Modelle - Janusköpfe der Software-Entwicklung - oder: Mit Janus von der A- zur S-Klasse

Wolfgang Hesse

Fachbereich Mathematik und Informatik, Univ. Marburg,
Hans Meerwein-Str., D-35032 Marburg
hesse@informatik.uni-marburg.de

Zusammenfassung: Modelle spielen heute eine hervorragende Rolle in der Software-Entwicklung. Sie haben sowohl deskriptiven Charakter (als Nachbilder eines "realen" Weltausschnitts) als auch präskriptiven (als Vorbilder und Baupläne für zu konstruierende Systeme). Software-Entwicklung lässt sich als Folge von Modelltransformationen auffassen, die von Objekten der Anwendungswelt über Modell-Objekte zu Software-Objekten (den Bausteinen unserer Systeme) führt und deskriptive sowie präskriptive Elemente miteinander verbindet. Eine deutlichere Trennung dieser drei Welten und der darin liegenden Objekte und Klassen kann für mehr terminologische Klarheit, durchsichtigere Transformationsprozesse und eine bessere Wiederverwendbarkeit der Ergebnisse sorgen. Ein entsprechend erweitertes Metamodell bietet Platz für Glossar-artige Beschreibungen von Anwendungs-Objekten und damit auch für Ontologien in der Softwaretechnik.

1 Einleitung: Modelle als Nach- und Vorbild

Modelle sind – das ist heute nahezu unbestritten - zum zentralen Element der Software-Entwicklung geworden und aus deren wichtigsten Phasen nicht mehr wegzudenken. Selbst die so genannten agilen Methoden, bei denen Spezifikation und Dokumentation eine eher untergeordnete Rolle spielen, wollen auf Modelle nicht ganz verzichten – wenngleich natürlich – der Grundbotschaft folgend – die Modellierung "agil" (sprich: möglichst schnell und unverbindlich) erfolgen soll [Amb 05].

Im Hauptstrom der Softwaretechnik haben die Modelle dagegen weitgehend die Rolle der Spezifikation eingenommen. Wo früher über Spezifikationssprachen, operationale oder algebraische, formale oder halb-formale Spezifikation debattiert wurde, beherrschen heute die "Unified Modeling Language" und "Modell-getriebene Architektur" bzw. "Modell-getriebene Entwicklung" die Szene. Ist das nur eine Umbenennung – ein Angebot des alten Weines in neuen Schläuchen? Offenbar nicht, denn Modelle bieten einerseits mehr, andererseits weniger als Spezifikationen. Weniger insofern, als nicht jedes Modell wie eine Spezifikation genau festlegen muss, was das künftige Produkt – in unserem Falle die Software – leisten soll. Eine (richtig verstandene) Spezifikation gilt als Maßstab für die Überprüfung des Produkts: Man muss es an ihr messen und damit dessen Korrektheit zeigen oder widerlegen können. Das muss bei einem Modell nicht unbedingt der Hauptzweck sein. Umgekehrt kann ein Modell auch mehr sein als eine Spezifikation: Während diese ausschließlich Vorbildcharakter hat, kann ein Modell auch Nachbild eines Gegenstands oder Sachverhalts sein oder – wie wir sehen werden – sogar Vor- und Nachbild zugleich.

Grundsätzlich ist ein Modell ein Stellvertreter für ein Original [Sta 73, Lud 02], der diesem Original in gewisser (begrenzter) Hinsicht gleicht oder ähnelt. Entscheidend ist dabei, ob das Modell *vor* oder *nach* dem Original entsteht: Im erstgenannten Falle ist es *präskriptiv*, hat also *Vorbildcharakter,* im zweiten ist es *deskriptiv* (hat *Nachbild-charakter*). Betrachten wir dazu Beispiele: Das Papphäuschen des Architekten – heute weitgehend ersetzt durch eine "virtuelle" 3D-Darstellung am Bildschirm – oder die oben genannte Spezifikation eines Computer-Programms sind *vor* den zugehörigen Originalen da, haben also Vorbildcharakter für diese. Dagegen sind Landkarten in den meisten Fällen *Nachbilder* einer bereits vorhandenen Landschaft und die von den Geographen so genannten "thematischen Karten" zeigen sehr schön, wie das Hervorheben bestimmter Aspekte (z.B. topographischer, klimatischer, politischer Art) zu ganz verschiedenen Darstellungen des gleichen Originals führt.

Was sind nun die Modelle in der Software-Entwicklung: Vor- oder Nachbilder? Sie sind in der Regel beides zugleich – und das macht sie gerade so interessant. Ich vergleiche sie deswegen gern mit den Eingangsskulpturen in alten römischen Patrizierhäusern. Diese stellten den doppelgesichtigen Türgott *Janus* dar, der sowohl nach draußen als auch nach drinnen schaut. Ähnlich geht es uns heute mit unseren UML-Modellen. Sie sollen einerseits einen Ausschnitt aus der Anwender-Welt darstellen, dienen uns also als Nachbild dieses Weltausschnitts. Andererseits nutzen wir sie als Vorlage für die zu erstellende Software (also als Vorbild und Spezifikation), anhand derer wir (unter anderem) auch die Güte des Endprodukts überprüfen und nachweisen können.

Abb. 1: Der doppelgesichtige Gott Janus auf einer römischen Münze

Das Modewort von der *Modell-getriebenen Entwicklung* (*model-driven development / architecture – MDD* bzw. *MDA, vgl*. [M-M 03]) betont diesen Vorbild- und Spezifikations-Aspekt - und suggeriert insofern noch nichts Neues: Wir werden angehalten, Modelle als Grundlage unserer (Produkt-) Entwicklung überhaupt erst einmal anzufertigen, dabei systematisch vorzugehen und verschiedene Aspekte zu trennen wie z.B. die (Plattform-abhängigen) Implementierungs-Aspekte von den (Plattform-unabhängigen) Anwendungs-Aspekten. Dem "doppelgesichtigen" Charakter unserer Modelle wird MDD dagegen noch zu wenig gerecht. Die Modelle sind nämlich nicht nur

Treiber der Entwicklung *(model driven)*, sondern zugleich Getriebene – u.a. der Anwendungswelt, der darin erhobenen Anforderungen, der Entwicklungsziele etc..

Dazu kommt der - ebenfalls nicht ganz neue – Transformationscharakter der Software-Entwicklung. Bei MDD spielt er eine große Rolle: Aus dem Plattform-unabhängigen Modell PIM soll – womöglich in mehreren, so weit wie möglich automatisierten Transformationsschritten ein Plattform-spezifisches Modell PSM abgeleitet werden. Damit wird ein verbindliches, am Beginn der Transformationskette stehendes PIM suggeriert. Tatsächlich beginnt die Transformationskette aber viel früher und an ihrem Beginn steht kein Modell, sondern die "nackte" Anwendungswelt – allenfalls ein Modell aus einem früheren Projekt, das aber nur in seltenen Fällen wieder verwendbar, d.h. auf ein neues Projekt übertragbar ist.

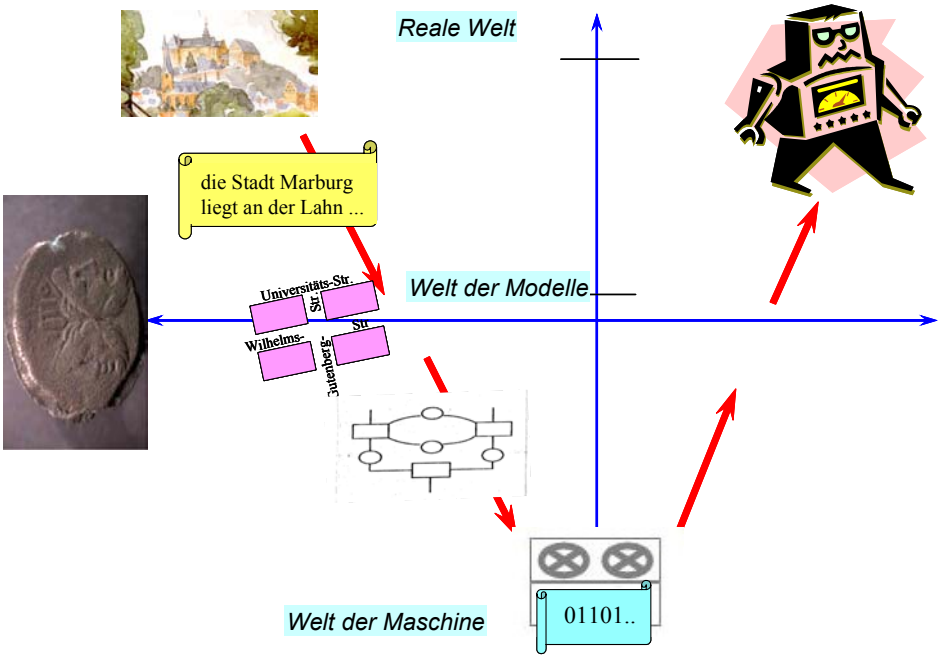

Abb. 2: Vom Anwendungsmodell zum ausführbaren Programm

Das erste Modell ist also in der Regel nicht ein UML-basiertes PIM, sondern die Domänen- und Anforderungsbeschreibung, d.h. ein noch sehr anwendungsnahes, implementierungsfernes Nachbild der Anwendung. Aus diesem müssen dann Schritt für Schritt ähnlichere Vorbilder für das zu entwickelnde Computer-Programm gewonnen werden, das am Ende der Modellkette steht. Begleitet also den römischen Patrizier ein einziger Janus, der an der Pforte des Hauses "drinnen" von "draußen" scheidet, so sind es in der Software-Entwicklung üblicher Weise mehrere "Janüsse", die uns als (Analyse-, Entwurfs-, Implementierungs-, etc.) Modelle durch den gesamten Entwicklungsprozess begleiten (vgl. Abb. 2).

In den folgenden Abschnitten wollen wir uns diesen Modellen und ihren verschiedenen Ausprägungen näher zuwenden. Dabei steht der Gedanke der Software-Entwicklung als einer Folge von Modell-Transformationen im Mittelpunkt. Diese sollten jedoch nicht mit einem intuitiv gefundenen, Vor- und Nachbildaspekte vermischenden PIM beginnen, sondern schon weit früher bei der Domänen- und Anforderungsanalyse. Gerade wenn die Modelle eine entscheidende Rolle für die weitere Programmentwicklung spielen sollen, werden Fragen nach ihrer Entstehung, ihrem Bezug zur Anwendungs-Domäne, ihrer Stabilität und Qualität immer wichtiger. Dem könnte man durch eine Ausweitung des MDD-Ansatzes auf die frühen Phasen der Software-Entwicklung und ein entsprechend erweitertes UML-Metamodell Rechnung tragen. Daraus ergeben sich erweiterte Möglichkeiten zur Modellierung und Weitergabe von Domänenwissen - etwa in der Form von Ontologien.

2 Modelle im Software-Entwicklungsprozess

In diesem Abschnitt betrachten wir den Software-Entwicklungsprozess mit seinen heute üblichen Zwischenstufen, aber speziell unter dem Blickwinkel der dabei verwendeten Modelle und ihres Vor- und Nachbildcharakters.

Modell	als Nachbild	als Vorbild
Anwendungsfall-Beschreibungen	• anwendungs-nah • narrativ, vorwiegend in natürlicher Sprache, kaum formal	• enthalten Anforderungen an das zu erstellende System
Glossar, Objektliste	• widerspiegeln wesentliche Konzepte des Anwendungsbereichs	• enthalten Kandidaten für Dateien, Datenbank-Tabellen
UML-Klassen- und Objektdiagramme (PIM)	• strukturieren (Konzepte für) Gegenstände des Anwendungsbereichs	• bilden (mögliche) Vorlage für Dateien, Datenbank-Tabellen
UML-Dynamik-Diagramme	• strukturieren Abläufe, Vorgänge, Prozesse des Anwendungsbereichs	• bilden (mögliche) Vorlage für Programme, Routinen
UML-Diagramme (PSM)	• strukturieren Gegenstände des Anwendungsbereichs (so weit noch benötigt)	• strukturieren Gegenstände des Implementierungs-bereichs
Code	• reflektiert Gegenstände und (vorher existierende) Abläufe im Anwendungs-bereich	• steuert die Behandlung der Gegenstände und Abläufe im Anwendungsbereich (in seiner künftigen Form)

Abb. 3: Modelle im Software-Entwicklungsprozess als Nach- und Vorbilder

Die obige Tabelle (Abb. 3) zeigt einige typische Zwischenstufen wie Anwendungsfall-Beschreibungen *(use case models),* Glossare bzw. Objektlisten (wie sie z.B. Jacobson vorschlägt), UML-Diagramme, den entstehenden Code. Alle Stufen haben sowohl Nach- als auch Vorbild-Charakter – wir haben es also, um im obigen Bild zu bleiben, nicht mit einem einzigen Janus-Modell, sondern mit einer ganzen Kette von solchen Modellen zu tun. Allerdings verlagert sich der Schwerpunkt – in der Regel von mehr nach-bildenden Modellelementen hin zu mehr Modellelementen mit Vorbildfunktion. Der Sinn vieler Modell-Transformationen besteht nun gerade darin, einzelne Elemente dahingehend umzuformen, dass sie besser als Vorbild für die angestrebte Lösung dienen können.

Als (Standard-) Beispiel wollen wir *Personen* betrachten, die z.B. in der Kundendatei eines Versandunternehmens registriert werden sollen. Die Originale (die für das Unternehmen in vielfacher Ausprägung vorhanden und wichtig sind) sind Menschen aus Fleisch und Blut – sie dienen als Vorbilder für die nachfolgende Modellierung. Die Menge der Merkmale, mit denen man diese Originale beschreiben kann, ist offen, ja sogar potentiell unendlich – wir könnten zu jeder Merkmalsmenge immer weitere und noch feinere Merkmale finden. Ein mögliches erstes Modell bildet die Anforderungsbeschreibung, sie enthält in narrativer Form (z.B. als Teil der Anwendungsfälle) eine Beschreibung derjenigen Merkmale, die für das Unternehmen und die geplante Anwendung relevant sind. Sie enthält möglicherweise aber auch schon Elemente des Nachbild-Originals, etwa Vorgaben, wie bestimmte Kundenmerkmale später am Bildschirm des Sachbearbeiters oder eines online-Bestellers repräsentiert werden sollen.

Schon hier liegt also in der Regel ein "Janus-Modell" vor, d.h. eine Mischung von Merkmalen des Vorbilds mit solchen des Nachbilds. Eine gute Anforderungsbeschreibung zeichnet sich allerdings dadurch aus, dass sie diese Aspekte deutlich voneinander trennt – schon allein deshalb, weil die letztgenannten noch Gegenstand möglicher Entwurfsentscheidungen (oder –revisionen) sein können, die erstgenannten dagegen in der Regel nicht. Oder – wie M. Jackson es formuliert: "It is essential to distinguish clearly between the properties of the Problem Domain that are given, and those which the Machine must enforce." Als Beispiel betrachte man einen Fahrstuhl, zu dessen *inhärenten* (Vorbild-) *Eigenschaften* es gehört, dass er auf der Fahrt von Ebene 2 zur Ebene 4 an Ebene 3 vorbeikommt. Eine (im Gegensatz dazu *verhandelbare*) *Anforderung* an die zu entwickelnde Fahrstuhl- Software (das Nachbild) könnte darin bestehen, dass an keiner Ebene vorbeigefahren wird, falls für diese eine Mitfahr-Anforderung besteht (vgl. [Jac 02]).

Betrachten wir als nächstes zwei UML-Klassendiagramme, die für zwei verschiedene Möglichkeiten stehen, unsere oben genannten Personen zu modellieren (vgl. Abb. 4). Was stellen sie dar: Nachbild (der Personen aus Fleisch und Blut) oder Vorbild (z.B. für deren Repräsentation in der Kundendatei oder am Bildschirm)? Das Modell *Person_1* betont offenbar die deskriptive Sicht. Alle Attribut-Typbezeichnungen haben Bezug zum Anwendungsbereich und lassen mögliche alternative Realisierungen offen. So könnte z.B. der Typ *VName* für eine Liste möglicher Vornamen stehen und eine diesbezügliche Plausibilitätsprüfung ermöglichen.

Die Repräsentation für Werte vom Typ *Datum, GTyp* und *AdrTyp* ist noch offen, gleiches gilt für Angaben der Körper- oder Konfektionsgröße. Dagegen ist das Modell *Person_2* deutlicher präskriptiv: Hier wurden die Repräsentations-Entscheidungen schon getroffen, wie z.B. durch *String* (mit Maximallänge) für *Name, Vorname, Straße* und *Stadt*, durch *Boolean* für *GTyp*, *Integer* für die Größenangaben sowie durch Zerlegung von *Datum* und *AdrTyp* (auf eine ganz bestimmte Weise) in Bestandteile. Dazu folgen die Namen von Attributen und Operationen bestimmten, durch die Programmiersprache oder -richtlinien vorgegebenen Konventionen, die nichts mit dem Anwendungsfeld zu tun haben.

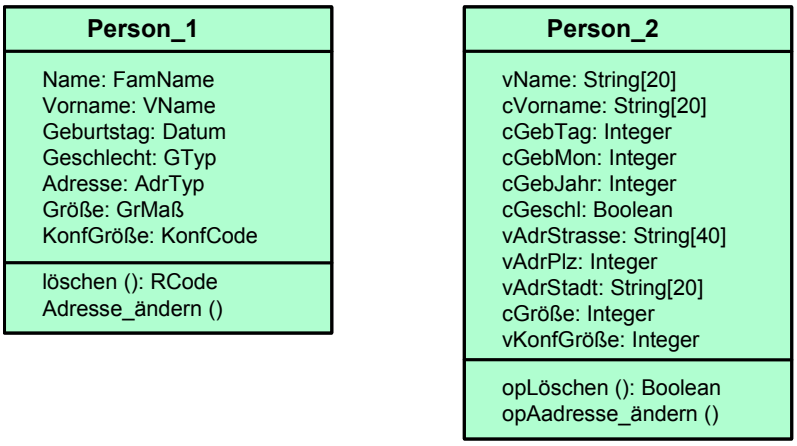

Abb. 4: Zwei Modelle für die Klasse "Person"

Im Code – auch dieser ist ein Modell - überwiegen eindeutig die Vorbild-artigen Elemente (für das zugehörige Nachbild-"Original", das auszuführende Programm), aber z.B. in "sprechenden" Variablen-, Typ- und Operationsbezeichnungen finden sich immer noch Elemente, die an das ursprüngliche Vorbild aus der Anwendungswelt erinnern. Alle genannten Modelle haben – wohlgemerkt – ihre Berechtigung und sollten im Sinne der Entflechtung von Anwendungs- und Lösungswissen sowie der Bündelung von Entwurfs- bzw. Implementierungsentscheidungen so wenig wie möglich miteinander vermischt werden. Dies kommt nicht nur der oben kolportierten Forderung von M. Jackson, sondern auch dem bekannten Prinzip des "separation of concerns" entgegen.

3 Das "Softwaretechnik-Dreieck"

Wir können uns die geschilderte Situation anhand einer Graphik veranschaulichen, dem folgenden "Softwaretechnik-Dreieck" (s. Abb. 5)[1]. Sie ist inspiriert durch das so genannte "semiotische Dreieck", das in vielen Versionen existiert und in seiner Urform Aristoteles zugeschrieben wird.

[1] Dieses Dreieck sollte nicht mit dem "software-technischen Dreieck" von P. Schefe verwechselt werden (vgl. [Sce 99]).

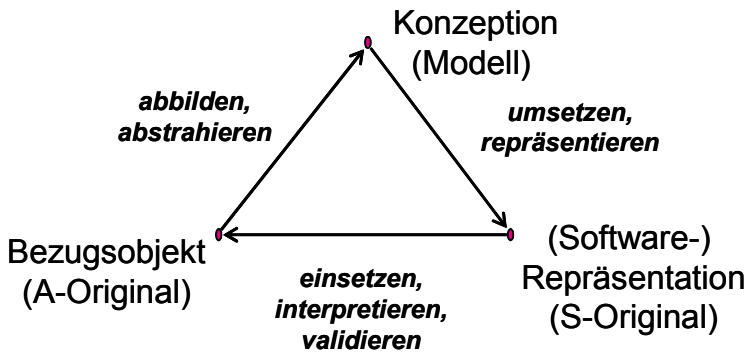

Abb. 5: Softwaretechnik-Dreieck

In der hier gezeigten Form ist es dem Tetraeder aus dem FRISCO-Bericht (vgl. [FHL 98] nachempfunden, der zusätzlich noch den Beobachter bzw. Interpreter in der Mitte des Dreiecks zeigt. In dieser Sichtweise fasst man die Entwicklung und Nutzung von Informationssystemen als Zeichenprozesse auf. Zu jedem Zeichen gehört ein Gegenstand (hier das "Bezugsobjekt", bei FRISCO: *sign referent*), auf den das Zeichen verweist und eine Zeichen-Repräsentation (ein Symbol, FRISCO: *sign token*), also die Erscheinung des Zeichens selbst. Der Bezug zwischen beiden wird hergestellt durch die Auffassung (FRISCO: *conception*) des Beobachters, der als Zeichen-Hersteller Bezugsobjekte durch Symbole repräsentiert und als Zeichen-Nutzer Symbole als Bezugsobjekte interpretiert.

Software-Systeme und die darin vorkommenden Bausteine sind bei dieser Betrachtung also Zeichen-Artefakte (Repräsentationen, rechts im Dreieck), die einen Weltausschnitt und darin vorkommende Gegenstände (Bezugsobjekte, links im Dreieck) repräsentieren und damit Bezug auf diese nehmen. Modelle sind "Konzeptionen" (oben im Dreieck) und dienen u.a. dazu, diesen Bezug explizit zu machen, zu dokumentieren, zu diskutieren, zu verfeinern etc. Beim Entwicklungsprozess bilden sie die Grundlage für verschiedene mögliche Repräsentationen. Dafür verantwortlich sind die System-Entwickler, die Weltausschnitte für ihre Zwecke abstrahieren, in (Nachbild-) Modelle abbilden und diese dann als Vorbilder für die Umsetzung in Computer-Repräsentationen nutzen. Die beiden oberen Kanten des Dreiecks symbolisieren diese Übergänge: vom Bezugsobjekt, dem "A(nwendungs)-Original" über die Konzeption im Modell zur Software-Repräsentation - dem "S-Original". Die dritte Kante des Dreiecks können wir als Einsatz und damit Interpretation des S-Originals im Umfeld des A-Originals deuten: Das Software-System wird in die Anwendungswelt eingepflanzt ("implementiert") und dort im Einsatz überprüft ("validiert")

Natürlich ist das S-Original selbst auch ein Modell (des "A-Originals" oder eines Teils davon). Trotzdem ist es als Nachbild und Ziel des ganzen Entwicklungsprozesses auch "Original". Wir können diese Situation vielleicht am ehesten mit der eines malenden oder bildenden Künstlers vergleichen: Auch er arbeitet mit zwei "Originalen" – seinem darzustellenden Gegenstand (der verwirrender Weise in diesem Kontext oft als

"Modell" bezeichnet wird, etwa in der Redeweise vom "Modell stehen") und dem fertigen Kunstwerk. Dazwischen stehen oft Skizzen, Entwürfe, Blaupausen, die den Transformationsprozess vom (realen) Original zum Kunstwerk-Original dokumentieren und deren Merkmale sich schrittweise vom Ausgangs- zum Ziel-Original hin verändern. In analoger Weise lässt sich die Software-Entwicklung als Folge von Modelltransformationen vom A-Original über verschiedene Analyse-, Entwurfs- und Implementierungsmodelle hin zum S-Original auffassen.

4 Konsequenzen für Terminologie und Metamodelle

Die verschiedenen Beziehungen zwischen Modellen und Originalen werfen auch tief liegende terminologische Fragen auf. Was ist z.B. eine "Entität"? Ist sie das (A-) Original – also in unserem obigen Beispiel die Person aus Fleisch und Blut – ist sie eine der Modellversionen auf dem Wege zur Implementierung oder gar das S-Original – eine Bitsequenz in der gespeicherten Kundendatei oder eine Pixel-Anordnung, die die Person auf dem Bildschirm repräsentiert? Oder anders gefragt: Was können wir über die Entität "Erika Mustermann" aussagen? Dass sie 171 cm lang, weiblichen Geschlechts und am 5.6.1978 geboren ist oder dass sie aus 7 (oder 11?) Attributen besteht, in der Datei *$Kunden* abgelegt ist und im Speicher 50 KByte belegt?

Genau genommen haben wir es mit vielen verschiedenen Entitäten zu tun, die ganz unterschiedlichen Welten angehören. Der Abb. 5 folgend unterscheiden wir 3 Welten und die darin befindlichen Entitäten: *A-Entitäten* (wie unsere Original-Musterfrau, aber auch so abstrakte Dinge wie Konten, Verträge, Bilanzen oder Geschäftsprozesse), *M-Entitäten* (z.B. konkretisierte Objekt-Diagramme für die in Abb. 4 dargestellten Klassen) und *S-Entitäten* wie z.B. eine Datei, eine Zeile in einer Datenbank-Tabelle oder eine ausgefüllte Bildschirmmaske.

A-Entitäten haben i.a. (potentiell) unendlich viele Attribute: Wir können z.B. unsere Musterfrau mit immer feineren, noch weiter detaillierten Merkmalen beschreiben. Jede dieser endlichen Beschreibungen stellt aber schon selbst notwendigerweise eine Abstraktion und damit ein Modell dar. Also entziehen sich A-Entitäten (außer in trivialen, technischen Ausnahmefällen) unserer vollständigen Erfassung: Sobald wir sie "realisieren" – d.h. in unsere vorgeblich "reale" Modell- und Computer-Welt überführen – werden sie der (Anwendungs-) Realwelt enthoben und damit "irreal". "Realisierung" und Computerisierung sind somit zwangsläufig Schritte der Entfernung von der Realität. Womit keineswegs ausgeschlossen sein soll, dass die entstehenden Software-Systeme Nützliches leisten, Teil einer neuen Realität werden und dort sehr reale Dienste übernehmen können. D.h. auch die S-Objekte werden letztendlich Teil der Realwelt und damit zu neuen A-Objekten, weisen aber als Artefakte in der Regel nicht die genannten Beschreibungsprobleme auf.

Ähnliche Unterscheidungen wie bei "Entität" sind natürlich ebenfalls bei Begriffen wie "Objekt", "Gegenstand", aber auch bei "Vorgang", "Aktivität", "Prozess" etc. möglich. Beim Objektbegriff ist die Begriffsverwirrung besonders eklatant und auffällig (vgl. [H-B 01]): Die Objekte, von denen in der Analyse, z.B. bei den Anforderungen und in

den Anwendungsfällen die Rede ist, sind in der Regel A-Objekte. Objekte im UML-Objektdiagramm oder am Kopf der *"Lebenslinien"* im Interaktionsdiagramm sind von Haus aus M-Objekte. Beziehen wir allerdings "lebensweltliche" Akteure mit ein, so machen hier auch A-Objekte Sinn. Dagegen sind die Objekte, aus denen man Systeme zusammensetzt, von denen man also bei der Komponenten-Technologie oder bei Kompositions-Werkzeugen spricht, S-Objekte.

Müssen wir also in Zukunft alle unsere Entitäten, Objekte, Vorgänge, Prozesse mit A-, M- oder S- indizieren, um deutlich zu machen, welcher Welt sie angehören sollen? Wie bei manchen ähnlich gelagerten terminologischen Problemen wird das nicht praktikabel und auch nicht immer notwendig sein, da wir meistens aus dem Kontext die richtige Zuordnung vornehmen können. Wenn wir uns allerdings präzise ausdrücken wollen oder wenn die Zuordnung aus dem Kontext nicht unmittelbar klar ist – wie z.B. bei der Beschreibung von Modell-Transformationen -, kann eine solche Unterscheidung sehr hilfreich sein und viele Missverständnisse vermeiden helfen.

Eine damit eng zusammenhängende Frage betrifft die Metamodelle. Die folgende Abb. 6 zeigt einen kleinen Ausschnitt aus dem UML (1)-Metamodell für "classifier". Müssen wir jetzt *"class"* in *A-class, M-class* und *S-class* differenzieren - entsprechend der drei genannten Kategorien von Objekten? Fangen wir mit dem Einfachen an: M-Klassen haben wir selbstverständlich – das ist der Standardfall für unsere Entwicklungs- und Modelltransformations-Werkzeuge, wo wir im Wesentlichen M-Objekte manipulieren. S-Klassen finden wir auch schon im Metamodell – unter dem Stichwort "component" – deren Exemplare sind gerade die Bausteine, aus denen wir unsere Systeme bauen.

Es bleiben die A-Klassen. Ihre Exemplare sind die Objekte unserer Anwendungs-Welt. Wie wir oben gesehen haben, entziehen sich diese in der Regel unserem direkten Zugriff. Wir können uns ihnen nur durch Beschreibungen nähern – dafür bietet das Metamodell zunächst einmal Anwendungsfälle (*"use cases"*, vgl. Abb. 6) an. Allerdings steht ein Anwendungsfall meist nicht für eine (homogene) Klasse von Anwendungs-Objekten, sondern beschreibt ein Stück Funktionalität, die sich durchaus auf mehrere Anwendungs-Objekte erstrecken kann.

Es sieht also auf den ersten Blick so aus, als ob im Metamodell für A-Klassen keine Notwendigkeit bestünde, da sich ihre Exemplare, die A-Objekte sowieso unserem Zugriff entziehen. Um das genauer zu prüfen, wollen wir drei Fälle unterscheiden. Tatsächlich existiert eine solche Notwendigkeit immer dann nicht, wenn sich die Anwendung schon selbst auf artifizielle, total spezifizierte Objekte bezieht – in anderen Worten, wenn M-Objekte oder S-Objekte selbst Gegenstände der Anwendung sind. Das ist z.B. bei den Objekten von "Metasystemen" wie Software-Werkzeugen, Klassenbibliotheken oder Dateiverwaltungsprogrammen der Fall. Für solche Objekte braucht man offensichtlich keine zusätzliche A-Klassen-Kategorie.

Der zweite Fall betrifft den Ersatz von A-Objekten durch *Repräsentanten*. Wie könnte ein Repräsentant für unsere o.g. Muster-Person aussehen? Es müsste eine "möglichst konkrete" Beschreibung sein, d.h. eine, die so viele Merkmale und Einzelheiten wie möglich über die konkreten Exemplare aufzunehmen imstande ist. Die oben angespro-

chenen Anwendungsfälle weisen zwar in diese Richtung, können aber das Gewünschte in der Regel nicht leisten. So sind sie auf ein Anwendungs-Projekt begrenzt, beschreiben Funktionalität und sind damit zur Objektbeschreibung nur indirekt geeignet.

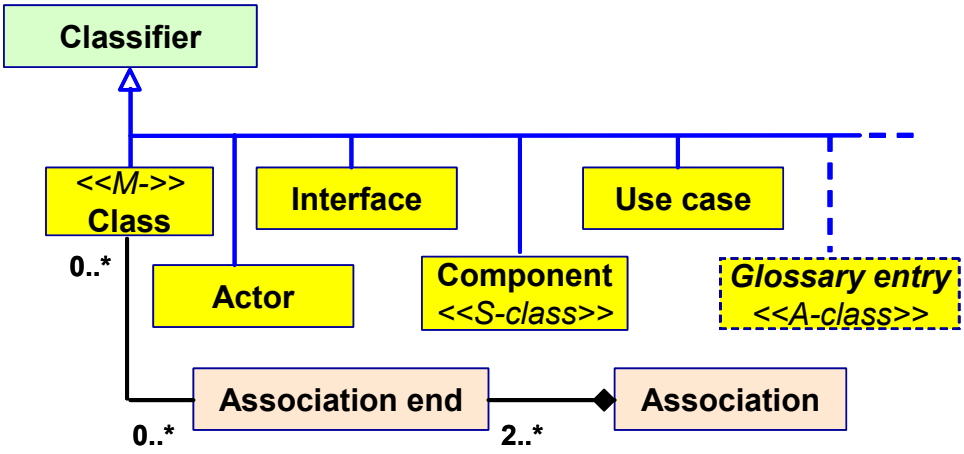

Abb. 6: UML-Metamodell für "classifier" (erweiterter Auszug, n. [Neu 02])

Vielmehr können wir uns Glossar-artige Beschreibungen als Repräsentanten für A-Objekte vorstellen, wie sie z.B. in den Ansätzen KCPM oder ORM vorgeschlagen werden (vgl. [M-K 02] [SMJ 02]). In solche offenen, "potentiell unendlichen" Beschreibungen kann man alles nur erdenklich Wissenswerte über eine Klasse von A-Objekten aufnehmen. Eine so geartete "A-Klasse" ist also ein offenes, projekt-neutrales Glossar zur Beschreibung einer Klasse von A-Objekten, das für die Modellierung – genauer: für die Abstraktion zu M-Objekten – alle Möglichkeiten offen lässt. In ein entsprechend erweitertes Metamodell (s. Abb. 6) müsste man als zusätzliche Alternative zu *classifier* so etwas wie *glossary entry* aufnehmen.

Für A-Objekte können wir uns natürlich noch – das ist der dritte Fall – andere Repräsentanten vorstellen, wie z.B. Webseiten für einzelne Personen, Artikel oder Reiseangebote. Oder – wenn wir die neueste, sich gerade ausbreitende Technologie betrachten – RFID-Chips, die eine Stellvertreter-Rolle für die sie tragenden A-Objekte übernehmen sollen – also z.B. für bestimmte Produkte in einem Supermarkt. Was könnte eine sinnvolle A-Klasse – also ein Platzhalter für eine Menge solcher A-Objekte sein? Hier können wir zwei Standpunkte einnehmen: einen *extensionalen* oder eine *intensionalen*. Im ersten Fall ist die Klasse eine Menge von *links* – auf Webseiten, auf RFID-Chips - die es uns erlaubt, alle Objekte der Klasse zu identifizieren und ggf. anzusteuern. Für eine intensionale Charakterisierung ist wiederum das Glossar eine geeignete Form: es dient dazu, alles mögliche Wissen über die besagte Menge von Objekten in möglichst offener und erweiterbarer Form zu erfassen.

Fassen wir also zusammen: Modelle bilden bei der Software-Entwicklung das entscheidende Bindeglied zwischen den Objekten der Anwender- und der Software-Welt. Sie weisen (wie der römische Türgott Janus) in beide Welten und können einerseits Merkmale der abgebildeten Anwendungs-Objekte, andererseits Merkmale der repräsentierenden Software-Objekte – oft auch in gemischter Form - enthalten. Will man eine Mischung vermeiden und die Transformationsprozesse zwischen den "Welten" verdeutlichen, so empfiehlt sich die Unterscheidung von Anwendungs- (A-), Modell- (M-) und Software- (S-) Objekten. Objekte der "Realwelt" (vom Typ "A" oder später auch "S"), die sich einem direkten Zugriff entziehen, lassen sich möglicherweise durch Stellvertreter-Objekte (z.B. Webseiten, *links* oder RFID-Chips) repräsentieren.

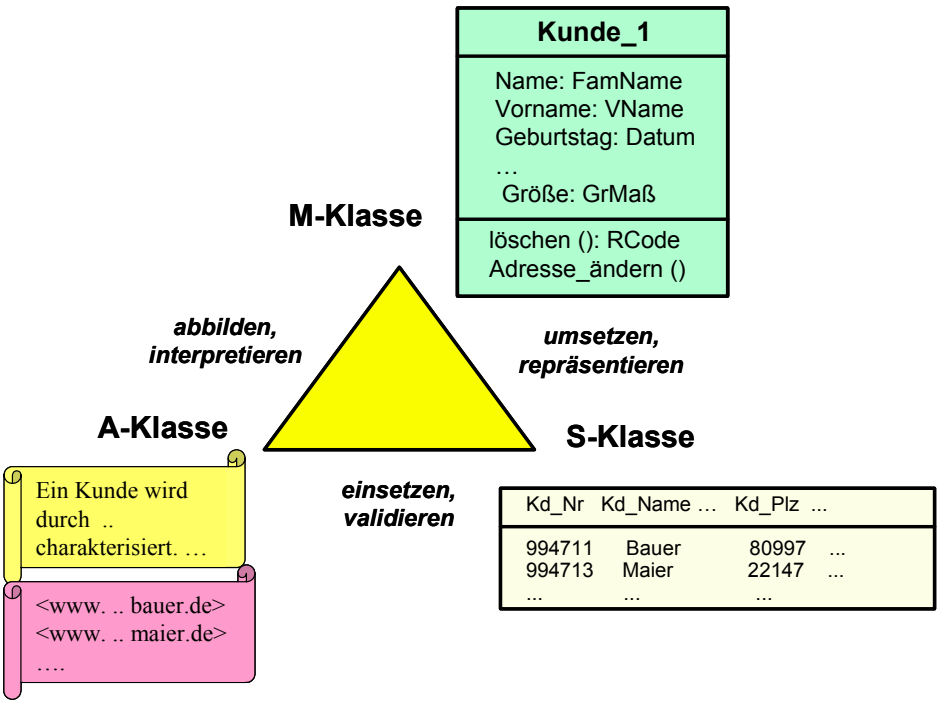

Abb. 7: Drei Kategorien von Klassen für Anwendungs-, Modell- und Software-Objekte

Die genannte Klassifizierung von Objekten induziert eine ebensolche von Klassen. Während M- und S-Klassen im UML-Metamodell bereits einen festen Platz haben, bilden die *use cases* bislang nur einen unzureichenden und wenig passenden Ersatz für A-Klassen. Im erweiterten Metamodell (Abb. 6) werden sie durch Glossar-Einträge (für eine offene, intensionale Sicht) und ggf. durch Aufzählungen bzw. Verweise auf die zugehörigen (Stellvertreter-) Objekte (extensionale Sicht) ergänzt. In Abb. 7 sind die drei Kategorien von Klassen – illustriert am Standard-Beispiel "Kunde" - im Softwaretechnik-Dreieck dargestellt.

5 Eine MDA-Erweiterung: Ontologien als A-Klassenbibliotheken

Der MDA-Ansatz beinhaltet schon eine ähnliche Kategorisierung von Modellen, wie wir sie zwischen M- und S- Objekten (und den entsprechenden Klassen) vorschlagen. Ein PIM entspricht unseren (Plattform-unabhängigen) M-Objekten und -Klassen, während ein PSM Vorlage für (Plattform-abhängige) S-Objekte und -Klassen in unserem Sinne sein kann. Die rechte Seite des Dreiecks auf Abb. 7 wird also durch PIM und PSM's hinreichend abgedeckt. Dagegen sind die Transformationsprozesse auf der linken Seite des Dreiecks noch nicht in den Blickpunkt der MDA-Initiative gerückt. Hier scheint uns eine weitere Unterscheidung hilfreich, wenn nicht sogar notwendig.

Modelle sind gezielte Abstraktionen von Realwelt-Gegenständen. "Gezielt" deswegen, weil sie einen bestimmten Projekt- oder Anwendungszweck verfolgen und dieser notwendiger Weise den Ausschlag gibt, was als relevant erachtet wird und ins Modell eingehen soll und was nicht. Modelle sind also immer projekt-spezifische Anwendungssichten – analog den externen Modellen im Drei-Schichtenmodell für Datenbanken. Oder in der MDA-Terminologie: Modelle sind (als PIM) Plattform-unabhängig, aber Anwendungs(projekt)-spezifisch (ASM). Will man eine Domäne projektübergreifend modellieren, so braucht es dazu ein Anwendungs-unabhängiges Modell (*application-independent model*, *AIM*). Dies kann man sich als eine Sammlung von A-Objekten und A-Klassen – z.B. in der oben vorgeschlagenen Form und als Glossar organisiert – vorstellen.

Das ist nun gerade auch der Ansatzpunkt von *Ontologien* in der Softwaretechnik [Hes 02, Hes 05]. Nach T.Gruber ist eine Ontologie "*a formal explicit specification of a shared conceptualization"* [Gru 93]. Wichtig ist hier die Kennzeichnung *"shared"*. Sie unterscheidet gerade Ontologien als projekt-übergreifende AIM's von den "normalen" ASM-Modellen. Inwieweit das "formal" und "explizit" geschehen kann, hängt von der Zielsetzung ab, mit der man eine Ontologie aufstellt. Soll sie eine Domäne wirklich unabhängig von speziellen Anwendungsprojekten beschreiben, so erfordert das eine besondere Offenheit: Prinzipiell muss sie potentiell unendlich viele Merkmale aufnehmen können, wie wir es oben von unseren A-Klassen gefordert haben. Eine solche Beschreibung wird aus pragmatischen Gründen nie vollständig formal sein können – und explizit ist sie nur in Bezug auf die bereits erfassten Merkmale. Zu diesen können immer noch weitere, zunächst implizite Merkmale hinzukommen, wenn man sich dafür entscheidet, sie explizit zu machen. Glossare erfüllen nach unserer Ansicht diese Forderungen am besten. Allerdings müssen sie in einem wohldefinierten Grundgerüst ontologischer Kategorien verwurzelt sein [Gua 98], [Gui 05].

Folgt man dieser Sicht, so steht am Beginn eines Software-Projekts neben dessen spezifischen Anforderungen eine "Ontologie", d.h. eine verallgemeinerte, Glossar-artige Bibliothek von A-Klassen als Domänen-Beschreibung (vgl. [H-K 04]). Sie spielt – in der oben erweiterten MDA-Terminologie – die Rolle des AIM und ist im Wesentlichen ein *Nachbild* der Anwendungs-Domäne. Aus ihr werden – geleitet durch die Anforderungen – in einer Folge von Transformationsschritten weitere Modelle abgeleitet: z.B. ein projekt-spezifisches, aber Plattform-unabhängiges ASM/PIM (mit

*Vor- und Nachbild*charakter) und daraus ein Plattform-abhängiges PSM, das als *Vorbild* für das entstehende System dienen kann. Aus dem ASM/PIM kann aber auch neues, zusätzliches Domänen-Wissen in das AIM zur weiteren Verwendung in anderen Projekten einfließen. Eine entsprechende Erweiterung der MDA-Grundstruktur ist in Abb. 8 dargestellt.

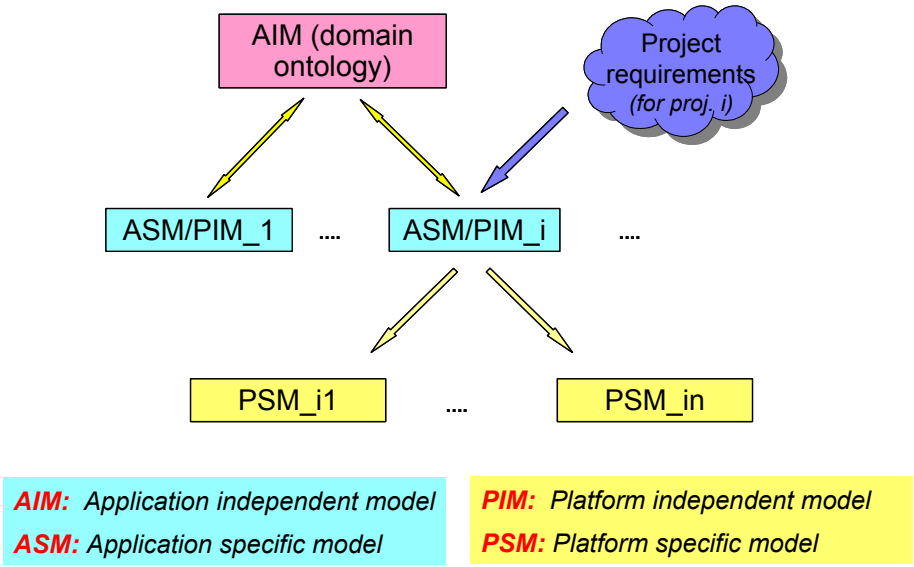

Abb. 8: MDA, erweitert auf der Grundlage von Ontologien

6 Fazit und Ausblick

Der MDA/MDD-Ansatz hat den Modellen in der Software-Entwicklung zur zentralen Bedeutung verholfen. Sie haben – das unterscheidet sie von den Spezifikationen – in gleichem Maße (aber in unterschiedlichen Ausprägungen) Vor- und Nachbildcharakter. Während die Übergänge vom Projekt-spezifischen, Plattform-unabhängigen Modell (PIM) zur Implementierung (PSM) Gegenstand umfangreicher Forschungen sind und sich gute Möglichkeiten zu deren (teilweiser) Automatisierung erwarten lassen, ist der Weg vom (oft auf viele Köpfe und Dokumente verteilten) Domänenwissen und von den Anforderungsbeschreibungen hin zum PIM noch weit weniger erschlossen.

Eine notwendige Voraussetzung dazu ist eine klare, übersichtliche, gut strukturierte, an Standards orientierte Dokumentation dieses Wissens – z.B. in Form von Glossaren -, die wiederum auf einer klaren, eindeutigen und systematischen Terminologie beruht. Dies beinhaltet z.B. eine sichtbare Trennung von Elementen der Anwendungs-, Modell- und (Software-) Systemwelt, die auch zur Beschreibung der notwendigen Transformationen unentbehrlich ist. Eine entsprechende Erweiterung des UML-Metamodells wäre wünschenswert.

Darüber hinaus wird eine Ausweitung des MDA-Ansatzes vorgeschlagen, welche die "frühen" Übergänge von Domänen- und Anforderungsbeschreibungen bis zum PIM in den Modelltransformations-Prozess einbezieht. Dabei könnten Ontologien, die das langlebige, projekt-übergreifende Wissen über bestimmte Anwendungsbereiche bündeln und dokumentieren, eine hervorragende Rolle spielen und zur Wiederverwendung nicht nur von gebrauchsfertigen Software-Komponenten, sondern auch von Analysen, Modellen und Entwürfen beitragen.

Eine solcherart erweiterte *OBMDA* (*Ontology-Based Model Driven Architecture*) könnte damit – dem Gotte Janus folgend - nicht nur den Blick nach drinnen (in die Modell- und Software-Welt), sondern auch nach draußen (in die Domänen- und Anwendungs-Welt) unterstützen und dazu passende Transformationswege eröffnen.

Dank

Für die kritische Durchsicht des Manuskripts und hilfreiche Kommentare danke ich Herrn Thomas Kühne aus Darmstadt sowie für weitere Anregungen den mir unbekannten Gutachtern dieses Beitrags.

Literaturhinweise:

[Amb 05] S. W. Ambler: Agile Modeling (AM) Home Page: Effective Practices for Modeling and Documentation. http://www.agilemodeling.com/ (15.11.2005)

[FHL 98] E. Falkenberg, W. Hesse, P. Lindgreen, B.E. Nilsson, J.L.H. Oei, C. Rolland, R.K. Stamper, F.J.M. Van Assche, A.A. Verrijn-Stuart, K. Voss: FRISCO - A Framework of Information System Concepts - The FRISCO Report. IFIP WG 8.1 Task Group FRISCO. Web version: http://www.mathematik.uni-marburg.de/~hesse/papers/fri-full.pdf (1998)

[Gru 93] T. Gruber: A translation approach to portable ontologies. Knowledge Acquisition, 5(2), pp. 199-220 (1993)

[Gua 98] N. Guarino: Formal Ontology and Information Systems. In: Proc. FOIS '98, Trento (Italy) June 1998, Amsterdam IOS Press pp 3-15

[Gui 05] G. Guizzardi: Ontological Foundations for Structural Conceptual Models. Ph. D. thesis, Univ. of Twente 2005

[H-B 01] W. Hesse, H. v. Braun: Wo kommen die Objekte her? Ontologisch-erkenntnistheoretische Zugänge zum Objektbegriff. In: K. Bauknecht et al. (eds.): Informatik 2001 - Tagungsband der GI/OCG-Jahrestagung, Bd. II, S. 776-781. books_372ocg.at; Bd. 157, Österr. Computer-Gesellschaft 2001

[H-K 04] W. Hesse, B. Krzensk: Ontologien in der Softwaretechnik. Proc. Workshop "Ontologien in der und für die Softwaretechnik" bei der Modellierung 2004 in Marburg. GI / Univ. Marburg, März 2004

[Hes 02] W. Hesse: Das aktuelle Schlagwort: Ontologie(n). in: Informatik Spektrum, Band 25.6 (Dez. 2002)

[Hes 05] W. Hesse: Ontologies in the Software Engineering process. in: R. Lenz et al. (Hrsg.): EAI 2005 - Tagungsband Workshop on Enterprise Application Integration, GITO-Verlag Berlin 2005 und: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-141/

[Jac 02] M. Jackson: Some Basic Tenets of Description. Software and Systems Modeling (SoSym), Vol. 1, no. 1 pp. 5 - 9

[Lud 02] J Ludewig: Modelle im Software Engineering - eine Einführung und Kritik. In: M. Glinz et. al (Hrsg.): Proc. Modellierung 2002. Springer LNI P-12 (2003)

[M-K 02] H.C. Mayr , Ch. Kop: A User Centered Approach to Requirements Modeling. In: M. Glinz, G. Müller-Luschnat (Hrsg.): Modellierung 2002 - Modellierung in der Praxis - Modellierung für die Praxis, pp. 75-86, Springer LNI P-12 (2003)

[M-M 03] J. Miller, J. Mukerji: MDA Guide. Version 1.1.1, Object Management Group 2003.

[Neu 02] H.A. Neumann: Analyse und Entwurf von Softwaresystemen mit der UML (2. Aufl.). Carl Hanser Verlag 2002

[Sce 99] P. Schefe: Softwaretechnik und Erkenntnistheorie. Informatik-Spektrum Bd. 22, S. 122-135 (1999)

[SMJ 02] P. Spyns, R. Meersman, M. Jarrar: Data modelling versus Ontology engineering, SIGMOD Record 31 (4), Dec. 2002

[Sta 73] H.Stachowiak: Allgemeine Modelltheorie. Springer, Wien 1973