIGN="LEFT">«IF member.deprecated-»
                    @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="
                    #888888">listOf «IF member.isEnum()»«member.type.name» enum«ELSEIF
                    member.isState()»state«ELSE»«member.type»«ENDIF»</FONT></TD>
91            </TR>
92       «ELSE-»
93          <TR>
94            <TD href="#" TOOLTIP="«tooltip»" ALIGN="LEFT">«IF member.deprecated-»
                    @Deprecated «ENDIF-»«IF member.isState()»«statemember.name»«ELSE»«member
```

```
                .name»«ENDIF»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">«IF member.
                isEnum()»«member.type.name» enum«ELSEIF member.isState()»state«ELSE»«
                member.type»«ENDIF»</FONT></TD>
95            </TR>
96          «ENDIF-»
97            «ENDLET»
98        «ENDLET-»
99    «ENDFOREACH-»
100   </TABLE>>];
101   «EXPAND complexMembers FOR this-»
102   «ELSE-»
103      «LET service.getORMapping(this).getMappingEntity(this) AS entity-»
104     «EXPAND labelDataobject FOR this-»
105     «EXPAND simpleMembers(entity) FOR this-»
106     «EXPAND measureMembers(entity) FOR this-»
107     «EXPAND enumMembers(entity) FOR this-»
108     «EXPAND stateMember(entity) FOR this-»
109     </TABLE>>];
110     «EXPAND complexMembers FOR this-»
111       «ENDLET-»
112 «ENDIF»
113 «ENDDEFINE»
114
115 «DEFINE complexMembers FOR DataObject»
116   «IF !complexmembers.isEmpty-»
117   «FOREACH complexmembers AS member-»
118      «IF (member.dataobject.dataModel().name != this.dataModel().name)-»
119         «EXPAND labelDataobjectForForeignDatamodels FOR member.dataobject-»
120   </TABLE>>];
121      «ENDIF-»
122   «ENDFOREACH-»
123   «ENDIF-»
124 «ENDDEFINE»
125
126
127 «DEFINE labelDataobjectForForeignDatamodels FOR DataObject»
128 «LET this.dataModel().getSolution().name AS solution-»
129   «this.name» [label=<
130   <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
131     <TR>
132       <TD href="../«solution»/«this.dataModel().name».html" TITLE="«this.name»" TARGET=
            "_top" COLSPAN="2" border="1" ALIGN="CENTER" height="32" BGCOLOR="#«
            getSolutionColor(solution)»"><FONT POINT-SIZE="12" FACE="Univers">«IF this.
            deprecated-»@Deprecated «ENDIF-»«this.name»</FONT></TD>
133     </TR>
```

```
134 «ENDLET»
135 «ENDDEFINE»
136
137
138 «DEFINE labelDataobject FOR DataObject»
139 «LET this.dataModel().getSolution().name AS solution-»
140    «IF transient-»
141    «this.name» [label=<
142    <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
143      <TR>
144        <TD COLSPAN="2" border="1" ALIGN="CENTER" height="32" BGCOLOR="#«getSolutionColor
             (solution)»"><FONT POINT-SIZE="8">&laquo;transient&raquo;<BR/></FONT><FONT
             POINT-SIZE="12" FACE="Univers">«IF this.deprecated-»@Deprecated «ENDIF-»«
             this.name»</FONT></TD>
145      </TR>
146        «ELSEIF embedded-»
147    «this.name» [label=<
148    <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
149      <TR>
150        <TD COLSPAN="2" border="1" ALIGN="CENTER" height="32" BGCOLOR="#«getSolutionColor
             (solution)»"><FONT POINT-SIZE="8">&laquo;embedded&raquo;<BR/></FONT><FONT
             POINT-SIZE="12" FACE="Univers">«IF this.deprecated-»@Deprecated «ENDIF-»«
             this.name»</FONT></TD>
151      </TR>
152        «ELSEIF abstract-»
153    «this.name» [label=<
154    <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
155      <TR>
156        <TD COLSPAN="2" border="1" ALIGN="CENTER" height="32" BGCOLOR="#«getSolutionColor
             (solution)»"><FONT POINT-SIZE="8">&laquo;abstract&raquo;<BR/></FONT><FONT
             POINT-SIZE="12" FACE="Univers">«IF this.deprecated-»@Deprecated «ENDIF-»«
             this.name»</FONT></TD>
157      </TR>
158          «ELSE-»
159    «this.name» [label=<
160    <TABLE BGCOLOR="#f5f5f5" BORDER="1" CELLBORDER="0" CELLSPACING="3" CELLPADDING="2">
161      <TR>
162        <TD COLSPAN="2" border="1" ALIGN="CENTER" height="32" BGCOLOR="#«getSolutionColor
             (solution)»"><FONT POINT-SIZE="12" FACE="Univers">«IF this.deprecated-»
             @Deprecated «ENDIF-»«this.name»</FONT></TD>
163      </TR>
164    «ENDIF-»
165 «ENDLET»
166 «ENDDEFINE»
167
```

```
168 «DEFINE simpleMembers(Entity entity) FOR DataObject»
169    «FOREACH simplemembers AS member-»
170       «LET entity.getMappingForMember(member) AS mapping-»
171       «LET mapping == null ? "[manual mapped]" : "["+ mapping.table.name+"."+mapping.
              dbField + "]" AS mappingInfo-»
172       «IF member.list-»
173    <TR>
174      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF member.deprecated-»
           @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">
           listOf «member.type»</FONT></TD>
175    </TR>
176       «ELSE-»
177    <TR>
178      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF member.deprecated-»
           @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">
           «member.type»</FONT></TD>
179    </TR>
180       «ENDIF-»
181       «ENDLET-»
182       «ENDLET-»
183    «ENDFOREACH-»
184 «ENDDEFINE»
185
186 «DEFINE measureMembers(Entity entity) FOR DataObject»
187    «FOREACH measureMembers AS member-»
188      «LET entity.getMappingForMember(member) AS mapping-»
189      «LET mapping == null ? "[manual mapped]" : "["+ mapping.table.name+"."+mapping.
             dbField + "]" AS mappingInfo-»
190      «IF member.list-»
191    <TR>
192      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF member.deprecated-»
           @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">
           listOf «member.type»</FONT></TD>
193    </TR>
194       «ELSE-»
195    <TR>
196      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF member.deprecated-»
           @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">
           «member.type»</FONT></TD>
197    </TR>
198       «ENDIF-»
199       «ENDLET-»
200       «ENDLET-»
201    «ENDFOREACH-»
202 «ENDDEFINE»
```

```
203
204 «DEFINE enumMembers(Entity entity) FOR DataObject»
205    «FOREACH enummembers AS member-»
206      «LET entity.getMappingForMember(member) AS mapping-»
207      «LET mapping == null ? "[manual mapped]" : "["+ mapping.table.name+"."+mapping.
             dbField + "]" AS mappingInfo-»
208      «IF member.list-»
209    <TR>
210      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF member.deprecated-»
           @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">
           listOf «member.type.name» enum</FONT></TD>
211    </TR>
212       «ELSE-»
213    <TR>
214      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF member.deprecated-»
           @Deprecated «ENDIF-»«member.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="#888888">
           «member.type.name» enum</FONT></TD>
215    </TR>
216       «ENDIF-»
217       «ENDLET-»
218       «ENDLET-»
219    «ENDFOREACH-»
220 «ENDDEFINE»
221
222 «DEFINE stateMember(Entity entity) FOR DataObject»
223    «IF statemember != null»
224    «LET entity.getMappingForMember(statemember) AS mapping-»
225    «LET mapping == null ? "[manual mapped]" : "["+ mapping.table.name+"."+mapping.
         dbField + "]" AS mappingInfo-»
226    <TR>
227      <TD href="#" TOOLTIP="«mappingInfo»" ALIGN="LEFT">«IF statemember.deprecated-»
           @Deprecated «ENDIF-»«statemember.name»</TD><TD ALIGN="RIGHT"><FONT COLOR="
           #888888">state</FONT></TD>
228    </TR>
229    «ENDLET-»
230    «ENDLET-»
231    «ENDIF»
232 «ENDDEFINE»
```

Listing A.4: Template dotModel

```
1 import wfsdatamodel;
2
3 import wfsservicemodel;
4 import wfsmodel;
```

```
5  import wfsormapping;
6
7  extension com::wieland::wfs::mdsd::servicemodel::GenExtensions reexport;
8  extension com::wieland::wfs::mdsd::ormappingmodel::Extensions;
9  extension com::wieland::wfs::mdsd::datamodel::Extensions;
10 extension org::openarchitectureware::util::stdlib::io;
11
12
13 extension com::wieland::wfs::mdsd::ormapping::extensions::ormappingCommon;
14 extension com::wieland::wfs::mdsd::ormapping::extensions::java::ormappingJava;
15 extension com::wieland::wfs::mdsd::datamodel::extensions::java::datamodel;
16 extension com::wieland::wfs::mdsd::servicemodel::Extensions;
17
18 Void addDotFileRelativePath(String path):
19    JAVA com.wieland.wfs.mdsd.datamodel.graph.extensions.GraphHelper.
          addDotFileRelativePath(java.lang.String);
20
21 String getSolutionColor(String solution):
22    switch(solution) {
23    case "APS": "ffcc99"
24    case "BASIS": "c0c0c0"
25    case "BDE": "ccffcc"
26    case "Logistik": "c0ffff"
27    case "MDE": "99ccff"
28    case "PDE": "339966"
29    case "Personal": "ff99cc"
30    case "QDE": "cc99ff"
31    case "Versand": "ffff99"
32    default: "cccccc"
33    };
34
35 String getMappingTooltip(List[Mapping] mappings):
36   mappings.size == 0 ? "[manual mapped]" : "[" + mappings.first().table.name + "." +
         mappings.first().dbField + "]" + getMappingTooltipExists(mappings.withoutFirst()
         )
37 ;
38
39 String getMappingTooltipExists(List[Mapping] mappings):
40   mappings.size == 0 ? "" : " [" + mappings.first().table.name +"." + mappings.first()
         .dbField + "]" + getMappingTooltipExists(mappings.withoutFirst())
41 ;
42
43 cached List[Mapping] getMappingsForEmbeddedAttribute(DataService this, DataObject
       dataobject, MemberDeclaration member):
44    let result = {}:
```

```
45       let complexMembers = getComplexMembersUsingEmbedded(dataobject):
46       (
47          addMappingsForComplexMembers(result, {}.add(member), complexMembers, 0)
48          -> result
49       );
50
51 Void addMappingsForComplexMembers(DataService this, List[Mapping] result, List[
       ComplexMemberDeclaration] memberHierarchy, List[ComplexMemberDeclaration] members,
        int i):
52    i >= members.size ?
53       Void
54    :(
55       let member = members.get(i):
56       (
57          !member.dataObject().isEmbedded() ?
58          (
59             let mapping = findMappingForMemberHierarchy(member.dataObject(), {}.addAll(
                  memberHierarchy).add(member)):
60                mapping != null ? result.add(mapping) : null
61          ):(
62             let complexMembers = getComplexMembersUsingEmbedded(member.dataObject()):
63                addMappingsForComplexMembers(result, {}.addAll(memberHierarchy).add(
                      member), complexMembers, 0)
64          )
65       )
66       -> addMappingsForComplexMembers(result, memberHierarchy, members, i+1)
67    );
68
69 cached Mapping findMappingForMemberHierarchy(DataService this, DataObject dataObject,
       List[ComplexMemberDeclaration] memberHierarchy):
70    let hierarchyReverted = memberHierarchy.reverse():
71       let entity = getORMapping(dataObject).getMappingEntity(dataObject):
72       (
73          entity == null ?
74             null
75          :(
76          let lastMember = hierarchyReverted.get(hierarchyReverted.size-1):
77          (
78             hierarchyReverted.remove(lastMember) ->
79             (
80                getMappingForMember(entity, hierarchyReverted.id(), lastMember)
81                )
82          )
83        )
84       );
```

```
85
86 cached List[ComplexMemberDeclaration] getComplexMembersUsingEmbedded(DataObject this):
87    dataModel().dataobjects.complexmembers.select(e|e.aggregation && e.dataobject ==
          this);
```

Listing A.5: Hilfsfunktionen als Extensions der Sprache Xtend

Literaturverzeichnis

[1] *aiSee – Graphvisualisierung*. http://www.absint.com/aisee/index_de.htm. Version: 2012

[2] *Eclipse Documentation*. http://help.eclipse.org/indigo/index.jsp. Version: 2012

[3] *Graphviz*. http://de.wikipedia.org/wiki/Graphviz. Version: 2012

[4] *Graphviz - Graph Visualization Software*. http://graphviz.org/. Version: 2012

[5] *yWorks*. http://www.absint.com/aisee/index_de.htm. Version: 2012

[6] ALLWEYER, T.: *Bpmn 2.0- Business Process Model and Notation:*. Books on Demand GmbH, 2009

[7] BEYDEDA, S. ; BOOK, M. ; GRUHN, V.: *Model-driven software development*. Springer, 2005 (Springer eBooks collection: Computer science)

[8] BIEN, Adam: *Java EE 5 Architekturen : Patterns und Idiome*. Frankfurt, M. : Entwickler.press, 2007

[9] BOSTOCK, Mike: *radial tree*. http://www.flickr.com/photos/mbostock/4368442997/in/photostream/. Version: 2012

[10] CARD, S.K. ; MACKINLAY, J.D. ; SHNEIDERMAN, B.: *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers, 1999 (Morgan Kaufmann Series in Interactive Technologies)

[11] CLAUS, V. ; SCHWILL, A.: *Duden Informatik: Ein Fachlexikon für Studium und Praxis*. Dudenverl., 2001

[12] COMMISSION, International E.: *IEC 62264 Enterprise-control system integration*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35480

[13] DEURSEN, Arie van ; KLINT, Paul ; VISSER, Joost: Domain-specific languages: an annotated bibliography. In: *SIGPLAN Not.* 35 (2000), Nr. 6, S. 26–36

[14] FORSCHNER, Alexander ; REICHERT, Manfred (Hrsg.) ; RINDERLE-MA, Stefanie (Hrsg.) ; KREHER, Ulrich (Hrsg.): *Fortschrittliche Datenflusskonzepte für flexible Prozessmodelle*. June 2009

[15] FOUNDATION, The E.: *Rich Eclipse-based Editors and IDE Integration*. `http://www.eclipse.org/Xtext/#features`. Version: 2012

[16] HOFMANN, Britta: *Vom Umgang mit Menschen - Benimmregeln für interaktive Systeme nach ISO 9241-110*. `http://www.fit-fuer-usability.de/archiv/einfuehrung-in-die-iso-9241-110/`. Version: 2008

[17] HORVATH, Zoltan: *Eine kompakte Einführung in die Erstellung einer openArchitectureWare Cartridge*. `http://www.oio.de/public/opensource/tutorial-oAW-Cartridge-Beispiel-Entwicklung-Anleitung.htm`. Version: 2009

[18] IP, Wieland Werke AG A.: *Architektur im Überblick*. `http://wwos/sites/wipi/Intern/IF%20Wiki/Architektur%20JEE.aspx`. Version: 2012

[19] IP, Wieland Werke AG A.: *Produktorganisation der WFS*. `http://wwos/sites/wipi/WFS/Produkt%20Wiki/Forms/Prozesstyp%20solution.aspx`. Version: 2012

[20] ITWISSEN: *UML-Profil*. `http://www.itwissen.info/definition/lexikon/UML-Profil-UML-profile.html`. Version: 2012

[21] JOUAULT, F. ; ALLILAIRE, F. ; BÉZIVIN, J. ; KURTEV, I. ; VALDURIEZ, P.: ATL: a QVT-like transformation language. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* ACM, 2006, S. 719–720

[22] KLEPPE, Anneke G. ; WARMER, Jos ; BAST, Wim: *MDA Explained: The Model Driven Architecture: Practice and Promise*

[23] KRUSKAL, J. B. ; LANDWEHR, J. M.: Icicle Plots: Better Displays for Hierarchical Clustering. In: *The American Statistician* 37 (1983), Nr. 2

[24] PIETREK, Georg (Hrsg.) ; TROMPETER, Jens (Hrsg.): *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Frankfurt am Main : Entwickler.Press, 2007

[25] PREIM, B. ; DACHSELT, R.: *Interaktive Systeme: Grundlagen, Graphical User Interfaces, Informationsvisualisierung*. Springer, 2010

[26] REICHERT, M. ; DADAM, P.: ADEPT flex—supporting dynamic changes of workflows without losing control. In: *Journal of Intelligent Information Systems* 10 (1998), Nr. 2, S. 93–129

[27] SCHUMACHER, Jochen: *Wertschöpfung ohne Verschwendung durch den Einsatz von MES*. http://www.competence-site.de/produktions-management/Wertschoepfung-ohne-Verschwendung-durch-den-Einsatz-von-MES. Version: 2012

[28] SHNEIDERMAN, Ben: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*, IEEE Computer Society, 1996

[29] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Heidelberg : dpunkt, 2007

[30] VALCKENAERS, P. ; BRUSSEL, H. V.: Holonic Manufacturing Execution Systems. In: *CIRP Annals - Manufacturing Technology* 54 (2005), Nr. 1, S. 427 – 432

[31] WANNER, Gerhard ; SIEGL, Stefan: Modellgetriebene Softwareentwicklung auf Basis von Open-Source-Werkzeugen – reif für die Praxis? In: *Informatik-Spektrum* 30 (2007), S. 340–352

[32] WIKIPEDIA: *Model Driven Architecture*. http://de.wikipedia.org/wiki/Model_Driven_Architecture. Version: 2012

Name: Peter Fuchs Matrikelnummer: 491849

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Peter Fuchs