Universität Bremen

Diplomarbeit

Modellgetriebene Software-Entwicklung: Vergleich von leichtgewichtiger und schwergewichtiger Methode am Beispiel des gerichtlichen Mahnverfahrens

Model-Driven Software Development: Comparison between light- and heavy-weighted methods by way of the judical dunning procedure

25 Mai 2009

Gutachter: Prof. Dr. Karl-Heinz Rödiger
Dr. Dieter Müller

Verfasser: Radek Eckert

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die mir beim Erstellen meiner Diplomarbeit durch ihre fachliche und persönliche Unterstützung zur Seite gestanden haben. Allen voran danke ich Prof. Dr. Karl-Heinz Rödiger für die Ermöglichung und konstruktive Unterstützung dieser Arbeit. Des weiteren danke ich meiner Freundin Christine und meiner Schwester Anja für das Korrekturlesen der Arbeit.
Außerdem bedanke ich mich ganz besonders bei meinen Eltern, die mir dieses Studium ermöglicht haben, und bei meiner ganzen Familie für die emotionale Unterstützung.

Kurzfassung

Die vorliegende Diplomarbeit beschäftigt sich mit modellgetriebener Software-Entwicklung und verschiedenen Methoden dieser. Seitdem die *Object Management Group* (OMG) im Jahre 2000 eine Spezifikation für *Model-Driven Architecture* (MDA) herausgebracht hat, wird dieser Ansatz unterschiedlich interpretiert und durchgeführt. Beim Ansatz der MDA wird der Programmcode durch automatisierte Transformationen aus Modellen erzeugt. Dafür setzt MDA mehrere Modelltransformationen und die Nutzung der UML als Modellierungssprache voraus. Diese Voraussetzungen werden in der Branche nicht als zwingend betrachtet. Dadurch entstehen ähnliche Ansätze, welche wie MDA Programmcode aus Modellen generieren, jedoch nicht genau der MDA entsprechen.

In dieser Arbeit werden zwei Methoden Modellgetriebener Software-Entwicklung neben der MDA dargestellt. Jede dieser Methoden weicht an einer Stelle von der Spezifikation der MDA ab. Die erste weicht von MDA insofern ab, indem direkt Programmcode aus UML-Modellen generiert wird; es findet also nur eine Modelltransformation statt. Die zweite Methode weicht von MDA dadurch ab, dass sie nicht die UML zum Modellieren der Modelle nutzt, sondern das *Eclipse Modelling Framework* (EMF).

Ziel der Arbeit ist es, mit den zwei unterschiedlichen Methoden Modellgetriebener Software-Entwicklung (englisch *Model-Driven Software Development*), jeweils dieselbe Anwendung zu erstellen, um die Methoden miteinander vergleichen zu können. Der Vergleich dient hauptsächlich dazu zwei Dinge zu erforschen. Zum Einen sollen die Vor- und Nachteile von mehreren Modelltransformationen während der Modellgetriebenen Software-Entwicklung erforscht werden, und zum Anderen die Vor- und Nachteile der Nutzung von UML für die Modellgetriebene Software-Entwicklung.

Als Anwendung soll das gerichtliche Mahnverfahren (GMV) entwickelt werden. Für die Entwicklung beider Methoden wird das Generator-Framework *openArchitectureWare* (oAW) unter Eclipse genutzt.

Abstract

This diploma thesis illustrates several different possible implementations of Model-Driven Software Development. Since the Object Management Group (OMG) released a specification for Model-Driven Architecture (MDA) in the year 2000, it has been interpreted in different ways, resulting in diverse implementations. In the MDA approach, source code is generated automatically through model transformation. Therefore MDA requires several model transformations and the use of UML as the modelling language. These requirements are not regarded as mandatory in the industry. Thus, similar approaches were formed, such as the MDA that generate source code from models, but which according to OMG do not correspond to the MDA.

In this work, two methods of Model-driven Software Development are shown in addition to the MDA. Each method differs in one way from the MDA specification. In the first method, source code is generated from UML models; therefore only one model transformation may take place. The second method differs from the MDA in that it creates models using the Eclipse Modeling Framework (EMF) instead of UML.

The aim of this thesis is to generate the same application using these two different methods of Model-Driven Software Development and to compare them with each other. This comparison is being done primarily to investigate two things. First, the advantages and disadvantages of the use of UML for Model-Driven Software Development are explored and second, the pros and cons of several model transformations during the Model-Driven Software Development process.

The application shall be developed by way of judicial dunning procedure. The openArchitectureWare (oAW) generator framework will be used under Eclipse for the development of both methods.

Inhaltsverzeichnis

1 Einleitung

Im Zuge der Rationalisierung der Software-Entwicklung werden neue Vorgehensweisen erprobt. Eine dieser Vorgehensweisen ist modellgetriebene Software-Entwicklung (englisch *Model-Driven Software Development*, MDSD). MDSD hat sich scheinbar in der IT-Branche etabliert; viele Unternehmen setzen bereits darauf. Daraus resultiert, dass vermehrt Software-Projekte mit diesem Vorgehensmodell realisiert werden. Aber modellgetriebene Software-Entwicklung ist nicht gleich modellgetriebene Software-Entwicklung. Wie so oft in der IT-Branche wird der Ansatz der modellgetriebenen Software-Entwicklung unterschiedlich interpretiert und umgesetzt. Daraus ergeben sich schwergewichtigere und leichtgewichtigere Ansätze. Diese Ansätze sind das Thema der Arbeit; sie sollen miteinander verglichen werden.

Modellgetriebene Software-Entwicklung hat in den letzten Jahren selbst eine Entwicklung von einem theoretischen Ansatz zu einer oft verwendeten Praktik gemacht. Zuerst erstellte die *Object Management Group* (OMG) einen Leitfaden zur modellgetriebenen Software-Entwicklung, genannt *Model-Driven Architecture* (MDA). Dieser Ansatz ist sehr anspruchsvoll und war auf Grund mangelnder Unterstützung durch Werkzeuge in der Praxis nicht durchführbar. Daraus ist eine vereinfachte Version entstanden: MDA-Light.

Grundsätzlich kann man sagen, dass bei der modellgetriebenen Software-Entwicklung der Großteil des Programmcodes durch eine oder mehrere Transformationen automatisch aus einem Modell erzeugt wird. Bei schwergewichtigen Ansätzen wird zuerst ein plattformunabhängiges Modell (englisch *Platform Independent Model*, PIM) erstellt. Dieses PIM wird in einem oder mehreren Schritten um Plattform spezifische Informationen erweitert. Durch diese Transformationen entstehen plattformspezifische Modelle (englisch *Platform Specific Model*, PSM). Bei der letzten Transformation wird dann der Programmcode generiert. Wie der Name schon sagt, ist MDA-Light ein vereinfachter, leichtgewichtiger Ansatz, in dem der Programmcode direkt aus einem Modell generiert wird. Nachdem immer mehr Tools doch MDA unterstützen können und weitere Nicht-OMG Standards entwickelt wurden, ist MDSD als Oberbegriff für modellgetriebene Software-Entwicklung anzusehen. Die verschiedenen Vorgehensweisen modellgetriebener Software-Entwicklung, die zugehörigen Begriffe und die Zusammenhänge dieser werden in Kapitel 2 dargestellt. Insbesondere die beiden Methoden, die in dieser Arbeit verglichen werden.

In der modellgetriebenen Software-Entwicklung spielt das Modell die zentrale Rolle; es wird in Modellierungssprachen erstellt und soll den Programmcode größtenteils ersetzen. Der Slogan hierfür heißt: „Das Modell ist der Code“. Aus diesem Grund werden in Kapitel 3 dieser Arbeit Modelle diskutiert. Laut OMG ist eine domänenspezifische Sprache (englisch *Domain* Specific Language, DSL) ein wichtiger Faktor für den Erfolg von MDA/MDSD. Mit diesen speziell auf einen Problemraum zugeschnittenen Sprachen kann man konkrete Sachverhalte dieser Problemräume darstellen bzw. modellieren. Domänen können fachliche wie auch technische Wissensgebiete sein. Die Domäne der Architekturen von Web-Applikationen ist eine technische, im Gegensatz dazu ist das Versicherungs- bzw. Bankwesen eine fachliche Domäne. Wenn eine DSL nun in einem Metamodell formalisiert ist, hat man eine auf den Problemraum zugeschnittene Modellierungssprache. Beim MDA-Light-Ansatz wird die DSL mit sogenannter leichtgewichtiger Metamodellierung erstellt, indem das UML Metamodell erweitert wird. Während bei sogenannter schwergewichtiger Metamodellierung ein eigenes Metamodell erstellt wird. In dieser Arbeit wird auch hinterfragen, welche Vorteile eine DSL mit eigenem Metamodell im Zusammenhang mit MDA/MDSD hat. Dazu wird bei der Realisierung der schwergewichtigen Methode eine propritäre DSL für eine Versicherung erstellt und in einem eigenen Metamodell formalisiert.

Um die Methoden miteinander vergleichen zu können, wird dieselbe Anwendung erst mit einer leichtgewichtigen und dann mit einer schwergewichtigen Vorgehensweise implementiert. Als Anwendung soll ein gerichtliches Mahnverfahren (GMV) eines Versicherungsunternehmens realisiert werden. Das GMV wird in Kapitel 4 erläutert.

In Kapitel 5 werden die Implementierungen beider Methoden behandelt. Dabei wird jeweils der gesamten Entwicklungsprozess vom Metamodell, über das Modell, die Modelltransformationen bis hin zu den individuellen Änderungen dargestellt.

In Kapitel 6 werden beide Ansätze miteinander verglichen; dabei soll unter anderem ausgewertet werden, inwieweit sich die Aufwände der verschiedenen Vorgehensweisen unterscheiden. Im Speziellen soll in dieser Arbeit ermittelt werden, welchen Nutzen mehrere Modelltransformationen in der modellgetriebenen Software-Entwicklung haben. Weiter ist der Unterschied einer propritären DSL im Gegensatz einer Erweiterung der UML eine weitere Frage die es in dieser Arbeit zu klären gilt. Verglichen werden die verschiedenen Methoden anhand der Effizienz der Methoden, der Qualität der Modelle und der Vollständigkeit des generierten Codes.

Abschließend wird in Kapitel 7 ein Fazit der gewonnen Erkenntnisse gezogen und ein Ausblick weiterer Möglichkeiten modellgetriebener Software-Entwicklung dargestellt.

Da ich während meiner Tätigkeit als Studentische Aushilfe bereits mit MDSD in Berührung gekommen bin und diese Vorgehensweisen äußerst interessant finde, will ich mich auf diesem Gebiet spezialisieren. Die Fragen dieser Arbeit sind auch deshalb von besonderem Interesse für mich, um die verschiedenen Ansätze nicht nur theoretisch, sondern auch praktisch zu beherrschen. Während dieser Tätigkeit konnte ich leider nur eine Teilaufgabe dieser Vorgehensweisen erlernen, deshalb ist die korrekte Durchführung dieser Vorgehensweisen ein Ziel dieser Arbeit. Weitere Ziele sind die Erforschung der Vor- und Nachteile von mehreren Modelltransformationen, sowie der Vor- und Nachteile der Nutzung einer propritären DSL.

2 Modellgetriebene Software-Entwicklung

In der objektorientierten Vorgehensweise ist eine durchgängige Überführung eines Modells in Programmcode möglich. Sie wird *Forward Engeeniring* genannt. Es gibt Entwicklungswerkzeuge die Änderungen im Code sogar automatisch im Modell nachvollziehen können. Dabei spricht man von *Roundtrip Engineering*. So hat Modellieren eine wichtige Rolle eingenommen. Eine Standard-Notation für die Modellierung (UML) hat sich etabliert. Man spricht von modellbasierter Software-Entwicklung. In diesem Kapitel wird die modellgetriebene Software-Entwicklung beschrieben, die als eine spezielle Variante der modellbasierten Software-Entwicklung anzusehen ist. Es gibt jedoch einen grundlegenden Unterschied zwischen modellbasiert und modellgetrieben: in der modellbasierten Software-Entwicklung wird der Programmcode per Hand aus dem Modell erstellt, während in der modellgetriebenen Software-Entwicklung der Code automatisch aus dem Modell erzeugt wird. Nachfolgend wird erst die Historie der Modellgetriebenen Software-Entwicklung vorgestellt, um dann die verschiedenen Vorgehensweisen zu erläutern.

2.1 Historie modellgetriebener Software-Entwicklung

Die Software-Industrie hat sich in den letzten Jahrzehnten zu einer der größten Industrien der Welt entwickelt (nach [DeMarco 97], S.2 f). Hohe Software-Entwicklungskosten haben eine ökonomische Auswirkung auf Unternehmen. Schlechtes Software-Design, das die Produktivität beeinflusst, hat eine noch größere Auswirkung. Obwohl die Software-Branche eine der produktivsten Industrien ist, hat sie einen schlechten Ruf. Statistiken besagen, dass ca. 70% der Software-Projekte Budget und Termin überschreiten (nach [Poitrek et al 07], S.10 f). Man spricht von der anhaltenden Software-Krise. Aus diesem Grund befindet sich die Branche in einem stetigen Wandel. Es wird nach einer Technologie gesucht, welche Sicherheit und Standards in die Software-Entwicklung einbringen soll. Hier bietet sich der Vergleich mit der Entwicklung in der Autoindustrie an. Herr Ford hat in den 20er-Jahren das Montageband eingeführt. Diese Rationalisierung war Anfang einer Revolution in der Autoindustrie und auch in anderen Industrien. Dieser Standard hat sich bis heute bewährt. Nach so einer Rationalisierung bzw. Automatisierung, die solche weitgehenden Auswirkungen hat, sucht man immer noch in der Software-Branche. Ob eine derartige Entwicklung möglich ist, bleibt eine spannende Frage.

Die Geschichte der Software-Branche zeigt, dass es schon mehrere Rationalisierungen dieser gab. Bereits zu Beginn der Software-Industrie wurde von *Assembler* über höhere Programmiersprachen wie *Cobol* oder *Fortran* die Entwicklung von Software rationalisiert. Dabei wurde die Software-Entwicklung auf eine höhere Abstraktionsebene gehoben. Abbildung 2.1 zeigt die Rationalisierung der Software-Entwicklung über den Abstraktionsgrad. Anfang der 90er Jahre gab es einen ersten Versuch modellbasierter Software-Entwicklung durch *Computer Aided Software Engeneering* (CASE). CASE-Werkzeuge erfüllten die Erwartungen aber nicht, da sie nicht ausgereift und proprietär waren (nach [Stahl/Völter 05], S.12). Nachdem CASE als gescheitert galt, hielt die Objektorientierung Einzug in die Software-Entwicklung. Objektorientierung war wieder eine echte Rationalisierung des Software-Entwicklungsprozesses. OO-Modellierungswerkzeuge haben die Kluften zwischen Modell, Design und Implementierung überwunden. Dadurch entstand der Notationsstandard UML und Werkzeuge, welche *Roundtrip Engineering* ermöglichen; UML-Modelle und Code konnten nun ohne Probleme im Einklang gehalten werden. Durch die Objektorientierung im Zusammenspiel mit diesen neuen Werkzeugen ergaben sich neue Möglichkeiten der Software-Entwicklung. Agile Vorgehensweisen wie z.B. Extreme Programming (XP) sind entstanden und Probleme größerer Komplexität konnten in Angriff genommen werden.

Vorläufiger Höhepunkt des Wandels der Software-Entwicklung ist der im Jahre 2000, von der OMG verabschiedete Standard, genannt MDA. Die OMG versteht MDA als weiteren Schritt aus

der Software-Entwicklung eine Ingenieursdisziplin zu machen. Dafür ist der Abstraktionsgrad wieder weiter gestiegen. Die drei hauptsächlichen Ziele von MDA sind: Portabilität, Interoperabilität und Wiederverwendung. Wie diese Ziel erreicht werden sollen und alles weitere zu MDA, wird im Abschnitt 2.2 erläutert. Leider gab es zu Beginn keine Unterstützung durch Werkzeuge, so dass MDA nicht anwendbar war. In den folgenden Jahren (bis ca. 2003) kristallisierte sich eine vereinfachte Version von MDA heraus, genannt MDA-Light. Dieser Ansatz ist nirgends definiert und kein Standard. Es wird dabei, ähnlich dem MDA-Standard Programmcode, aus UML-Modellen generiert. Ein Vorgehen wird im Absatz 2.3 dargestellt. Wieder ein paar Jahre später (bis ca. 2005) wurde der MDA-Ansatz aufgegriffen, um ihn ohne die von OMG geforderten Standars zu nutzen. Dieser Ansatz wird in der Literatur MDSD genannt und ist MDA bis auf wenige Details sehr ähnlich. Obwohl modellgetriebene Software-Entwicklung, also MDSD, schon länger ein Thema war, ist sie erst nach der Veröffentlichung von MDA wieder in den Fokus gekommen und mit Hilfe der Ideen der MDA neu verwirklicht worden. MDSD ist durch die Auflockerung der Standars praktikabler als MDA und wird im Abschnitt 2.4 abgehandelt. MDA ist bis heute den Beweis schuldig geblieben, mit freien Werkzeugen und unter Berücksichtigung aller Standards, durchführbar zu sein. Es gibt zwar freie Tools wie z.B. oAW und AndroMDA, welche seit kurzen den Anspruch erheben echte MDA-Tools zu sein. Ob man mit ihnen tatsächlich MDA mit allen OMG Standards machen kann, ist zu überprüfen.

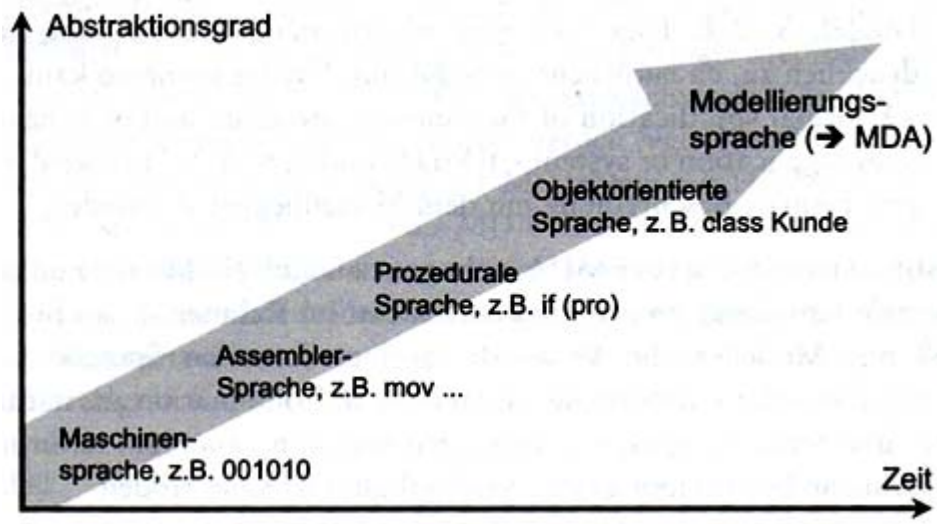

Abbildung 2.1: Rationalisierung der Software-Entwicklung durch Steigerung der Abstraktionsgrades ([Petrasch/Meimberg 06], S.46 Abb. 2-4)

2.2 Model-Driven Architecture (MDA)

In diesem Abschnitt werden die Ziele und zentralen Konzepte von MDA abgehandelt. Laut OMG sind die Hauptziele von MDA: Portabilität, Interoperabilität und Wiederverwendung (nach [MDA 03], Abschnitt 2.1). Um diese Ziele verwirklichen zu können, trennt die OMG die Geschäftslogik von der Plattformtechnologie ab. Diese Trennung erfolgt in den verschiedenartigen Modellen, welche nachfolgend erläutert werden.

Ein *Computation Independent Model* (CIM) ist vergleichbar mit einer Anforderungsanalyse und Bestandteil des MDA-Ansatzes der OMG. Es spezifiziert die Anforderungen der Anwendung unabhängig von der Realisierung und soll somit ein besseres Verständnis der Aufgaben der Anwendung ermöglichen. Üblicherweise werden im CIM auch domänenspezifische Zusammenhänge beschrieben die ein gemeinsames Vokabular eines Problemraums definieren

(nach [Piotrek et al. 07], S.14). Zur Notation sind sowohl beliebige Diagrammarten, als auch umgangssprachliche Beschreibungen möglich.

Das PIM spezifiziert formal das zu entwickelnde System unabhängig von Plattformen, unter Berücksichtigung der Anforderungen aus dem CIM. Dadurch ist es portierbar und für verschiedene Plattformen wiederverwendbar. Genau das ist mit Portabilität gemeint; die Plattformunabhängigkeit von Software-Systemen. Interoperabilität im Sinne der OMG bedeutet Herstellerunabhängigkeit durch Standardisierung. Dazu hat die OMG diverse Standards herausgebracht, die im Abkürzungsverzeichnis zusammengefasst werden und im weiteren Verlauf dieses Abschnitts miteinander in Zusammenhang gebracht werden. Bereits erstellte Architekturen, Modelle und Transformationsregeln können wiederverwendet werden und machen somit Expertenwissen in Form von Software verfügbar; auch das versteht die OMG unter Wiederverwendung. Die Plattformtechnologie wird in einem oder mehreren PSM‘s beschrieben, wobei diese automatisch durch Transformationsregeln aus dem PIM erzeugt werden. Diese Regeln müssen für jede Plattform neu programmiert werden. Mit Plattform sind Zielplattformen gemeint wie z.B. J2EE/Java, Webservices oder propiertäre Frameworks. Dabei sagt MDA erst einmal nichts über den Abstraktionsgrad der Plattform aus. Abstraktion im Sinne der MDA bedeutet das Weglassen von Details; je mehr technische Informationen man weglässt, desto höher wird der Abstraktionsgrad. Durch einen höheren Abstraktionsgrad erreicht man höhere Allgemeingültigkeit, bessere Wiederverwendbarkeit, Portierbarkeit und Lesbarkeit (nach [Petrasch/Meimberg 06], S.45). Aus einem PSM wird in einer letzen automatisierten Transformation Programmcode generiert. Dies ist eine textuelle Version des PSM und kann deshalb auch als Modell bezeichnet werden. Abbildung 2.2 verdeutlicht das Grundprinzip der MDA.

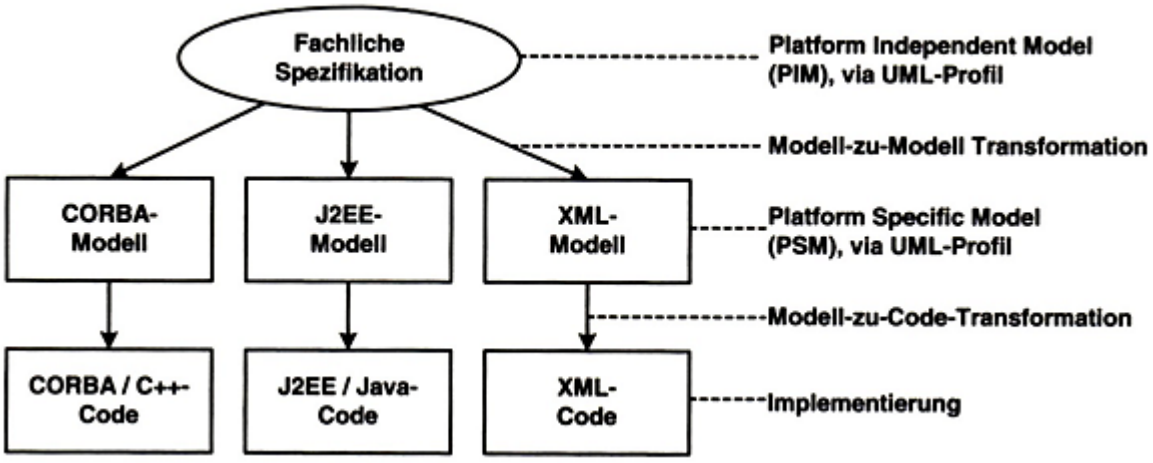

Abbildung 2.2: MDA-Grundprinzip ([Stahl/Völter 05], S.18 Abb.2-2)

Wie der Name schon sagt, sind Modelle zentraler Bestandteil modellgetriebener Software-Entwicklung. Dafür sollten die Modelle so präzise und maschinenlesbar entworfen werden, dass sich die Erzeugung der Anwendung automatisieren lässt. Bei MDA werden die Modelle in UML erstellt. UML ist die verbreitetste Modellierungssprache und ein Standard der OMG. Die Modelle müssen laut MDA-Spezifikation nicht zwingend in UML erstellt werden, jedoch verlang MDA eine Modellierungssprache die MOF-konform ist ([MDA 03], Abschnitt 3.3). *Meta Object Facility* (MOF) ist ebenfalls ein OMG-Standard der Kern des MDA-Ansatzes ist (nach [Stahl/Völter 05], S.375 f). In MOF werden Metamodelle beschrieben. MOF ist folglich ein Metametamodell. Ab der Version 2.0 ist das UML-Metamodell jetzt MOF-konform und entspricht den MDA-Richtlinien. Meinen Recherchen nach, gibt es zur Zeit neben UML keine andere Modellierungssprache, welche auf der aktuellen MOF basiert. Aus diesem Grund sind Modelle für die MDA bis jetzt zwingend mit UML zu erstellen. Mehr über Modelle findet man in Kapitel 3 dieser Arbeit. Ein weiterer Standard Namens XMI basiert auch auf MOF. XMI ist in XML definiert und zum Speichern und Austauschen der Modelle gedacht. Leider benutzen die

verschiedenen UML-Tools unterschiedliche XML-Dialekte für ihre XMI-Schnittstellen, sodass ein problemloser Datenaustausch, wie er von der OMG gedacht war, nicht immer machbar ist (nach [Stahl/Völter 05], S.376 f).

Nachfolgend soll die schrittweise Erstellung einer Anwendung mit MDA skizziert werden. Zuerst wird ein CIM erstellt um die domänenspezifischen Zusammenhänge und die Funktionalitäten der Anwendung umgangssprachlich zu beschreiben. Dieses CIM dient später als Basis für die DSL und das PIM. Wenn man sich im Klaren darüber ist, was man entwickeln möchte, erstellt man einen Prototypen als Referenzimplementierung. Spätestens jetzt sollte man sich für eine Architektur bzw. für die zu nutzenden Plattformen entscheiden. Durch die Analyse der Referenzimplementierung extrahiert man drei Arten von Code:

- Generischen Code, der für alle Anwendungen gleich ist
- Schematischen Code, der für alle Anwendungen semantisch gleich ist (Entwurfsmuster)
- Individuellen Code, anwendungsspezifischer Anteil, der nicht verallgemeinerbar ist

Der schematische und generische Anteil des Codes, kann aus Modellen generiert werden. Dazu wird ein Anwendungsmodell (PIM) erstellt, aus welchem wiederum alle Teilanwendungen der Applikation ableitbar bzw. generierbar sind. Das erstellte Anwendungsmodell nutzt die erstellte DSL. Im Prozess der Generierung der Teilanwendungen müssen Transformationen durchgeführt werden. Die Regeln für diese Transformationen werden mit einer neuen Modellabfragesprache von der OMG erstellt. Dieser Standard heißt QVT und ist erst 2005 spezifiziert worden (nach [Piotrek et al. 07], S.71). Die Zusammenhänge dieser Vorgänge werden noch einmal in Abbildung 2.3 verdeutlicht.

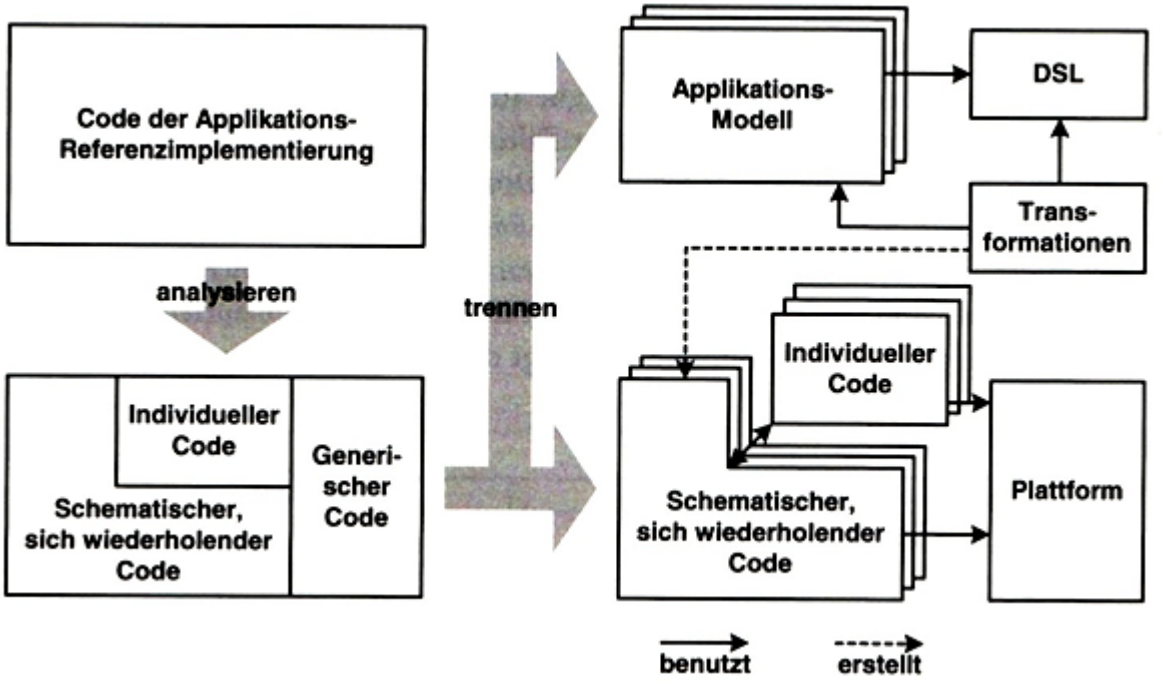

Abbildung 2.3: Grundidee modellgetriebener Software-Entwicklung ([Stahl/Völter 05], S.17)

Um Abbildung 2.3 richtig im Sinne der MDA zu interpretieren, sollte man erwähnen, dass sich hinter dem Kasten Transformationen mehrere Transformationsschritte verbergen. Um Teilanwendung aus dem PIM zu generieren, wird in einer ersten Transformation (M2M) ein PSM erzeugt, welches ein Modell ist dass mit plattformspezifischen Informationen angereichert wird. Diesen Schritt kann man im Sinne der OMG beliebig oft machen, insofern man mehrere verschiede Plattformen hat und die Informationen dieser nur separat in das Modell einfügen will. Dadurch wäre man flexibler im Austausch der einzelnen Plattformen. Hier sollte man erwähnen

das PIM und PSM relative Konzepte sind. So kann das erste generierte PSM spezifisch für eine Plattform sein, aber immer noch unabhängig von einer weiteren, die noch durch weitere Transformation hinzugefügt werden soll. Erst in der letzten Transformation wird aus dem PSM Programmcode generiert, dieser *Platform Specific Implementation* (PSI) genannte Programmcode wird aus sogenannten Templates bzw. Cartridges erzeugt. Diese Templates sind in Template-Sprachen, neuerdings auch in QVT, geschrieben und enthalten Codemuster der Anwendung, welche so allgemein gehalten sind, dass sie auf alle möglichen Teilanwendungen zutreffen. Aus diesen Templates wird Programmcode als Ausgabe generiert, der einem Schema entspricht und in die passenden Dateien geschrieben wird. Um den korrekten Ablauf kümmert sich ein Generator-Framework, das auf die eigenen Bedürfnisse konfiguriert wurde. Abschließend wird der generierte schematische Programmcode der Anwendung bzw. der Teilanwendungen per Hand um den individuellen Code ergänzt, um die Anwendung komplett lauffähig zu machen. Übersetzen des Codes und Testen der Anwendung sei hier nur kurz erwähnt, da es sowieso in jeden Software-Entwicklungsprozess gehört.

Zu Beginn meiner studentischen Tätigkeit im Jahre 2006 gab es noch keine frei verfügbaren Generatoren, die MDA vollends mit all seinen Standards unterstützten. Mittlerweile soll oAW auch Modell zu Modell (M2M) Transformationen durchführen können, wobei die Modelle über UML-Tools auch MOF konform sein können ([Piotrek et al. 07], S.192). Ob man nun aber den gesamten MDA Prozess mit allen Standards der OMG in der Praxis durchführen kann, gilt es zu überprüfen. Trotz allen Schwierigkeiten bei der Verwirklichung von MDA-Projekten, bietet der Ansatz laut vieler hier zitierter Literatur einen guten Einstieg in die modellgetriebene Software-Entwicklung, so dass alle in der Praxis verwendeten Ansätze MDA als Basis haben. Des weiteren wird behauptet das MDA eine Standardisierungsinitiative der OMG zum Thema MDSD sei (nach [Stahl/Völter 05], S.5).

2.3 MDA-Light

MDA-Light ist eine vereinfachte Version von MDA. Diese ist durch die Skepsis der Branche gegenüber den verfügbaren Werkzeugen entstanden. Die Skepsis der Branche bezog sich auf die Nutzbarkeit eines PSM in der Praxis (nach [Bohlen/Starke 03], S.52 f). Aus diesem Grund wurde in der Praxis auf Werkzeuge gesetzt, die direkt aus einem Modell Programmcode generierten. Diese Vereinfachung des methodischen Ansatzes aus pragmatischen Gründen ist nicht neu und wird *Early Adaptors* Phänomen genannt. Auf diese Weise entstehen durchaus brauchbare Technologien.

AndroMDA spielt bei der Entstehung von MDA-Light eine wichtige Rolle. In ([Bohlen/Starke 03], S.56) und ([Richly et al. 05], Abschnitt 2) wird die Art und Weise wie AndroMDA Programmcode erzeugt als MDA-Light bezeichnet. Hierzu sollte jedoch erwähnt werden das MDA-Light kein Standart ist. AndroMDA ist ein Nachfolgeprojekt von UML2EJB, indem Java-Code aus UML-Modellen generiert wurde. Die Beschränkung auf EJB gilt für AndroMDA keinesfalls, es sind alle Zielplattformen aus UML-Modellen generierbar. Das Weglassen von PSM ist aber nicht der einzige Unterschied zu MDA.: die OMG sieht in der Erstellung einer DSL einen Erfolgsfaktor für die Modellierung und Generierung von Anwendungen. [Bohlen/Starke 03] und [Richly et al. 05] schreiben nichts von der Erstellung einer DSL in ihren Zusammenfassungen zu MDA-Light. Das Anreichern des Anwendungsmodells ist dennoch zu vergleichen mit der Erstellung einer DSL. Man spricht in diesem Zusammenhang auch von leichtgewichtiger und schwergewichtiger Metamodellierung ([Petrasch/Meimberg 06], S.73 ff). Bei MDA-Light wird das PIM um Informationen angereichert, welche die Transformationen steuern sollen. Dabei handelt es sich um Stereotypen und TaggedValues, die im UML-Profile definiert werden. Stereotypen können UML-Modellelementen bestimmte Bedeutungen geben und TaggedValues können ihnen zusätzliche Eigenschaften verleihen. Ein solch angereichertes PIM wird marked PIM genannt, obwohl es ein PIM ist, das teilweise

mit plattformspezifischen Informationen angereichert wurde. Man könnte auch sagen, dass die Transformation vom PIM zum PSM per Hand vorweg genommen wurde.

([Bohlen/Starke 03], S.54) beschreiben den Ablauf der Software-Entwicklung mit AndroMDA in sieben Schritten (siehe dazu auch Abbildung 2.4):

1. Templates programmieren, die festlegen wie aus Geschäftsmodell-Klassen notwendige Klasse in der Zielplattform/Programmiersprache werden.
2. Modellieren in UML, wie PIM in MDA.
3. Anreichern des Modells; ein marked PIM entsteht.
4. Generator starten, um schematischen Programmcode zu erzeugen.
5. Generierter Programmcode wird um individuellen Programmcode ergänzt.
6. Build-Skript kompiliert und erzeugt komplette Anwendung.
7. Anwendung testen.

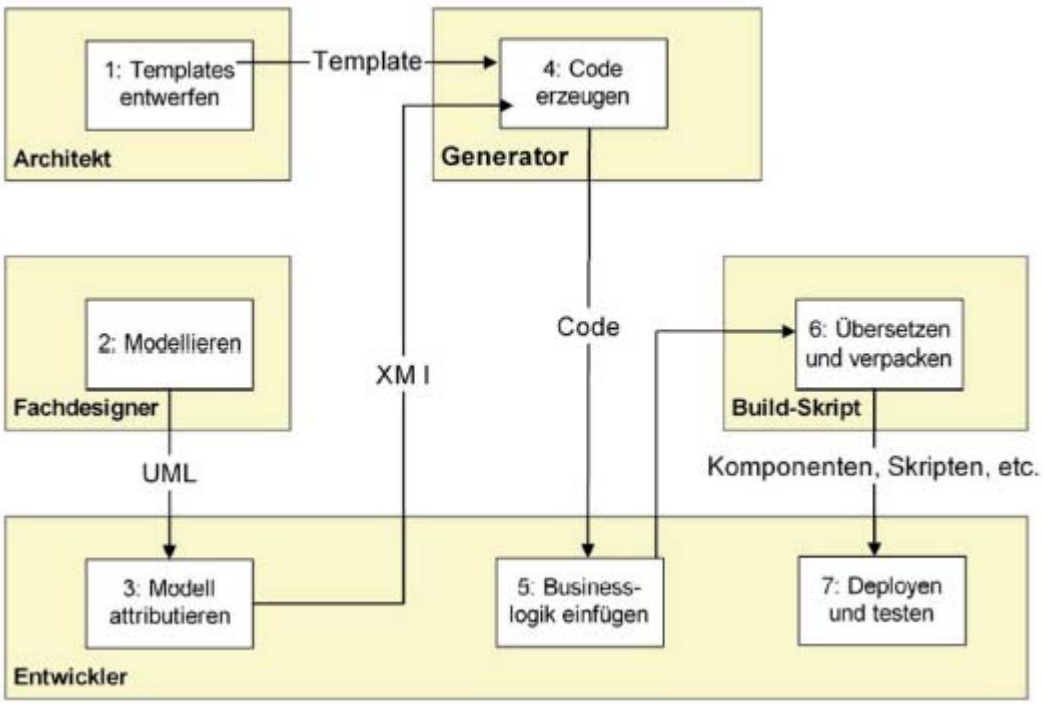

Abbildung 2.4: Entwicklungszyklus in AndroMDA-Kontext ([Bohlen/Starke 03], S.4 Abb. 2)

In dieser Arbeit wird mit dem MDA-Light-Ansatz, der in Kontext von AndroMDA entstanden ist, entwickelt. Dazu werden die Schritte 1-7 mit dem UML-Werkzeug TopcaseD und dem Generator-Framework *openArchitektureWare* durchgeführt. Beide Tools sind in Eclipse integriert und lassen sich leichtgewichtig nutzen wie oben beschrieben wurde.

2.4 Model-Driven Software Development

MDSD ist der Oberbegriff für modellgetriebene Software-Entwicklung und beschreibt nicht nur einen Ansatz, sondern leichtgewichtigere und schwergewichtigere Ansätze. Genau genommen sind die Ansätze aus 2.3 und 2.4 ebenfalls MDSD-Ansätze, da die Standards der OMG für MDSD nicht als zwingend betrachtet werden, um modellgetriebene Software-Entwicklung praktikabler zu machen (nach [Stahl/Völter 05], S.5). In dieser Arbeit werden die Ansätze aus 2.3 und 2.4 aber weiterhin MDA und MDA-Light benannt, um eine klare

Unterscheidung zu ermöglichen. Weiterhin wird modellgetriebene Software-Entwicklung als MDSD bezeichnet. Der Ansatz von MDSD stimmt weitestgehend mit MDA überein. Es gibt zwei Punkte in denen sich MDA und MDSD unterscheiden können. Der erste Punkt ist die Erstellung eines PSM. Der theoretische Nutzen der Trennung von PIM zu PSM ist erkennbar, man erhält einen größeren Freiheitsgrad beim Generieren von Programmcode. Der zusätzliche Zwischenschritt zum Programmcode ist für Projekte, welche diese Flexibilität nicht brauchen jedoch nicht praktikabel (nach [Stahl/Völter 05], S.27). Der zweite Punkt ist die Nutzung von Standards, im Speziellen vom Standard der Modellierungssprachen. Für MDSD ist es nicht unbedingt nötig, ein MOF-konformes Tool wie UML zu nutzen. Es ist zwar oft von Vorteil, da man mit UML eine breite Unterstützung erhalten kann, jedoch kann man mit einer eigenen Modellierungssprache/DSL genau so gut modellgetrieben entwickeln. MDA ist als eine schwergewichtige MDSD-Implementierung anzusehen.

Die Ziele von MDSD sind mit den Zielen von MDA, sowie den Zielen jeder Rationalisierung der Software-Entwicklung bis auf einen Punkt gleich zu setzen. Sie wurden aus ([Stahl/Völter 05], S.14 f) entnommen und werden hier noch mal zusammen gefasst:

- Steigerung der Entwicklungsgeschwindigkeit durch Automation
- Steigerung der Qualität von Software durch Einsatz von automatisierter Transformationen und formal definierter Modellierungssprachen
- Redundanzvermeidung, bessere Wartbarkeit und Handhabbarkeit von Technologiewandel durch die so genannte Trennung von Verantwortlichkeiten
- Mehr Wiederverwendung von definierten Architekturen, Modellierungssprachen und Transformationen, dadurch Nutzung von Expertenwissen
- Probleme größerer Komplexität leichter lösbar durch Abstraktion
- Erzeugung eines Produktiven Umfeldes durch Best Practices und Prozessbausteine
- Portabilität und Interoperabilität durch Standardisierung

Der letzte Punkt kann nur erfüllt werden, wenn man einen MDSD-Ansatz wählt, der auf Standards wie die der OMG setzt. Die Ziele klingen mal wieder nach den heilsbringerischen Versprechen, die in der Branche zu Recht skeptisch betrachtet werden. Hierzu sollte erwähnt werden, dass zum Erreichen dieser Ziele natürlich gewisse Voraussetzungen und Investitionen nötig sind ([Piotrek et al. 07], S.18 f). Als Voraussetzungen gelten: organisatorische Rahmenbedingungen, bestehende Vorgehensmodelle, vorhandene Projektkultur, Know-how der Mitarbeiter, Unterstützung des Managements und vorgeschriebene Werkzeuge. Zu den erforderlichen Investitionen zählen die Ausbildung von Mitarbeitern, die Entwicklung eines passenden Generators und die Anpassung des Entwicklungsprozesses.

Wie man an den Voraussetzungen und Investitionen sehen kann, entstehen bei einer MDSD-Vorgehensweise größere Aufwände als bei herkömmlichen Vorgehensweisen um Software zu entwickeln. Da man Aufwände schätzen kann, sollte es theoretisch möglich sein, eine Kosten-Nutzen-Analyse zu erstellen, um zu berechnen, ob ein MDSD-Ansatz für eine Entwicklung in Frage kommt. Ein Szenario unter dem sich ein MDSD Vorgehensmodell auszahlt ist, wenn es um eine Anwendung mit langer Lebensdauer geht, wobei sich die Aufwände während der Wartungsphase auszahlen. Ein weiteres Szenario ist, wenn es um die Entwicklungen mehrerer Anwendungen in derselben Architektur geht. Die zusätzlichen Aufwände fallen nur einmal an, während der Nutzen sich mit jeder weiteren Anwendung vervielfacht (nach [Piotrek et al. 07], S.17).

In dieser Arbeit wird nach einem schwergewichtigen MDSD-Ansatz entwickelt, der MDA bis auf ein Detail entspricht. Anstatt einer MOF-konformen Modellierungssprache will ich eine eigene DSL auf EMF Basis erstellen, um den Nutzen dieser eigens für eine Domäne erstellten Sprache zu untersuchen. Dadurch wird die Erweiterung einer bereits sehr umfangreichen Modellierungssprache wie UML vermieden. Dazu wird die DSL in *Ecore* formuliert und somit eine Modellierungssprache mit dem EMF erstellt, um damit PIM zu modellieren und in PSM zu transformieren. Der Ansatz aus Abschnitt 2.3 wird bis auf den

Standard der Modellierungssprache angewendet; mit Transformationen von PIM über PSM zum PSI. Die Modelltransformationen werden bei dieser Vorgehensweise ebenfalls mit dem Generator-Framework oAW erstellt. Dabei wird unter anderem auch untersucht, ob man die Vorteile von MDA auch ohne die Standards der MDA-Spezifikation nutzen kann.

3 Modelle

In diesem Kapitel werden die Modelle im Sinne modellgetriebener Software-Entwicklung behandelt. Der Ausdruck Modell ist aus dem lateinischen Wort modulus hergeleitet und bedeutet das Maß oder Maßstab. In Lexika werden zur Erklärung des Ausdrucks oft die Synonyme Muster und Entwurf genannt.

„Die Modelle in der Informatik sind im Allgemeinen abstrakte Abbilder oder Vorbilder zu konkreten oder abstrakten Originalen“ ([Kastens/Büning 08], S.18).

In der modellgetriebenen Software-Entwicklung ist ein Modell eine abstrakte Abbildung eines Software-Systems. Modelle für MDSD können über grafische Diagramme, strukturierte Texte oder in Tabellenform dargestellt werden. Darüber hinaus unterschiedet man zwischen dynamischen Modellen und statischen Modellen. Dynamische Modelle stellen ein Verhalten des Systems dar, z.B. Aktivitätsdiagramme in UML. Statische Modelle repräsentieren feste Strukturen, z.B. Klassendiagramm in UML. Modelle werden in sogenannten Modellierungssprachen erzeugt, besser gesagt modelliert. Definiert werden Modellierungssprachen in Metamodellen über Schlüsselwörter oder grafische Symbole.

3.1 Modellierungsprachen

In diesem Abschnitt werden verschiedene Arten von Modellierungssprachen behandelt um dann ein paar Modellierungsprachen vorzustellen, die sich für MDSD eignen. Wie schon erwähnt, unterscheiden sich Modellierungssprachen von der Art und Weise wie mit Ihnen modelliert wird. Beispiele für grafische Modellierungssprachen sind UML, EPK oder ER.

UML ist geeignet für dynamische und statische Modelle durch die Vielzahl an Diagrammtypen. Die meisten UML-Tools unterstützen Diagramme wie z.B. Klassendiagramme, Komponentendiagramme, Aktivitätsdiagramme, Anwendungsfalldiagramme, Zustandsdiagramme, Sequenzdiagramme und noch einige mehr. UML bietet somit Möglichkeiten, Sachverhalte auf verschiedene Arten zu repräsentieren. In den meisten Fällen werden zur Generierung von Programmcode nur Klassen- und Aktivitätsdiagramme verwendet. UML eignet sich selbstverständlich für MDSD.

Ereignisgesteuerte Prozessketten (EPK) stellen Geschäftsprozesse grafisch dar. EPK ermöglichen die Abbildung einer Funktion zu einem Prozess in zeitlich logischer Abfolge und eignen sich nur bedingt zur Modellierung von Geschäftsprozessen für die Generierung (nach [Mielke 02], S.38). Zur Modellierung von Geschäftsprozessen für die Generierung hat sich ([Mielke 02], S.140) für UML-Aktivitätsdiagramme entschieden, die er jedoch noch erweiterte. Generierung aus Geschäftsprozessmodellierung ist ein spannendes Thema für sich, das ich im Rahmen meiner Arbeit auch gerne untersucht hätte, jedoch würde das den Rahmen dieser Arbeit sprengen. Aus diesem Grund wird in dieser Arbeit nur aus statischen Modellen generiert.

Entity Relationship (ER) ist eine Modellierungssprache mit der die persistente Datenstrukturen modelliert werden (Datenmodellierung).

Beispiel für Modellierungssprachen, die Modelle in strukturierten Texten darstellen ist XML. Auf XML-Basis werden eigene XML-basierte Sprachen definiert. Prinzipiell eignen sich dazu alle Dateiformate, die man über einen Parser auswerten kann. Für XML spricht, dass es offen verfügbare Parser zur Auswertung von XML-Dateien gibt. OMG hat XMI (ein XML-Derivat) definiert, um MOF-basierte Modelle textuell darzustellen, bzw. über dieses Format auszutauschen. Wenn man in UML modelliert und daraus generieren will, werden die

Modelle in XMI gespeichert und ausgewertet. Wenn man direkt in XML modellieren würde, könnte man sich den Schritt des Zeichnens sparen und würde die Informationen direkt in die XML-Datei schreiben. Es gibt Projekte welche aus Performanz-Gründen direkt in XML modellieren.

Ein Beispiel für eine auf XML Basierende Modellierungssprache ist *Business Process Execution Language* (BPEL). BPEL wurde 2002 von IBM und Microsoft eingeführt und dient zur Beschreibung von über Webservices kommunizierenden Geschäftsprozessen.

Zur Erstellung eigener Modellierungssprachen gibt es im Umfeld von Eclipse entsprechende Werkzeuge: das EMF- und GMF-Framework. Mittels eines Metamodells, welches in *Ecore* definiert wird, können grafische und textuelle Modellierungssprachen erstellt werden. Mittels EMF kann man textuelle Modellierungssprachen daraus erstellen und GMF kann daraus grafische Repräsentation erstellt. Außerdem hat man dann eine Modellierungssprache die speziell auf die eigenen Bedürfnisse zugeschnitten ist. Eine Modellierungssprache die für eine Domäne erstellt wurde nennt man Domänen spezifische Sprache oder *Domain Specific Language* (DSL). In dieser Arbeit werden die EMF für den MDSD-Ansatz erprobt, um eine eigene DSL zu definieren. Für die Entwicklung mit MDA-Light wird in UML modelliert.

3.2 Metamodelle

Die OMG definierte für die modellgetriebene Software-Entwicklung vier Metaebenen. Die in UML erstellten Modelle befinden sich auf der Metaebene M1. Auf der Ebene M0 befindet sich eine konkrete Ausprägung des modellierten Sachverhaltes. Auf Ebene M2 befindet sich z.B. das UML-Metamodell und eigene DSL. Metametamodelle wie MOF und *Ecore* definieren wiederum Metamodelle und befinden sich auf Ebene M3. Zum besseren Verständnis dient Abbildung 3.1.

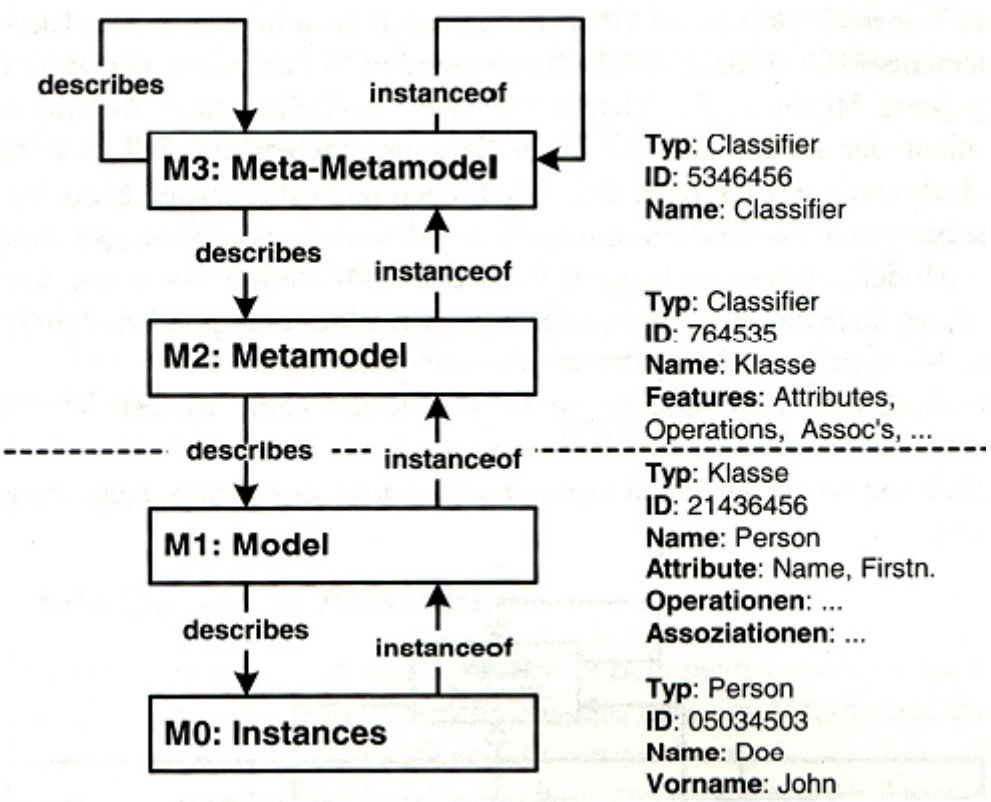

Abbildung 3.1: Die vier Metaschichten der OMG ([Stahl/Völter 05], S.93)

In Metamodellen wird die abstrakte Syntax und die statische Semantik einer Modellierungssprache definiert (nach [Stahl/Völter 05], S.67 f). Im Sinne von MDSD werden durch Metamodellierung eigene DSL erzeugt, die meistens auf UML basieren. In UML wird eine DSL durch die Erweiterung des UML-Metamodells definiert. Dazu gibt es eine leichtgewichtigere und eine schwergewichtigere Möglichkeit der Erweiterung (nach [Petrasch/Meimberg 06], S.73 ff). In der leichtgewichtigen wird das UML-Profil mit Stereotypen erweitert. Viele Werkzeuge verfügen bereits über Sätze von UML-Profilen, die für bestimmte Domänen erweitert wurden. Diese DSL sind aber immer noch sehr allgemein gehalten.

Für leichtgewichtige Metamodellierung werden neue UML-Sprachkonstrukte erzeugt, indem man Stereotypen erstellt, die von bereits vorhandenen UML-Elementen wie z.B. UML::Class oder UML::Attribute erben. Diese Stereotypen kann man dann nach belieben anpassen. Diese neuen Elemente sind dann formal in einem UML-Metamodell definiert. Abbildung 3.2 zeigt ein Beispiel eines neuen Stereotypen, der von der Metaklasse UML::Class erbt.

Abbildung 3.2: Definition eines Stereotypen in UML 2 ([Stahl/Völter 05], S.100 Abb.6-11)

Für eine schwergewichtige Erweiterung von UML würde man das Metamodell der UML anpassen, also eine eigene *Metaclass* in das Modell einfügen. Abbildung 3.3 zeigt drei neue Metaclasses: *Entity*, *Component* und *ComponentInterface*. Diese Anpassung des Metamodells befindet sich in einem neuen Package namens *PersistentComponent*. Durch diese eigenen Erweiterungen soll das eigentliche Metamodell besser unterschieden werden können (nach [Pertasch/Meimberg 06], S.75 ff).

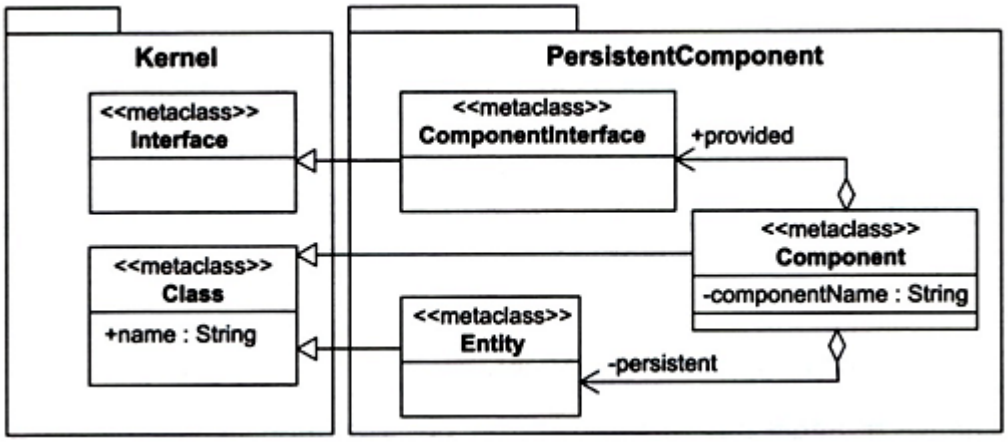

Abbildung 3.3: Schwergewichtige Erweiterung der UML mittels Metamodellierung ([Petrasch/Meimberg 06], S.77 Abb. 2-25)

Man kann auch Metamodelle für eigene DSL definieren ohne UML zu nutzen. Die Darstellungsmöglichkeiten von UML-Werkzeugen sind beschränkt; man kann oft nicht einmal alles darstellen, was das UML-Metamodell erlaubt (nach [Poitrek et al. 07], S.64 f). Mit einer eigens erstellten Modellierungssprache hat man einen höheren Freiheitsgrad während der Modellierung. Dazu definiert man selbst ein Metamodell, ohne ein vorgegebenes wie das der UML zu erweitern. Diese Art von Metamodellierung betrachtet man auch als schwergewichtig. Eine Möglichkeit, ein solches Metamodell zu erstellen, bietet EMF. EMF ist ein Framework mit dem eigene Modellierungssprachen auf Eclipse-Ecore Basis erstellt werden können. Das Ecore-Metamodell befindet sich, wie MOF, auf der Metaebene M3. Die

Bezeichnungen an dieser Stelle sind leicht irreführend, da Metamodelle auf Ebene M2 sind. Das Ecore-Metamodell ist aber ein Metamodell für Metamodelle, also auch ein Metametamodell. Eclipse stellt in dieser Richtung einige Werkzeuge zur Verfügung. So kann man aus einem in *Ecore* definierten Metamodell mittels EMF eine eigene Modellierungssprache generieren. Weiter gibt es noch die Möglichkeit, mithilfe von GMF, aus dieser Sprache eine grafische Modellierungssprache zu generieren. Dabei wird jedem Sprachelement der neuen Modellierungsprache ein grafisches Element zugeordnet, um dann mit diesen Zuweisungen eine eigene neue grafische Modellierungssprache zu generieren. Diese Sprache kann man dann, unter Eclipse, mit dem integrierten Werkzeug TOPCASED nutzen. TOPCASED ist ein grafisches Modellierungswerkzeug für viele Formate, u. a. kann man damit auch in UML modellieren.

Eclipse-Tools, besser gesagt Tools, die in Eclipse integrierbar sind, bieten viele Möglichkeiten der Modellierung. *XText* ist beispielsweise ein Framework zur Erstellung eigener textueller DSL, *XText* gehört zum oAW-Generator Framework. Mit *XText* definierte Sprachen, und damit erzeugte textbasierte Modelle, lassen sich mit oAW verarbeiten.

In dieser Arbeit wird eine leichtgewichtige Erweiterung der UML für MDA-Light durchgeführt. Für MDSD soll eine eigene Domänen spezifische Modellierungssprache erstellt werden, die mit Hilfe von *Ecore* definiert wird, um sie dann mit EMF zu konstruieren.

3.3 Domänenspezifische Modelle

„Domänenspezifische Modellierung ist die abstrahierende Darstellung von Sachverhalten aus einem Problemraum unter Verwendung einer Modellierungssprache, welche auf den Problemraum zugeschnitten ist.“ ([Piotrek et al. 07], S. 56).

Um domänenspezifische Modelle erstellen zu können braucht man also eine DSL. Durch den Einsatz einer DSL lässt sich die Effizienz der Modellierung und der Auswertung der Modelle steigern. Die besondere Herausforderung dabei ist die Definition einer DSL, mit Unterstützung von Domänen-Experten. Die Verwendung von DSL in der Informatik ist nicht erst seit modellgetriebener Software-Entwicklung ein Thema: so wurde bereits in der 50er Jahren eine Sprache für Fertigungsmaschinen namens „Automatically Programmed Tools“ definiert. Ein weiteres weit verbreitetes Beispiel ist die SQL, als Datenbanksprache für relationale Datenbanken (nach [Piotrek et al. 07], S. 57). DSL sind also nicht nur Modellierungs-DSL, sondern auch Programmierungs-DSL. Nachfolgend werden für diese Unterscheidung die Begriffe *Domain Specific Programming Language* (DSPL) und *Domain Specific Modelling Language* (DSML) benutzt. Bei DSPL ist die Domäne eine technische. Fachliche Domänen sind z.B. die Versicherung- oder Finanzbranche. Die Unterscheidung zwischen technischen und fachlichen Domänen wird unter anderem auch in ([Stahl/Völter 05], S.66) gemacht. Meines Erachtens nach ist es lediglich eine Sichtweise. Man könnte auch sagen, dass die sogenannten technischen Domänen fachliche Domänen in einem technischen Fach sind. Die OMG selbst schränkt Domänen auf fachliche Domänen ein (nach [MDA 03]).

In der MDSD haben DSML insofern eine große Bedeutung, da mit ihnen die Modelle erstellt werden, aus welchen der Code generiert wird. Ausdrücke in DSML sollen gleichermaßen von Mensch und Maschine verstanden werden. Auf menschlicher Seite können Sachverhalte in Modellen effizient dargestellt werden; zur besseren Kommunikation und als Basis für die Maschine. Die Modelle stehen dann Compilern/Interpretern bzw. Generatoren zur weiteren Verarbeitung zur Verfügung. Die Vorteile einer DSML bzw. domänenspezifischer Modelle sind (nach [Piotrek et al. 07], S. 59 f):

- Spezifikationen lassen sich schneller beschreiben und steigern die Effizienz während der Analyse.
- Änderungsanforderungen lassen sich schneller beschreiben und steigern die Effizienz der Wartung.
- Die DSL hat eine eindeutige Semantik. Daher weniger Interpretationsspielraum, der zu Missverständnissen führen kann.
- Die Erstellung von Generatoren/Interpretern ist einfacher, da sie für eine spezifische Domäne sind.
- Software-Qualität und Vertrauen in Software steigt, durch geringere Fehlerwahrscheinlichkeit während der M2T Transformation.

Bei der Definition einer geeigneten DSML, werden in den meisten Fällen die IT und die Fachabteilung beteiligt sein. Ob die DSML technisch oder fachlich motiviert sein soll, hängt von der Art der zu entwickelnden Anwendung ab. Wenn eine langlebige Anwendung, unabhängig von wechselnden Plattformen oder mehreren Anwendungen, in derselben Domäne entwickelt werden soll, dann bietet sich eine fachliche DSML an. Wenn aber verschiedene Anwendungen auf derselben technischen Plattform entstehen sollen, dann bietet sich eine technische DSML an.

In dieser Arbeit wird für die Entwicklung mit dem schwergewichtigen MDSD-Ansatz eine eigene propietäre DSML für den Problemraum Mahnverfahren für Versicherungsanwendungen erstellt. Im Gegensatz dazu wird für MDA-Light das UML-Metamodell um die benötigten domänenspezifischen Elemente erweitert, und somit auch eine Art DSML erstellt.

3.4 Modellvalidierung

Das Modellieren soll in der MDSD größtenteils das Programmieren ersetzen. Aber auch diese Disziplin ist fehleranfällig. Bei komplexen Problemstellungen, denen komplexe Modelle folgen, sind Fehler kaum vermeidbar. Fehler im Modell werden oft erst im generierten Code festgestellt. Um solche Fehler zu vermeiden, sollte man die erstellten Modelle prüfen bzw. validieren. Eine Form der Modellvalidierung sind sogenannte Reviews mit Modellierungsexperten (nach [Piotrek et al. 07], S.68). Experten kennen sich mit Modellierungsrichtlinien aus und können gemeinsam die Modelle überprüfen. Diese Form ist jedoch zeitaufwendig und teuer, was klare Nachteile sind.

([Piotrek et al. 07], S.67) sind der Meinung, dass man die Modellvalidierung in den Prozess des Generierens integrieren sollte. Dann können während des Generierens automatisch Fehler im Modell festgestellt werden. Die Fehler im Modell könnten behoben werden und würden sich nicht auf den generierten Code auswirken. Dazu braucht man automatisierte Formen der Modellvalidierung. OCL ist eine formale Sprache, die eine automatisierte Modellvalidierung bietet. Dazu werden Contraints/Regeln für Modellelemente definiert, die später überprüft werden können. Wenn man beispielsweise im Metamodell eine Klasse mit bestimmten Eigenschaften erzeugt hat, kann man diese Eigenschaften in OCL definieren und später überprüfen. Unter Eclipse gibt es Tools zur Modellvalidierung, die alle auf OCL basieren. Abbildung 3.4 zeigt wie Modellvalidierung die in den Entwicklungsprozess integriert werden sollte.

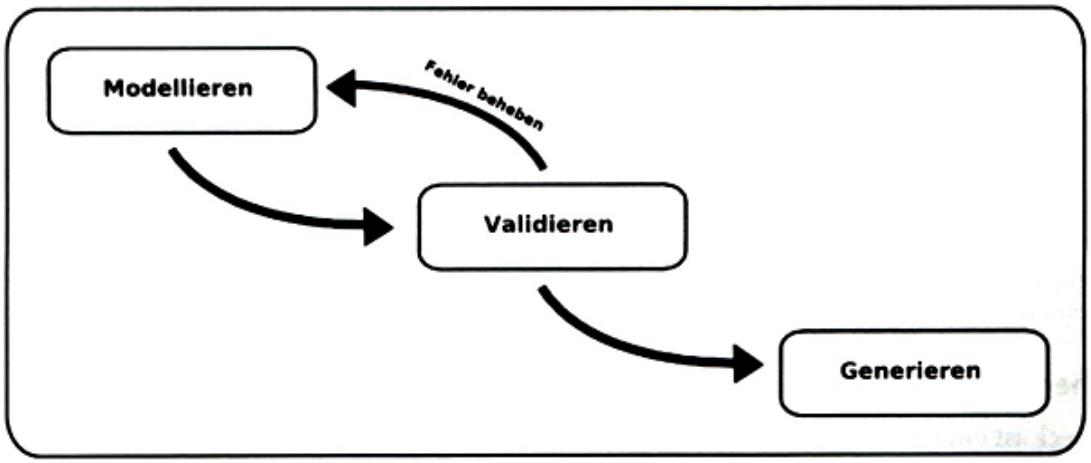

Abbildung 3.4: Modellvalidierung im Entwicklungprozess ([Piotrek et al. 07], S.67 Abb.3.11)

Man kann beim Validieren der Modelle aber nur semantische Korrektheit und Vollständigkeit prüfen. Ob die Geschäftslogik die in den Modellen repräsentiert wird Sinnvoll ist, lässt sich nicht automatisiert prüfen. Dafür sind die oben genannten Reviews mit Experten nötig. Ein schlechtes Design lässt sich also eben so wenig bei der modellgetriebenen Software-Entwicklung vermeiden, wie bei der modellbasierten Software-Entwicklung.

4 Mahnverfahren

Um beide Ansätze modellgetriebener Software-Entwicklung miteinander vergleichen zu können, wird dieselbe Anwendung mit beiden Ansätzen entwickelt. Als Anwendung soll das gerichtliche Mahnverfahren eines Versicherungsunternehmens erstellt werden. In diesem Kapitel wird alles Wissenswerte zum Thema Mahnverfahren erläutert und die Leistung der Anwendung abgegrenzt. Die Informationen über das Mahnverfahren wurden weitestgehend sinngemäß aus [GMV 08a] übernommen.

4.1 Vorgerichtliches Mahnverfahren (VMV)

Zum vorgerichtlichen Mahnverfahren zählt jede Handlung zwischen Schuldner und Gläubiger, die vor dem Antrag auf Mahnbescheid stattfindet. Wenn ein Schuldner mit Zahlungen in Verzug gerät, kann man als Gläubiger eine Mahnung verschicken, oder diese durch ein Inkassobüro oder einen Rechtsanwalt verschicken lassen. In Verzug gerät man als Schuldner, wenn man einen kalendermäßig bestimmten Leistungspunkt verstreichen läßt, die Leistung ernsthaft und endgültig verweigert oder durch Zustellung einer Mahnung. Die Mahnung ist also eine Möglichkeit den Schuldner in den Status des Verzugs zu versetzen. Durch die Reform des Schuldrechts hat sich eine wichtige Änderung ergeben: Der Schuldner kommt nämlich in jedem Fall spätestens 30 Tage nach Fälligkeit und Zugang einer Rechnung oder einer gleichwertigen Zahlungsaufstellung in Verzug (§ 286 Absatz 3 BGB). Diese Frist kann auch verkürzt werden, eben durch eine Mahnung oder einen kalendermäßig bestimmten Zeitpunkt. Für Verbraucher gilt, dass der Gläubiger in der Rechnung oder Zahlungsaufstellung auf die Verzugsfolgen hingewiesen werden muss, sonst gilt die 30-Tages-Frist nicht. Der Verzug ist allerdings keine Voraussetzung für einen Mahnbescheid, hierfür reicht die Fälligkeit. Der Verzug wird zum Berechnen der Verzugszinsen benötigt, welche auch in einem Mahnbescheid geltend gemacht werden können. Wie viele Mahnungen man verschickt bzw. verschicken lässt, bevor man einen Mahnantrag beim Gericht stellt, bleibt jedem Gläubiger selbst überlassen. Auch Ratenzahlungen und Vergleiche sind der Kulanz des Gläubigers überlassen.

4.2 Gerichtliches Mahnverfahren (GMV)

Sollte im Zuge des VMV keine Einigung zwischen Gläubiger und Schuldner zustande gekommen sein, hat der Gläubiger die Möglichkeit des gerichtlichen Mahnverfahrens. Das gerichtliche Mahnverfahren kann nach § 688 Absatz 1 der Zivilprozessordnung (ZPO) nur für die Ansprüche des Gläubigers durchgeführt werden, welche die Zahlung einer Geldsumme in Euro zum Gegenstand haben. Wer z.B. die Lieferung von Ware durchsetzen will, kann nicht auf das gerichtliche Mahnverfahren setzen und muss den Klageweg beschreiten. Der Zahlungsanspruch muss fällig sein. Die Fälligkeit einer Zahlung bestimmt sich nach § 271 des Bürgerlichen Gesetzbuches (BGB); sie ist grundsätzlich sofort fällig, wenn nicht eine Zahlungsfrist oder ein Zahlungstermin vereinbart wurde, wie es bei Rechnungen oft der Fall ist. Hierzu sollte man erwähnen, dass eine Mahnung nicht Voraussetzung eines gerichtlichen Mahnverfahrens ist.

Einen Mahnantrag stellt man beim zuständigen Gericht. Die zuständigen Gerichte kann man z.B. in [Maschinelles GMV 09] nachschlagen. In der Regel sind das die örtlich zuständigen Amtsgerichte, die in diesem Fall Mahngericht genannt werden. Der Antrag darf also nur in zugelassenen Formen beim Mahngericht gestellt werden. Je nach Bundesland kann dies wahlweise erfolgen:

- in Papierform (Vordrucke)
- durch mündliche Erklärung zu Protokoll der Geschäftsstelle (§§ 702 Absatz 1, 129a Zivilprozessordnung)
- durch Übermittlung des Antrages auf einem Datenträger (Diskette)
- durch Übermittlung des Antrages mittels DFÜ-Verbindung
- per Internet

Bei einem schriftlichem Antrag sind bestimmte Formulare zu verwenden, wie § 703c Absatz 2 der Zivilprozessordnung (ZPO) bestimmt. Die Formulare sind im Schreibwarenhandel erhältlich. In manchen Bundesländern wird ein maschinelles Antragsverfahren durchgeführt. Dafür muss immer ein bestimmtes, maschinell lesbares Formular verwendet werden, das ebenfalls im Schreibwarenhandel erhältlich ist. Wird nicht das entsprechende Formular verwendet, wird der Antrag vom Mahngericht zurückgewiesen.

In den zentralen Mahngerichten besteht die Möglichkeit, die Anträge elektronisch zu übermitteln. Dafür wird eine Datenübermittlung per Datenträgeraustausch (DTA) mittels Disketten oder per Datenfernübertragung (DFÜ) zugelassen. Die elektronische Datenübermittlung ist mittlerweile in fast allen Bundesländern möglich, wobei in einigen Bundesländern einzelne Übertragungsarten (DFÜ oder DTA) ausgeschlossen sind. Formulare sind dabei überflüssig, hierfür benötigt der Antragsteller lediglich eine bestimmte Software und muss sich vorher beim entsprechenden Mahngericht registrieren lassen. Vorteil der elektronischen Datenübermittlung ist, dass das Mahngericht den beantragten Mahnbescheid schneller bearbeiten und an den Schuldner versenden kann. In der Regel vergehen zwischen Beantragung und Versendung des Mahnbescheids an den Schuldner nur drei bis fünf Werktage, während auf dem herkömmlichen schriftlichen Weg oft zwei bis sechs Wochen vergehen. Außerdem sind elektronische Anträge weniger fehleranfällig als Anträge in Papierform. Derzeit werden etwa 70 Prozent aller Mahnanträge elektronisch übermittelt.

Durch die Entwicklung des "E-Government", also bei der elektronischen, internetgestützten Verwaltung, kann der Antrag mittlerweile in allen Bundesländern auch per Internet gestellt werden. Die meisten Bundesländer setzen beim Online-Mahnverfahren auf zwei Systeme:

- Online-Mahnantrag [Online-Mahn 09]
- Profimahn [Profimahn 09], das durch das EGVP (Elektronisches Gerichts- und Verwaltungspostfach) ersetzt werden soll [EGPV 09]

Online-Mahnantrag ist ein interaktives Antragsformular, das relativ einfach bei der Antragstellung hilft. Profimahn und EGVP ist dagegen für Antragsteller gedacht, die häufig Mahnanträge stellen und auch bisher schon mittels Datenträgeraustausch elektronisch ihre Daten übermittelt haben.

Damit das Mahngericht den Antrag annimmt, muss dieser korrekt und vollständig ausgefüllt sein. Dazu ist in [GMV 08a] der Inhalt eines Mahnantrages ausführlich beschrieben. Im Folgenden soll dieser Inhalt kurz dargestellt werden:

- Bezeichnung der Parteien mit vollständiger Adresse, also Gläubiger und Schuldner
- Bezeichnung des Mahngerichts, bei dem der Antrag gestellt wird
- Bezeichnung des Anspruchs. Art, Grund und Umfang muss klar erkennbar sein, damit der Schuldner entscheiden kann, ob er sich dagegen zur Wehr setzt. Deshalb sollte Gegenstand, Datum des Vertrags oder Vorgangs, eine typische Anspruchsbezeichnung, z. B. Kaufpreis, (Typenbezeichnung), Rechnung (Nr. XXX) mit Datum der genaue Geldbetrag, der gefordert wird angegeben werden
- Erklärung, dass der Anspruch nicht von einer Gegenleistung abhängt
- Bezeichnung des Gerichts, welches bei einem Verfahren zuständig wäre, falls der Schuldner Widerspruch einlegt
- Unterschrift bei schriftlichem, elektronische Signatur bei einem Elektronischen Antrag

Falls der Antrag, unabhängig von der Form, unvollständig oder falsch eingereicht wurde, bekommt man vom Mahngericht ein Monierungsschreiben in schriftlicher Form. Das Verfahren wird erst fortgesetzt, wenn das Monierungsschreiben vollständig beantwortet wurde. Dafür wird vom Mahngericht einen Vordruck für die Monierungsantwort mit übersendet (nach [Maschinelles GMV 09], S.23 ff).

Damit das Mahngericht ein Mahnbescheid erlassen kann, braucht es also einen fehlerfreien und vollständigen Antrag und einen Kostenvorschuss in Form der Gerichtsgebühren, die der Gläubiger vorstrecken muss. Dieser richtet sich nach dem Streitwert, er kann in ([Maschinellen GMV 09], S.83) nachgeschlagen werden. Diese Kosten und gegebenenfalls andere Kosten, wie z.B. Anwaltskosten können in den Mahnbescheid aufgenommen werden, und sind vom Schuldner zu erstatten.

Wenn alle Voraussetzungen erfüllt sind, wird der Mahnbescheid erlassen und dem Antragsgegner zugestellt. Über diese Zustellung wird der Antragsteller schriftlich benachrichtigt. Sollte der Mahnbescheid nicht zugestellt werden können, wird der Antragsteller ebenfalls benachrichtigt. Er bekommt eine Nichtzustellungsnachricht mit dem Grund der Nichtzustellung sowie einem Antrag auf Neuzustellung eines Mahnbescheids, der schon teilweise ausgefüllt wurde, z.B. mit der Geschäftsnummer des ursprünglichen Mahnbescheids. Die Verwendung des Vordrucks ist zwingend bei einer Nichtzustellung.

Nachdem der Mahnbescheid dem Schuldner zugestellt wurde, hat dieser nun zwei Wochen Zeit, die Forderung zu begleichen oder einen Widerspruch einzulegen. Ein Widerspruchsvordruck wird mit dem Mahnbescheid zugestellt. Wird der Widerspruch eingelegt, kann das Mahnverfahren als Zivilprozess weitergeführt werden. Vorausgesetzt, eine der Parteien hat die Durchführung des streitigen Verfahrens beantragt. Das kann der Antragsteller bereits im Voraus mit dem Mahnantrag tun, für den Fall des Widerspruchs.

Sollte der Antragsgegner keinen Widerspruch einlegen und auch die Forderung nicht begleichen, kann der Antragsteller nun einen Antrag auf Erlass eines Vollstreckungsbescheides stellen, der ohne weiteres vollstreckbar ist. Beim maschinellen Verfahren ist auch ein maschineller Vordruck zu verwenden, beim schriftlichen ein schriftlicher Vordruck. Außerdem dürfen seit Zustellung des Mahnbescheids an den Schuldner keine sechs Monate verstreichen.

Genau wie der Mahnbescheid, wird auch der Vollstreckungsbescheid per Post zugestellt und der Antragsteller davon benachrichtigt. Sollte der Vollstreckungsbescheid nicht zugestellt werden können, wird genauso verfahren wie beim Mahnbescheid; der Antragsteller wird benachrichtigt und hat die Möglichkeit, einen Neuantrag des Vollstreckungsbescheid zu stellen. Ebenso gibt es auch für den Vollstreckungsbescheid die Möglichkeit des Widerspruches binnen zwei Wochen für den Antragsgegner.

Erst nachdem der Vollstreckungsbescheid erlassen und nicht widersprochen wurde, bekommt der Antragsteller einen vollstreckbaren Vollstreckungsbescheid zugesendet. Mit diesem vollstreckbaren Bescheid kann der Gläubiger nun Vollstreckungsmaßnahmen ergreifen. Der Gläubiger kann nun z.B. den zuständigen Gerichtsvollzieher mit der Zwangsvollstreckung des Schuldners beauftragen. Dafür genügt es, den Vollstreckungsbescheid mit der Bitte um Zwangsvollstreckung an die Gerichtsvollzieherverteilungsstelle des zuständigen Vollstreckungsgerichts zu senden.

In Abbildung 4.1 wird der gesamte Ablauf des maschinellen gerichtlichen Mahnverfahrens in Form eines Dokumentenflussdiagramms dargestellt.

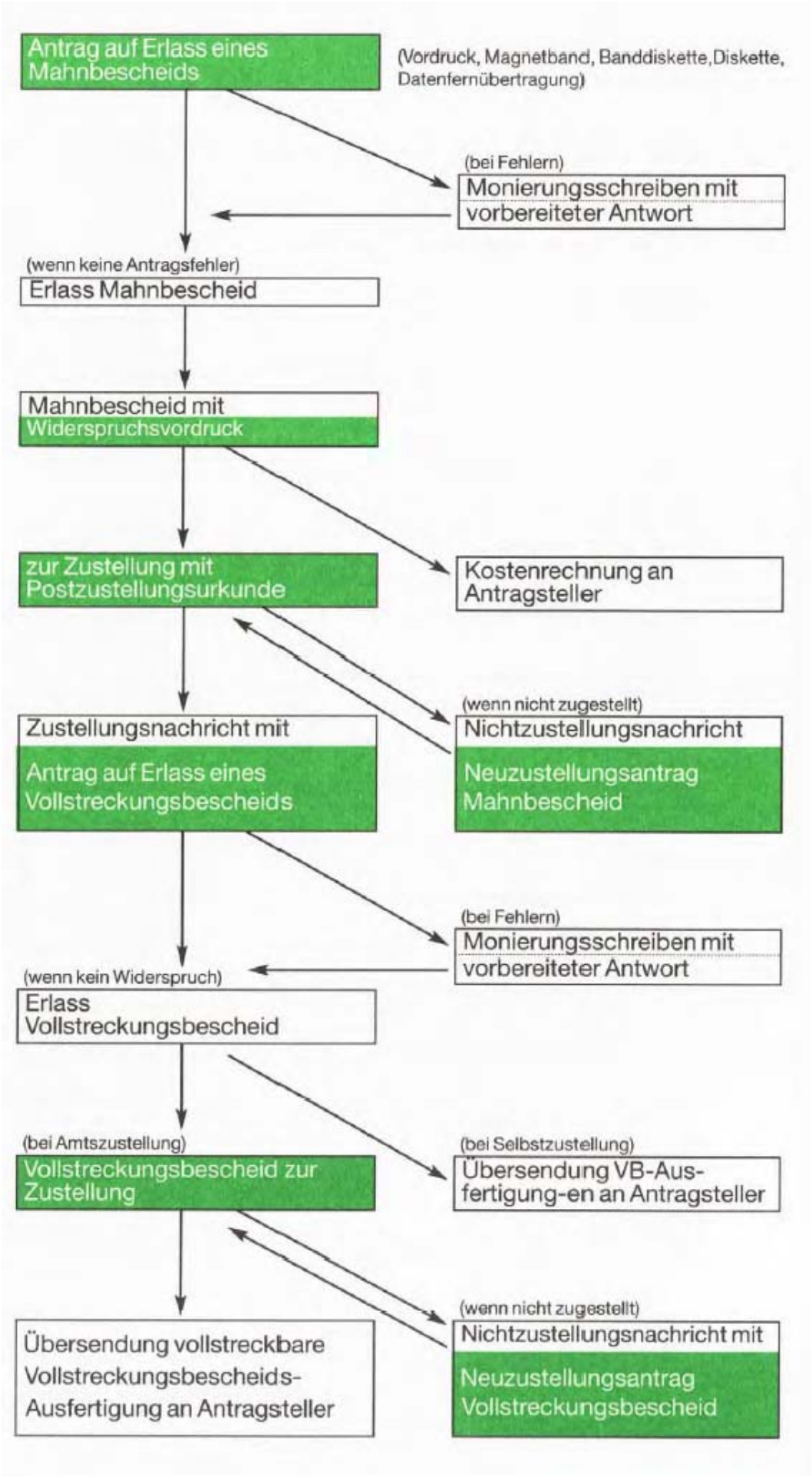

Abbildung 4.1: Verfahrensablauf maschinelles gerichtliches Mahnverfahren ([Maschinelles GMV 09], S. 85, Anhang 3)

Will man auch Verzugszinsen geltend machen, sollte man zuerst eine Mahnung an den Schuldner schicken, um das genaue Datum zur Berechnung der Zinsen zu ermitteln, siehe Abschnitt 4.1. Abbildung 4.2 zeigt den Ablauf beim Widerspruch des Antragsgegners.

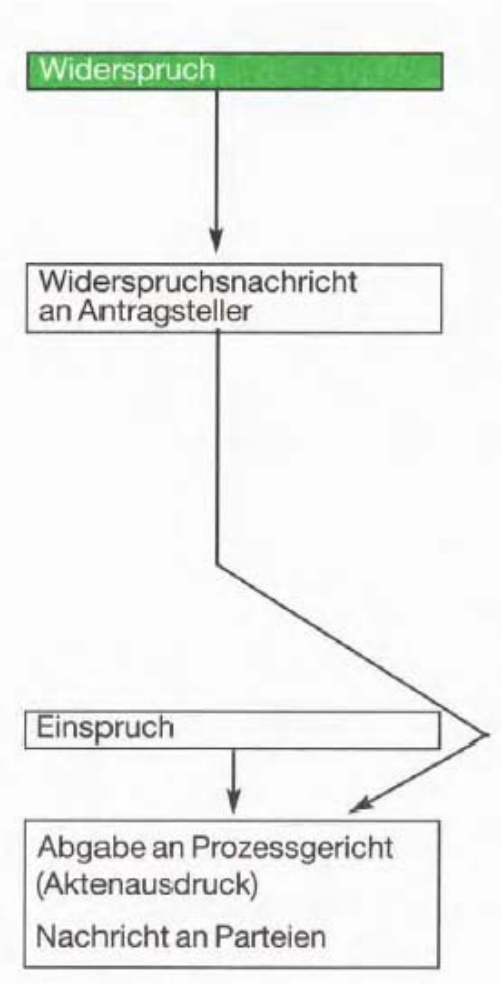

Abbildung 4.2: Verfahrensablauf bei Widerspruch ([maschinelles GMV 09], S.85 Anhang 3)

4.3 Anwendung

Als Anwendung wird nur das GMV für eine Versicherung erstellt. Dafür werden alle automatisierbaren Vorgänge von der Anwendung ausgeführt. Die Vorgänge, welche manuelles Eingreifen von Sachbearbeitern voraussetzen, sollen simuliert werden. Dazu werden Meldungen auf dem Bildschirm ausgegeben, und die Anwendung wird automatisch weiterarbeiten. Die Schnittstelle zum Gericht und das Verhalten des Schuldners wird ebenfalls simuliert und durch Bildschirmausgaben dokumentiert. Das VMV wird bewusst nicht implementiert, da es bei der Erstellung der Anwendung nicht um die Anwendung selbst geht, sondern um die verschiedenen Ansätze, mit denen entwickelt wird. In der Tat sind auch VMV und GMV bei den Versicherungen getrennte Anwendungen, die oft unabhängig von einander realisiert und betrieben werden. Die Anwendung soll also die automatisierbaren Aktivitäten eines realen GMV-Systems realisieren. Die Aktivitäten einer solchen Anwendung, die nicht automatisierbar sind, sollen simulieren werden. Die Klassifizierung der Aktivitäten einer GMV-Anwendung folgt in Abschnitt 5.1 dieser Arbeit.

5 Implementierung

In diesem Kapitel wird die Implementierung der beiden Methoden genau beschrieben. Dazu wird in Abschnitt 5.1 zuerst ein CIM für die Anforderungen an die Anwendung erstellt. Dieses CIM gilt für beide Vorgehensweisen gleichermaßen, da aus den dort erstellten Modellen kein Code generiert werden soll. In Abschnitt 5.2 werden das Generator-Framework oAW und weitere Werkzeuge vorgestellt, die für die Anwendung verwendet werden. Abschnitt 5.3 beinhaltet die Implementierung der leichtgewichtigen Methode. Abschnitt 5.4 die der schwergewichtigen.

5.1 Computation Independent Model (CIM)

Ein CIM skizziert die Anforderungen der Anwendung umgangssprachlich oder unter Verwendung von beliebigen Diagrammarten. Für eine Versicherungsanwendung ist es von großer Bedeutung, Daten dauerhaft speichern zu können. Aus diesem Grund wird die Nutzung einer Datenbank Voraussetzung für diese Anwendung. Üblicherweise sind die Daten der Kunden einer Versicherung dauerhaft abrufbar. Es wird wohl keine Versicherung geben, die Ihre Kundendaten nicht bereits in Datenbanken speichert. Da die Anwendung dieser Arbeit aber nicht auf eine Datenbank einer Versicherung zugreifen kann, müssen die Stammdaten in einer selbst aufgesetzten Datenbank gespeichert werden. Zudem gibt weitere Daten die dauerhaft abrufbaren sein sollen. Zu diesen Daten gehören: die an das Gericht verschickten Anträge, die vom Gericht empfangenen Nachrichten, der Zustand indem ein Mahnfall sich gerade befindet und eine Historie die den Ablauf des Mahnfalls dokumentiert. Die Anwendung soll in jedem Fall eine Datenverwaltung von Stammdaten und operativen Daten ermöglichen. Um dauerhaft abrufbare Elemente dieser Anwendung zu kennzeichnen wird der Begriff Entität verwendet. Weiter werden Entitäten für diese Anwendung so definiert, dass Sie nur die Datenobjekte repräsentieren und die Kommunikation mit einer passenden Datenbanktabelle ermöglichen. Die Spalten der Datenbanktabelle werden durch Variablen innerhalb einer Entität repräsentiert. Dafür gibt es zwei Arten von Feldern: Schlüsselfelder und normale Felder. Entitäten sollen nicht die Geschäftslogik implementieren. Die Artefakte, die für die Geschäftslogik zuständig sind, werden Geschäftsobjekte genannt. Zum Starten der Anwendung wird zudem mindestens ein Prozess benötigt, der in einer realen Versicherungsanwendung immer zyklisch aufgerufen wird. Diese Artefakte werden hier Geschäftsprozesse genannt. Die Aktivitäten der Artefakte, welche Geschäftslogik implementieren, werden in technische, fachliche und simulierte Aktivitäten gruppiert.

Um das Verhalten der Anwendung für jeden einzelnen Mahnfall steuern zu können, werden die Mahnereignisse, die eintreten können, in einem Aufzählungstypen definiert. Diese Mahnereignisse werden aus der Beschreibung des GMV, den Abbildungen 4.1, 4.2 und meinem Verständnis für Datenbankanwendungen hergeleitet. Darüber hinaus werden auch Mahnaktivitäten, die auf Mahnereignisse folgen, sowie ein Zustand eines Mahnfalls der daraufhin eintritt, in Aufzählungstypen definiert. Zum besseren Verständnis der Zusammenhänge von Mahnereignissen, Mahnaktivitäten und des Zustandes eines Mahnfalls wird ein Mapping in Tabellenform erstellt. Tabelle 5.1 stellt dieses Mapping dar.

Tabelle 5.1: Mappping von Mahnereignissen über Mahnaktivitäten zu GMV-Zuständen

Mahnereignis	**Mahnaktivität**	**GMV-Zustand**
GMV gestartet	Mahnfall erstellen MB erstellen	MB erstellt
MB erstellt	MB abschicken	MB beantragt
MB beantragt	Nachricht abwarten und auswerten	MB erlassen **oder** MB moniert
MB erlassen	Nachricht abwarten und auswerten	MB widersprochen **oder** MB nicht widersprochen **oder** MB nicht zustellbar
MB moniert	MB Monierung klären	MB Monierung geklärt
MB Monierung geklärt	MB erstellen	MB erstellt
MB nicht zustellbar	Adressermittlung	Adresse ermittelt **oder** Adresse nicht ermittelbar
MB widersprochen	Prozessverfahren starten	GMV beendet
MB nicht widersprochen	VB erstellen	VB erstellt
VB erstellt	VB abschicken	VB beantragt
VB beantragt	Nachricht abwarten und auswerten	VB erlassen **oder** VB moniert
VB erlassen	Nachricht abwarten und auswerten	VB widersprochen **oder** VB nicht widersprochen **oder** VB nicht zustellbar
VB moniert	VB Monierung klären	VB Monierung geklärt
VB Monierung geklärt	VB erstellen	VB erstellt
VB nicht zustallbar	Adressermittlung	Adresse ermittelt **oder** Adresse nicht ermittelbar
VB widersprochen	Prozessverfahren starten	Prozessverfahren start
VB nicht widersprochen	Vollstreckung starten	Vollstreckung start
Schuldner bezahlt	GMV beenden	GMV beendet
Adresse nicht ermittelbar	GMV beenden mit Wiedervorlage	Wiedervorlage
Adresse ermittelt	MB erstellen **oder** VB erstellen	MB erstellt **oder** VB erstellt
Vollstreckung gestartet	GMV beenden	GMV beendet
Prozessverfahren gestartet	GMV beenden	GMV beendet

Die Abkürzung MB steht für Mahnbescheid, VB für Vollstreckungsbescheid. Ein Mahnereignis aus der linken Spalte löst direkt die Mahnaktivität der gleichen Zeile, in der Spalte rechts daneben, aus. Nachdem diese Mahnaktivität ausgeführt wurde, wechselt der Zustand des Mahnfalls in einen GMV-Zustand der gleichen Zeile, in der Spalte rechts daneben. Der Wechsel in einen anderen Zustand kann gleichzeitig auch wieder ein

Mahnereignis sein. Zum besseren Verständnis für das Verhalten der Anwendung dient das UML-Aktivitätsdiagramm aus Abbildung 5.1.

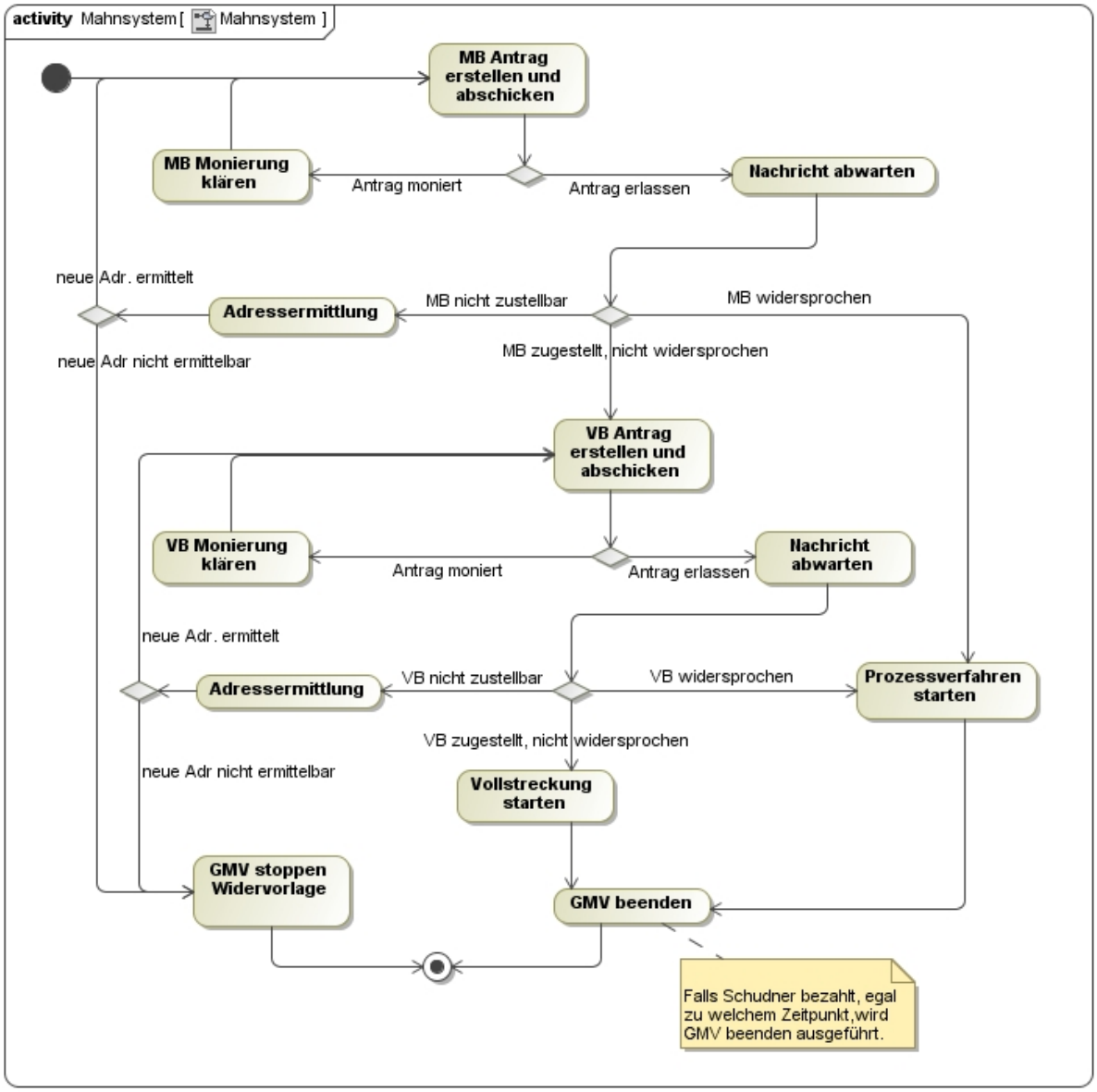

Abbildung 5.1: Aktivitätsdiagramm für das Verhalten der Anwendung bei einem Mahnfall

Zusammenfassend kann man sagen, dass die Anwendung neben einer Datenverwaltung auch den Ablauf vieler Mahnfälle steuern und dokumentieren soll.

5.2 Tools

Dieser Abschnitt stellt kurz die verwendeten Tools vor, mit welchen die Implementierung realisiert wird. Es werden ausschließlich freie Tools verwendet, die ebenfalls auf der CD mit den Quellen mitgeliefert werden.

5.2.1 Eclipse

Eclipse ist nicht nur eine frei verfügbare Entwicklungsumgebung für Java-Projekte, sondern vielmehr auch ein durch Plugins erweiterbares Werkzeug. Prinzipiell steuert Eclipse nur den Ablauf von Plugins, durch das Plugin-Manifest *plugin.xml*. Die verwendeten Funktionalitäten werden von den Plugins zur Verfügung gestellt. Man kann unter [Eclipse 09] verschiedene vorgefertigte Versionen herunterladen; hier sind bereits eine Vielzahl von Plugins installiert. In dieser Arbeit wird die aktuelle Version 3.4.2 benutzt, die vorgefertigt als *Eclipse Modelling Tools* installiert wird. Diese Version enthält eine ganze Reihe von Plugins, welche für die Implementierung benötigt werden; wie z.B. das EMF und das GMF Plugin. Um modellgetrieben entwickeln zu können, wird Eclipse um das oAW-Plugin in der Version 4.3.1 erweitert. Da großer Wert darauf gelegt wird, nicht all zu viele verschiede Tools zu benutzten und mit auszuliefern, wird Eclipse auch um ein UML-Tool und ein Tool zur Bereitstellung von Datenbank-Funktionalitäten erweitert. Zum Modellieren mit UML wird TOPCASED in der Version 1.0.3 installiert. Um sich zur Kontrolle der Anwendung die dauerhaften Daten direkt unter Eclipse anschauen zu können, wird ein Plugin für die Datenbank installiert. Eclipse wird darüber hinaus auch als Laufzeitumgebung genutzt. Die speziell an die Anforderungen dieser Arbeit erstellte Eclipse Version wird auf der CD mit ausgeliefert. Eine Anleitung zur Nutzung dieser befindet sich im Anhang.

5.2.2 openArchitectureWare (oAW)

oAW ist eine Open-Source-Lösung zum Bau von MDA/MDSD-Generatoren, die in Java implementiert ist. Ursprünglich war oAW ein kommerzielles Produkt namens *b+m Generator Framework*, welches 2003 als oAW3 bei SourceForge frei verfügbar wurde. Ab Version 4 ist oAW zu einem Unterprojekt bei Eclipse geworden und gewann dadurch zunehmend an Bedeutung in Großprojekten bzw. im akademischen Bereich (nach [Piotrek/Trompeter 07], S.192).

Bei oAW wird der Arbeitsprozess des Generators durch den sogenannten Workflow beschrieben und gestartet. Der Workflow wird im XML-Format erstellt und in einer .oaw-Datei gespeichert. Im einzelnen werden Konfigurationen wie z.B. das benutzte Metamodell, das erstellte Anwendungsmodell und das auszuführende Root-Template beschrieben. Gestartet wird der Generator direkt aus der Eclipse-Umgebung mit “RUN AS | OAW WORKFLOW”.
Die Templates für die M2T-Transformationen werden in der eigenen Templatesprache *xPand* geschrieben. XPand-Dateien werden mit der Endung .xpt gespeichert. Als Ausgabe werden Textdateien mit beliebiger Endung erzeugt. Bei oAW 4.x kommt xPand2 zum Einsatz. xPand2 verfügt über eine kleine Anzahl von Schlüsselwörtern die von << >> eingeschlossen sind. Tabelle 5.2 fasst den benutzten Sprachumfang von xPand2 zusammen (nach [Piotrek et al. 07], S.195).

Tabelle 5.2: Benutzten xPand2 Sprachumfang

<<IMPORT *Namespace* >>	Importiert einen Namespace um Elemente aus dem Modell nicht mit voll qualifizierten Namen Nutzen zu müssen.
<<EXTENSION *Qualifier.Name* >>	Deklaration einer Extension, die Funktionen zur Verfügung stellt.
<<DEFINE *Name* FOR *Metaklasse* >>	Definition eines Templates, bestehen aus Name, optionaler Parameterliste und Name der Metaklasse, wofür die Definition gilt.

<<EXPAND *DefName* FOR/FOREACH *Ausdruck*>>	Expandiert einen DEFINE-Block im gleichen Template oder einem neuen Template. Der Ausdruck steuert für welches Modellelement der DEFINE-Block expandiert wird.
<<FILE *Ausdruck* [*Ausgabepfad*] >> ...Inhalt... <<ENDFILE>>	Leitet die Ausgabe in eine Datei.
<<FOREACH *Ausdruck* AS *Var* [ITERATOR *ItName*] [SEPERATOR *SepString*]>> ...Inhalt... <<ENDFOREACH>>	Expandiert den Inhalt für jeden Ausdruck. Das aktuelle Modellelement wird in Var repräsentiert.
<<IF *Ausdruck*>> ...Inhalt... <<ELSE IF *Ausdruck*>> ...Inhalt... <<ELSE>> ...Inhalt... <<ENDIF>>	Expandiert den Inhalt anhängig vom Ausdruck.
<<PROTECT CSTART *KommentarBeginn* CEND *KommentarEnde* ID *id* [DISABLE]>> ...Inhalt... <<ENDPROTECT>>	Kennzeichnung eines geschützten Bereichs. Der Inhalt wird bei neuerlichen Generierung nicht überschrieben. Nach CSTART und CEND müssen die Zeichenketten für Kommentar Beginn und Kommentar Ende der Zielsprache stehen.
<<LET *Ausdruck* AS *Varname* >> ...Inhalt... <<ENDLET>>	Definition einer Variablen, die im Innern des Blocks an Varname gebunden ist.
<<REM>> Kommentar <<ENDREM>>	Kommentarblock.

Wie man aus Tabelle 5.2 entnehmen kann, ist der Umfang von xPand2 begrenzt. Mit xTend kann der Sprachumfang des oAW-Generators funktional erweitert werden. xTend benutzt die auf Java basierende oAW Expression Language zur Beschreibung von Funktionen. Die Dateiendung von xTend-Dateien ist .ext. Funktionen werden wie folgt definiert:

```
ReturnType extensionName(paramType1 paramName 1, paramType2 ... ) :
Expression-using-params ;
```

Mit Check stellt oAW eine eigene oAW-Sprache zur Beschreibung von Constrains zur Verfügung, die ebenfalls in der oAW Expression Language verfasst wird. Die verfassten Dateien mit der Endung .chk werden in OCL umgewandelt um Modelle validieren zu können. In dieser Arbeit wird oAW Version 4.3.1 für Modelltransformationen genutzt. Sie ist als Plugin unter Eclipse auf der CD mit enthalten.

5.2.3 IBM DB2

Als Datenbank wird IBM DB2 Version 9.5 Express installiert, welche unter [DB2 09] frei verfügbar ist. DB2 wird als Datenbank genutzt, da ich bereits damit gearbeitet habe und somit auf meinen Wissen in diesem Bereich zurückgreifen kann. Eine ausführbare Installationsversion wird auf der CD mit übergeben.

5.2.4 TOPCASED

Toolkit in Open source for Critical Applications & SystEms Development (TOPCASED) ist eine Open-Source-Werkzeugsammlung zum Entwickeln kritischer Anwendungen (nach [Piotrek et al. 07], S.66). TOPCASED ist ein Eclipse Projekt und verfügt über eine Vielzahl Editoren. Man kann damit sowohl UML2, als auch EMF und GMF Modelle modellieren. Es gibt zwar Tools die Ansehnlichere Modelle erzeugen, aber um nicht mit Verschiedenen Tools modellieren zu müssen, habe ich mich für dieses Tool entschieden. In dieser Arbeit wird TOPCASED Version 1.0.3 benutzt, und wird ebenfalls als Plugin unter Eclipse auf der beigelegten CD mit ausgeliefert.

5.3 Leichtgewichtige Methode

In diesem Abschnitt wird die komplette Implementierung der leichtgewichtigen Methode dargestellt. Es wird empfohlen sich, in Anlehnung an diesen Abschnitt, die Quellen auf der CD, unter dem Projekt MDA-Light, anzusehen, da hier nur teilweise Quellcode zum besseren Verständnis gezeigt wird.

5.3.1 Metamodell

Für das Metamodell dieser Vorgehensweise wird das UML Metamodell mit leichtgewichtiger Metamodellierung erweitert. Dazu wird mit TOPCASED ein neues UML Profil erstellt, welches zuerst nur das UML Metamodell beinhaltet. Das Profil wird *gmv* benannt. Anschließend wird das Profil um die Elemente erweitert, die man zum Generieren der Anwendung benötigt. Die benötigten Elemente werden im CIM beschrieben. Es werden nachfolgend die Stereotypen *Entität*, *Geschäftsobjekt*, *Geschäftsprozess*, *Schlüssel*, *Feld*, *Dummy*, *Fachl* und *Techn* erzeugt. Die Stereotypen *Entität*, *Geschäftsobjekt* und *Geschäftsprozess* erweitern das UML-Element *uml::Class*. Sie sollen später im Modell die verschiedenen Arten von Klassen repräsentieren, die dann während des Generierens unterschiedlich behandelt werden. Die Stereotypen *Schlüssel* und *Feld* erweitern das Element *uml::Property* und stehen für die Attribute einer Klasse. Die Stereotypen *Dummy*, *Fachl* und *Techn* erweitern *uml::Operation* und repräsentieren die verschiedenen Arten von Methoden. Die in 5.1 erklärten Aufzählungstypen werden mit *uml::Enumeration* realisiert. Man kann in TOPCASED das Profil grafisch, oder über einen sogenannten Baumeditor, erweitern. Ein Baumeditor zeigt eine Hierarchie der Elemente in Form von strukturierten Texten. Diese Art von Editoren wird häufig unter Eclipse genutzt zum Editieren von Dateien, die auf XML basieren. Mir gefällt es besser mit dem Baumeditor über das Kontextmenü zu arbeiten, da es einfacher ist, die erstellten Stereotypen einem UML-Element zuzuweisen. Dazu markiert man den erstellten Stereotypen, wählt den Menüpunkt „UML Editor | Stereotyp | Create Extension“ aus, dann wählt man ein UML-Element aus der Auswahl. Das UML-Profil wird also über einen Baumeditor in Form von strukturiertem Text erstellt. TOPCASED bietet auch die Möglichkeit, aus dem in strukturierten Texten erstellten Metamodell, ein Diagramm zu generieren. Abbildung 5.1 zeigt ein solch generiertes Klassendiagramm aus dem hier dargestellten Metamodell.

Mit Hilfe von Stereotypen werden Klassen, Methoden und Attribute unterschiedlichen Kategorien zugeordnet. Die Stereotypen selbst werden durch Attribute, auch *TaggedValues*, Eigenschaften oder benutzerdefinierte Eigenschaften genannt, charakterisiert. Mit diesen Eigenschaften kann man einzelnen Elementen aus dem Modell spezifisches Verhalten zuweisen. Der Stereotyp *Entität* enthält zwei solche Eigenschaften vom Typ String; namens *tabname* und *beschreibung*. In *tabname* steht der Tabellenname der Datenbanktabelle, falls der Tabellenname abweichend vom Namen der Entität selbst ist. Der Stereotyp *Feld* enthält

die Attribute *spaltenname*, *länge*, *nullzulässig* und *beschreibung*. In *spatenname* steht der Namen der Spalte für die Datenbank, den das Feld repräsentiert, falls sich der Name von dem Namen des Feldes unterscheidet. Für String-Felder steht in *länge* ein Integer-Wert, der angibt, wie lang die Zeichenkette werden kann. *Nullzulässig* ist ein Boolean-Wert, der angibt, ob die Spalte in der Datenbank nullable ist. Standardmäßig sind die String-Felder leer, die Länge ist auf null und *nullzulässig* auf false gesetzt. Der Stereotyp *Schlüssel* erbt von *Feld* und besitzt keine eigene Eigenschaften, da über den Namen des Stereotypen in den Transformationsregeln besser unterschieden werden kann, ob es sich um ein Schlüsselfeld oder ein normales Feld handelt. Die Stereotypen *Geschäftsobjekt*, *Geschäftsprozess*, *Dummy*, *Techn* und *FachI* enthalten jeweils nur die Eigenschaft *beschreibung*. Daraus kann eine Dokumentation generiert werden. Über Inhalte, innerhalb dieser Zeichenkette, kann man auch die Generierung manipulieren.

Da oft erst beim Codieren der fachlichen Methoden gemerkt wird, dass man noch weitere Sprachmittel benötigt, ist MDSD eine iterative Vorgehensweise. In dieser Arbeit werden beide Vorgehensweisen auch iterativ angewandt. Zum Beispiel werden zu den Elementen aus dem CIM iterativ weitere Aufzählungstypen hinzugefügt. Dabei handelt es sich um nachfolgende Typen:

- *Versicherungsart*, beinhaltet die verschiedenen Versicherungsarten
- *Antragsart*, beinhaltet die verschiedenen Antragsarten
- *Zahlungsart*, beinhaltet die verschiedenen Zahlungsarten
- *AntragsZustand*, beinhaltet die verschiedenen Zustände, in welchen ein Antrag intern im GMV sein kann
- *NachrichtsZustand*, beinhaltet die verschiedenen Zustände, in welchen eine Nachricht intern im GMV sein kann

In der Abbildung 5.2 wird das Profil dargestellt. Die Stereotypen sind in der oberen Reihe gruppiert und mit einem S in dem kleinen Kästchen vor dem Namen markiert. Die Aufzählungstypen sind unten angeordnet und mit einem E, welches für Enumeration steht, markiert. Die Namen der Stereotypen und Eigenschaften enthalten keine Umlaute, da der Generator Elemente mit Umlauten im Namen nicht erkennen kann.

Abbildung 5.2: gmv-Metamodell

5.3.2 Anwendungsmodell

Die Standard-Sprachmittel der UML und die Erweiterungen in dem Profil, worin das Metamodell aus 5.3.1 definiert wird, dienen als Basis für das Anwendungsmodell. Wenn man mit UML modelliert, sind diese UML-Sprachelemente immer verfügbar, man kann sie nicht ausblenden. Das hat einige Vor- und Nachteile. Diese werden in Kapitel 6 erläutert.

Das Modell trägt den Namen *Mahnsystem*. In der Abbildung 5.3 ist das Anwendungsmodell *Mahnsystem* dargestellt. Für die Stammdaten der Versicherung werden drei Entitäten modelliert: *Versicherungsnehmer*, *Vertrag* und *Konto*. Die Entitäten *Versicherungsnehmer*, *Vertrag* und *Konto* würden nicht benötigt, wenn die Anwendung auf einer bestehenden Versicherungsanwendung aufsetzen würde. Dann würde man lediglich eine Schnittstelle benötigen, welche die Daten dieser drei Entitäten zur Verfügung stellt.

Eine Entität steht sowohl für eine Zeile in der Datenbank als auch für das Java-Objekt, welches Kommunikation zur Datenbank bereitstellt und die Daten kapselt. UML bietet standardmäßig primitive Datentypen, wie Date oder String, welche weitestgehend mit der Vielzahl von Programmiersprachen übereinstimmen. Das wird hier erwähnt, da ein Unterschied zum Modellieren mit einer eigenen DSL besteht. Im weiteren Verlauf dieses Abschnittes wird nicht mehr darauf hingewiesen, wenn ein Datentyp aus dem Sprachraum der UML stammt, lediglich wenn ein selbst erstellter Datentyp benutz wird.

Die Entität *Versicherungsnehmer* enthält die persönlichen Daten eines Kunden. Diese Daten werden in den Feldern *name*, *vorname*, *geburtsdatum*, *wohnort*, *plz* und *strasse* gespeichert. Das Feld *geburtsdatum* ist vom Typ Date. Die restlichen Felder sind vom Typ String. Die Entität *Vertrag* enthält die Vertragsdaten bestehend aus den Feldern *versicherungsart*, *zahlungsart* und *zahlungsbetrag*. Das Feld *versicherungsart* ist vom Typ *Versicherungsart*, der im Profil selbst definiert wurde. Das Feld *zahlungsart* ist vom Typ *Zahlungsart*, ebenfalls aus dem Profil. Das Feld *zahlungsbetrag* ist eine Gleitkommazahl, also vom Typ *Float*.

Darüber hinaus besteht eine Beziehung vom Typ Composite zwischen den Entitäten *Versicherungsnehmer* und *Vertrag*. Das bedeutet, ein *Vertrag* kann nur in Verbindung mit einem *Versicherungsnehmer* existieren. Wie eine Komposition programmtechnisch aufgelöst wird, ist in Abschnitt 5.3.3 beschrieben. Die Multiplizität der Beziehung ist 0 zu n. Das bedeutet, ein Versicherungsnehmer kann keinen bis beliebig viele Verträge haben. Weiter besteht zwischen *Konto* und *Vertrag* eine Komposition mit der Multiplizität 1 zu n. Ein *Vertrag* muss also mindestens ein *Konto* haben. In der Regel wird er aber eine Vielzahl an Konten haben. Ein Konto repräsentiert eine fällige Zahlung zum Vertrag. Die Entität *Konto* besteht aus den Feldern *zahlungsdatum* vom Typ *Date* und *gmv_Starten* vom Typ Boolean. In einer Inkassoanwendung würde Konto noch mehr Felder benötigen, aber für ein Mahnsystem reichen diese beiden Felder aus. Desweiteren besitzen diese und alle weiteren Entitäten noch ein Schlüsselfeld *oid* vom Typ Integer. Das Feld *oid* steht für Objektidentität und dient als eindeutige Kennzeichnung jeder Entität.

Die Entität *Mahnfall* enthält *beginn_Datum* vom Typ Date, *ende_Datum* vom Typ Date, *mahnbetrag* vom Typ Float, *gerichts kennzeichen* vom Typ String und den aktuellen Status eines Mahnfalls. Das Feld *status* ist vom Typ *gmvZustand* aus dem erstellten UML-Profil. Ein Mahnfall kann nur in Verbindung mit einem Versicherungsnehmer bestehen, also gibt es eine Komposition zwischen den beiden Klassen. Ein Versicherungsnehmer kann keinen bis beliebig viele Mahnfälle haben. Ein Mahnereignis besteht nur mit einem Mahnfall, deshalb besteht hier wieder eine Komposition mit der Multiplizität 1 zu n. Ein Mahnereignis enthält die Felder *datum* vom Typ Date und *mahnereignis* vom selbst erstellten Aufzählungstypen *Mahnereignis*. In der Tabelle, die durch diese Entität repräsentiert wird, soll die Historie des Mahnverfahrens gespeichert werden.

Jede Art von Kommunikation mit dem Gericht wird als Dokument in Datenbanktabellen gespeichert. Jedes Dokument enthält die persönlichen Daten eines Versicherungsnehmers, sowie das aktuelle Datum in einem Date gespeichert, das eindeutige Gerichtskennzeichen in einer String- und *unser_zeichen* in einer Integer-Variablen. Das Feld *unser_zeichen* ist immer die Objektidentität des Mahnfalls. Mögliche Kommunikation mit dem Gericht können ausgehende Anträge oder ankommende Nachrichten sein. Deshalb erben die Entitäten *Antrag* und *Nachricht* aus der Entität *Dokument*. Die Tabellen *Antrag* und *Nachricht* dienen gleichzeitig als Historie der Dokumente und als Stapel der ein- und ausgehenden Dokumente. Die Klasse *Nachricht* erweitert die Klasse *Dokument* um die Felder *nachrichtsart* vom selbst erstellten Typ *Mahnereignis*, und *status* vom ebenfalls selbst erstellten Typ *nachrZustand*. Die Klasse *Antrag* wird erst um das Feld *status* und *Antragsart* von den eigenen Typen *antrZustand* und *Antragsart* erweitert. Weiter wird *Antrag* um alle noch ausstehenden Informationen, die ein Antrag enthalten muss, erweitert (nach [Maschinelles GMV 09], S.67 ff). Dabei handelt es sich um Zeichenketten, die Informationen über das Mahngericht, das Amtsgericht, den Antragsteller und den Grund des Antrags enthalten.

Die Geschäftslogik der Anwendung wird in den Geschäftsobjekten und Geschäftsprozessen realisiert. Ein Geschäftsobjekt *Mahnfall* steuert das Verhalten der Anwendung für einen Mahnfall. Es enthält Entitäten vom Typ *Mahnfall* und *Mahnereignis*, über welche die Daten mit der Datenbank abgeglichen werden. Darüber hinaus enthält es eine Reihe von Methoden, welche die Mahnaktivitäten aus Abschnitt 5.1 implementieren. Nachfolgend eine vereinfachte Auflistung der Methoden:

- *Mahnfall_erstellen*
- *MB_erstellen*
- *MB_beantragt_setzen*
- *MB_Monierung_klären*
- *MB_erlassen_setzen*
- *VB_erstellen*
- *VB_beantragt_setzen*
- *VB_Monierung_klären*
- *VB_erlassen_setzen*
- *Nachricht auswerten*
- *Adressermittlung*
- *Vollstreckung_starten*
- *Prozessverfahren_starten*
- *Schuldner bezahlt*
- *GMV beenden*

Die beiden Methoden *Vollstreckung_starten* und *Prozessverfahren_starten* sind vom Stereotypen Dummy. Das bedeutet, sie simulieren eine Aktivität und dokumentieren sie per Bildschirmausgabe. Die restlichen Methoden sind alle vom Stereotyp *fachl*; sie üben also fachliche Aktivitäten aus. Die Parameter und Funktionen der Methoden werden ausführlich in Abschnitt 5.3.4, bei den individuellen Implementierungen, erläutert.

Das Geschäftsobjekt *Gerichtsschnittstelle* repräsentiert die Schnittstelle zum Gericht. Es besteht aus den technischen Methoden: *empfang_Dokument* und *verschick_Dokument*. Und aus den Dummy-Methoden *simulier_Gericht* und *simulier_Schuldner*. Die Parameter und Aktivitäten werden ebenfalls in Abschnitt 5.3.4 erläutert.

Außerdem enthält das Modell noch einen Geschäftsprozess *Mahnsystem* mit den fachlichen Methoden *verarb_eink_Dokumente*, *verarb_ausg_Dokumente* und *erstell_neue_Mahnfälle*. Genau genommen sollten diese drei Methoden selbst Prozesse sein. In dieser Arbeit werden sie als Methoden realisiert, um das Testen der Anwendung zu vereinfachen und das Modell nicht allzu kompliziert aussehen zu lassen. Die genauen Aktivitäten des Geschäftsprozesses werden auch bei den individuellen Implementierungen dargestellt. Abbildung 5.3 zeigt das Anwendungsmodell.

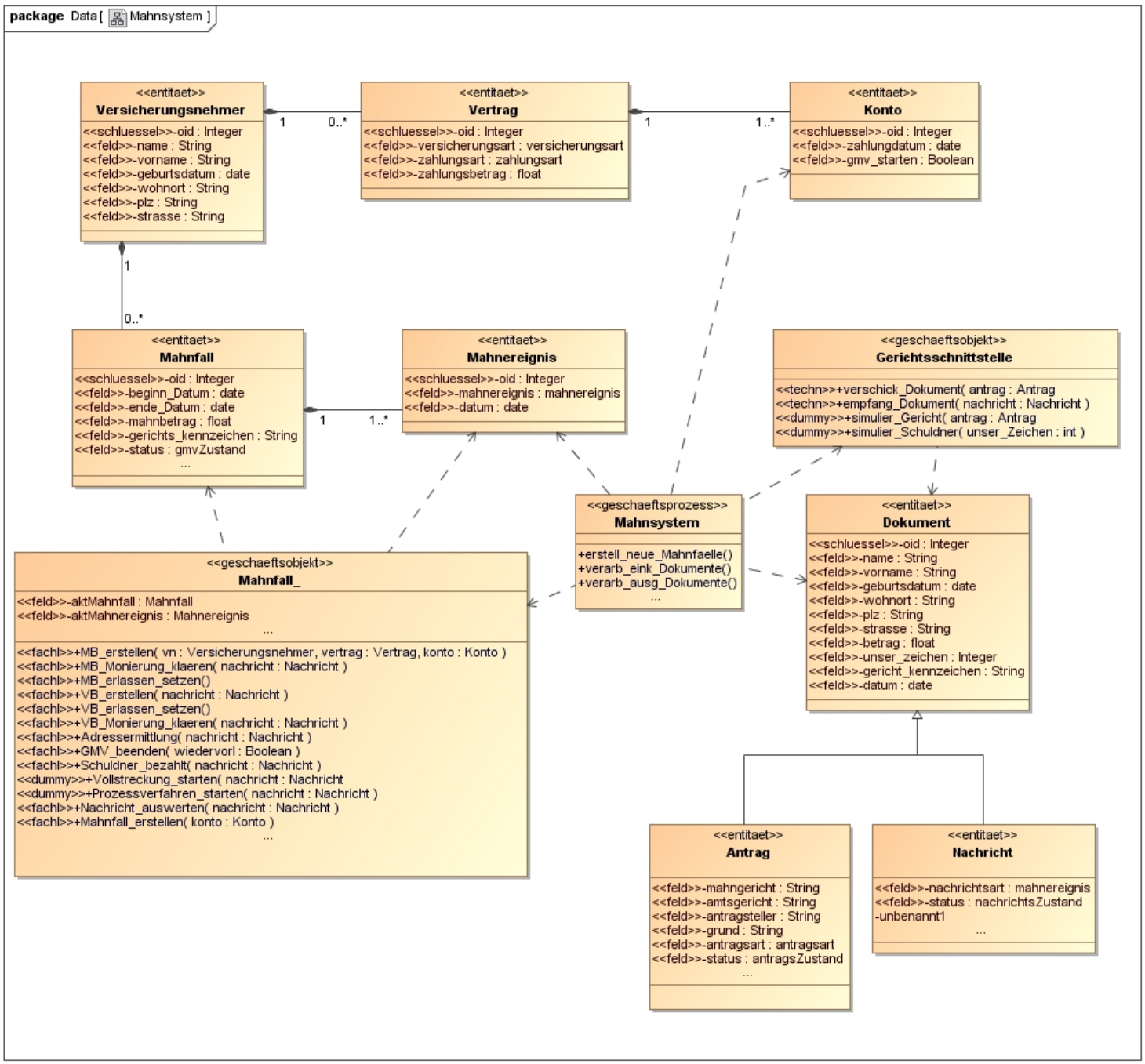

Abbildung 5.3: Anwendungsmodell mit UML modelliert.

Die gestrichelten Linien zeigen Aufruf-Beziehungen der Objekte. Das Geschäftsobjekt *Mahnfall* erzeugt beispielsweise Entitäten von *Mahnfall* und *Mahnereignis*. Der Geschäftsprozess *Mahnsystem* erzeugt Mahnereignisse, benutzt Dokumente, Konten, die Gerichtsschnittstelle und das Geschäftsobjekt *Mahnfall*.

Angereichert wird das Modell, indem die Attribute der Stereotypen gefüllt werden. In großen Unternehmen wie Versicherungen bestehen oft Namenskonventionen für Dateinamen, Klassennamen usw. Aus diesem Grund werden die Tabellennamen für Entitäten mit dem Präfix *Db* versehen. Der Tabellenname für Konto heißt demnach *DbKonto*. Alle Entitäten werden auf dieser Weise angereichert. Weiter wird in allen String-Feldern der Wert für die maximale Länge der Zeichenkette gesetzt. Die Eigenschaften *spaltenname* und *nullzulässig* werden in nur einem Feld exemplarisch geändert. Die Eigenschaft *spaltenname* bei dem Feld *nachrichtsart* der Entität *Nachricht* wird auf *Nachricht_von_Gericht* gesetzt. Die

Eigenschaft *nullzulässig* bei dem Feld *amtsgericht* der Entität *Antrag* wird auf true gesetzt. Um während der Testphase besser den zeitlichen Ablauf nachvollziehen zu können, sollen einige Felder, die als Date deklariert sind, in der Datenbank als Timestamp deklariert werden. Betroffen sind die Felder *beginn_Datum* und *ende_Datum* aus *Mahnfall*, *datum* aus *Mahnereignis* und *datum* aus *Dokument*. Im Modell sollen sie aber weiterhin als *Date* deklariert bleiben. Für einen solchen Fall bedient man sich einer Eigenschaft, um die Generierung zu beeinflussen. Für diese Felder wird in der Eigenschaft *beschreibung* die Zeichenkette „TIMESTAMP“ hinzugefügt, um zu steuern, dass diese Felder für die Datenbank einen anderen Datentypen bekommen als für den Programmcode.

5.3.3 Modelltransformation

Bei der MDA-Light Vorgehensweise gibt es nur eine Transformation vom Modell zur PSI. Für diese Transformation wird ein oAW Generator konfiguriert. Dazu wird eine Workflow-Datei in XML erstellt. In ihr wird unter anderem beschrieben, welche Metamodelle benutzt werden und an welcher Stelle man sie findet. Weiter wird der Pfad angegeben, wo die generierten Artefakte hingeschrieben werden sollen und wo die geschützten Bereiche für individuelle Implementierungen liegen. Darüber hinaus wird das Anwendungsmodell und das xPand-Hauptskript angegeben. Als Metamodelle werden das UML2-Metamodell und das GMV-Profil angegeben. Listing 5.1 zeigt einen Codeausschnitt, der ein Metamodell und das Hauptskript *RootGMV* definiert.

```
<!-- UML meta model -->
<metaModel class="oaw.uml2.UML2MetaModel"/>

<!-- execute template -->
<expand value="templates::RootGMV::Root FOR model"/>
```

Listing 5.1: Workflow-Ausschnitt

Die benutzen Werkzeuge speichern die Modelle im XMI-Format, einem XML-Derivat. Ein Modell wird also in einer Baumstruktur in XML gespeichert. Der Wurzelknoten ist das Modell. Die darunterliegenden Knoten sind Pakete oder Klassen. Unter den Klassenknoten liegen Knoten für Methoden, Attributen, Assoziationen und weitere Daten bezüglich der Klasse. So kann man sich beim Generieren durch das Model arbeiten, indem man nacheinander die Knoten expandiert und zum Knoten passende Codeausschnitte in die Ausgabedatei schreibt. Das Modell Mahnsystem wurde als Paket *Mahnsystem* definiert. Das Hauptskript *RootGMV.xpt* expandiert direkt die Knoten, welche für die Klassen stehen (Entitäten, Geschäftsobjekte und Geschäftsprozesse) und ruft den dazugehörigen DEFINE-Block auf. Ein DEFINE-Block wird für ein Modellelement definiert, was auch ein Knoten in dem XMI-Baum ist. Diese Blöcke sind auf verschiedene Skripte verteilt, in welchen wiederum neue Skripte mit weiteren DEFINE-Blöcken aufgerufen werden können. Für eine Entität werden die Skripte *cr_java_dbh*, *cr_java_entity* und *cr_db2_table* aufgerufen. Für einen Geschäftsprozess wird das Skript *cr_java_gp.xpt* und für ein Geschätfsobjekt *cr_java_gp.xpt* aufgerufen.

Für Geschäftsprozesse wird jeweils nur eine Java-Klasse generiert. Geschäftsprozesse besitzen keine eigenen Daten, enthalten nur Methoden. In dem aufgerufenen Skript wird also nur der Knoten für den Geschäftsprozess und die Knoten für die Methoden expandiert. Das Skript enthält nur schematischen Code und Markierungen für den geschützten Bereich des individuellen Codes. Der schematische Code wird durch das Expandieren der Knoten des Modells erzeugt. Der individuelle Code wird per Hand, in der generierten Datei, in den geschützten Bereich geschrieben und somit bei jedem Generator Durchlauf wieder in

dieselbe Stelle eingefügt. Das Skript für Geschäftsprozesse wird genau erklärt, um die Arbeitsweise des Generators verständlicher zu machen. Der Code des Skripts ist in Listing 5.2 dargestellt. In Zeile 1 bis 3 steht ein einfacher Kommentar. Zeile 4 importiert das Metamodell, Zeile 5 die mit xTend erstellen Erweiterungen. In Zeile 7 wird der Knoten, welcher für den Geschäftsprozess steht, expandiert. Dadurch kann man innerhalb des DEFINE-Blocks (Zeile 7-60) auf alle modellierten Sprachelemente des Geschäftsprozesses zugreifen. Der Aufruf des DEFINE befindet sich im Hauptskript in Form eines EXPAND-Befehls. Zeile 9 definiert den Pfad und die Ausgabedatei, welche aus den Zeichen *Gp*, plus den Namen des Geschäftsobjektes besteht. Zeile 22 und 25 sind schematischer Code. Dieser Code ändert sich von Geschäftsprozess zu Geschäftsprozess und wird somit aus den Elementen des Modells gesteuert. Zeile 26 bis 37 ist ein statischer Code, der für alle Geschäftsprozesse gleich aussieht. Generell kann man sagen, dass alle Zeichen, die blau gefärbt sind in dem Listing, statischen, unveränderlichen Code darstellen. Diese Zeilen werden direkt in die Ausgabedatei geschrieben. In den Zeilen 18/19, 38/39 und 52 bis 54 werden geschützte Bereiche erzeugt. Individueller Code, in der Klasse des Geschäftsprozesses, wird in den modellierten Methoden, der Main-Methode und für benötigte Imports, erwartet. Jede Art von individuellen Änderungen außerhalb der geschützten Bereiche wird beim nächsten Generieren überschrieben. In den Zeilen 42 bis 56 werden die modellierten Methoden erzeugt. Die FOREACH Schleife legt in jedem Durchlauf ein UML-Element vom Typ *Operation* in dem Objekt *meth* ab. Somit sind die Modell-Informationen über die Methode in dem Objekt *meth* enthalten. Müssten die Operationen anhand des Stereotyps unterschieden werden, hätte man an dieser Stelle über EXPAND einen DEFINE Block aufrufen können, welcher nur für einen bestimmten Stereotyp gilt. Dann wären die Methodendaten über die *this* Variable innerhalb des Blocks verfügbar. In diesem Skript wird davon abgesehen, da dies ein einfacher Abschnitt ist, indem nur Methodenrümpfe und geschützte Bereiche generiert werden. Die Methoden in Geschäftsprozessen haben noch nicht einmal Parameter oder Rückgabewerte.

```
«REM»
Template für ein Geschäftsprozess.
«ENDREM»
«IMPORT meta::gmv»
«EXTENSION tools»

«DEFINE Class(String packageName) FOR gmv::geschaeftsprozess»

«FILE "../MDA-Light/src-gen/" + packageName + "/gp/Gp"+name+".java"-»
/* Gp«name».java
 * Generiert aus cr_java_gp.xpt
 * «getDate()»
 */
package «packageName».gp;
import  «packageName».enumeration.*;
import  «packageName».en.*;
import  «packageName».go.*;
«PROTECT CSTART "/*" CEND "*/" ID packageName+"-Go"+name+"-Imports"»
«ENDPROTECT»

/**
 * Geschäftsprozess «name».
 */
//-----------------------------------------------------------------------
public class Gp«name»
//-----------------------------------------------------------------------
{

   /**
    * main
    */
   //-----------------------------------------------------------------------
   public static void main(String[] args)
   //-----------------------------------------------------------------------
   {
«PROTECT CSTART "/*" CEND "*/" ID packageName+"-Gp"+this.name+"Main"»
«ENDPROTECT»
   }

«FOREACH this.getOperations() AS meth-»


   /**
    * «meth.name»
    */
   //-----------------------------------------------------------------------
   public static void «meth.name»()
   //-----------------------------------------------------------------------
   {
«PROTECT CSTART "/*" CEND "*/" ID packageName+"-Gp"+meth.getType()+"
 -Meth"+meth.name»
«ENDPROTECT»
   }
«ENDFOREACH»

}
«ENDFILE»
«ENDDEFINE»
```

Listing 5.2: Skript zum generieren eines Geschäftsprozesses

In dem Skript wird nur eine Extension benutzt. Die Extension *getDate*() wird in Zeile 12 aufgerufen. Sie ruft wiederum eine Funktion einer per Hand codierten Java-Klasse auf, die auch als Erweiterung des Generators dient. Diese Funktion liefert das aktuelle Datum mit Uhrzeit. Diese Information wird in jeder generierten Java-Klasse in einen Kommentarblock zu Beginn der Datei geschrieben, mit dem Namen des Skripts aus welchem generiert wurde. Nachfolgend wird die Funktion aus der xTend-Datei *tools*.*ext* dargestellt.

```
String getDate():
JAVA gmvTools.Tools.getToday();
```

Listing 5.3: Extension aus xTend-Datei

Das Skript zum Generieren von Geschäftsobjekten unterscheidet sich vom Skript für Geschäftsprozesse nur ein wenig. Es werden ebenfalls Methoden generiert, diesmal jedoch mit Parametern, falls welche modelliert wurden. Außerdem kann ein Geschäftsobjekt auch Daten haben. Dazu gibt es noch einen weiteren geschützten Bereich zum Ende der Klasse, falls weitere Methoden implementiert werden, die nicht modelliert wurden, oder so trivial sind, dass sie nicht im Modell sein sollen.

Für eine Klasse aus dem Modell, die als Entität gekennzeichnet ist, werden Java-Entitäten, Java-DB-Handler und eine DDL-Datei zum Erzeugen einer DB-Tabelle generiert.

Das Skript für die DDL-Datei generiert zu 100% schematischem Code. Individueller Code ist in diesen DDL nicht vorgesehen. Der Code dieser DDL wird in SQL verfasst und besteht aus dem Create-Table-Befehl. Der Name der Tabelle wird aus der Eigenschaft *tabname* entnommen. Ist an der Stelle kein Name eingetragen, wird der Name der Klasse als Tabellenname generiert. Darüber hinaus werden die Attribute der Klasse als Spalten der Tabelle eingetragen, je nach dem wie ihre Eigenschaften gesetzt sind. Dafür wird in dem Skript eine Extension aufgerufen, welche ein Mapping von UML-Datentypen zu DB2-Datentypen realisiert. Felder, die als Date deklariert sind, werden in Abhängigkeit von dem Eintrag im Feld *beschreibung* umgesetzt. Wenn die Beschreibung die Zeichenkette TIMESTAMP enthält, wird in der Datenbank der Typ Timestamp deklariert. Bei der Generierung des Programmcodes wird weiterhin Date erzeugt.

Für die Generierung von Geschäftsprozessen, Geschäftsobjekten und DDL wird keine Referenzimplementierung benötigt, um den Code zu analysieren. Der Code ist übersichtlich und kann ohne einen Prototyp abstrahiert werden. Für die Java-Entitäten und Java-DB-Handler ist es schon schwieriger, die verschiedenen Anteile von individuellen, generischen und schematischen Code zu analysieren. Aus diesem Grund wird zum Analysieren jeweils ein Prototyp einer Entität und eines DB-Handlers als Referenzimplementierung ausgewertet. Die Auswertung einer Referenzimplementierung erleichtert die Erstellung eines Skripts, welches aufwendige Artefakte generieren soll.

Die Auswertung der DB-Handler ergibt, dass die SQL-Code-Behandlung als generischer Code implementiert werden kann. Für die DB-Handler werden Zugriffsmethoden für die Standard-Zugriffe auf die jeweilige Tabelle generiert. Es sind die sogenannten CRUD-Methoden (*create*, *retrieve*, *update*, *delete*). Um Prepared-Sql im Java komfortabel zu erzeugen, werden noch weitere Methoden generiert, welche Zeichenketten aus den Attributen oder SQL-Strings zusammenstellen. Für individuelle Zugriffsmethoden der jeweiligen Tabellen gibt es geschützte Bereiche. Die DB-Handler sind gekapselt und nur in der zugehörigen Entität sichtbar.

Die Auswertung der Java-Entitäten ergibt keinen generischen Code. In Java-Entitäten werden die Attribute als private Variablen, die Getter- und Setter-Methoden für die Variablen und die Aufrufe der Zugriffsmethoden der DB-Handler erzeugt. Weiter haben Java-Entitäten geschützte Bereiche für individuelle Methoden. Zum Generieren der Variablen aus den modellierten UML-Attributen wird ein Prinzip genutzt, welches oft Verwendung in einem

Codegenerator findet. Dieses Prinzip wird kurz vorgestellt. Um einen syntaktisch korrekten Java-Code in einer Zeichenkette zu generieren, wird eine Extension aufgerufen. Zum Beispiel, um eine private Java-Variable zu erzeugten, übergibt man als Parameter der Extension den Namen des Attributs und das UML-Element, welches für den Typ dieses Attributs steht. Diese Extension selektiert dann über eine Switch-Anweisung den richtigen Typ aus und erzeugt für diesen Typ die richtige Zeichenkette. So wird gleichzeitig ein Mapping von UML-Typen zu Java-Typen gemacht und eine Zeichenkette mit dem Java-Code erstellt. Listing 5.4 zeigt diese Extension.

```
String getJavaAttrType(String name, uml::PrimitiveType typ):
      switch (typ.name) {
            case "String":  " private String           "+name
            case "Boolean": " private Boolean          "+name
            case "EDate":   " private Date             "+name
            case "Integer": " private int              "+name
            case "EFloat":  " private float            "+name
            default:        "Fehler: typ "+typ+" nicht gefunden!!!"
      };
```

Listing 5.4: Mapping von UML-Typ zu Java-Typ in einer Extension

Um Aufzählungstypen zu generieren, gibt es kein gesondertes Skript. Wird innerhalb des Skripts für Java-Entität ein Feld vom Datentyp *uml::Enumeration* gefunden, wird ein DEFINE-Block mit der UML-Enumeration als Parameter aufgerufen. In diesem Block wird dann ein Java Aufzählungstyp mit dazugehörigen Werten generiert.

Der Generator löst auch für Klassen des Stereotyps *Entität* die Beziehungen auf. Dies geschieht in den Skripten für Entitäten. Da im Modell nur Kompositionen und Vererbungen vorkommen, werden nur diese Beziehungen aufgelöst.

Bei der Komposition wird auf der Kind-Seite ein Fremdschlüssel erzeugt. Dieses Feld ist vom gleichen Typ wie das Schlüsselfeld der Entität, zu der die Beziehung besteht. Der Fremdschlüssel wird in den Artefakten Entität, DB-Handler und Datenbanktabelle generiert. In diesem Feld muss immer die Objektidentität des Vaterobjekts stehen. Darüber hinaus werden Gib-Methoden generiert, um die Objekte der anderen Seite der Beziehung zu bekommen. Sollte die Multiplizität der Beziehung größer als 1 sein, bekommt man eine Liste der Objekte, zu welchen eine Beziehung besteht.

Hierarchien werden so aufgelöst, dass die Superklasse nicht alleinstehend generiert wird, sondern in der erbenden Klasse aufgelöst wird. Generell gibt es folgende Arten der Abbildung von Vererbungen auf die relationale Datenbank (OR-Mapping):

- Jede Klasse alleine für sich (eigene DB-Tabelle und Java-Klasse)
- Alle Klassen zusammen (eine DB-Tabelle und eine Java-Klasse für die ganze Hierarchie)
- Superklassen werden in die unterste Subklasse aufgenommen (eine DB-Tabelle und eine Java-Klasse für jeweils einen Zweig der Hierarchie)

In der ersten Art müsste die Beziehung, wie bei der Komposition, über Objektidentitäten zur Laufzeit aufgelöst werden. In der zweiten Art bedarf die Auflösung der Beziehung keines Fremdschlüssels, dafür aber einer Objektart, welche alle Entitäten der Vererbungshierarchie vereint. Dabei würden auch überflüssige Felder in der Tabelle gehalten. Die dritte Art löst die Beziehung ebenfalls ohne den Fremdschlüssel. Es werden keine überflüssigen Felder mitgeführt und eine Objektart braucht man auch nicht. Diese Variante scheint die einfachste zu sein, deshalb wird sie realisiert. Für das Modell aus 5.3.2 bedeutet das, dass die Entität *Dokument* nicht generiert wird. Alle Felder aus Dokument werden in die Entitäten *Antrag* und *Nachricht* übernommen. Dies gilt für Datenbanktabelle, DB-Handler und Java-Entität.

Bei der Generierung der Methoden werden die Stereotypen nicht unterschieden. Die Stereotypen *Fachl*, *Techn*, und *Dummy* dienen lediglich zur Kennzeichnung der verschiedenartigen Methoden. In dieser Arbeit wird für die fachlichen Methoden der individuelle Code programmiert. Technische und Dummy Methoden würden in einer realen Anwendung weiter entwickelt werden.

Im Anhang A, befindet sich eine Erläuterung zu den Quellen auf der CD und wie man die Anwendung generiert.

5.3.4 Individuelle Implementierungen

In diesem Abschnitt wird der individuelle Codeanteil der Anwendung dargestellt. Der individuelle Codeanteil der Anwendung besteht aus der Geschäftslogik. Diese wird innerhalb der modellierten Methoden der Geschäftsprozesse und Geschäftsobjekte realisiert. Dazu werden die Methoden-Rümpfe mit geschützten Bereichen für individuelle Änderungen generiert. Innerhalb dieser geschützten Bereiche wird der individuelle Code der Anwendung implementiert. Zur Darstellung des individuellen Code wird jede fachliche Methode aus dem Modell mit ihren Aktivitäten kurz erläutert. Für den Geschäftsprozess *Mahnsystem* werden die modellierten und die Main-Methode implementiert. In der Main-Methode wird der Ablauf des Prozesses festgelegt. Dies geschieht über Aufrufe der modellierten Methoden. Zuerst werden alle neuen Mahnfälle erstellt, dann alle ausgehenden Dokumente verarbeitet, um abschließend alle ankommenden Dokumente zu verarbeiten. Die Methode *erstell_neue_mahnfälle* holt alle Objekte der Klasse *Konto*, in welchen das Feld *gmv_starten* gesetzt ist. Anschließend wird für jede dieser Objekte die Methode *Mahnfal_erstellen* des Geschäftsobjekts *Mahnfall* aufgerufen und das Objekt *Konto* übergeben. Die Methode *verarb_ausg_dokumente* holt alle Objekte der Klasse *Antrag*, in welchen der Status auf *erstellt* gesetzt ist. Das bedeutet, der Antrag wurde erstellt und ist bereit zum Verschicken. Anschließend wird die Methode *verschick_Dokument* des Geschäftsobjektes *Gerichtsschnittstelle* mit dem Objekt *Antrag* als Parameter aufgerufen. Die Methode *verarb_eink_Dokumente* holt alle Objekte der Klasse *Nachricht*, in welchen der Status auf *empfangen* gesetzt ist. Anschließend wird die Methode *Nachricht_auswerten* von Geschäftsobjekt *Mahnfall* mit der Nachricht als Parameter aufgerufen.

Das Geschäftsobjekt *Gerichtsschnittstelle* besteht aus vier modellierten Methoden. Dazu werden noch fünf weitere Methoden implementiert. Nachfolgend werden die Methoden des Geschäftsobjekts aufgelistet mit ihren Funktionalitäten und Parametern:

- *verschick_Dokument* bekommt als Parameter das Objekt *Antrag* und soll diesen verschicken. Da in dieser Arbeit keine reale Schnittstelle zu einem Gericht implementiert wird, wird an dieser Stelle die dummy Methode *simulier_Gericht* aufgerufen.
- *empfang_Dokument* bekommt als Parameter das Objekt *Nachricht* und schreibt diese in die Datenbank.
- *simulier_Gericht* bekommt als Parameter einen Antrag und erstellt daraus eine Nachricht. Um zu simulieren, ob der Antrag erlassen oder moniert wird, ruft man die Methode *Antwort_MB_beantragt* auf. Falls noch kein Gerichtskennzeichen eingetragen ist, wird die Methode *gerichtsKz_gen* aufgerufen. Anschließend wird *empfang_Dokument* aufgerufen.
- *simulier_Schuldner* bekommt als Parameter einen Antrag und erstellt daraus eine Nachricht. Um zu simulieren ob der Antrag zustallbar ist, ihm widersprochen oder nicht widersprochen wird, ruft man die Methode *antwort_MB_erlassen* auf. Anschließend wird *empfang_Dokument* aufgerufen.
- *antwort_MB_beantragt* simuliert eine mögliche Antwort des Gerichts auf einen Antrag auf MB. Die Wahrscheinlichkeit, dass der Antrag erlassen ist, wird auf 70% und dass er moniert ist, wird auf 30% festgelegt.
- *antwort_VB_beantragt* arbeitet wie *antwort_MB_beantragt*, nur für VB.

- *antwort_MB_erlassen* simuliert eine mögliche Antwort vom Schuldner auf einen erlassenen MB. Die Wahrscheinlichkeit, dass der Antrag nicht zustellbar ist, wird auf 10%, dass er zugestellt und nicht widersprochen ist, wird auf 30%, dass er zugestellt und widersprochen ist, wird auf 30% und die Wahrscheinlichkeit das der Schuldner zu diesem Zeitpunkt bezahlt hat, wird auf 20% festgelegt.
- *antwort_VB_erlassen* arbeitet wie *antwort_MB_erlassen*, nur für VB.
- *gerichtsKz_gen* generiert einen zufälligen String, der 16 Zeichen lang ist, als Gerichtskennzeichen.

Das Geschäftsobjekt *Mahnfall* dient als zentrale Klasse zur Verarbeitung einzelner Mahnfälle. Dazu hat es zwei Felder; *aktMahnfall* und *aktMahnereignis,* mit welchen es die Entitäten *Mahnfall* und *Mahnereignis* verwaltet. Nachfolgend werden die Methoden des Geschäftsobjekts mit ihren Funktionalitäten und Parametern aufgelistet:

- *Mahnfall_erstellen* bekommt als Parameter die Entität Konto, für die ein Mahnfall erstellt werden soll. Über diese Entität werden die Entitäten Vertrag und Versicherungsnehmer geholt, zu denen eine Beziehung besteht. Aus den Daten der Entitäten wird eine Entität Mahnfall erstellt und in die Datenbank geschrieben. Abschließend wird *MB_erstellen* aufgerufen.
- *MB_erstellen* bekommt als Parameter Entitäten Versicherungsnehmer, Vertrag und Konto, für die ein Antrag auf Mahnbescheid erstellt werden soll. Aus den Daten der Entitäten wird eine Entität Antrag erstellt und in die Datenbank geschrieben.
- *MB_erlassen_setzen* ändert den Status den Mahnfalls auf MB erlassen und ruft *Simulier_Schuldner* in der Gerichtsschnittstelle auf.
- *MB_Monierung_klären* bekommt eine Entität Nachricht übergeben. Aus dieser Nachricht wird wieder ein neuer Antrag erstellt und in die Datenbank geschrieben.
- *VB_erstellen* bekommt eine Nachricht übergeben. Aus dieser wird ein Antrag auf VB erstellt und in die Datenbank geschrieben.
- *VB_erlassen_setzen* wie *MB_erlassen_setzen*, nur für VB.
- *VB_Monierung_klären* wie *MB_Monierung_klären*, nur für VB.
- *Nachricht_auswerten* bekommt eine Nahricht übergeben und wertet diese aus. Je nach Zustand des Mahnfalls und Art der Nachricht, wird der Mahnfall weiter verarbeitet durch einen Aufruf der passenden Methode.
- *Adressermittlung* bekommt eine Nachricht übergeben. Prüft ob eine neue Adresse ermittelbar ist durch Aufruf der Methode *Adr_ermitteltbar*. Wenn eine neue Adresse ermittelbar ist, dann wird aus der Nachricht ein neuer Antrag erstellt und in die Datenbank geschrieben. Wenn keine neue Adresse ermittelbar ist, wird *GMV_beenden* aufgerufen mit Wiedervorlage.
- *Schuldner_bezahlt* bekommt eine Nachricht übergeben und ruft *GMV_beenden* auf, ohne Wiedervorlage.
- *Prozessverfahren_starten* bekommt eine Nachricht übergeben und ruft *GMV_beenden* auf, ohne Wiedervorlage.
- *Vollstreckung_starten* bekommt eine Nachricht übergeben und ruft *GMV_beenden* auf, ohne Wiedervorlage.
- *GMV_beenden* bekommt eine *true* oder *false* übergeben, je nachdem ob der Mahnfall auf Wiedervorlage gelegt werden soll.
- *GoMahnfall* bekommt ein Integer-Wert übergeben. Dieser Konstruktor lädt den Mahnfall mit übergebener Objektidentität in das Feld *aktMahnfall*.
- *Antrag_Erweiterbar* bekommt eine Entität Versicherungsnehmer übergeben. Prüft durch Aufruf der Methode *antragErweiterbar* aus der Entität Antrag, ob man einen Antrag erweitern kann.
- *MB_erweitern* bekommt Entitäten von Versicherungsnehmer, Vertrag, Konto und die OID des erweiterbaren Antrags übergeben. Erweitert den Antrag um ein weiteres offenes Konto.
- *Adr_Ermittelbar* berechnet eine mögliche Antwort eines Adressermittlungsverfahrens über festgelegte Wahrscheinlichkeiten. Die Wahrscheinlichkeit das die Adresse

ermittelbar ist wird auf 70%, die Wahrscheinlich das sie nicht ermittelbar ist auf 30% festgelegt.

Darüber hinaus tragen die Methoden, welche durch Mahnereignisse ausgelöst werden diese Mahnereignisse in die Datenbank ein und ändern den Status des Mahnfalls, wenn nötig. Für die Entitäten Konto, Antrag und Nachricht werden ebenfalls individuelle Methoden benötigt.

Für Konto wird die Methode *GibNeueMahnfälle* implementiert, welche einen Vektor zurückgibt der mit allen Konten gefüllt, in denen das Flag *gmv_Starten* gesetzt wird. Dazu wird im DB-Handler ein Methode realisiert für den Zugriff auf die Datenbank. Für Nachricht wird die Methode *GibEmpfangeneNachrichten* implementiert welche einen Vektor zurück gibt, der mit allen empfangenen Nachrichten gefüllt ist, die noch nicht verarbeitet sind. Dafür wird wieder eine Zugriffsmethode im DB-Handler realisiert. Für Antrag werden drei individuelle Methoden, für Zugriffe auf die Datenbank, benötigt. *GibErstelltAnträge* gibt einen Vektor zurück, der mit allen Anträgen gefüllt ist, die erstellt aber noch nicht verschickt sind. *AntragErweiterbar* prüft, ob auf der Datenbank bereits ein Antrag zur als Parameter übergebenen Entität Versicherungsnehmer liegt, der noch nicht abgeschickt ist. Diese Methode wird benötigt um mehrere offenen Konten in einem Antrag zusammenzufassen. *LeseMitUnserZeichen* liest einen Antrag auf der Datenbank, über das Feld *unser_Zeichen* und gibt diesen zurück.

In der DB-Handler Schicht wird noch eine Klasse *DbConnection* erstellt. Diese Klasse ist nur individuell implementiert und ermöglicht den einzelnen DB-Handlern eine Verbindung zur Datenbank herzustellen.

5.3.5 Zusammenfassung

In diesem Abschnitt wird die Entwicklung der Anwendung mit der leichtgewichtigen Vorgehensweise noch einmal zusammengefasst. Die Basis für die leichtgewichtige Vorgehensweise ist der Ansatz aus Abschnitt 2.3. Darin wird der Ansatz in sieben Arbeitsschritte aufgeteilt. Diese sieben Arbeitsschritte werden in ähnlicher Form, aber nicht zwingend in der selben Reihenfolge, durchgeführt. Im ersten Arbeitsschritt werden die Skripte programmiert, diese legen fest wie die Klassen aus dem Modell zu Klassen der Zielplattform Java werden. Dazu wird teilweise eine Referenzimplementierung zu Hilfe genommen. Dieses Vorgehen ist laut dem Ansatz aus 2.3, der sich an [Bohlen/Starke 03] orientiert, nicht vorgesehen. Zur gleichen Zeit wird parallel ein UML-Profil um Stereotypen erweitert, die während der Generierung abgefragt werden. Dieser Punkt ist ebenfalls nicht vorgesehen oder nicht explizit erwähnt in [Bohlen/Starke 03]. Ich gehe davon aus, dass sich dieses Vorgehen hinter Arbeitsschritt 2 verbirgt. Ansonsten ist Arbeitsschritt 3 nicht durchführbar, wenn man vorher keine Stereotypen erzeugt hat, welche Eigenschaften definieren. Arbeitsschritt 1, 2 und 3 werden de facto abwechselnd nebeneinander durchgeführt. Wenn man beispielsweise eine neue Eigenschaft benötigt, fügt man diese einem Stereotypen hinzu, attributiert diese Eigenschaft und programmiert dann das Verhalten des Generators in den Skripten. Zwar sind die Aufgaben der Arbeitsschritte klar getrennt, dennoch lassen sie sich nicht unabhängig voneinander durchführen. Arbeitsschritt 4 fließt zum Prüfen des Verhaltens des Generators auch zyklisch in die Arbeitsschritte 1-3 ein. Laut dem Ansatz aus 2.3 wird die Anwendung zum Schluss getestet. Dieses Vorgehen ist in dieser Implementierung aufgeteilt worden. Der generierte Code wird bereits vor Erstellen des individuellen Programmcodes getestet, um spätere Fehler klar im individuellen Programmcode identifizieren zu können. Dazu wird ein Java-Programm geschrieben, welches die Funktionalitäten des generierten Codes testet. Anschließend wird Arbeitsschritt 5 durchgeführt, und der individuelle Programmcode erstellt. Die Nutzung eines Build-Skripts aus Arbeitsschritt 6 ist unter Eclipse nicht nötig, da Eclipse Java-Anwendungen automatisch

erzeugt. Arbeitsschritt 7 wird abschließend durchgeführt wie im Ansatz aus 2.3. Dazu werden 2 Java-Programme geschrieben. Das Erste füllt die Tabellen der Entitäten Versicherungsnehmer, Vertrag und Konto mit Daten für zu testende Mahnfälle. Das Zweite leert die Tabellen bei Bedarf wieder, um erneut testen zu können. Um die Anwendung zu testen wird *GpMahnsystem* solange immer wieder ausgeführt, bis keine Verarbeitung von Mahnfällen mehr auf dem Konsole dokumentiert wird. Dabei werden die Bildschirmausgaben, die zum Dokumentieren der Anwendung ausgegeben werden, überprüft. Darüber hinaus werden die Inhalte der Datenbank auf korrekte Inhalte geprüft.

5.4 Schwergewichtige Methode

In diesem Abschnitt wird die komplette Implementierung der schwergewichtigen Methode dargestellt. Es wird empfohlen sich in Anlehnung an diesen Anschnitt, die Quellen auf der CD anzuschauen.

5.4.1 DSL

In diesem Abschnitt werden die Anforderungen, die an eine DSML gestellt werden, definiert. Eine DSML muss folgende Kriterien erfüllen (nach [Sirotin 08], S.76):

1. Sie wird in mindestens einer Phase des Software Lebenszyklus verwendet.
2. Sie besitzt eine formale Syntax und eine reichhaltige Semantik.
3. Durch ihre Semantik ist sie auf einen speziellen Problemraum begrenzt.
4. Sie kann durch einen Interpreter ausgewertet werden.

In den folgenden Abschnitten wird eine geeignete DSML für den Problemraum Mahnverfahren für Versicherungsanwendungen erstellt. Im der Zusammenfassung der schwergewichtigen Methode (Abschnitt 5.4.6) wird gezeigt, dass diese Sprache die hier definierten Kriterien erfüllt.

5.4.2 Metamodelle

In diesem Abschnitt wird die DSML für den Problemraum Mahnverfahren einer Versicherung erstellt. Die Sprache erhebt nicht den Anspruch vollständig zu sein, lediglich den Anspruch, die Anwendung dieser Arbeit ausreichend beschreiben zu können, um Programmcode aus solchen Modellen generieren zu können. Genau genommen werden zwei Sprachen erstellt; eine um PIM zu modellieren und daraus zu transformieren (M2M), und eine weitere um PSM darzustellen, zu modellieren und ebenfalls daraus zu transformieren (M2T). In Abschnitt 5.3.1 wird die UML bereits um Sprachelemente erweitert, welche zum Generieren einer solchen Anwendung ausreichen. Deshalb lehnt sich die hier erstellte Sprache an diese erweiterten UML-Elemente an. Zum Erstellen dieser Sprache wird das EMF genutzt. Eigene Metamodelle werden dabei in Ecore erstellt. Ecore ist ein Metametamodell genau wie MOF. Eigentlich basiert Ecore auf einer frühen Version von MOF, jedoch haben sich beide Modelle unabhängig voneinander weiter entwickelt (nach [Schmidt 07], S.2), weshalb sich beide Modelle mittlerweile unterscheiden. In diesem Abschnitt wird jeweils ein Metamodell, sprich eine DSML, für plattformunabhängige und für plattformspezifische Modelle erstellt. Die Baum-Darstellung beider DSML ist auf der Abbildung 5.5 zu sehen. Es werden folgende Sprachelemente von Ecore benutzt:

- EClass, steht für eine Klasse
- EEnum, steht für ein Aufzählungstyp
- EEnum Literal, steht für ein Literal eines Aufzählungstyps
- EAttribut, steht für ein Attribut
- EReference, steht für eine Beziehung
- EString, steht für eine Zeichenkette
- EInt, steht für einen Integerwert

Zum Definieren einer DSML erzeugt man ein Metamodell als Ecore-Modell (nach [oAW 09], S.4 ff). Das Ecore-Modell, welches Metamodell für PIM wird, heißt *gmv.ecore*. Das Ecore-Modell, welches Metamodell für PSM wird, heißt ps.*ecore*. Ecore-Modelle editiert man mithilfe eines Baumeditors. Der erste Knoten des Baums ist die Klasse *Model*. Diese Klasse

wird zum Wurzelknoten im eigentlichen Modell und muss Beziehungen zu allen Klassen vom Type EClass unterhalten. Sonst wird die Baumstruktur im eigentlichen Modell nicht unterstützt. Über diesen Wurzelknoten wird das Modell im Generator expandiert. Die Sprache besteht aus folgende Klassen: *Entität*, *Geschäftsobjekt*, *Geschäftsprozess*, *Attribut*, *Aktion*, *Parameter* und *Beziehung*. Alle Klassen haben jeweils ein Attribut *name* vom Ecore-Typ EString. In der Klasse *Beziehung* ist der Name jedoch vom selbst erstellten Aufzählungstyp *Beziehungsart*. Die Klassen *Parameter* und *Attribut* bekommen dazu noch ein Attribut vom selbst erstellten Aufzählungstyp *Datentyp*. Die Klassen werden in Beziehung zueinander gebracht. Das passiert mithilfe der EReferences. Zuerst werden drei EReferences für die Klasse *Model* erstellt. Es wird jeweils eine Beziehung zu *Entität*, *Geschäftsobjekt* und *Geschäftsprozess* erstellt. Die Multiplizität wird jeweils auf 0 zu n gesetzt. Das bedeutet, dass die Klassen *Entität*, *Geschäftsobjekt* und *Geschäftsprozess* Bestandteil von *Model* werden. Ein Modell kann durch die erstellten Beziehungen keine bis beliebig viele Entitäten, Geschäftsobjekte und Geschäftsprozesse beinhalten. Die Entität hat eine 1 zu n Beziehung mit dem Attribut. Durch das Setzen der Eigenschaft *containment* auf true wird das Attribut zum Bestandteil von der Entität. Auf dieser Weise entsteht die Baumstruktur des Modells. Weiter erhält Entität eine EReference mit der Klasse *Beziehung*, wobei Beziehung ebenfalls Bestandteil von Entität wird. Die Beziehung wiederum erhält ein weiteres Attribut *Ziel* vom Typ der hier erstellten Klasse *Entität*. Auf diese Weise werden Beziehungen zwischen Entitäten im Modell realisiert. Die Art der Beziehung wird über die Eigenschaft *name* festgelegt. Das Geschäftsobjekt beinhaltet keine bis beliebig viele Attribute und Aktionen. Der Geschäftsprozess enthält ausschließlich beliebig viele Aktionen. Weiter kann eine Aktion keinen bis beliebig viele Parameter enthalten.

Für die versicherungs- und mahntypischen Attribute werden dieselben Aufzählungstypen wie in Abschnitt 5.3.1 erstellt. Hierzu kommen noch zwei Aufzählungstypen, um eine sicherere Nutzung von Datentypen und Beziehungen zu gewährleisten. Wenn man eine eigene proprietäre DSML erstellt, sollte man einen Satz an Datentypen erstellen. Da man beim Generieren davon abhängig ist, dass die Datentypen richtig geschrieben werden, sollte man für ein Minimum an Typensicherheit sorgen. Hierfür gibt es zwei Möglichkeiten. Bei der ersten Möglichkeit erstellt man für jeden benötigten Datentypen und jede genutzte Beziehungsart im Metamodell einen eigenen Typen in Form von EClass oder EData. Die zweite Möglichkeit besteht darin, sich eigene Aufzählungstypen zu erstellen, welche alle benötigten Datentypen oder Beziehungsarten als Literale beinhalten. So kann beim Modellieren über eine Auswahl ein Datentyp oder eine Beziehungsart ausgewählt werden, ohne sich zu vertippen. Hier wird die zweite Möglichkeit gewählt. Dazu werden die Aufzählungstypen *Datentyp* und *Beziehungsart* erstellt. Die Beziehungsart enthält *erbt_von*, *besteht_nur_mit* und *Superklasse* als Literale, da keine weiteren Beziehungen während der Generierung eine Rolle spielen. Der Datentyp enthält *zeichenkette*, *geldbetrag*, *datum*, *ganzeZahl*, *merker*, *zeitstempel*, *antragsart*, *nachrichtsart*, *gmvZustand*, *mahnereignis*, *EnVersicherungsnehmer*, *EnVertrag*, *EnKonto*, *EnAntrag*, *EnNachricht*, *EnMahnfall* und *EnMahnereignis*. Dabei fällt auf, dass die modellierten Entitäten aus dem PIM ebenfalls in den Aufzählungstypen mit aufgenommen werden. Die verwendeten Datentypen und Beziehungsarten werden für ein PIM so gewählt, sodass sie auch fachliche Mitarbeiter einer Versicherung verstehen.

Aus den Metamodellen in Form von Ecore-Modellen kann man Diagramme zur besseren Darstellung generieren. Abbildung 5.4 zeigt das Metamodell für PIM als generiertes Diagramm. Abbildung 5.5 zeigt die Metamodelle für PIM und PSM nebeneinander aus Bildschirmausschnitten des Baumeditors.

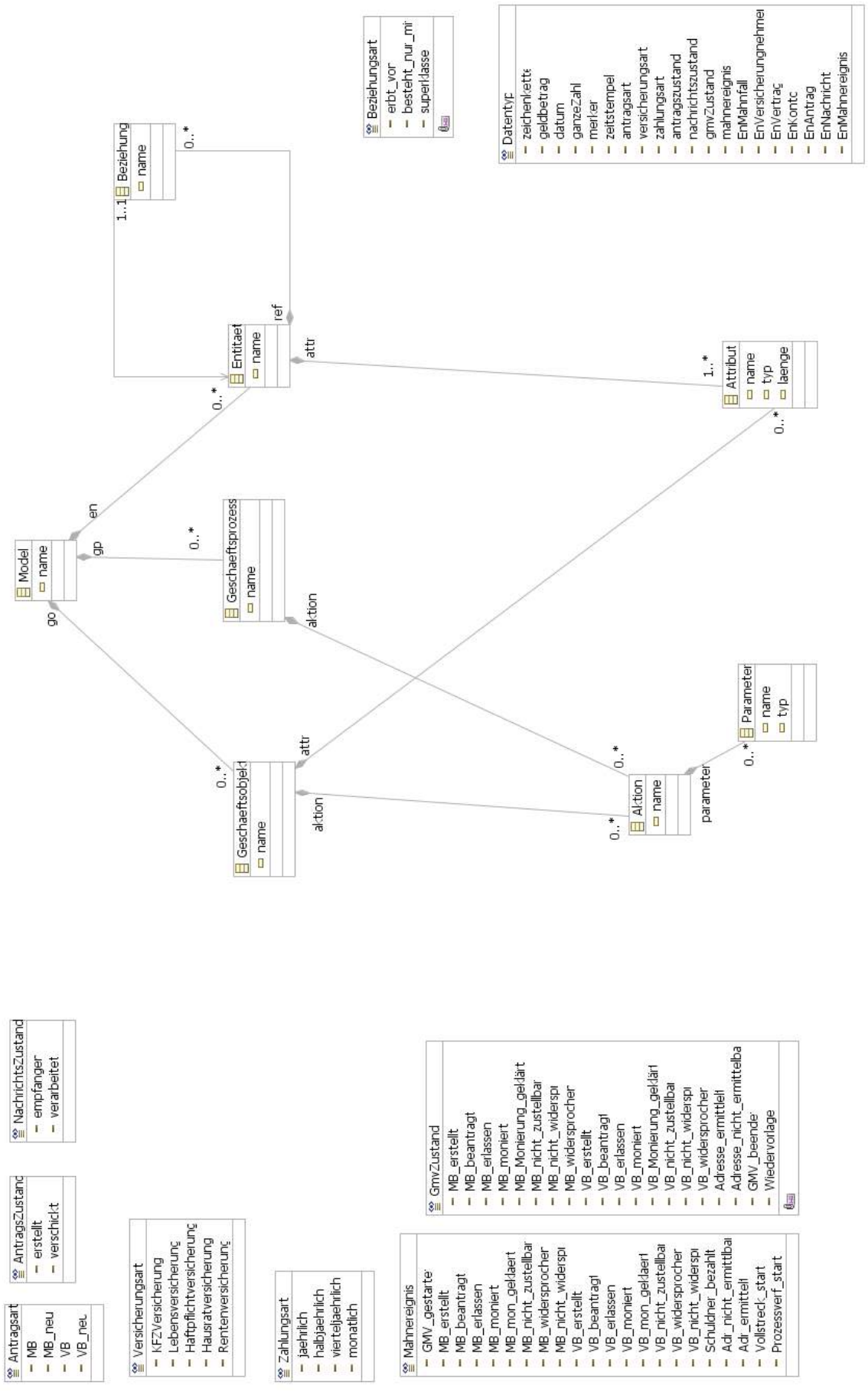

Abbildung 5.4: Metamodell für PIM in generiertem Ecore-Diagramm dargestellt.

Das oben erstellte Metamodell für PIM dient als Basis für das Metamodell für PSM. Es wird abgeändert und erweitert. Das Metamodell heißt *ps.ecore* und wird um die Metaklasse *Schlüsselattribut* erweitert. Den Metaklassen *Entität*, *Attribut* und *Aktion* werden weitere Attribute hinzugefügt. Der Aufzählungstyp *Datentyp* wird geändert und ein weiterer Aufzählungstyp *Aktionsart* wird hinzugefügt. Der Schlüsselattribut enthält die Attribute *name* und *spaltenname* vom Typ String, und *typ* vom Aufzählungstyp *Datentyp*. Die Metaklasse *Attribut* wird um die Attribute *spaltenname* vom Typ String und *nullzulässig* vom Typ Boolean erweitert. Die Entität wird um *tabname, beschreibung* und *schlüssel* vom Typ *Schlüsselattribut* erweitert. Die Aktion wird um das Attribut *art* vom Aufzählungstyp *Aktionsart* erweitert. Der Aufzählungstyp *Aktionsart* besteht aus den Literalen *fachliche*, *technische* und *dummy*, äquivalent zu dem Metamodell aus 5.3.1. Die Datentypen aus dem Aufzählungstypen *Datentyp* werden an die Java-Plattform angepasst. Die Beziehungsarten behalten ihre sprechende Bedeutung. Abbildung 5.6 zeigt das Metamodell für PSM in einem generierten Diagramm. Wie schon erwähnt editiert man die Metamodelle in Form eines Baumeditors. Über das Kontextmenü und Eclipse-View, die ein Fenster mit den Eigenschaften eines Elementes anzeigt, werden neue Elemente hinzugefügt oder bestehende geändert. In der Abbildug 5.5 werden Bildschirmausschnitte beider Metamodelle nebeneinander dargestellt. Um die erstellten Metamodelle als DSML nutzen zu können, muss noch mit Hilfe von EMF ein Editor generiert werden. Aus dem Projekt *gmv.datamodel*, in welchem das Metamodell *gmv.ecore* definiert ist, wird der Editor für PIM generiert. Der Sprachraum heißt *gmv*. Aus dem Projekt *ps.datamodel*, in welchem das Metamodell *ps.ecore* definiert ist, wird der Editor für PSM generiert.

Die generierten Editoren stehen ebenfalls in Form von Projekten in Eclipse zur Verfügung (nach [oAW 09], S.7 f). Diese Projekte werden in das Plugin Verzeichnis von Eclipse kopiert. Nach dem Neustart von Eclipse stehen die so erzeugten Sprachen dem Modellierer zur Verfügung. Über das Menü kann nun ein Gmv-Modell (PIM) bzw. Ps-Modell (PSM) erstellt und editiert werden. Die Namen der Sprachen sind so kurz gewählt, damit sie gleichzeitig als Dateiendung der Modelldateien und als Namespace innerhalb der Skripte zur Abfrage der Sprachelemente dienen können. Die erstellten Editoren sind ebenfalls Baumeditoren, welche genauso über das Kontextmenü und Views bedient werden.

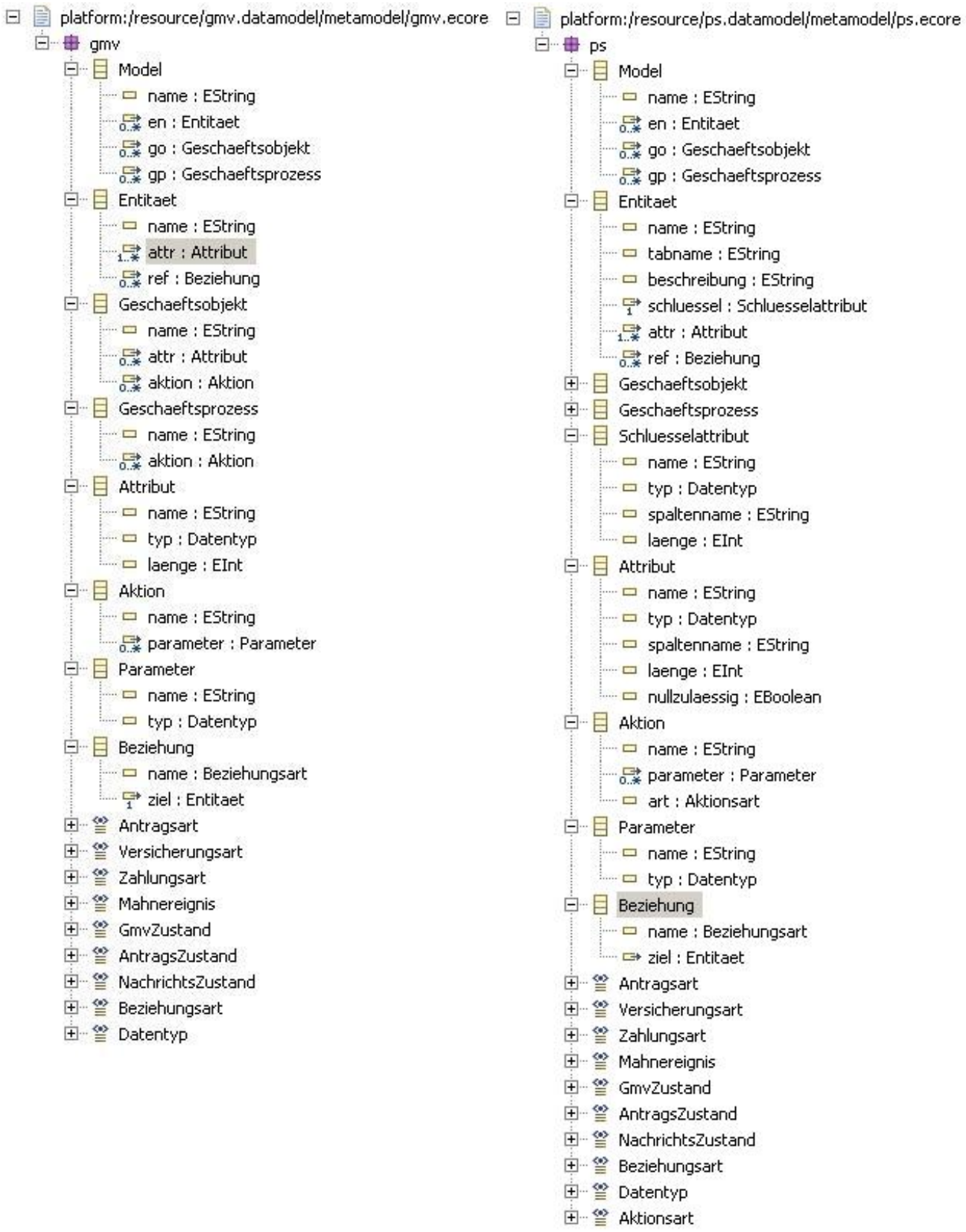

Abbildung 5.5: Metamodelle für PIM und PSM nebeneinander im Baumeditor

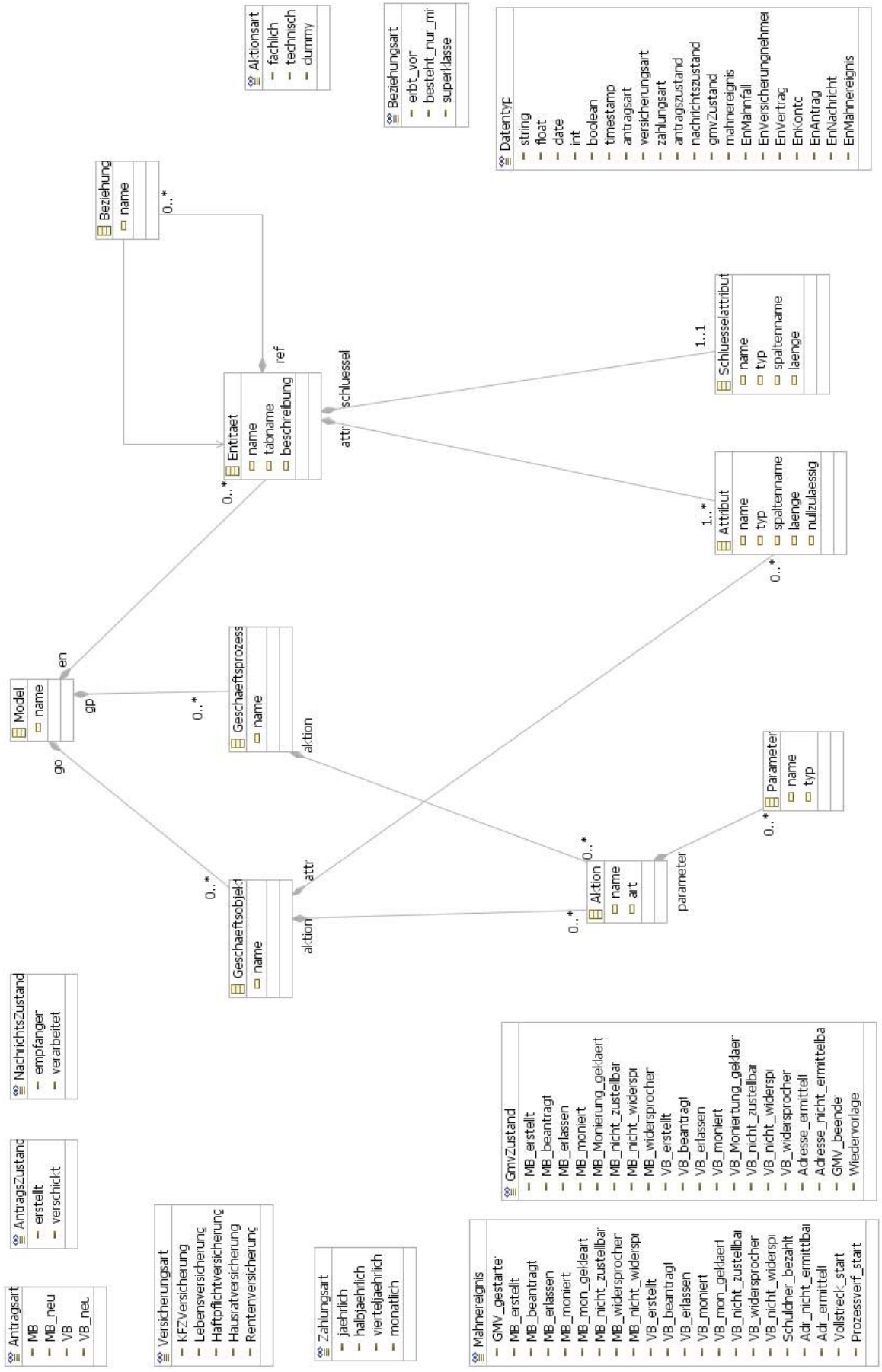

Abbildung 5.6: Metamodell für PSM in generiertem Ecore-Diagramm dargestellt.

5.4.3 Anwendungsmodelle

In diesem Abschnitt werden die Anwendungsmodelle vorgestellt. Die Geschäftslogik dieser Modelle stimmt mit der Geschäftslogik der leichtgewichtig implementierten Anwendung überein. Die Anwendung der leichtgewichtigen Implementierung ist lauffähig und die Geschäftslogik getestet. Trotzdem unterscheiden sich die Modelle in der Darstellungsweise; der Art wie sie erstellt und genutzt werden. Mehr dazu findet man Kapitel 6. Für PIM wird ein neues *Gmv-Model* erstellt. In diesem Modell wird zuerst der Wurzelknoten *Model* erstellt und Mahnsystem benannt. Weiter werden diesem Knoten die Entitäten, Geschäftsobjekte und der Geschäftsprozess aus Abschnitt 5.3.2 mit all ihren Attributen, Aktionen und Beziehungen hinzugefügt. Die Anreicherung des Modells findet hier während einer Transformation von PIM zu PSM statt. Die Inhalte der Anreicherung unterscheiden sich von der Anreicherung aus der leichtgewichtigen Implementierung. Die Längen der Zeichenketten werden bereits in das PIM mit aufgenommen. In dieser Arbeit wird eine Objektidentität als Primärschlüssel der Tabelle verwendet und jeder Entität als Schlüsselattribut vom Typ Integer in dem PSM hinzugefügt. Dazu bekommt jede Entität automatisch einen Tabellennamen hinzugefügt, der aus den Zeichen *Db_Tab_* und dem eigentlichen Namen der Entität besteht. Weiter werden die fachlichen Datentypen in plattformspezifische Datentypen umgewandelt. Attribute bekommen die Eigenschaften *spaltenname* und *nullzulässig* hinzugefügt, welche nicht automatisch mit Inhalten gefüllt werden, lediglich mit Defaultwerten. Leider kann man diesmal nicht ohne weiteres grafische Modelle aus den Modellen des Baumeditors generieren. Deshalb werden die Modelle in ihrer ursprünglichen Form dargestellt. Abbildung 5.7 zeigt das plattformunabhängige Modell, Abbildung 5.8 das plattformspezifische Modell.

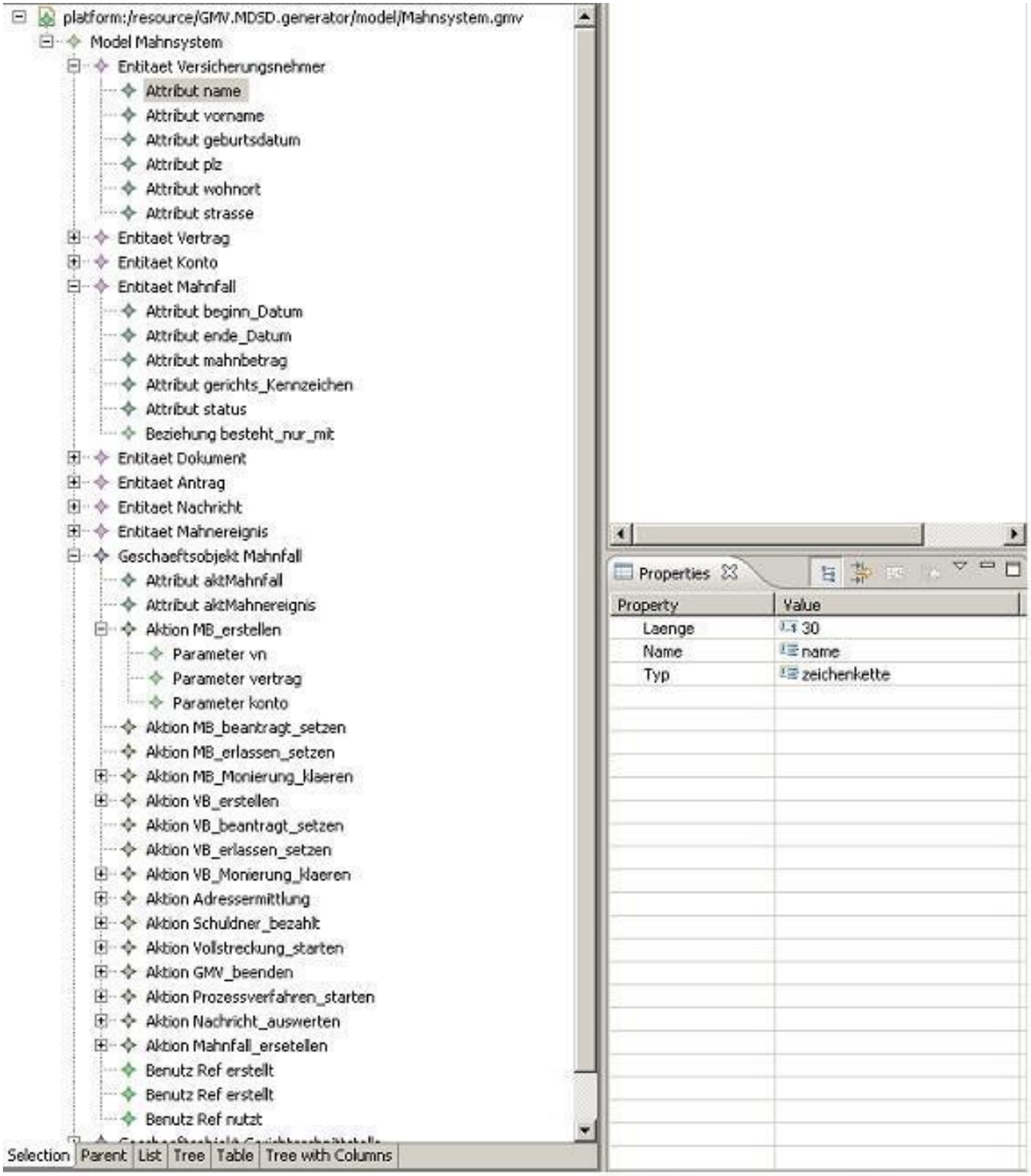

Abbildung 5.7: PIM des Mahnsystems

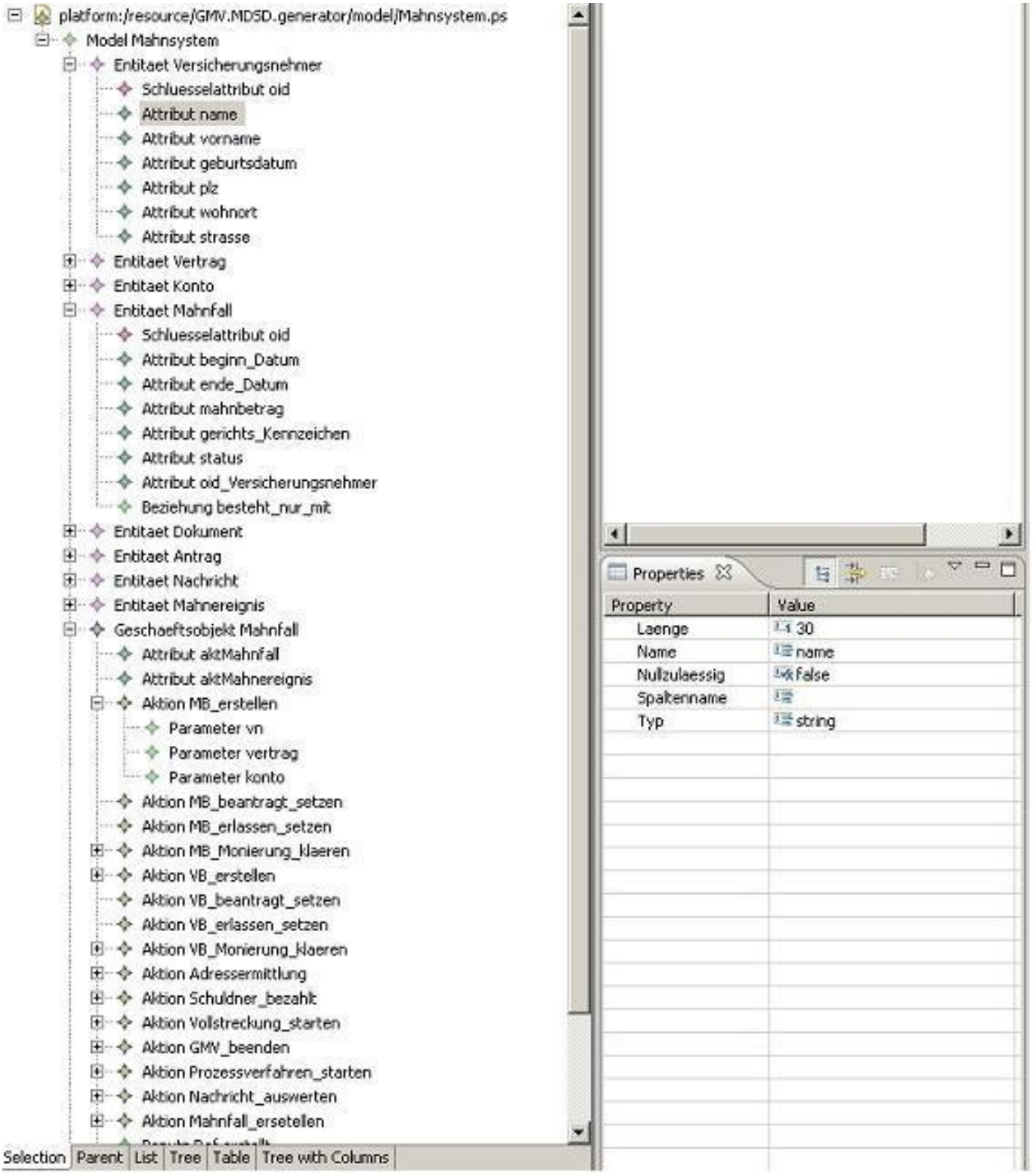

Abbildung 5.8: PSM des Mahnsystems

5.4.4 Modelltransformationen

Bei der schwergewichtigen Methode gibt es zwei Arten von Modelltransformation. Für die Transformation von PIM zu PSM wird die so genannte M2M (Modell-to-Modell) Transformation durchgeführt. Für die Transformation von PSM zu PSI wird eine M2T (Modell-to-Text) Transformation durchgeführt. Eine Transformation kann das Eingangsmodell verändern und erweitern. xPand erlaubt das Modell zu überschreiben (nach [oAW 09], S.213 ff). Die Veränderung von PIM wird nicht in das ursprüngliche Modell zurückfließen. Leider konnte keine Anleitung gefunden werden, in welcher steht, wie man eine M2M Transformation durchführt, die verschiedene Metamodelle als Basis hat. Aus diesem Grund wird in dieser Arbeit ein eigener Ansatz für M2M Transformation mit xPand verfolgt. Diese Transformation erlaubt es verschiede Metamodelle für Eingangs- und Ausgangsmodell zu nutzen. Um diese Vorgehensweise zu entwickeln, wird mit dem PSM-Editor ein kleines Modell erstellt und die dazugehörige XMI-Datei untersucht. Aus dieser Vorlage werden Transformationsregeln erstellt. Anschließend wird ein xPand Skript für die Transformation erstellt. Während der Transformation wird bei jeder Entität ein Schlüsselattribut hinzugefügt. Außerdem werden jeder Aktion und jedem Attribut weitere Eigenschaften hinzugefügt und die Datentypen von fachlichen Datentypen in plattformspezifische Datentypen transformiert. Diese M2M Transformation ist de facto eine T2T Transformation, weil ein strukturierter Text (PIM als XMI-Datei) in einen strukturierten Text (PSM als XMI-Datei) transformiert wird. Diese Texte werden jedoch in grafischen Modellen oder durch Modelle in Baumstrukturen dargestellt.

Die zweite Transformation aus dem PSM in die PSI wird wieder mit xPand Skripten durchgeführt. Dazu werden leicht abgeänderte Skripte der leichtgewichtigen Methode wiederverwendet. Die nötigen Änderungen resultieren aus den anderen Sprachelementen und einem anderen Konzept für die Beziehungen. Die Elemente sind zwar ähnlich benannt, haben jedoch andere Präfixe und andere Bestandteile innerhalb eines expandierten Elements. Die Struktur der Skripte bleibt dennoch die gleiche. Die generierten Sourcen sind identisch mit den generierten Sourcen der leichtgewichtigen Vorgehensweise.

5.4.5 Individuelle Implementierungen

Da die generierten Sourcen identisch sind, wird der individuelle Programmcode aus der leichtgewichtigen Vorgehensweise kopiert und in die Source dieser Vorgehensweise eingefügt. Aus diesem Grund wird der individuelle Programmcode nicht erneut erläutert.

5.4.6 Zusammenfassung

Während der Entwicklung mit der schwergewichtigen Methode wird ein DSML für die Domäne Mahnverfahren für Versicherungsanwendungen erstellt. Diese Sprache wird in zwei Varianten erstellt. Die erste zum Modellieren von PIM und zum generieren aus PIM (M2M). Die zweite zum Darstellen und Modellieren von PSM und zum Generieren aus PSM . Das PIM wird in Anlehnung an das Modell aus der leichtgewichtigen Vorgehensweise erstellt, da die Geschäftslogik dieses Modells alles beinhaltet, was zur Verwirklichung dieser Anwendung benötigt wird. Für die Transformation von PIM zu PSM wird ein eigener Ansatz verwendet, um Modellinstanzen eines Metamodells in Modellinstanzen eines anderen Metamodells transformieren zu können. Während der Transformation werden unter anderem plattformspezifische Daten in Form von Datentypen erstellt. Die Skripte zur Generierung des

schematischen Programmcodes werden größtenteils aus der leichtgewichtigen Vorgehensweise übernommen. Der Individuelle Programmcode wird eins zu eins übernommen. Getestet wird die Anwendung genauso wie bei der leichtgewichtigen Vorgehensweise.

In Anschnittes 5.4.1 werden Anforderungen an DSML definiert. Hier wird begründet, dass diese Kriterien erfüllt sind.

1. Die DSML wird in mehreren Phasen des Software Lebenszyklus verwendet. In der Analysephase wird das PIM *Mahnsystem* basierend auf der DSML *gmv.ecore* erstellt. In dieser Phase wird das Modell in einem Baumeditor iterativ verfeinert. In der Designphase (M2M-Transformation) werden die in der DSML definierten Metaklassen ausgewertet. Zum Beispiel werden für eine Entität die technischen Attribute, wie Fremdschlüssel, in das Ausgangsmodell aufgenommen. In der Kodierungsphase wird aus dem PSM *Mahnsystem,* basierend an der DSML *ps.ecore,* der Java-Code generiert. Dafür werden die Metaklassen in den oAW-Skripten ausgewertet. Zum Beispiel werden für eine Entität die Zugriffsmethoden in den Entitätsklassen und die DB-Schicht (Tabellen, DB-Handler) generiert.

2. Die DSML besitzt eine formale Syntax und eine reichhaltige Semantik. Die Syntax der Sprachelemente ist differenziert definiert. Zum Beispiel lässt die Syntax nicht zu, dass ein Geschäftsprozess eigene Daten besitzt. Der generierte Baumeditor bietet keine Möglichkeit ein Attribut bei einem Geschäftsprozess zu erfassen (im Gegensatz zu UML). Das fachliche Wissen (semantischer Inhalt) ist in zahlreichen fachlichen Auswahllisten auf der Ebene der Sprache (GmvZustand, Mahnereignis, Antragsart und einiges mehr) zur Verfügung gestellt.

3. Die Semantik der Sprache unterstützt nur den Problemraum Mahnwesen einer Versicherung. Die Sprachelemente *Antragsart*, *Mahnereignis*, *GmvZustand*, *AntragsZustand*, *NachrichtsZustand* kapseln das Wissen und Verhalten eines gerichtlichen Mahnverfahrens. Die Sprachelemente *Versicherungsart* und *Zahlungsart* beziehen sich auf das Versicherungswesen. Alle anderen Elemente dienen lediglich der Strukturierung der Sprache.

4. Die Sprache wird durch den oAW-Generator interpretiert. Dies wird in dieser Arbeit ausführlich beschrieben und oft exerziert.

6 Vergleich

In diesem Kapitel werden die beiden Vorgehensweise dieser Arbeit, modellgetrieben Software zu entwickeln, miteinander verglichen. Dafür werden zuerst die Vergleichskriterien dargestellt und anschließend werden die Vorgehensweisen anhand der Kriterien bewertet.

6.1 Vergleichskriterien

Zum Vergleich der beiden Vorgehensweisen werden vier Kriterien festgelegt.

Das erste Kriterium ist die Effizienz der einzelnen Arbeitsschritte (Abschnitt 6.2). Die Effizienz wird durch die Zeit gemessen, welche zur Verarbeitung des jeweiligen Arbeitsschrittes benötigt wird. Dazu wird in vergleichbaren Zeitblöcken gearbeitet und die verbrachte Zeit in einer Art Leistungsnachweis eingetragen. Die Arbeitsschritte, welche gemessen werden, sind:

1. Konfiguration der Tools
2. Metamodellierung
3. Modellierung
4. Modelltransformation
5. Individuelle Implementierung
6. Test
7. Wartung

Das zweite Kriterium ist die Qualität der Modelle (Abschnitt 6.3). Die Qualität der Modelle wird bewertet durch:

1. die intuitive Verständlichkeit der Modelle
2. die intuitive Bedienbarkeit der Modellierungssprache
3. die Fehleranfälligkeit der Modelle

Das dritte Kriterium, nach dem die Ansätze verglichen werden, ist die Vollständigkeit des generierten Programmcodes (Abschnitt 6.4). Diese Vollständigkeit wird anhand der individuellen Implementierungen gemessen.

Das letzte Kriterium bewertet die Vor- und Nachteile mehrerer Modelltransformationen in der modellgetriebenen Software-Entwicklung (Abschnitt 6.5).

6.2 Bewertung der Effizienz der Arbeitsschritte

Die Effizienz der Arbeitsschritte wird bewertet, indem man verwendete Aufwände der einzelnen Arbeitsschritte miteinander vergleicht. Dazu muss nach jedem Arbeitsschritt der jeweiligen Vorgehensweise ein gleicher, oder zumindest annähernd gleicher, Zwischenstand bestehen. Nachfolgend wird die Effizienz der Arbeitsschritte bewertet.

6.2.1 Konfiguration der Tools

Aus eigener Erfahrung weiß ich, dass es unter Eclipse oft Stunden, manchmal sogar Tage dauern kann, ein Tool richtig zum Laufen zu bringen. Deshalb wird das Konfigurieren der Tools auch als ein Arbeitsschritt in den Methoden der modellgetriebenen Software-Entwicklung betrachtet. Zur Konfiguration der Tools werden alle Aufwände gezählt, die

anfallen, bis ein reibungsloser Ablauf der Vorgehensweise gewährleistet ist. Bei den Tools, die von beiden Methoden genutzt werden, wie zum Beispiel Java oder DB2, wird die benötigte Zeit nicht mit aufgenommen.

Es werden folgende Aufwände berücksichtigt:

- Testmodelle in das Generator-Framework oAW einbinden
- Erstellung und Nutzung eines Metamodells
- Einbinden des Metamodells in den Generator, um die Schnittstelle zwischen Modellierung und Generierung zu testen
- Generierung erster Artefakte

Die dabei anfallenen Aufwände belaufen sich

- für die leichtgewichtige Methode auf 11 Stunden und
- für die schwergewichtige Methode auf 6 Stunden

Dieses Ergebnis resultiert aus Problemen mit dem UML-Profil sowohl beim Modellieren, als auch beim Generieren. Dass es Probleme beim Importieren von UML Modellen und Profilen geben kann, ist mir bekannt. Dass es mit dem EMF so reibungslos funktioniert, erstaunt um so mehr. Die größten Probleme gab es beim Einbinden des UML-Profils in das Modell. Wenn man das UML-Profil ändert, muss man es erneut einbinden. Dabei sind gelegentlich die bereits eingepflegten Stereotypen und Eigenschaften wieder verschwunden. Dieses Problem ist auch während der eigentlichen Entwicklung aufgetreten, wird dennoch zur Konfiguration des Tools gezählt. Man kann sagen, dass dieses Problem spezifisch für TOPCASED ist. Aber durch meine Erfahrung mit mehreren frei verfügbaren UML-Tools kann ich sagen, dass Probleme mit dem Modellaustausch keine Seltenheit sind. Beim Entwickeln mit EMF gab es keinerlei Probleme in diese Richtung. Die Änderungen am Metamodell werden automatisch im Modell erkannt. Aus diesem Grund wird die schwergewichtige Methode in diesem Arbeitsschritt als Sieger gewertet.

6.2.2 Metamodellierung

Zu diesem Arbeitsschritt zählt die Erweiterung des UML-Metamodells bei der leichtgewichtigen Methode und die Erstellung der DSL mit beiden Metamodellen bei der schwergewichtigen Methode. Die Aufwände dieses Arbeitsschrittes sind leider verfälscht durch die Tatsache, dass die benötigten Sprachelemente bei der Durchführung der schwergewichtigen Methode bereits bekannt waren. Das heißt, dass während der leichtgewichtigen Methode Denk- und Tipparbeit geleistet wird. Bei der schwergewichtigen Methode wird das Konzept aus der leichtgewichtigen Methode übernommen, sodass die Denkarbeit größtenteils nicht mehr geleistet werden muss.

Die angefallenen Aufwände belaufen sich

- für die leichtgewichtige Methode auf 12 Stunden
- für die schwergewichtige Methode auf 13 Stunden
  oder nach der Berücksichtigung der Wiederverwendung auf 20 Stunden

Man sieht, dass trotz des wiederverwendeten Konzeptes die schwergewichtige Methode mehr Aufwand benötigt. Dafür gibt es Gründe. Der erste ist, dass man zwei Metamodelle erstellt und sich beim Erstellen dieser auch Gedanken über die Trennung von plattformunabhängigen und plattformspezifischen Daten machen muss. Der zweite Grund ist, dass man sich um die Bereitstellung der Beziehungen, Attribute, Methoden, Datentypen usw.

selbst kümmern muss. Konzepte für diese Anforderungen werden auf Metamodellebene ausgearbeitet. UML stellt auf der Metaebene bereits die Konstrukte Klasse, Attribut, Methode, Beziehung, Stereotype und Eigenschaft zur Verfügung. Bei der Erweiterung des Metamodells werden diese Elemente im Profile benutzt. In diesem Arbeitsschritt ist die leichtgewichtige Methode als Sieger zu bewerten.

6.2.3 Modellierung

Dieser Arbeitsschritt beinhaltet die Erstellung des Anwendungsmodells. Aus der schwergewichtigen Methode fließen nur die Aufwände zur Erstellung des PIM in diesen Arbeitsschritt. Die Aufwände für die Erstellung des PSM fließen in den Arbeitsschritt Modelltransformationen mit ein. Die Aufwände dieses Arbeitsschrittes sind ebenfalls verfälscht dadurch, dass das Konzept des Modells der leichtgewichtigen Methode in der schwergewichtigen wiederverwendet wird.

Die angefallenen Aufwände belaufen sich

- für die leichtgewichtige Methode auf 23 Stunden
- für die schwergewichtige Methode auf 13 Stunden
  oder nach der Berücksichtigung der Wiederverwendung auf 20 Stunden

Je größer die Modelle werden, umso mehr Aufwand ist bei der UML-Methode zu erwarten. Die Grafiken zu editieren und das Layout zu gestalten, nimmt mehr Zeit in Anspruch, als die Texte über die Kontextmenüs zu erfassen. Dieser Arbeitsschritt ist mit leichtem Vorteil für die schwergewichtige Methode zu bewerten.

6.2.4 Modelltransformation

Aufseiten der leichtgewichtigen Methode zählt die M2T Transformation des marked-PIM zur PSI zu diesem Arbeitsschritt. Aufseiten der schwergewichtigen Methode zählen sowohl die M2M Transformation, als auch die M2T Transformation zu diesem Arbeitsschritt. Erneut sind die Aufwände verfälscht, da die Skripte der leichtgewichtigen Vorgehensweise für die schwergewichtige Vorgehensweise übernommen werden. Die Skripte müssen lediglich abgeändert werden. Durch die Struktur der Skripte kann man sehen, dass die Aufwände für die Generierung der PSI größtenteils gleich sind. Dennoch gibt es auf Seiten der schwergewichtigen Methode einen größeren Aufwand; durch die M2M Transformation.

Die angefallenen Aufwände belaufen sich

- für die leichtgewichtige Methode auf 120 Stunden
- für die schwergewichtige Methode auf 35 Stunden
  oder nach der Berücksichtigung der Wiederverwendung auf 140 Stunden

Bei diesem Arbeitsschritt wird die leichtgewichtigere Methode als effizienter gewertet.

6.2.5 Individuelle Implementierung

Der individuelle Programmcode ist bei beiden Vorgehensweisen identisch, aus diesem Grund wird dieser Arbeitsschritt als gleich effizient gewertet.

6.2.6 Test

Der generierte Code wird sofort nach dem Erstellen der Skripte getestet. Dadurch wird die Korrektheit des Skriptes sofort geprüft. Die Aufwände des Testens des generierten Programmcodes fließen somit in den Arbeitsschritt Modelltransformation mit ein. Dadurch wird die Lokalisierung der Fehlerstelle beim Testen des individuellen Codes erleichtert. Der Test des individuellen Codes ist gleichzeitig auch der Test der gesamten Anwendung.

Der generierte und individuelle Code ist in den beiden Vorgehensweisen identisch. Da im Zuge der leichtgewichtigen Entwicklung die Anwendung getestet wurde, wird bei der schwergewichtigen Methode auf das Testen verzichtet. Es wird lediglich geprüft, ob der generierte Code identisch ist. Aus diesem Grund wird dieser Arbeitsschritt als gleich effizient gewertet.

6.2.7 Wartung

Für diesen Arbeitsschritt wird

- eine plattformunabhängige Änderung an dem Modell und
- eine plattformspezifische Änderung an dem Modell vorgenommen

Bei Änderungen dieser Art, muss ggf. auch der individuelle Code angepasst werden.

Als plattformunabhängige Änderung wird in die Entitäten *Versicherungsnehmer* und *Dokument* das Feld *hausnummer* aufgenommen. Zuvor war die Hausnummer in dem Feld *strasse* gespeichert. Nun wird im Zuge einer Wartungsaktion die Hausnummer in einem eigenen Integer-Feld gespeichert.

Dazu wird für die leichtgewichtige Methode das Modell angepasst, die PSI erneut generiert und in allen Methoden, die Anträge erstellen oder Nachrichten auswerten, individuell angepasst. Abschließend werden die DB-Tabellen neu erzeugt und die Anwendung getestet.

Für die schwergewichtige Methode erfolgt die Wartungsaktion in folgenden Schritten: PIM anpassen, PSM generieren, PSI generieren, individuellen Programmcode anpassen, DB-Tabellen erzeugen und Anwendung testen. Wie man sieht, enthält die plattformunabhängige Wartungsaktion bei der schwergewichtigen Methode einen Arbeitsschritt mehr als bei der leichtgewichtigen Methode. Dieser Schritt besteht aus dem Generieren des PSM. Für diesen Arbeitsschritt fällt aber so gut wie kein Aufwand an, da man nur einmal die M2M-Transformation starten muss. Aus diesem Grund wird die plattformunabhängige Wartungsaktion als gleich effizient für beide Vorgehensweisen gewertet.

Als plattformspezifische Änderung wird der Datentyp für die Schlüsselattribute der Entitäten geändert. Weiter wird die Namenskonvention für Tabellennamen geändert.

Im Zuge einer Wartungsaktion mit der leichtgewichtigen Methode wird der Typ der Schlüsselfelder von *Integer* auf *Long* geändert. Bei den Namenskonventionen der DB-Tabellen wird das Präfix von *Db* auf *Db_* im Marked-PIM geändert. In jeder Entität muss

sowohl die Eigenschaft *tabname,* als auch der Typ des Schlüssels manuell geändert werden. Nachdem man die Änderungen manuell in das Marked-PIM eingepflegt hat, folgen die Schritte: PSI generieren, DB-Tabellen erzeugen und Anwendung testen. Individuelle Anpassungen am Programmcode sind nicht nötig.

Im Zuge einer Wartungsaktion mit der schwergewichtigen Methode wird der Typ vom Schlüsselattribut von *int* auf *long* geändert. Bei den Namenskonventionen für Tabellennamen wird das Präfix von *Db_Tab_* auf *DbTab* geändert. Dafür wird zuerst ein neuer Datentyp *long* in der plattformspezifischen Version der DSML erstellt. Dann wird der Editor neu generiert, um den neuen Datentypen im Modell auch anzeigen zu können. Weiter wird die M2M Transformation angepasst; der Typ des Schlüsselattributs und das Präfix der Tabellennamen werden berücksichtigt. Abschließend folgen dieselben Schritte wie zuvor bei der leichtgewichtigen Methode: PSI generieren, DB-Tabellen erzeugen und Anwendung testen. Auf den ersten Blick erscheint die Wartungsaktion mit der schwergewichtigen Methode aufwendiger, da mehr verschiedene Arbeitsschritte nötig sind. Jedoch werden bei dieser Methode die Arbeitsschritte größtenteils automatisch durchgeführt, während die plattformspezifischen Anpassungen bei der leichtgewichtigen Methode manuell eingepflegt werden müssen.

Die angefallenen Aufwände belaufen sich

- für die leichtgewichtige Methode auf 3 Stunden
- für die schwergewichtige Methode auf 1,5 Stunden

Für diesen Arbeitsschritt wird die schwergewichtige Methode als Sieger bewertet.

6.3 Bewertung der Qualität der Modelle

Die Qualität von Modellen wird üblicherweise über Reviews durch Experten bewertet. Dabei wird die fachliche Korrektheit und Vollständigkeit des Modells geprüft. Leider kann für diese Arbeit kein Expertenwissen genutzt werden. Aus diesem Grund werden hier nur Kriterien zur Wertung verwendet, von welchen ich glaube sie wirklich bewerten zu können, durch die Erfahrung während dieser Arbeit. Diese Kriterien sind die intuitive Verständlichkeit der Modelle, die Fehleranfälligkeit beim Modellieren und die intuitive Bedienbarkeit der Modellierungssprache.

Zur Bewertung der intuitiven Verständlichkeit der Modelle kann man die Modelle nebeneinander auf dem Bildschirm betrachten. Dabei fällt auf, dass ein grafisches Modell wie das Klassendiagramm aus 5.3.2 viel mehr Informationen auf den ersten Blick preisgibt. Man sieht sofort welche Entitäten eine Beziehung zueinander haben. Weiter sieht man auch die Typen der Felder und die Parameter der Methoden auf den ersten Blick. In dem Modell aus 5.4.3, welches in einem Baumeditor dargestellt wird, müsste man erst die Knoten der Elemente expandieren, um die Parameter der Methoden zu sehen oder um fest zu stellen, ob Entitäten Beziehungen zueinander haben. An dieser Stelle sind die UML-Modelle ganz klar als Sieger zu bewerten.

Die intuitive Bedienbarkeit der Modellierungssprache wird nach eigenem Ermessen beurteilt. Beim Modellieren mit UML erhält man Unterstützung durch eine Symbolleiste, welche eine Vielzahl an verfügbaren Elementen bietet. Durch diese grafischen Symbole lassen sich Standard UML-Elemente direkt per Mausklick in das Modell einfügen. Um ein Sprachelement des selbstdefinierten Profils zu erstellen, muss man erst das passende UML-Element erstellen und dann den Stereotypen ändern. Will man den Stereotypen eines Elementes ändern, geschieht das bei den Eigenschaften des Elements, in einem dafür vorgesehenen

Fenster, oder über ein Kontextmenü. Beim Modellieren mit der eigenen DSML wählt man neue Elemente über das Kontextmenü, mit der rechten Maustaste. Die auswählbaren Elemente entsprechen den benötigten Sprachelementen, man muss keinen Stereotypen mehr ändern. Die grafischen Elemente sind ein Vorteil von UML, die direkt erstellbaren Sprachelemente ein Vorteil der proprietären DSML. UML bietet fertige Konzepte, beispielsweise für Beziehungen. Bei der eigenen DSML werden eigene Konzepte für Beziehungen ausgearbeitet. Diese eigenen Konzepte sind nicht so intuitiv wie die der UML. Deshalb bietet das Modellieren mit UML insgesamt eine intuitivere Bedienbarkeit.

Die Fehleranfälligkeit der Modelle wird auch über die eigenen Erfahrungen aus dieser Arbeit bewertet. Wenn man in UML einem Attribut einen Typ zuweisen will, öffnet man bei den Eigenschaften ein Drop-Down-Menü und bekommt hunderte Typen zur Auswahl. Auf dieses Phänomen trifft man an vielen Stellen beim Modellieren mit UML. Der Sprachumfang ist riesig und man wird dadurch oft überfordert. Als Folge davon kann man schnell Fehler in das Modell einpflegen, wie z.B. durch die Verwendung von Sprachelementen welche im generierten Programmcode nicht vorgesehen sind und vom Generator nicht abgefangen werden. Beim Modellieren mit der eigenen DSML gibt es derlei Probleme nicht, es gibt nur die eigenen Sprachelemente, man kann gar keine anderen auswählen. Man kann trotzdem beim Modellieren mit DSML Fehler machen, wie durch das Weglassen von Eigenschaften. Beispiel dafür sind die Längen der Zeichenketten. Aber die Wahrscheinlichkeit einen Fehler zu begehen ist bei der Modellierung mit der eigenen DSML wesentlich geringer. Dadurch, dass die UML keine Beschränkung der Vielfalt ermöglicht, ist sie fehleranfälliger. Aus diesem Grund werden UML-Modelle als anfälliger für Fehler gewertet.

Um derart Fehler beim Modellieren zu vermeiden, sollten Modelle während des Entwicklungsprozesses automatisch validiert werden. Wie man Modelle automatisch validiert, kann in Anschnitt 3.4 nachgelesen werden. In dieser Validierung kann man die Vollständigkeit der Modelle und die verwendeten Sprachmittel überprüfen. Beispielsweise kann man über Constrains sowohl bei UML-Modellen als auch bei den EMF-Modellen prüfen, ob jede Entität einen Namen hat. Dies wäre eine Validierung der Vollständigkeit eines Modells. Eine Prüfung, ob wirklich nur Sprachmittel des erforderlichen Namensraums genutzt werden, macht nur Sinn für die UML-Modelle. Dazu würde man prüfen, ob die verwendeten Elemente aus dem erstellten Profil sind, indem man überprüft, ob die Elemente das Präfix *gmv*:: anstatt *uml*:: haben. Bei den EMF-Modellen braucht man nicht zu überprüfen, ob man wirklich Elemente des benötigten Namensraums nutzt, da es keine anderen Elemente gibt. In dieser Arbeit werden die Modelle nicht validiert, da der Entwicklungsprozess von den Metamodellen über die Modelle bis zur Generierung des Programmcodes ausschließlich von mir durchgeführt wird. Da ich die benötigten Bestandteile des Modells verinnerlicht habe, und den generierten Programmcode immer selbst überprüfe, habe ich auf das Validieren der Modelle verzichtet.

Von den drei Kriterien zur Bewertung der Qualität der Modelle, schneiden die UML-Modelle bei zwei Kriterien besser ab. Eigene DSML liefern viel Potential für MDSD, aber alles in allem überwiegen die Faktoren der grafischen Darstellung und der Werkzeugunterstützung. Das Potential eigener DSML liegt darin, die Sprachmittel auf das Nötigste einzuschränken. Mann kann durch den höheren Freiheitsgrad bei der Erstellung des Metamodells, die Modelle weiter einschränken, wodurch beim Modellieren die Fehleranfälligkeit reduziert werden kann.

Eclipse bietet über das GMF die Möglichkeit, aus den Editoren für die eigene DSML grafische Editoren zu generieren. Dazu wird jedem Element der Sprache ein grafisches Element oder ein Platzhalter innerhalb eines grafischen Elements zugewiesen. Somit soll sich mit nicht allzu großem Aufwand ein Box-and-Line-Editor, ähnlich zu UML-Klassendiagrammen erzeugen lassen. Leider ist es in dieser Arbeit nicht möglich, ein solches grafisches Werkzeug für eigene DSML zu erzeugen, da der Aufwand zu groß wäre. Bei Interesse wird eine Arbeit empfohlen, welche eine Erstellung eines solchen grafischen Editors mit EMF und GMF zu Thema hat [Kuhn 09].

6.4 Bewertung der Vollständigkeit des generierten Programmcodes

Wie weiter ober schon erwähnt, wird die PSI durch nahezu identische Skripte erzeugt. Der generierte Programmcode und die individuellen Implementierungen sind identisch. Aus diesem Grund kann man die Bewertung der Vollständigkeit des generierten Programmcodes aus dem Vergleich streichen.

6.5 Bewertung mehrerer Modelltransformationen

Hier soll die Verwendung von mehreren Modelltransformationen, im Gegensatz zu einer Transformation vom Modell zum Programmcode, bewertet werden.

Bei der Entwicklung mit der schwergewichtigen Methode wird aus dem PIM ein PSM generiert, aus welchem wiederum die PSI generiert wird. Bei der Modelltransformation von PIM zu PSM wird das PIM automatisch erweitert, verändert und in einem neuen Modell gespeichert. Bei den Erweiterungen handelt es sich um Schlüsselattribute und Tabellennamen der Entitäten. Bei den Änderungen handelt es sich um die Datentypen der Attribute. Aus fachlichen Datentypen werden plattformspezifische Datentypen. Diese Erweiterungen und Änderungen werden automatisch mit der Skript-Sprache xPand erzeugt.

Bei der Entwicklung mit der leichtgewichtigen Methode werden diese Informationen manuell in das Modell eingetragen. Man hätte sie aber ebenso während der Generierung der PSI automatisch aus dem Modell in den Code generieren können. Das heißt, alle automatischen Transformationen, welche durch zwei nacheinander folgende Transformationen realisiert werden, können auch durch eine Transformation realisiert werden. Die Trennung der Transformationen dient also zum Kapseln der Transformationsregeln. Dies kann man auch gut an der üblichen Arbeitsweise der oAW M2M-Transformationen sehen. Dabei wird die Transformation von PIM zu PSM direkt in die Transformation zur PSI mit eingebunden, ohne das PSM weiter editieren oder betrachten zu können. Es existiert lediglich im Speicher. Will man das PSM dennoch betrachten, wird der Generator so konfiguriert, dass man das PSM physikalisch speichert. Die Kapselung der PIM zu PSM Transformation hat einen Vorteil. Bei einem Wechsel der Plattform muss nicht erst in den Skripten nach den Regeln für diese Transformation gesucht werden. Das Konzept der Transformationen von PIM über PSM zu PSI wird in der Theorie eben so beschrieben. Man ist flexibler beim Austausch von Plattformen. Der Nachteil dieses Konzeptes ist der größere Aufwand.

Bei der schwergewichtigen Methode dieser Arbeit ist das PSM weiter editierbar. Bei Attributen der Entität ist zum Beispiel die Eigenschaft *nullzulässig* hinzugekommen und standardmäßig auf false gesetzt. Man kann sie auf true setzten und daraufhin Code generieren. Wird aber nach einer beliebigen Änderung des PIM das PSM neu generiert, ist diese Änderung rückgängig gemacht (durch den Standardwert false überschrieben). Diese Vorgehensweise in der schwergewichtigen Methode dieser Arbeit führt also zu Inkonsistenzen. Dies ist natürlich ein klarer Nachteil. Um solche Probleme zu beheben, müsste man das neue PIM mit dem PSM zusammenführen, damit keine per Hand eingepflegten Daten aus dem PSM verloren gehen. Leider bietet oAW auf Anhieb keine Lösung dieses Problems. Zu dieser Inkonsistenz ist es gekommen, da das PSM auf jeden Fall darstellbar sein sollte.

6.6 Zusammenfassung

Um zwei verschiedene Vorgehensweisen, wie die dieser Arbeit, unabhängig voneinander bewerten zu können, bräuchte man eigentlich zwei unabhängige Entwickler oder Enwickler-

Teams mit gleichen Kompetenzen. Diese würden dann dieselbe Anwendung mit verschiedenen Methoden modellgetriebener Software-Entwicklung implementieren. Bewertet würden die Anwendungen und die Methode der Entwicklung dann von einer neutralen Jury anhand vorher bestimmter Kriterien. Ein solch enormer Aufwand ist in einer Diplomarbeit wie dieser nicht zu leisten. Deshalb wird dieser Vergleich unter gegebenen Umständen vollzogen. Wieso man eigentlich zwei unabhängige Entwicklungen bräuchte, kann man gut bei der Bewertung der Kriterien sehen: Wenn der Vergleich von ein und derselben Person durchgeführt wird, kann es zu Verfälschungen von Messergebnissen kommen, da bei der Durchführung der zweiten Methode, Ergebnisse der ersten wiederverwendet werden.

Dennoch kann man treffende Aussagen über die einzelnen Kriterien des Vergleichs machen. Bei der Bewertung der einzelnen Arbeitsschritte steht es drei zu zwei für die schwergewichtige Methode. Dies liegt vor allem an den Fehlern des Modellierungs-werkzeugs TOPCASED. Dieses beeinflusst die Bewertung der Konfiguration der Tools. Würde man diesen Arbeitsschritt als gleich effizient werten, wäre die Bewertung nach Arbeitsschritten ausgeglichen. Zählt man den gesamten Aufwand, unter Berücksichtigung der Wiederverwendung, ist die leichtgewichtige Methode im Vorteil. Aus diesem Grund wird die leichtgewichtige Methode als effizienter gewertet.

Bei der Qualität der Modelle wird das UML-Modell in zwei von drei Kategorien als besser gewertet, durch grafische Diagramme und die gebotene Werkzeugunterstützung. Dennoch sind UML-Modelle fehleranfälliger. Um Fehler während der Modellierung mit UML zu vermeiden, sollten die Modelle validiert werden. Die Qualität der Modelle wird bei der leichtgewichtigen Methode als besser bewertet.

Bei der schwergewichtigen Methode fällt auf, das durch das Editieren des PSM Inkonsistenzen entstehen. Um das zu vermeiden, sollte das PSM nicht editierbar sein, es sei denn man verfügt über einen Mechanismus, der das editierte PSM mit einem PIM zusammenführen kann. Konfigurations-Managemant-Tools könnten sich dafür eignen (nach [Völter 05]). Für die Anwendung dieser Arbeit sind mehrere Modelltransformationen nicht nötig, da man keinen Nutzen davon hat diesen Aufwand zu tätigen. Wäre die Entwicklung dieser Anwendung jedoch so gedacht, dass sie auf verschiedenen Plattformen lauffähig werden soll, dann wären mehrere Modelltransformationen sinnvoll. Mehrere Modelltransformationen dienen lediglich der Kapselung der Transformationsregeln. Da die Anwendung dieser Arbeit nicht für mehrere Plattformen gedacht ist, ist der Vorteil dieser Kapselung (leichtere Wartbarkeit bei plattformspezifischen Änderungen) von geringerer Bedeutung.

Alles in allem ist die leichtgewichtige Methode besser geeignet für die Entwicklung der Anwendung dieser Arbeit. Diese Aussage lässt sich jedoch nicht verallgemeinern. Ob man nun die Transformationsregeln kapselt, hängt von der Anwendung ab, welche man entwickeln möchte. Ob man die UML oder das EMF zum Modellieren benutzt ist, denke ich, eine Geschmacksfrage. Die einen sind überzeugt von UML, andere wiederum, sind überfordert von mit Komplexität der Tools.

Jede Methode hat ihre Stärken und Schwächen. Ist man in der Lage die Vorteile von UML in die selbstdefinierte DSML zu übernehmen, hat man beide Ansätze vereint und man kann sich entscheiden, ob man mit eigenen DSL einen leicht- oder schwergewichtigen Ansatz verfolgt.

7 Fazit und Ausblick

Ein Ziel der Arbeit war die Durchführung beider Methoden modellgetriebener Software-Entwicklung, um praktische Erfahrungen währenddessen zu sammeln. Dieses Ziel ist erreicht worden. Die lauffähigen Projekte werden auf der CD übergeben, sowie die benötigten Tools und Systeme.

Ebenfalls ein Ziel dieser Arbeit war die Erforschung mehrerer Modelltransformationen im Gegensatz zur direkten Generierung von Programmcode aus dem Anwendungsmodell. Es wird dadurch erforscht, dass während der leichtgewichtigen Methode das Modell direkt in Programmcode transformiert wird, und während der schwergewichtigen eine weitere Transformation von PIM zu PSM stattfindet.

Dabei ist herausgekommen, dass mehrere Transformationen lediglich zur Kapselung von Transformationsregeln dienen. Da dieses Konzept mit größerem Aufwand verbunden ist, macht es nur Sinn mehrere Modelltransformationen während der Entwicklung zu nutzen, wenn man flexibel im Austausch von Plattformen bleiben will. Wenn man sich für eine schwergewichtigere Methode von MDSD entscheidet, sollte man in jedem Fall darauf achten, dass man das PSM nicht mehr editieren kann oder dass es einen Mechanismus gibt, PIM und verändertes PSM zusammenzuführen. Ansonsten entstehen Inkonsistenzen, wie bei der schwergewichtigen Methode dieser Arbeit. Für Änderungen im Programmcode sind die geschützten Bereiche gedacht. Ein ähnliches Konzept wäre für M2M-Transformationen auch denkbar. Wenn die manuellen Änderungen des PSM geschützt wären, könnte der Generator sie bei der Transformation wieder an dieselbe Stelle einfügen. Leider wird ein solches Konzept nicht von oAW unterstützt.

Ein weiteres Ziel war die Untersuchung verschiedener Modellierungssprachen für MDSD. Dazu wurde während der leichtgewichtigen Methode mit UML modelliert, während der schwergewichtigen mittels einer selbstdefinierten domänenspezifischen Modellierungssprache (DSML). Dabei wird festgestellt, dass die UML trotz ihrer Komplexität immer noch besser geeignet ist für modellgetriebene Software-Entwicklung. Die grafische Darstellung der Modelle und die Werkzeugunterstützung während des Modellierens sind ein Grund dafür. Ein weiterer Grund sind die Konzepte, welche nicht nur beim Modellieren, sondern auch beim Auswerten der Modelle und während der Erstellung der Transformations-Regeln mehr Komfort bieten. Zu diesen Konzepten zählt beispielsweise die Auflösung der Beziehungen mit UML. Bei der eigenen DSML muss man zur Auflösung von Beziehungen eigene Konzepte entwickeln und realisieren. Das betrifft sowohl das Modellieren als auch die Transformationsregeln.

In Abschnitt 2.2 wird eine gewisse Skepsis gegenüber der Verwirklichung von MDA mit oAW geäußert. Diese Skepsis ist insofern widerlegt, da während der schwergewichtigen Entwicklung mehrere Modelltransformationen durchgeführt worden sind. Zwar wird in dieser Arbeit nicht explizit mit der MDA nach Spezifikation entwickelt, aber durch die M2M-Transformation dieser Arbeit ist das Verständnis dafür insofern gestiegen, dass solche Transformationen auf XMI-Ebene statt finden. Daher dürfte eine MDA nach der reinen Spezifikation durchführbar sein. Hier zeigt sich der Vorteil einer integrierten Entwicklungsumgebung (IDE) wie Eclipse. TOPCASED, oAW und EMF arbeiten auf derselben XMI-Basis.

Die in dieser Arbeit mit der leichtgewichtigen Methode erstellte Anwendung, eignet sich gut als Basis für ein reales Mahnsystem einer Versicherung. Der Generator ist so Konfiguriert, das eine Weiterentwicklung leicht durchführbar ist. Im Modell kann man erkennen, an welchen Stellen Bedarf für Änderungen ist. Die Anwendung aus der schwergewichtigen Methode eignet sich in dieser Form nicht für den realen Gebrauch, wegen der Inkonsistenzen zwischen PIM und PSM. Für einen realen Gebrauch der schwergewichtigen Methode, müsste diese noch verbessert werden. Die Inkonsistenzen zwischen PIM und PSM müssen beseitigt werden. Dies kann auf mehrere Arten passieren:

- man übernimmt die Eigenschaften die zu Inkonsistenz führen in das PIM
- ein zweites PSM wird eingeführt, indem die plattformspezifischen Änderungen manuell modelliert werden und während der M2T-Transformation werden beide PSM-Modelle berücksichtigt
- man führt einen Mechanismus zum Zusammenführen von Modellen während der M2M-Transformation ein

In der Arbeit ist es gelungen, die Datenbankschicht beinahe vollständig zu generieren. Wenn man Suchschlüssel in Form von einer weiteren Art von Schlüsselfeldern in das Modell mit aufnimmt, ist eine 100% Generierung der DB-Handler möglich. Diese Suchschlüssel könnten in UML als eingebettete Klasse, oder über weitere Eigenschaften definiert werden. Die Möglichkeit der individuellen Programmierung stünde natürlich weiterhin zur Verfügung.

An mehreren Stellen wird in dieser Arbeit auf die Stärken und Schwächen von verschiedenen Konzepten hingewiesen. Ein Hauptunterschied der beiden Vorgehensweisen ist die eingesetzte Modellierungssprache. Die UML ist grundsätzlich visuell und die verwendete DSML textuell. Beide Darstellungsarten haben ihre Berechtigung. Daher liegt es nahe eine graphische Darstellung der DSML zu entwickeln. Das GMF bieten die Möglichkeit einen solchen graphischen Editor für ein Ecore-Modell zu generieren. Eine weitere Forschung in diese Richtung ist erstrebenswert.

Die Komplexität der UML lässt sich zur Zeit nicht einschränken. Ein Tailoring der UML-Artefakte für die Erfordernisse des Projekts wäre wünschenswert. Weiß man aber, was genutzt werden soll und was überfüssig ist, kann eine Validierung des UML-Modells mittels OCL oder Transformationsregeln programmiert werden. Da eine Transformation das Modell interpretieren muss, kann auch an der Stelle eine Validierung stattfinden. Tendiert man aber zu einer DSML sollte der Aufwand in die Gestaltung eines Ecore-Modells investiert werden. Damit hätte man das Tailoring und die Validierung zur Zeit der Modellierung zur Verfügung.

Ein weiteres Defizit von UML ist die schwache Unterstützung der Prozessmodellierung. Dafür werden proprietäre Lösungen, meistens textuelle Sprachen, genommen. Es bietet sich an der Stelle an, eine DSML zu definieren (nach [Sirotin 08], S.78).

In der Arbeit wird nur statisches Verhalten des Modells ausgewertet und in den Transformationsregeln verarbeitet. Dynamische Aspekte des Modells könnten aber auch interpretiert und in der Generierung berücksichtigt werden. Zum Beispiel aus Zustands- oder Aktivitätsdiagrammen des Mahnverfahrens kann eine Zustandübergangsmatrix generiert werden. Diese würde dann in einem generischen Zustandsautomaten im Java interpretiert werden. Alternativ könnte auch für jedes Zustandsdiagramm ein statischer Code generiert werden (nach [Stehr 06]). Die Generierung aus dynamischen Modellen ist ein spannendes Thema für sich.

Gelingt es, die Vorteile von UML in einer selbstdefinierten DSML nachzubilden und die fehlenden Funktionalitäten und Konzepte herzustellen, hat man beide Ansätze vereint. Dadurch würde man die Vorteile beider Ansätze nutzen, und dann kann man sich entscheiden ob man mit der eigenen DSML einen leicht- oder schwergewichtigeren Ansatz verfolgt.

Trotz aller Schwierigkeiten hat sich die Arbeit auf jeden Fall gelohnt. Der Einsatz des Generators erleichtert die Wartung und senkt die Fehlerquote in dem schematischen Code drastisch. Weiter habe ich eine Menge über die Möglichkeiten und Probleme von M2M-Transformationen gelernt. Dinge, die man erst durch den praktischen Einsatz verinnerlicht. Auch bei den Möglichkeiten eigener DSML, im Gegensatz zur UML, ist mein Horizont enorm erweitert worden. Ich denke, das Wissen, welches ich durch die Realisierung der Methoden diese Arbeit erlangt habe, hätte ich nicht durch theoretische Recherche erlangen können.

Abkürzungsverzeichnis

OMG	Die Object Managment Group, Inc. wurde 1989 gegründet und ist eine internationale Organisation von über 800 Mitgliedern (Hardware-Hersteller, Software-Häuser, Endanwender). Im Rahmen objektorientierter Technologien der Software-Entwicklung wurden zahlreiche Richtlinien und Spezifikationen veröffentlicht.
MDA	Model-Driven Architecture ist eine Spezifikation der OMG, die im Jahr 2000 veröffentlicht wurde. Beschreibt ein modellgetriebenes Vorgehen welches Geschäftslogik von der Plattformtechnologie trennt. Setzt dabei auf weitere OMG Standards wie z.B. MOF, UML und XMI.
MDSD	Model Driven Software Development wurde als Oberbegriff modellgetriebener Software-Entwicklung eingeführt ([Stahl/Völter 05], S.5). Ähnelt dem MDA-Ansatz, setzt jedoch nicht zwingend auf OMG-Standards.
MOF	Die Meta Object Facility ist eine Spezifikation der OMG. Bietet eine konkrete Syntax zur Erstellung von Metamodellen.
Metamodell	Beschreibendes Modell für alle Modellarten.
DSL	Domain Specific Language ist eine Domänen spezifische Sprache zum beschreiben von Modellen. Wird in einem Metamodell definiert.
DSML	Domain Specific Modelling Language ist eine DSL zum Modellieren.
DSPL	Domain Specific Programming Language ist eine DSL zum Programmieren.
UML	Die Unified Modeling Language ist ebenfalls eine OMG-Spezifikation, die über MOF definiert wurde. UML hat sich als Modellierungssprache zur objektorientierten Software-Entwicklung durchgesetzt.
XMI	XML Metadata Intercharge ist eine OMG-Spezifikation die dem Modellaustausch (Interoperabilität) zwischen verschiedenen MDA-Tools dient.
EMF	Eclipse Modelling Framework erlaubt das Definieren eigener Modellierungssprachen.
GMF	Graphical Modelling Framework, mit dem man grafische Repräsentationen für Eclipse definieren kann.
Ecore	Eclipse MetaMetamodell, zur Erstellung von Sprachen und Tools.
CIM	Das Computation Independent Model ist eine weitgehend umgangssprachliche Beschreibung domänenspezifischer Zusammenhänge. Ist ein Bestandteil des MDA-Ansatzes der OMG.
PIM	Das Platform Independent Model beschreibt formalisiert die Anforderungen auf plattformunabhängiger Ebene (fachliches Referenz-Modell meistens in UML).
PSM	Das Platform Specific Model beschreibt alle Anforderungen speziell für eine Plattform.
PSI	Die Platform Specific Implementation ist der aus der letzten Transformation entstandene Programmcode (das letzte PSM); also textuelle Darstellung des Modells.
Marked PIM	Ist ein mit technischen Zusatzinformationen angereichertes PIM.
OCL	Object Constraint Language ist eine deklarative Sprache die semantische Anreicherungen von Modellen (i. d. R. UML) ermöglicht.

QVT	Queries Views Transformation ist eine neue Definition der OMG, eine explizite Modellabfragesprache für Transformationen zwischen den genannten Modellebenen.
M2M	Bezeichnet eine Modell zu Modell Transformation, z.B. von PIM zu PSM oder PSM zu PSM.
M2T	Bezeichnet eine Modell zu Text Transformation, z.B. von marked PIM zu PSI oder PSM zu PSI.
Templates	Bezeichnung für Skripte die Modelltransformation steuern.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Piotrek et al. 07] Poitek, G.; Trompeter, J.; Beltran, J.; Holzer, B.; Kamann, T.; Kloss, M.; Mork, S.; Niehues, B. und K. Thoms: Modellgetriebene Software-Entwicklung - MDA und MDSD in der Praxis, entwickler.press, Paderborn 2007

[Stahl/Völter 05] Stahl, T. und M. Völter: Modellgetriebene Software-Entwicklung - Techniken, Engineering, Management, dpunkt.verlag, Heidelberg 2005

[Petrasch/Meimberg 06] Petrasch, R. und O. Meimberg: Mobile Driven Architecture: Eine praxisorientierte Einführung in die MDA, dpunkt.verlag, Heidelberg 2006

[Kastens/Büning 08] Kastens, U. und H. Kleine Büning: Modellierung, Grundlagen und formale Methoden, Carl Hanser Verlag, München 2008

[MDA 03] MDA Guide Version 1.0.1. 2003

[Bohlen/Starke 03] Bohlen M. und G. Starke: MDA Entzaubert. Objektspektrum Jahrgang (2003), Nr.3, S.52 ff

[Richly et al. 05] Richly, S.; Lehner, W. und D. Schaller: GignoMDA – Modellgetriebene Entwicklung von Datenbankanwendungen. Datenbank-Spektrum , Jahrgang 2005, Nr.15

[Mielke 02] Mielke, C.: Geschäfsprozesse: UML-Modellierung und Anwendungsgenerierung, Forschung in der Softwaretechnik Herausgegeben von Helmut Balzert, Spektrum Akademischer Verlag, Heidelberg – Berlin 2002

[DeMarco 97] DeMarco, T.: Warum ist Software so teuer? ... und andere Rätsel des Informationszeitalters. Carl Hanser Verlag, München 1997

[Sirotin 08] Sirotin, V.: UML vs. DSL Teil 2: Stärken, Schwächen, Unterschiede und Mischformen. Objektspektrum Jahrgang (2008), Nr. 6, S. 76 ff

[Sirotin 09] Sirotin, V.: UML vs. DSL Teil 3: Stärken, Schwächen, Unterschiede und Mischformen. Objektspektrum Jahrgang (2009), Nr. 1, S. 76 ff

[Schmidt 07] Schmidt-Casdorff, C.: Model to Model: Neuer QVT-Standard will Modelltransformationen erheblich vereinfachen, Javamagazin Jahrgang (2007) Nr.11

[Maschinelles GMV 09] Die maschinelle Bearbeitung der gerichtlichen Mahnverfahren: Eine Informationsschrift und Anwendungshilfe, herausgegeben von den Justizverwaltungen der Bundesländer. Stand: 03/2009

[oAW 09] openArchitectureWare User Guide, Version 4.3.1

[Stehr 06] Stehr, J.: Modellgetriebene Entwicklung von Mautsystemen, Objektspektrum Jahrgang(2006) Nr.4, S.43

[Völter 05] Völter, M.: Katalisator-Effekt (Versionierung von Modellen), Javamagazin Jahrgang (2005), Nr.11, S. 90

Online Quellen

[GMV 08a] GMV: Gerichtliches Mahnverfahren. http://www.ratgeber-recht24.de/Gerichtliches_Mahnverfahren_Teil_1. Version: 2008. – [Online; Stand 02. Juni 2008]

[GMV 08b] GMV: Mahnverfahren Aktuell. http://www.mahnverfahren-aktuell.de/index.html. Version: 2008. – [Online; Stand 02. Juni 2008]

[Onlinemahn 09] http://www.online-mahnantrag.de. Version: 2009. – [Online; Stand 22. Mai 2009]

[Profimahn 09] http://www.profimahn.de. Version: 2009. – [Online; Stand 22. Mai 2009]

[EGPV 09] http://www.egvp.de. Version: 2009. – [Online; Stand 22. Mai 2009]

[Eclipse 09] http://www.eclipse.org. Version: 2009. – [Online; Stand 22. Mai 2009]

[Kuhn 09] http://wiki.eclipse.org/images/d/d0/DA_StefanKuhn.pdf. Version: 2009. – [Online; Stand 22. Mai 2009]

Anhang A – Nutzung der Implementierten Projekte

In diesem Anhang werden die Quellen auf der CD erläutert. Um die Quellen nutzen zu können, bedarf es gewisser Voraussetzungen. Um welche es sich dabei handelt und wie man diese gewährleistet, wird ebenfalls gezeigt.

Systemvoraussetzungen

Zu den Voraussetzungen zählen:

- Windows XP SP2, mindestens 1GB Hauptspeicher
- Java Version 1.6.x
- IBM DB2 Version 9.5
- Eclipse Version 3.4.x mit den in Abschnitt 5.2 beschriebenen Plugins

Eine ausführbare Installationsversion von DB2 wird gepackt auf der CD mitgeliefert. Die Eclipse Entwicklungs- und Laufzeitumgebung wird ebenfalls mit den entwickelten Projekten ausgeliefert. Die Projekte sind direkt in Eclipse integriert.

Installationsanleitung

Um die implementierten Projekte ohne großen Aufwand nutzen zu können, wird hier eine kurze Installationsanleitung angeboten. Eine Java-Installation wird nicht erläutert.

Die gepackte Version von Eclipse muss auf C:\ entpackt werden. Andernfalls müssten Pfade angepasst werden. Eclipse kann durch Aufruf von *C:\eclipse\eclipse.exe* gestartet werden.

Zur Installation von DB2 wird einfach das *setup* ausgeführt. Dabei ist zu beachten, dass man sich während der Installation den Benutzernamen und das Passwort notiert, da es im weiteren Verlauf benötigt wird. Nachdem DB2 installiert ist, erzeugt man eine Datenbank mithilfe der Steuerzentrale, die man im Programmordner von DB2 findet. Für die Entwicklung wurde der Datenbank- und Aliasname *db2t* genannt. Es wird empfohlen denselben Namen zu verwenden. Zum Erzeugen der benötigten Tabellen startet man das Befehlsfenster aus dem DB2-Programmordner. Eine Verbindung mit der Datenbank erzeugt man auf der Kommandozeile über den Befehl: „db2 connect to db2t“. Weiter erzeugt man die Tabellen, indem man die generierten DDL ausführt. Diese sollten sich nun unter *C:\eclipse\workspace\MDA-Light\src-gen\ddl* und unter *C:\eclipse\workspace\MDSD\src-gen\ddl* befinden. Die Tabellen erzeugt man, indem man innerhalb dieser Ordner für jede DDL den Befehl: „db2 –t –vf *name*.ddl“ ausführt.

Nutzungsanleitung

Um die Projekte zu sehen, sollte Eclipse in der *openArchtecturWare* Perspektive sein. Die gewählte Perspektive erkennt man oben rechts im Bild. In dieser Perspektive befindet sich links im Bild ein Package Explorer, welcher alle Projekte zeigt. Folgende Projekte sind zu sehen:

- MDA-Light: enthält die leichtgewichtige Implementierung
- MDSD: enthält die schwergewichtige Implementierung
- gmv.datamodel: enthält die Definition der DSML *gmv*
- ps.datamodel: enthält die Definition der DSML *ps*

- gmv.datamodel.edit: enthält den Gmv-Editor, generiert aus gmv.datamodel
- gmv.datamodel.editor: generiert aus gmv.datamodel
- gmv.datamodel.tests: generiert aus gmv.datamodel
- ps.datamodel.edit: enthält den Ps-Editor, generiert aus ps.datamodel
- ps.datamodel.editor: generiert aus ps.datamodel
- ps.datamodel.tests: generiert aus gmv.datamodel

Die Struktur der implementierten Projekte ist jeweils gleich. Der oAW Workflow, die Modelle und die Templates befinden sich in gleichnamigen Ordnern. Die generierten Klassen befinden sich im Ordner *src-gen*.

Damit *JDBC* eine korrekte Verbindung zur Datenbank erstellen kann, muss in der Klasse *DbConnection* der Benutzernamen und das Passwort eintragen werden. Die Klasse befindet sich bei beiden Implementierungen im Ordner *src-gen**mahnsystem.db*\.

Um die Anwendungen zu testen, lädt man zuerst die Testfälle in die Tabellen beider Implementierungen. Dazu führt man jeweils *gmvTestfaelleLaden.java* aus, welche sich in den Ordnern *src-gen**mahnsystem.test* befinden. Java-Programme kann man über das Kontextmenü „Run As | Java Application“ ausführen. Die Anwendungen startet man jeweils mit dem Geschäftsprozess *GpMahnsystem.java,* der im Ordner *src-gen**mahnsystem.gp* liegt. Diesen führt man sooft nacheinander aus, bis alle Mahnfälle verarbeitet sind. Die Verarbeitung der Mahnfälle wird auf der Konsole dokumentiert.

Zum Überprüfen der Tabelleninhalte wechselt man in die *DbEdit* Perspektive. Die Perspektive wechselt man über die Menüpunkte „Window | open Perspective | other“ oder direkt oben rechts in der Entwicklungsumgebung. Um das Datenbank-Plugin *DbEdit* nutzen zu können, muss es konfiguriert werden. Über das Kontextmenü von *DB2T*, welches für die Datenbank steht, öffnet man ein Fenster zum Konfigurieren unter „Connection | Configure“. In diesem Fenster ändert man die Felder *User*, *Schema* und *Password. User* und *Schema* sollten den erstellten Benutzernamen enthalten. Nach einem Klick auf *Connect,* sollte das Plugin funktionsfähig sein. Unter dem *DB2* Knoten kann man die erzeugten Tabellen sehen. In der Tabelle für die Mahnereignisse kann man den Ablauf eines Mahnfalls gut nachvollziehen, indem man die Tabelleninhalte über die Spalten *OID_MAHNFALL* und *OID* sortiert.

Den Programmcode der Anwendungen generiert man über das Kontextmenü eines Workflows.
Auf diesem Weg wird auch die M2M Transformation der schwergewichtigen Methode ausgeführt. So wird aus dem Model *Mahnsystem.gmv*, das Model *Mahnsystem.ps* generiert.

Zum Testen der Implementierungen gibt es noch zwei Anmerkungen:

1. In den xPand-Templates werden Fehler angezeigt, obwohl die Skripts fehlerfrei sind.

2. Wenn man im UML-Modell bzw. Klassendiagramm *Mahnsystem.umldi* etwas löscht, wird es nicht aus der dazugehörigen XMI-Datei *Mahnsystem.uml* entfernt. Dies ist ein Fehler von TOPCASED. Das Diagramm und die XMI-Datei müssen immer auf dem gleichen Stand sein. Damit es konsistent bleibt, muss man aus dem Diagramm gelöschte Elemente manuell in der XMI-Datei löschen. Dazu öffnet man die XMI-Datei mit einem Doppelklick in einem Baumeditor. Fügt man jedoch Elemente hinzu, dann sind beide Dateien konsistent. Der Generator generiert den Programmcode aus der XMI-Datei. Aus diesem Grund ist diese Konsistenz besonders wichtig.