”Realisierung eines metamodellbasierten Entwurfswerkzeuges für BDI-Agentensysteme”

Diplomarbeit an der Universität Hamburg
Fachbereich Informatik

Vorgelegt von Henry Becker
Hamburg, den 08.01.2005

Betreuer:

Prof. Dr. Winfried Lamersdorf
und
Dr. Daniel Moldt

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

Hamburg, den 08.01.2005

Inhaltsverzeichnis

1 Einleitung

Betrachtet man die rasante Entwicklung der Rechenleistung und der Speicherkapazität von Computersystemen sowie den steigenden Grad ihrer Vernetzung untereinander innerhalb der letzten Jahrzehnte, und prognostiziert gemäß dem Moor'schen Gesetz auch eine weitere Verdoppelung der Systemleistung alle 18 Monate, so wird einem schnell bewusst, daß sich die Grenzen dessen, was technisch mit vertretbarem Aufwand realisierbar ist, immer weiter verschiebt und den Einsatz der Informatik in immer komplexer werdenden Anwendungsbereichen ermöglicht.

Um der steigenden Komplexität moderner Anwendungen auch auf der softwaretechnischen Ebene nachzukommen, hat sich in den letzten Jahren das Konzept der Agenten und der multiplen Agenten als besonders vielversprechend in den Blickpunkt des allgemeinen Interesses gestellt. Die ursprünglich im Fachgebiet der Verteilten Künstlichen Intelligenz angesiedelte Agententheorie zieht mittlerweile größere Kreise und hat in zahlreichen Anwendungsgebieten wie dem Verteiltem Problemlösen, der Simulation komplexer Sachverhalte bis hin zur Robotik Einzug gehalten. Bereichert werden die theoretischen Überlegungen zu den Agenten auch durch Arbeiten auf den Gebieten der Kognitionswissenschaften oder der Philosophie, welche dem Agenten ein rationales Verhalten basierend auf einfachen mentalen Konzepten ermöglicht.

Das breite Spektrum der verschiedenen theoretischen Ansätze und Vorschläge in diesem Gebiet, als auch die steigende Zahl ihrer praktischen Implementationen, wie z.B. Agentenplattformen oder Agentenframeworks, stellt den Entwickler agentenbasierter Software vor eine schwierige Aufgabe. Im Gegensatz zur traditionellen Softwareentwicklung muß sich beim Entwurf eines Agentensystems zusätzlich in die vom jeweils eingesetzten Agentenframework bereitgestellten Konzepte eingearbeitet und sich dabei mit vielen technischen Details auseinandergesetzt werden. Dieser Umstand wird durch den derzeit noch bestehenden Mangel an etablierten Entwicklungsmethoden, vorhandenen Erfahrungen und Referenzprojekten verstärkt. Bei der Förderung der Akzeptanz eines Agentenframeworks kann ein geeignetes Werkzeug, welches den Entwicklungsprozess unterstützt, eine wesentliche Rolle spielen und dazu beitragen das Paradigma der agentenorientierten Softwareentwicklung einem größeren Publikum nahezubringen.

Im Bereich der objektorientierten Softwareentwicklung haben sich derartige Entwurfswerkzeuge zur Modellierung komplexer Strukturen und Vorgänge längst bewährt. Diese bieten eine durchgängige Unterstützung bei der Realisierung großer Softwareprojekte, indem sie das grafische Erstellen von Modellen, meist in Form von Diagrammen der Unified Modelling Language, unter Berücksichtigung verschiedener Abstraktionsebenen und Sichtweisen auf das zu entwickelnde System, ermöglichen. Diese von der Objekt Managment Group standardisierten Diagramme dienen nicht nur dazu eine gemeinsame Diskussionsbasis bei allen am Entwicklungsprozess beteiligten Personen zu etablieren, sondern können gleichzeitig für die Implementation, als Ausgangspunkt automatisierter Codegenerierung, genutzt werden.

Wie sich die Vorteile und die Erfahrungen mit derartigen modellgestützten Entwicklungsprozessen auf das Gebiet der Multiagentensysteme übertragen lassen, wird derzeit sowohl im universitären, als auch in industriellen Forschungsprojekten untersucht.

1.1 Zielsetzung der Arbeit

Im Rahmen dieser Arbeit sollen geeignete Möglichkeiten der Werkzeugunterstützung zum Entwurf und zur Modellierung der internen Strukturen von Agentenarchitekturen untersucht werden. Dabei wird konkret der Ansatz der „Model Driven Architecture“ verfolgt. Ziel dieser Untersuchung ist ein Entwurfswerkzeug für Agentensysteme, welches sich von bisherigen, bereits existierenden Werkzeugen darin unterscheidet, daß es auf einem leicht modifizierbarem, erweiterungsfähigem Metamodell basiert. Dieses Metamodell soll sämtliche statischen Aspekte der mentalen Konzepte „Belief“, „Plan“, „Goal“, sowie die des Agenten bzw. seiner Capability selbst enthalten. Das Entwurfswerkzeug soll anderen Werkzeugen einen standardisierten Zugriff auf das Metamodell und auf die Agentenmodelle selbst ermöglichen, welche damit in die Lage versetzt werden, adäquate Agentenspezifikationen zur Interpretation durch das Agenten-Framework „Jadex“ u.a. zu generieren. Um das in dieser Arbeit zu realisierende Entwurfswerkzeug auch an potentielle Änderungen des noch in der Entwicklung befindlichen Jadex-Frameworks und des Jadex zugrundeliegenden BDI-Agentenmetamodells angepassen zu können, sind Anforderungen bezüglich der Flexibilität und Wiederverwendbarkeit aufzustellen und umzusetzen.

1.2 Gliederung der Arbeit

Die vorliegende Diplomarbeit untergliedert sich in einen Grundlagenteil und in einen praktischen Teil. Dabei werden die zum besseren Verständnis der Arbeit benötigten Grundlagen ausführlich in den Kapiteln 2 und 3 erörtert.

Kapitel 2 konzentriert sich insbesondere auf die Themaktik der Softwareagenten, ihrer zugrundeliegenden Architekturen, sowie ihrer technischen Umgebungen der Agentenplattformen. Im Mittelpunkt dieser Überlegungen stehen die mentalen Konzepte kognitiver Agenten, wie Beliefs, Desires und Intentions. Die Integration dieser Konzepte in existierende Agentenframeworks wird anschließend am Beispiel, des ebenfalls am Fachbereich Informatik der Universität Hamburg entwickelten, Frameworks „Jadex“ dargelegt.

In Kapitel 3 werden die Konzepte „Modellierung“ und „Metamodellierung“ im Kontext der von der *Object Managment Group* vorgeschlagenen MDA (*Model Driven Architecture*) vorgestellt und die von der MDA verwendeten Standards: MOF *Meta Object Facility*, XMI *XML Metadata Interchange* und *UML Profile for MOF* untersucht, welche die technischen Grundlagen für den anschließenden praktischen Teil darstellen.

Der praktische Teil dieser Arbeit, der aus den Kapiteln 4 und 5 besteht, beschreibt das Vorgehen bei der Realisierung des metamodellbasierten Entwurfswerkzeuges für BDI-Agentensysteme. Sämtliche konzeptuellen Überlegungen zur Architektur des Werkzeuges, sowie ein zugehöriger prototypischer Entwurf entfallen dabei auf Kapitel 4. Weiterhin wird in Kapitel 4 erörtert, wie sich ein BDI-Agentenmetamodell für „Jadex“, mit Hilfe eines geeigneten UML-Tools erstellen läßt.

Auf diesen Entwurf aufbauend stellt Kapitel 5 die Herangehensweise bei der Implementation des Entwurfswerkzeuges dar. Ebenso wird in diesem Kapitel beschrieben, wie das Werkzeug zur Modellierung von BDI-Agenten eingesetzt werden kann.

2 Grundlagen

Dieses Kapitel bietet einen Überblick über das facettenreiche Gebiet der Agentensysteme und der Multiagentensysteme. Dabei steht die Klärung der Begriffe Agent, Agentenplattform und Agentenframework, wie sie in dieser Arbeit verwendet werden, im Vordergrund. Darauf aufbauend werden dann die mentalen Konzepte Belief, Desire und Intention analysiert und ihre konkrete Umsetzung an Beispielen näher erläutert. Anschließend wird Jadex als ein exemplarisches Beispiel für ein Framework zur Unterstützung von BDI-Agentenarchitekturen vorgestellt.

Ein gemeinsames, klares Verständnis des Agentenparadigmas ist gerade in diesem noch jungen Forschungsgebiet eminent wichtig, damit keine zu hohen Erwartungen mit agentenorientierten Ansätzen verbunden werden, die in der Realität nicht eingehalten werden können. Ein ähnliches Schicksal ereilte bereits dem Gebiet der künstlichen Intelligenz, welches als ein historischer Ursprung der Agententheorie betrachtet wird.

Auch die Frage nach dem Unterschied zwischen Agenten und bisherigen Programmen muß klar beantwortet werden, um konzeptuellen Fehlern bei Entwurf und Implementation agentenbasierter Softwaresysteme vorzubeugen. Wird das Agentenparadigma als generelle Lösung des Softwareentwicklungsproblems betrachtet, und jede Entität der Anwendungsdomäne oder der Software als Agent angesehen, führt dies meist zu komplexen unüberschaubaren Systemen mit zu vielen Agenten (siehe auch [Küh01] S.207), ohne einen klar erkennbaren Mehrwert gegenüber herkömmlichen Softwarelösungen zu bieten.

2.1 Agenten

Der Begriff Agent existiert in vielen Forschungszweigen der Informatik, aber auch in den Wirtschaftswissenschaften, in der Soziologie oder gar in der Psychologie. Derzeit hat sich eine klare und allgemeingültige Begriffsdefinition noch nicht durchgesetzt. Dies liegt insbesondere an der Tatsache, daß sowohl der Kontext in dem dieser Begriff

benutzt wird, als auch der Zweck einer solchen Definition zwischen den verschiedenen Autoren stark variiert.

Stattdessen finden sich in der Fachliteratur eine ganze Reihe von Ansätzen das Konzept „Agent“ unter verschiedenen Fragestellungen und Sichtweisen einzugrenzen.

> *At this moment, there is every appearance that there are more definitions than there are working examples of systems that could be called agent-based.* [Her96]

Eine weit verbreitete Vorstellung von einem Agenten ist die von einem System, ob künstlicher oder biologischer Natur, das etwas tut, also handelt bzw. agiert. Diese für konkrete Fragestellung recht unspezifische Sichtweise spiegelt sich auch in der Definition von Russel und Norwig [SR95] wieder.

Definition 1 *An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.*

Damit ein Agent jedoch auch sinnvoll, also gemäß seiner „Vorsehung“, handelt, bedarf er einiger grundlegender Eigenschaften. Zwei zentrale Eigenschaften, die in der Forschergemeinde am meisten Zuspruch finden, sind die der „Autonomie“ und der „Intelligenz“. Eine Definition für einen *autonomen* Agenten geben Stan Franklin und Art Graesser in [FG96]:

Definition 2 *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*

Ein autonomer Agent zeichnet sich demnach durch seine Fähigkeit aus, auf die von ihm wahrgenommene Umwelt autonom, also unabhäng von einer Fremdkontrolle, beispielsweise durch Menschen oder andere Systeme, unmittelbar Einfluß zu nehmen. Diese Einflußnahme kann sowohl eine Reaktion auf bestimmte, den Agenten stimulierende Veränderungen der Umwelt sein, man spricht in diesem Zusammenhang von *reaktiven* Agenten, oder auch selbstmotiviert durch Eigeninitiative des Agenten hervorgerufen werden. Agenten, welche selbständig aktiv werden um ihre Ziele bzw. Vorgaben zu erfüllen bezeichnet man gemeinhin als *proaktive* Agenten. Die Proaktivität ist ein wesentliches Element der Autonomie eines Agenten.

Die Art der Einflußnahme selbst kann eine physische oder rein virtuelle Interaktion mit ebenfalls in der Umwelt situierten Objekten, z.B. mit dem Agenten zugewiesenen Ressourcen sein. Sie kann aber auch durch eine Kommunikation mit anderen Agenten zustande kommen. Die soziale Fähigkeit von Agenten mit ihrer Umwelt zu kommunizieren setzt eine symbolische oder natürliche Sprache — die Agentensprache — voraus und ist ebenfalls charakteristisch für autonome Agenten.

Vereinen Agenten die Eigenschaften Autonomie, Reaktivität und Proaktivität in sich und verfügen darüber hinaus über soziale Fähigkeiten, so sprechen Wooldridge und Jennings in [MW95] von einer „weak notion of agency“. Also einer „schwachen“ Ausprägung des Agentenbegriffes, welche jedoch potentiell weniger umstritten ist, als eine ebenfalls von beiden Autoren angegebene „strong notion of Agency“. Die „strong notion of agency“ schlüsselt weitere charakteristische Merkmale auf, anhand derer ein System als Agent identifiziert werden kann. Diese zielen darauf ab, den Agenten zu „vermenschlichen“, bzw. sein Verhalten dem eines Menschen anzunähern. Sie bedingen damit, daß der Agent über mentale Zustände wie Wissen, Wünsche oder Überzeugungen verfügt.

Diese Zustände ermöglichen dann die Realisierung von Agenten, die aufrichtig, also nicht wissentlich falsche Informationen weitergeben, oder Agenten deren Handlungen nach streng rationalen Kriterien ablaufen. Solch *rationale* Agenten werden demnach nicht, wider besseren Wissens, Aktionen durchführen, die sie von ihren Zielen abbringen, sondern stets deliberativ also zielgerichtet agieren. In der Literatur findet sich daher auch häufig der Begriff des *deliberativer* Agenten. Das Merkmal, daß ein Agent stets wohlwollend agiert, also immer das tut, was ihm aufgetragen wurde und keine wiedersprüchlichen Ziele besitzt, zählt ebenfalls zu dieser stärkeren Begriffsausprägung.

Andere Autoren wie beispielsweise Lenny Foner stellen die „Intelligenz“ als notwendige Eigenschaft für einen Agenten in den Vordergrund, und sprechen demzufolge von „intelligent agents“.

Beide Attribute, das der Autonomie als auch das der „Intelligenz“ haben den Nachteil, daß sie sehr subjektiv und damit schwer einschätzbar sind. Zwar gibt es bezüglich der Intelligenz den sogenannten Touringtest, um bei einem gegebenen System ein intelligentes Antwortverhalten nachzuweisen. Dies bedingt jedoch die Anwesenheit eines aussenstehenden, ebenfalls intelligenten Beobachters des Systems. Dennoch haben sie sich als besonders geeignet erwiesen, den Begriff des Agenten klarer von anderen Konzepten abzugrenzen.

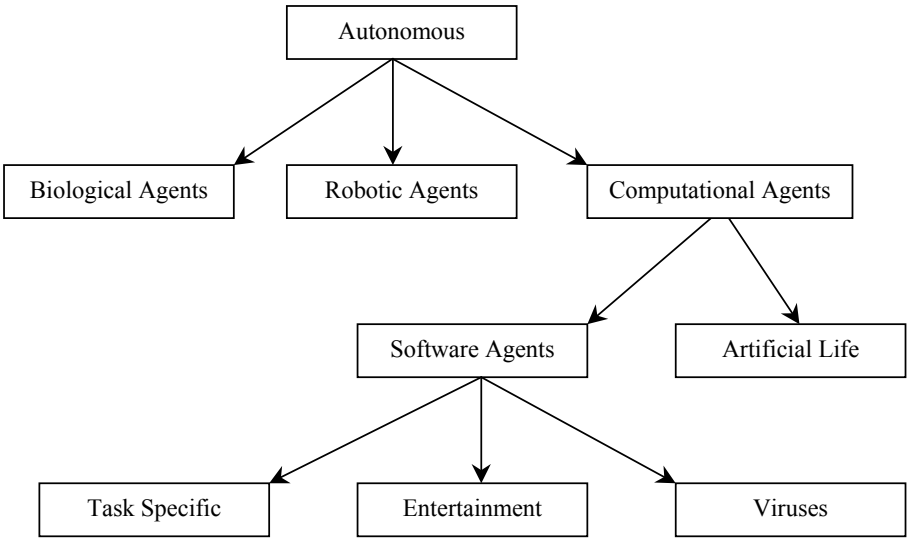


Abbildung 2.1: Agententaxonomie nach [FG96]

2.1.1 Software Agenten

Einen guten Überblick über die verschiedenen Bereiche, in denen sich autonome Agenten vorfinden, vermittelt die Agententaxonomie nach Franklin und Graesser, welche eine Klassifizierung in real existierende Entitäten wie „Biological Agents“, „Robotic Agents“ und in virtuelle Einheiten wie „Computational Agents“ vornimmt (siehe Abbildung 2.1). Mit real existierenden Systemen befassen sich unter anderem Arbeiten von Brustoloni in [Bru91].

Die Softwareagenten bilden demnach eine Untermenge aller derjenigen Systeme, die den Eingangs dargelegten Definitionen genügen. Dennoch umfasst die Menge der Softwareagenten ein weites Spektrum an Ausprägungen und Einsatzbereichen autonomer Agenten.

Hyacinth S. Nwana schlägt in [HSN97] eine Einteilung von Softwareagenten nach realen Einsatzgebieten vor:

Kollaborative Agenten kooperieren mit anderen Agenten, um eine Ziel gemeinsam zu erreichen. Der Benutzer des Systems erteilt einem Agenten eine Aufgabe, die dieser, falls er sie nicht selbst erfüllen kann, vollständig oder nur zum Teil an andere Agenten deligiert. Kooperative Agenten müssen also in der Lage sein, ihre Aktionen untereinander zu koordinieren. Ein Beispiel für den Einsatz kooperativer Agenten ist das Pleiades System.

Interface-Agenten bzw. Schnittstellenagenten dienen ihrem Benutzer als selbständige, persönliche Assistenten. Sie kooperieren in der Regel nicht mit anderen Agenten, sondern nur mit dem Benutzer selbst. Ziel dieser Kooperation ist es, den Benutzer einer Anwendung bei der Bewältigung von Problemen zu unterstützen. Dabei können sie anhand ihres vorgegebenen Wissens über die entsprechende Anwendung selbständig Lösungsvorschläge unterbreiten oder vom Benutzer erteilte Aufträge ausführen. Anhand der Wahrnehmung des Benutzerverhaltens sollte der Interface-Agent in der Lage sein, sich an diesen anzupassen, was eine Lernfähigkeit des Agenten voraussetzt.

Mobile Agenten sind in der Lage innerhalb eines Netzwerks zu migrieren, um beispielsweise mit Ressourcen zu arbeiten, die lokal nicht zur Verfügung stehen.

Information/Internet Agents übernehmen die Aufgabe im Auftrag ihres Benutzers Informationen zu suchen, zu sammeln und zu verwalten. Information Agents können über ein Benutzerprofil verfügen, um die Menge der Informationen nach sinnvollen Kriterien zu filtern. Als Informationsquelle für Information Agents kann z.B. das „world wide web“ dienen.

Anlehnend an die Klassifikation von Wooldrigde und Jennings schlägt Jörg Müller [Mül96] eine Klassifikation von Agenten bezüglich ihrer zugrundeliegenden Architektur, also ihrer internen Struktur, vor. Demnach lassen sich folgende vier Arten von Agentenarchitekturen unterscheiden:

Reaktive Agenten können auf eingehende Ereignisse, wie z.B. Nachrichten, anhand eines vorgegebenen Regelwerkes reagieren. Dabei wählt der Agent eine auf das Ereignis passende Regel aus einer Regelbasis aus, und wendet diese an. Der Vorteil reaktiver Agenten liegt vor allem in den kürzeren Anwortzeiten, da es ihnen an komplexeren Mechanismen zur Regelauswertung fehlt. Rein reaktive Agenten lassen sich vorwiegend für Simulationen von Systemen verwenden, welche aus einer großen Menge ebenfalls relativ einfach gebauter Entitäten bestehen, wie z.B. Ameisenpopulationen.

Deliberative Agenten verfügen im Gegensatz zu den reaktiven Agenten über einen eigenen Planungsalgorithmus zur Erreichung von Zielen. Der Planungsalgorithmus kann komplexere Aktionsfolgen aus einfachen Aktionen zusammenstellen, deren Durchführung überwachen und auf die erzielten Ergebnisse reagieren.

Da deliberative Architekturen im wesentlichen auf kognitionswissenschaftlichen Modellen ihrer Umwelt, ihrer Ziele und ihrer Aktionsfolgen basieren, findet man in der Literatur auch den Begriff der „kognitiven Agenten“ für diese Kategorie von Agenten.

Hybride Agenten besitzen eine interne Struktur, die sowohl deliberative Architekturmerkmale aufweist als auch die Vorteile von reaktiven Agenten bietet. Diese Architektur setzt sich in der Regel aus einem Planungsmodul und einem reaktiven Modul zusammen. Somit wird das entsprechend aufwendiger gestaltete Planungsmodul nicht bei jedem eintretenden Ereignis eingesetzt, sondern nur wenn dieses nicht von dem reaktiven Modul durch die Anwendung einer adäquaten Regel behandelt werden kann bzw. soll. Die Aufteilung von Verantwortlichkeiten auf die beiden Module ist ein praxisrelevantes Problem bei der Konkretisierung hybrider Architekturen.

Interagierende Agenten decken den Bereich der kollaborativen Agenten ab und zeichnen sich dementsprechend durch eine interne Repräsentation von Kooperationsmodellen aus, die sie mit anderen ebenfalls interagierenden Agenten gemeinsam haben.

Da ein deliberativer Agent über eine symbolische Abbildung seiner Umwelt, seiner Pläne bzw. Plansegmente und seiner Ziele verfügt, kann die von Anand S. Rao und Michael P. Georgeff vorgeschlagene BDI-Architektur (siehe [RG95]) als deliberative Architektur bezeichnet werden.

2.1.2 BDI-Agenten

Die philosophischen Grundlagen einer BDI-Architektur lassen sich auf Arbeiten von Michael E. Bratman zurückführen, der sich in [Bra87] besonders mit den Grundzügen rationalen und intentionalen Handelns von Menschen auseinandersetzt. Diese und ähnliche Untersuchungen ergeben, daß sich ein zielgerichtetes Agieren durch ein einfaches Modell mit folgenden Konzepten begründen läßt:

belief Das Belief ist eine Informations-Attitüde, mit der sich die Annahmen eines Agenten über sich selbst und seine Umwelt (also auch über andere Agenten) ausdrücken läßt. Diese Annahmen müssen nicht wahr sein.

desire Das Konzept des Desire drückt die Wünsche und Ziele aus, die der Agent erfüllen „möchte“. Wünsche haben einen schwächeren Charakter als Ziele. Das bedeutet, daß der Agent seinen Wunschzustand solange anstrebt, bzw. erhalten wird, bis dieser Zustand ihn von der Erfüllung anderer Ziele abbringt. Das Desire bezeichnet Bratman als Pro-Attitüde, im Sinne einer antreibenden, motivierenden Haltung.

intention Das Konzept der Intention drückt eine Absicht zu einer Handlung aus. Diese Handlung resultiert in der Regel aus einer mittel- bis langfristigen Planung. Derartige beabsichtigte Handlungen haben eine höhere Priorität als Desires. Die Intention ist ebenfalls eine Pro-Attitude.

Basiert die Architektur eines Agenten auf den oben genannten, mentalen Konzepten, so spricht man von einer BDI-Architektur bzw. von einem BDI-Agenten.

Definition 3 *BDI-Agenten sind deliberative bzw. kognitive Agenten, die über formal definierte Modelle ihrer Umwelt, ihrer Wünsche und ihrer Absichten verfügen, und sich durch die Fähigkeit des rationalen Handelns auszeichnen.*

Die vorgestellten Konzepte geben einen theoretischen Rahmen für Agenten als intentionale Systeme vor. Als eine erste formale Ausprägung dieser Konzepte wurde von Rao und Georgeff ([RG95]) ein BDI-Interpreter entwickelt.

2.1.3 Der BDI-Interpreter

Die abstrakte Architektur des BDI-Interpreters besteht im wesentlichen aus Datenstrukturen für die Beliefs, Desires und den Intentions, sowie aus einer Warteschlange (Eventqueue) in der externe und interne Ereignisse gesammelt werden. Der Interpreter selbst durchläuft nach einer Initialisierungsphase einen Zyklus in dem er zuerst die Warteschlange auf eingetroffene Ereignisse überprüft. Basierend auf dem ersten vorhandenen Ereigniss erstellt die Komponente „option-generator“ eine Liste mit möglichen Aktionen, die als Reaktion auf dieses Ereignis in Frage kommen. Eine weitere Komponente „deliberator“ wählt aus dieser Liste eine Menge von Aktionen aus und fügt sie der Datenstruktur der Intentions hinzu. Somit ist die Absicht, die ausgewählten Aktionen auszuführen, zum Ausdruck gebracht worden. Die Ausführung einer atomaren Aktion erfolgt im nächsten Schritt. Innerhalb der Ausführungszeit eingetroffene Ereignisse werden der Eventqueue hinzugefügt. Abschließend wird

überprüft, welche Wünsche und Ziele erfüllt worden sind, und welche nicht erfüllt werden können. Die Datenstrukturen für die Desires und Intentions werden entsprechend aktualisiert.

Die semi-formale Beschreibung eines BDI-Interpreters geben die Autoren Rao und Georgeff wie folgt an:

```
Initialize-state();
repeat
options := option-generator(event-queue);
selected-options := deliberate(options);
update-intentions(selected-options);
execute();
get-new-external-events();
drop-succesful-attitudes();
drop-impossible-attitudes();
end repeat
```

Dieses abstrakte Schema kann als Anleitung für konkrete BDI-Interpreter innerhalb einer BDI-Agentenarchitektur betrachtet werden, und dient damit als Grundlage für das von Georgeff und Lansky [GL87] spezifizierte Procedural Reasoning System (PRS). Implementierungen eines PRS kommen z.B. in der JAM-Architektur (vgl. [Hub00]) zum Einsatz.

2.1.4 Die JAM-Architektur

Die JAM-Architektur ist eine ausführbare BDI-Agentenarchitektur (siehe 2.2) bestehend aus einem Weltmodell, einer Planbibliothek, einem JAM Agent-Interpreter, einer Beobachterkomponente (Observer) und einer Intention-Komponente. Das Weltmodell ist eine die Beliefs des Agenten beschreibende Datenstruktur. Spezifikationen für die Beliefs, die Pläne und die Ziele eines JAM-Agenten werden in separaten Textdateien abgelegt. Die Struktur dieser Spezifikationen ist in Form einer BNF-Grammatik vorgegeben.

Eine Änderung des Weltmodells oder das Hinzufügen eines neuen Ziels initiieren einen Reasoning-Prozess innerhalb des Agent-Interpreters. Der Agent-Interpreter durchläuft, ähnlich wie der abstrakte BDI-Interpreter, einen Zyklus bis alle Ziele des Agenten erfüllt sind. In diesem Zyklus wird eine Liste von Plänen (Applicable Plan List - APL) erstellt, die sich aufgrund ihrer Spezifikation zur Bewältigung der

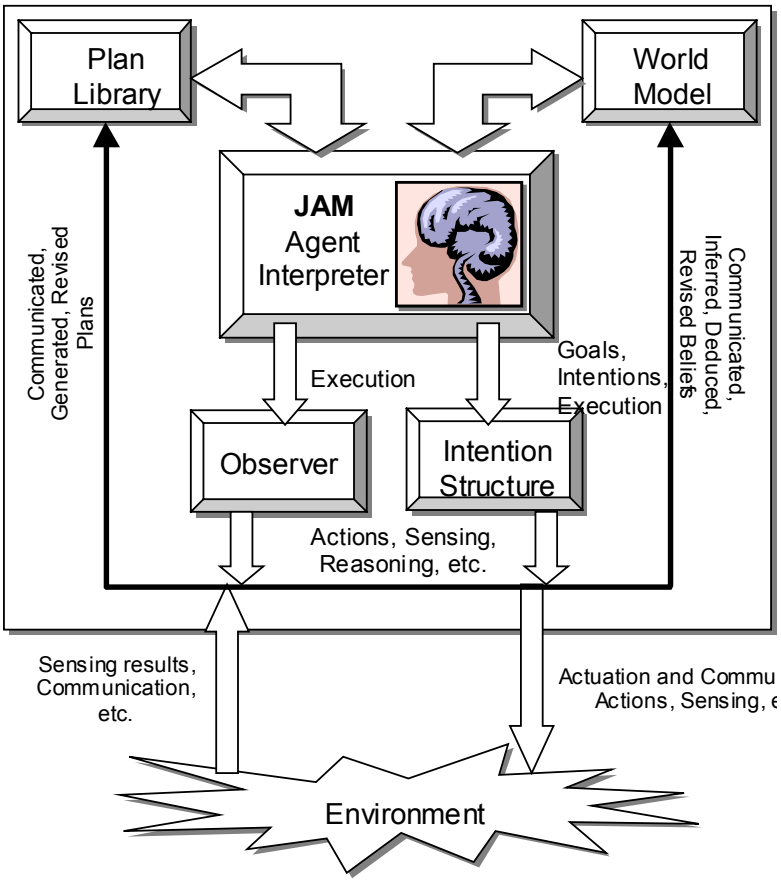


Abbildung 2.2: BDI-Architektur eines JAM-Agenten [Hub00]

im Welt-Modell kodierten Situation eignen. Diese Pläne werde schrittweise ausgeführt, und deren Ausführung wird von der Beobachterkomponente überwacht. Stehen zur Erfüllung eines Zieles mehrere Pläne in der APL zur Verfügung, wird die Nützlichkeit eines jeden Plans ermittelt und der Plan mit dem höchsten Nützlichkeitswert ausgewählt. Die Durchführung dieses Plan ist die aktuelle Intention des JAM-Agenten und wird in der Intention-Komponente abgelegt. Da die Nützlichkeit eines Ziels ebenfalls deklariert werden kann, ist der Agent-Interpreter in der Lage den nützlichsten Plan zur Erreichung des wichtigsten Zieles auszuwählen und anschließend auszuführen. Somit stellen sich die von einem JAM-Agenten ausgeführten Aktionen als ein durch rationale Beweggründe bestimmtes Handeln dar.

2.2 Multiagentensysteme

Agenten kommen in der Praxis sowohl alleinstehend, als *Single*-Agent, als auch in Organisationen von mehreren Agenten vor. Die in Abschnitt 2.1.1 geschilderte Klasse der Interface Agenten ist ein geeignetes Beispiel für einen Single-Agent, während die Klasse der kollaborativen Agenten ein aus mehreren Agenten bestehendes System voraussetzt. Ein solches System bezeichnet man als Multiagentensystem. Ein Multiagentensystem oder auch kurz MAS definieren Durfee und Lesser in [DL89] folgendermaßen:

Definition 4 *An MAS can be defined as a loosely coupled network of problem solvers that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver.*

Beim Lösen komplexer Probleme kann ein einzelner Agent an seine Grenzen stoßen, wenn er entweder nicht über ein ausreichendes Wissen bzw. über ausreichende Informationen verfügt, oder die für die Berechnung benötigten Ressourcen nicht ausreichen. Diesen Umstand versucht man zu beheben, indem das zu lösende Gesamtproblem in mehrere Teilprobleme mit jeweils geringerer Komplexität zerlegt wird. Einzelne Agenten können sich dann auf das Lösen der Teilprobleme spezialisieren. Eine derartige Herangehensweise läßt sich auch in natürlichen Organisationen beobachten, bei denen ebenfalls komplexe Probleme durch Kooperation verschiedener Akteure gemäß ihren individuellen Fähigkeiten gelöst werden.

Katia Sycara stellt in [SPVG03] für ein MAS folgende charakteristische Merkmale heraus:

- Jeder Agent hat nur unvollständige Informationen oder Fähigkeiten das Problem zu lösen und demnach eine eingeschränkten Gesamtsicht.
- Es existiert keine globale Systemkontrolle,
- die Daten sind dezentralisiert
- und die Berechnungen verlaufen asynchron.

Bei dem Entwurf von Multiagentensystemen müssen eine Reihe von Fragen beantwortet werden. So ist zunächst zu klären, wie das MAS strukturiert werden soll.

Welcher Agent welche Rolle im Gesamtsystem einnimmt, und welche Verantwortlichkeiten mit dieser Rolle verbunden sind. Da die Daten in einem MAS verteilt vorliegen müssen Interaktionsstrategien gefunden werden, die regeln, in welcher Form sich die Agenten gegenseitig ihr vorhandenes Wissen zur Verfügung stellen können.

Trotz all dieser zu klärenden Fragen, verspricht man sich von MAS eine Reihe von Vorteilen gegenüber herkömmlichen Lösungsansätzen. So können MAS dazu beitragen Probleme zu lösen, welche selbst inherent verteilt sind — und können zu robusteren Systemen führen, welche gegenüber unerwarteten Ausseneinflüssen weniger fehleranfällig sind — und sich besser skalieren lassen.

Weiterhin erlauben Multiagentensysteme bereits existierende „Legacy"-Systeme, wie z.B. Expertensysteme, miteinander zu verbinden. Das MAS übernimmt in diesem Fall die Rolle einer Middleware zwischen den „Legacy"-Systemen, welche jeweils durch einen Wrapperagenten nach aussen repräsentiert werden.

2.3 Agentenplattformen

Agentenplattformen bilden das Rückgrat eines Multiagentensystems, indem sie den Agenten die erforderliche Infrastruktur bereitstellen. Die „Foundation for Intelligent Physical Agents" kurz FIPA schlägt eine Reihe von Spezifikationen in den Bereichen „Agentenkommunikation", „Agentenmanagement" und „Nachrichtentransport" vor, um die Interoperation zwischen heterogenen Agentensystemen zu vereinfachen.

Den Kern dieser FIPA-Spezifikationen bildet die FIPA Abstract Architecture (siehe [FIP02]), welche als Grundlage für FIPA-konforme Agentenplattformen dient. Aus dieser abstrakten Architektur können die Entwickler eigene konkrete Architekturen ableiten, deren Interoperatibilität mit weiteren FIPA-konformen Plattformen gewährleistet wird.

Zu den Kernkomponenten der FIPA-Abstract Architecture gehören:

Agent Management System Das AMS ist verantwortlich für die Kontrolle der Lebenszyklen der einzelnen Agenten. Selbst als Agent realisiert regelt das AMS den Zugriff auf die Agentenplattform. Jeder Agent muß sich beim AMS anmelden und abmelden. Des weiteren lassen sich Agenten über das AMS suchen und modifizieren.

Directory Facilitator Der DF ist ein „yellow pages" Dienst, den Agenten in Anspruch nehmen *können*. Agenten können ihre Dienste beim DF registrieren,

deregistrieren und modifizieren. Informationen über diese Dienste stehen dann allen anderen Agenten der Agentenplattform zur Verfügung.

Agent Communication Channel Der ACC regelt den Nachrichtentransport für alle Agenten derselben Plattform, aber auch zu anderen FIPA-konformen Agentplattformen.

2.3.1 Die JADE-Plattform

Eine konkrete Agentenplattform, die auf der FIPA-Abstract Architecture basiert ist die von der Telecom Italia entwickelte JADE-Plattform ([BCT03]). Das Akronym JADE steht für Java Agent Development Framework. Zur Implementierung von Multiagentensystemen unter JADE steht dem Entwickler eine umfangreiche Java-Klassenbibliothek zur Verfügung. Des weiteren kann von technischen Aspekten, wie der Agentenverwaltung und dem Nachrichtenversand abstrahiert werden, da bereits beide Komponenten gemäß den FIPA-Spezifikationen umgesetzt wurden. Die Kontrolle über alle an der Agentenplattform angemeldeten Agenten wird durch einen Remote Monitoring Agent, dem RMA, sichergestellt. Dieser Agent kann optional mit einer grafischen Benutzeroberfläche gestartet werden und erlaubt das Überwachen, Starten und Beenden von Agenten, sowie deren Migration auf andere JADE-Agentenplattformen. Weitere JADE-Agenten erlauben das menügestützte Versenden von Nachrichten in der Agent Communication Language (ACL) und das Beobachten des Nachrichtenverkehrs zwischen frei wählbaren Agenten.

2.3.2 Die Jadex-Architektur

Das Jadex-Framework wird ebenfalls am Fachbereich Informatik der Universität Hamburg entwickelt ([BPL04]) und stellt eine umfangreiche Erweiterung der JADE-Plattform um eine BDI-Agentenarchitektur, ähnlich der von JAM, dar. Diese BDI-Agentenarchitektur beruht auf einem, mit XML Schema (siehe 3.5.2) beschriebenen, Metamodell. Die Struktur dieses Metamodells ist ein wesentlicher Bestandteil dieser Arbeit und wird daher in Abschnitt 4.7) gesondert angesprochen.

Der Entwicklungsprozess eines Jadex-Agenten läßt sich grob in zwei Bereiche einteilen. Zum einen muß die Architektur eines Jadex-Agenten deklarativ in Form eines XML-Dokumentes, dem Agent Description File (ADF), vorgenommen werden. Derartige Beschreibung sind mit dem in dieser Arbeit implementierten Entwurfswerk-

zeuges möglich. Zum anderen ist die Programmierung der vom Agenten ausführbaren Aktionen in Java erforderlich.

3 Der MDA-Ansatz

In diesem Kapitel werden die Grundüberlegungen, die später als Ausgangspunkt für die Realisierung des Entwurfswerkzeugs dienen sollen, näher betrachtet. Architekturen und architektonische Sichten als Instrument zur Spezifikation und Konstruktion von Softwarekomponenten, wie sie aus der komponentenbasierten Softwareentwicklung bekannt sind, werden auch im Kontext dieser Arbeit eingesetzt. Dies liegt besonders nahe, da die beiden Paradigmen *komponentenbasierte Softwareentwicklung* und *agentenbasierte Softwareentwicklung* zwar nicht identisch sind, jedoch viele Gemeinsamkeiten aufweisen.

Eine von der *Object Management Group* (OMG) vorgeschlagene Architektur zur Spezifikation von Software-Systemen ist der Ansatz der *Model Driven Architecture* (MDA). Die MDA ermöglicht eine Trennung der Spezifikation eines Systems von einer konkreten Plattform, auf der dieses System ausgeführt werden soll. Eine derartige Plattform stellt dabei nur eine Spezifikation technologischer Details dar, welche für die eigentliche Systemfunktionalität nicht relevant sind (vgl. [And03]). Um diese Trennung zu erreichen, führt die MDA die *Platform Independent Models* (PIM) und die *Platform Specific Models* (PSM) ein. Bevor diese Modelle untersucht werden, sollen die Begriffe Modell - Modellierung und Metamodell - Metamodellierung geklärt werden. Abschließend wird die Kerninfrastruktur der MDA, bestehend aus den ebenso von der OMG definierten Standards wie der *Unified Modeling Language* (UML), der *Meta Object Facility* (MOF) und des *XML Metadata Interchange* (XMI), vorgestellt.

3.1 Modellierung und Metamodellierung

Die Konzepte der *Modellierung* und der *Metamodellierung* sind in der Informatik längst etabliert, da sie eine Abstraktion von konkreten Gegenständen oder Prozessen zulassen und somit die Konstruktion und Darstellung immer komplexer werdender Softwarekomponenten, ihr Zusammenspiel sowie die ihnen zugrundeliegenden

Zusammenhänge vereinfachen, beziehungsweise erst sinnvoll ermöglichen. Derartige Abstraktionen finden zunehmend in fast allen Bereichen der Informatik statt, selbst wenn man dort nicht explizit von Modellierung spricht, oder zumindest ein intuitives Verständnis des Modellbegriffs voraussetzt. Daher soll hier auch keine allumfassende Definition dieser Begriffe gegeben werden, sondern nur Begriffsdefinitionen in dem für diese Arbeit relevanten Kontext der Softwareentwicklung.

3.1.1 Modell

Eine sprachliche Beschreibung eines Gegenstandsbereiches kann als eine Abbildung dieses Gegenstandsbereiches auf ein Zeichensystem verstanden werden, welche man im Fall einer isomorphen bzw. homomorphen Abbildung als Modell bezeichnet (vgl. [Str98]). Spezielle Anforderungen bezüglich der operationalen Handhabbarkeit an diese Modelle führen zu einem restriktiveren Modellbegriff.

Im folgenden bezeichnet der Begriff *Modell* eine Beschreibung eines Systems oder eines Systemteils mittels einer wohl definierten Sprache. Unter einer wohl definierten Sprache versteht man eine Sprache mit einer formal festgelegten Syntax und einer dazu angegebenen Semantik. Diese Sprache sollte des weiteren für eine automatische Interpretation durch einen Computer geeignet sein (vgl. [KWB03]). Diese Einschränkung läßt also keine natürlichen Sprachen zur Systembeschreibung zu, ist aber Voraussetzung für die automatische Transformation von einem Modell in ein anderes Modell, im Sinne der Model Driven Architecture (MDA).

Den Vorgang der Beschreibung von Systemen der realen oder einer imaginären Welt, welche im Gegensatz zu den Modellen selbst nicht in einer formalen Sprache definiert sein müssen, bezeichet man als Modellierung oder Modellbildung. Unter diesen Gesichtspunkten ist folglich das objektorientierte Programmieren, in einer zwangsläufig wohldefinierten Sprache wie z.B. Java, eine Form der Modellierung von Objekten der realen Welt.

3.1.2 Metamodell

Unter dem Begriff Metamodell versteht man allgemeinhin ein Modell das ein anderes Modell beschreibt. Welche Aspekte bei der Beschreibung eines Modells durch ein Metamodell im Vordergrund stehen, geht aus dem Begriff des Metamodells selbst nicht explizit hervor. Diese Aspekte können zum einen die zur Modellbildung verwendete Sprache und zum anderen der Prozess der Modellbildung selbst sein. Es

kann zwischen zwei vor allem in der Praxis relevanten Metamodellen unterschieden werden, einem sprachbasierten Metamodell und einem prozessbasierten Metamodell. Um kenntlich zu machen, um welchen Metamodellbegriff es sich jeweils handelt, führt [Str98] den Begriff des *Metaisierungprinzips* ein und definiert ihn wie folgt:

Definition 5 *Das Metaisierungsprinzip beschreibt denjenigen Aspekt eines Modells, der in der übergeordneten Modellierungsstufe abgebildet wird.*

Die Sprache, die zur Erstellung eines Modells verwendet wird, nennt [Str98], in Anlehnung an die Sprachstufentheorie der Logik, Objektsprache. Wird diese Objektsprache Gegenstand weiterer Untersuchungen, so kann dies wiederum in einer Sprache geschehen. Diese Sprache bezeichnet man in Bezug auf eine vorgegebene Objektsprache als Metasprache und das mit der Metasprache beschriebene Modell der Objektsprache als Metamodell. Da auch eine Metasprache wiederum Gegenstand einer Abbildung auf eine Sprache sein kann, lässt sich das Prinzip des Bildens von Metasprachen bezüglich ihrer Objektsprachen rekursiv anwenden. Das sprachliche Beschreibungsmodell einer Metasprache nennt man Metametasprache. Die Objektsprache und die Metasprache können identisch sein, was eine Einordnung der Sprache in eine Sprachebene erforderlich macht. Zur Kennzeichnung dieser Sprachebenen wird folgende Konvention vorgeschlagen. Die unterste Sprachebene, also die Ebene auf der die Objektsprache zur Erstellung eines Modells von Gegenständen der realen Welt anzusiedeln ist, wird mit *Ebene 0* bezeichnet. Die Abstraktionsebenen über dieser Modellebene werden aufsteigend durchnummeriert. Daraus ergibt sich eine hierarchische Darstellung des sprachbasierten Metamodells (siehe Abbildung 3.1.2).

Wie Eingangs bereits angesprochen gibt es in der Informatik auch eine weitere übliche Herangehensweise bei der Bildung von Metamodellen aus Modellen, bei der nicht die Abstraktion von der statischen Struktursemantik der Modellierungssprache im Vordergrund steht, sondern eine den Modellierungsprozess beschreibende dynamische Prozesssicht. Bei diesem Metaisierungsprinzip betrachtet man nicht die Sprache in der das Modell beschrieben wird, jedoch den Prozess mit dem ein Modell gebildet wird. Dieser Prozess der Modellbildung kann dann seinerseits wieder durch einen Prozess der Prozessbildung zur Modellbildung modelliert werden, dem sogenannten Metaprozess. Auch diese Rekursion lässt sich in Analogie zum sprachbasiertem Metamodell beliebig fortsetzen.

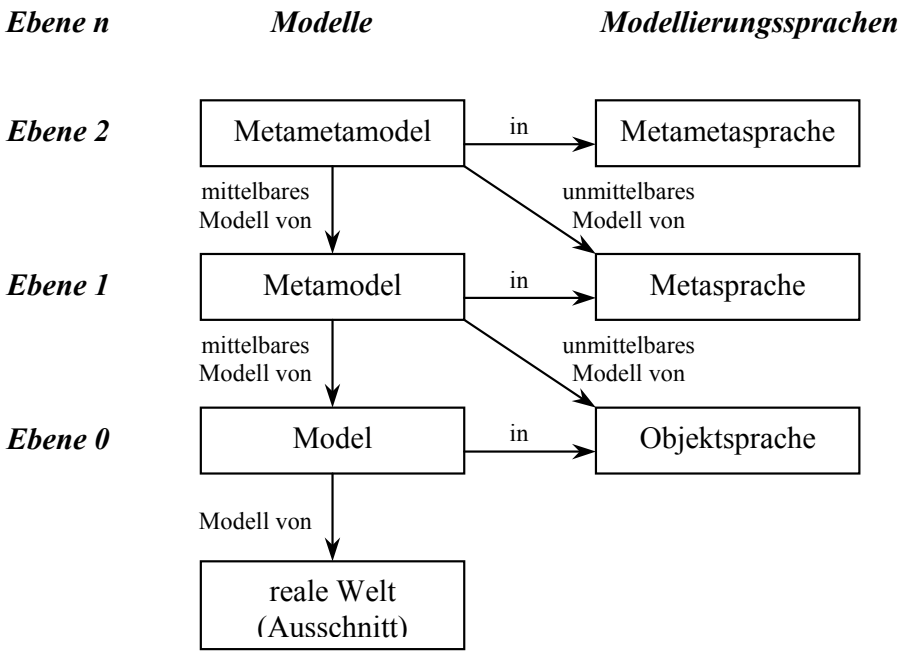


Abbildung 3.1: Sprachbasierte Modellhierarchie [Str98]

3.1.2.1 Stringenz des Metaisierungsprinzips

In der Theorie ließen sich zum einen beliebig viele Modellebenen als auch beliebige Metaisierungsprinzipien konstruieren. Dieser Sachverhalt erschwert nicht nur die Einordnung der Modelle in eine Modellhierarchie, sondern kann auch zu konzeptuellen Unstimmigkeiten bei der Verwendung von Metamodellen führen. Beispielsweise können von einem Gegenstand der realen Welt die dynamischen Aspekte durch die Notationsform der UML-Sequenzdiagramme beschrieben werden, welche ihrerseits wiederum entweder durch ein Modellierungsprozess oder auch durch das Metamodell der UML beschrieben werden. Die UML ist jedoch eine Modellierungssprache, die keinen Modellierungsprozess enthält (vgl. [Bur97]). Ein ähnliches Problem entsteht beim Beschreiben der statischen Struktur des oben erwähnten Gegenstandes, z.B. mit UML-Klassendiagrammen, deren Metamodelle sowohl ein Modellierungsprozess als auch das UML-Metamodell sein können. Auch in diesem Fall wäre der Modellierungsprozess als Metamodell für die UML-Klassendiagramme, welche nur statische Konstrukte aufweisen, konzeptuell ungeeignet. In [Jec00] legt der Autor daher die Stringenz des verwendeten Metaisierungsprinzips innerhalb einer Modellhierarchie nahe, was in Abbildung 3.2 illustriert wird. S sei dabei die Abstraktion von der statischen Strukturseman tik und P jeweils die prozessuale Sicht. Die Metamodelle auf

den orthogonalen Achsen sind durch stringente Metamodellierung bezüglich eines Metaisierungsprinzips entstanden.

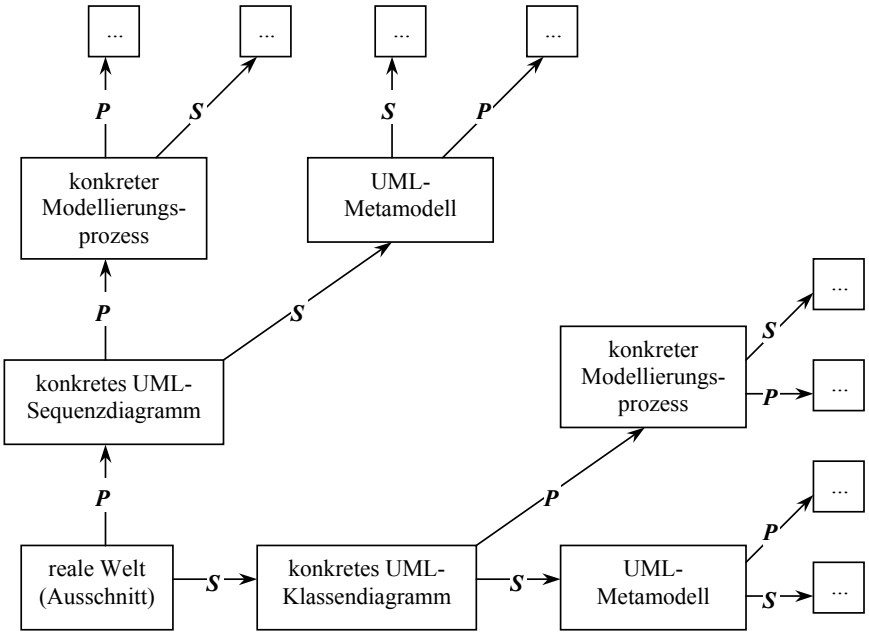


Abbildung 3.2: Einordnung von Metamodellen in eine gemeinsame Modellhierarchie [Jec00]

Für die Beschreibung dynamischer Prozesse haben sich vor allem petrinetzbasierte Ansätze empfohlen. Für den Einsatz der UML zur Beschreibung von Prozessen im Bereich der Softwareentwicklung bedarf es allerdings einer Methode, die von der UML unterstützt wird. Solche Methoden regeln hauptsächlich die zeitliche und logische Abfolge von einzelnen Modellierungsschritten und geben die zu verwendenden Diagrammtypen vor. Da das in dieser Arbeit zu erstellende Metamodell für BDI-Agentensysteme ausschließlich die statischen Aspekte solcher Systeme aufgreift, wird auf die prozessbasierte Metamodellierung nicht weiter eingegangen. Statt dessen kommt mit der Meta Object Facility (siehe Abschnitt 3.2) eine Modellierungsarchitektur zum Einsatz, die rein auf die Modellierung sprachlicher Konstrukte abzielt und in diesem Sinne bezüglich ihres Metaisierungsprinzips stringent ist.

3.1.3 Platform Independent Model (PIM)

Das *Platform Independent Model* (PIM) stellt in der MDA das Modell mit der höchsten Abstraktion eines bestimmten Aspektes dar. Diese Abstraktion ist völlig

unabhängig von einer Implementation in einer bestimmten Technologie, z.B. einer Programmiersprache. Der Vorteil eines PIM's besteht darin, daß eine bestimmte Domaine, z.B. ein Geschäftsvorgang, so modelliert werden kann, wie sie in der Realität existiert oder als sinnvoll erachtet wird. Wie diese modellierte Domaine später in einer konkreten Technologie umgesetzt wird, ist für den Modellierer nicht weiter relevant. Dieser Aspekt erleichtert die Arbeitsteilung bei der Softwareentwicklung ungemein, da Domainenexperten und Softwaretechniker unabhängig voneinander agieren können.

3.1.4 Platform Specific Model (PSM)

Das *Platform Specific Model* (PSM) geht aus dem PIM mittels einer Modelltransformation hervor. Es ist dabei unerheblich, ob aus einem PIM ein PSM oder mehrere PSM's erzeugt werden. Die PSM's sind speziell an eine konkrete Implementationstechnologie gebunden, sie sind demnach plattformspezifisch. Aus einem PIM, welches die Struktur eines Objektes „Person“ beschreibt, können z.B. konform zum Softwareentwicklungzyklus der MDA ein PSM für Java und ein PSM für eine relationale Datenbank generiert werden. Ersteres würde die Struktur des Personenobjektes dann in Form von Konstrukten wie „Klasse“, „Attribut“ etc. enthalten, während das PSM der relationalen Datenbank das Objekt „Person“ auf Konstrukte wie „Table“, „Column“ u.s.w. abbildet.

3.1.5 Code Model

Das Modell auf der untersten Abstraktionsebene wird im folgenden als *Code Model* bezeichnet. Die Code-Modelle werden mit Mechanismen, wie sie aus der traditionellen Softwareentwicklung bekannt sind, aus den PSM generiert. Diese Form der Modelltransformation ist aufgrund der Tatsache, daß benötigte technologische Details aus dem PSM extrahiert werden können, mit relativ geringem Aufwand möglich. In der Regel benutzt man für diese Transformation auf Templatemechanismen basierende Transformationswerkzeuge bzw. Codegeneratoren. Das Code-Modell für das obige Personenobjekt könnte demnach eine konkrete Instanz der Java-Klasse „Person“ oder auch ein Eintrag in der Tabelle „Person“ in einer relationalen Datenbank sein. Die Abbildung 3.3 veranschaulicht den Zusammenhang zwischen PIM, PSM und Code sowie die Brücken zwischen Modellen auf gleicher Abstraktionsebene.

Die Brücken zwischen zwei oder mehreren PSM, bzw. zwischen Code-Repräsentation dieser Modelle, sind bei der traditionellen Softwareentwicklung per se nicht gegeben,

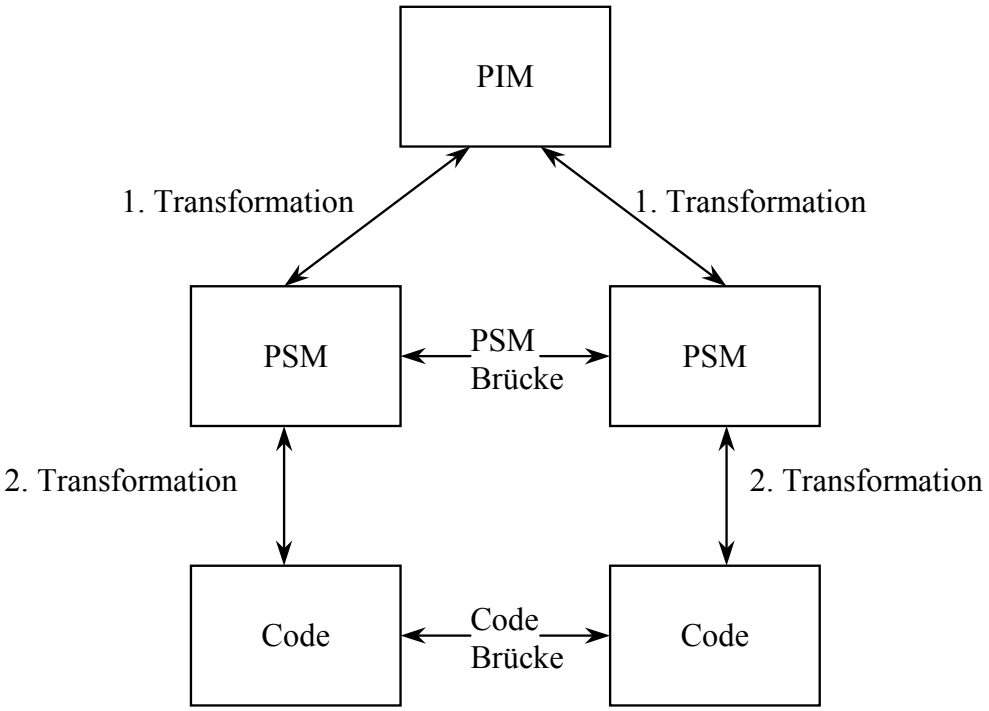


Abbildung 3.3: Modelltransformationen in der MDA [KWB03]

was die Integration von Anwendungen auf heterogenen Plattformen zusätzlich erschwert, da hierfür Konzepte der einen Plattform in Konzepte einer anderen, von der ersten verschiedenen Plattform, übersetzt werden müssen. Diese Übersetzung bedingt jedoch, daß explizites Wissen über die jeweils verwendeten Konzepte vorhanden sein muß. Liegt der Anwendung ein modellbasierter Entwicklungsprozess gemäß der MDA zugrunde, befindet sich das Wissen um domainenspezifische Konzepte in dem PIM und das Wissen um die technologischen Aspekte der für die Anwendung relevanten Plattform in dem PSM.

3.2 Die Meta Object Facility (MOF)

Die MOF ist für diese Arbeit von besonderem Interesse, da sie verspricht, im Zusammenspiel mit den weiteren Standards XMI und UML, die Grundlage zur Entwicklung von modellbasierten Entwicklungswerkzeugen, Metadaten Repositories u.s.w. zu schaffen.

Die Meta Object Facility, im folgenden nur noch mit MOF bezeichnet, ist einer der Standards, welche die Kerninfrastruktur der MDA bereitstellen. Die MOF befindet sich derzeit in der Version 2.0, die eine Revision der in der Industrie vielfach genutzten Vorgängerversion 1.4 darstellt. In dieser Arbeit wird aus technischen Gründen (vgl. Abschnitt 5.1) nur die Version 1.4 der MOF verwendet und im folgenden vorgestellt. Die komplette Spezifikation dieses Standards, der im April 2002 von der OMG verabschiedet wurde, findet sich in [OMG02a].

Die Motivation der MOF ist es die Beschreibung von Daten mittels Metadaten zu vereinheitlichen, den Austausch dieser Metadaten über die Systemgrenzen einer Anwendung hinaus zu vereinfachen und damit einen Beitrag zur Integration von wichtigen Anwendungen aus den Bereichen data warehousing, e-commerce und Informationsportalen zu leisten. *Metadaten* sind in der Terminolgie der OMG Modelle von Daten, die die statische Struktursemantik dieser Daten beschreiben.

Zu den Bestandteilen der MOF gehören im einzelnen:

- eine formal definierte abstrakte Sprache, das MOF-Metametamodell, zur Spezifikation von MOF-Metamodellen,
- eine Abbildung von beliebigen MOF-Metamodellen auf sprachunabhängige Schnittstellen, definiert in der CORBA Interface Definition Language

(CORBA IDL), die das Produzieren von Schnittstellen zum Verwalten von Metadaten jeglicher Art ermöglichen,

- eine Menge von „reflektiven“ CORBA IDL-Schnittstellen zum Verwalten von Metadaten, unabhängig vom Metamodell und
- eine Menge von CORBA IDL-Schnittstellen zur Verwaltung und Repräsentation von MOF-Metamodellen,
- einem XMI-Format zum Austauch von MOF-Metamodellen.

Die Gesamtheit dieser Bestandteile stellt ein geeignetes Rahmenwerk zur Unterstützung von auf Metadaten basierenden Werkzeugen dar. Dabei liegt der Fokus auf der Konstruktion und der Verwaltung beliebiger Metadaten, deren Gestalt nicht a priori bekannt sein muß, bzw. die erst im Nachhinein definiert wird. Um dieses Ziel zu erreichen, benutzt die MOF eine vierschichtige Metadaten-Architektur, wie sie durch die Standardisierungsbemühungen der ISO *(International Organization for Standardization)* und der CDIF *(Case Data Interchange Format)* im Bereich der klassischen Metamodellierung bekannt gemacht wurden. Diese vierschichtige Metadaten-Architektur, im Kontext der MOF mit MOF Metadaten-Architektur bezeichnet, ist eine Konvention zur Einordnung von Modellen, Metamodellen und Metametamodellen in eine einheitliche Ebenenhierarchie, ähnlich der aus Abschnitt 3.1.2. Die MOF Metadaten-Architektur ist jedoch mit dem MOF-Metametamodell auf vier Ebenen begrenzt. Diese Begrenzung ergibt sich aus der Tatsache, daß man das MOF-Metametamodell, welches auf der obersten Modellebene angesiedelt ist, mit den sprachlichen Modellierungskonstrukten, die das MOF-Metametamodell bereitstellt, beschreiben läßt. Ein Modell über dieser vierten Ebene einzuführen, würde somit aus konzeptueller Sicht keinen Mehrwert für diese Metadaten-Architektur ergeben. Sofern die Einteilung aller Modelle bzw. Metamodelle in diese vier Modellebenen vorausgesetzt werden kann, soll um Konfusionen bei der übermäßigen Verwendung des Präfix „meta“ zu vermeiden, anstelle von MOF-Metametamodell die Bezeichnung MOF-Modell benutzt werden. Die in der Literatur auch des öfteren gefundene Bezeichnung „MOF“ für das MOF-Modell erscheint ungünstig, da dann eine Verwechslungsgefahr des MOF-Modells mit der Meta Object Facility als solches bestünde. Die vierschichtige Metadaten-Architektur der MOF wird in der Tabelle 3.1 dargestellt.

Auf der obersten Modellebene M3 befindet sich als einziges Modell das MOF-Modell. Das MOF-Modell besteht aus einer Menge formal definierter Modellelemente, die der

Metaebene	Beschreibung	Beispiele
M3	Meta-Metamodell	MOF-Modell
M2	Metamodelle / Meta-Metadaten	Beschreibung von sprachlichen Konstrukten: Definition einer Klasse, einer Methode, eines Attributes, usw.
M1	Modelle / Metadaten	Beschreibung konkreter Klassen, Methoden und Attribute, z.B. Java-Klassen
M0	Instanzen / Daten	Instanz einer Klasse (z.B. Java-Klasse) zur Laufzeit, identifizierbare Objekte

Tabelle 3.1: Vierschichtige Metadaten-Architektur

Konstruktion von beliebigen Metamodellen dienen, sowie einem Regelsatz, welchem zu entnehmen ist, wie man diese Modellelemente benutzt. Die Metamodelle bzw. Metametadaten, deren Struktur und Semantik mit dem MOF-Modell beschrieben werden, befinden sich auf Ebene M2. Bekannte Vertreter dieser Ebene sind das UML-Metamodell, das *CWM* (Common Warehouse Metamodell) oder das Metamodell der CORBA-IDL. Das in dieser Arbeit erstellte Metamodell für BDI-Agentensysteme, welches ebenfalls durch das MOF-Metamodell beschrieben wird, läßt sich auch in diese Modellebene einordnen. Die Metamodelle, welche sich vollständig aus den Modellelementen des MOF-Modells konstruieren lassen, werden im folgenden auch als *MOF-konform* bezeichnet. MOF-konforme Metamodelle beschreiben Struktur und Semantik von Modellen bzw. Metadaten. Diese, der Modellebene M1 zugehörigen Modelle, können gemäß den oben genannten Beispielen der Modellebene M2, UML-Modelle, IDL-Schnittstellen oder CWM-Modelle sein. Ihre Aufgabe ist es, die Daten der untersten Modellebene M0, der sogenannten Informationsebene, mittels einer konkreten Syntax zu beschreiben.

Anmerkend sei erwähnt, daß die oben beschriebenen Modellebenen nicht explizit zum Standard der MOF gehören sondern lediglich die Beziehungen zwischen den verschiedenen Arten der Metadaten ausdrücken sollen. Konkrete Implementationen, die sich an der MOF orientieren, können mehr aber auch weniger Ebenen erforderlich machen.

3.3 Das MOF Modell

Das MOF-Modell übernimmt in der MDA die Rolle eines PIMs gemäß Abschnitt 3.1.3. In diesem Abschnitt soll ein kurzer Überblick über die abstrakte Syntax des MOF-Modells gegeben werden. Die abstrakte Syntax der MOF ist in [OMG02a] formal spezifiziert, indem die Eigenschaften eines jeden Modellelementes im einzelnen benannt und anschließend beschrieben werden. Des weiteren werden die Beziehungen der Modellelemente untereinander definiert. Die Tatsache, daß das MOF-Modell objektorientiert ist wird genutzt, um die Metamodellierungskonstrukte der MOF unter Zuhilfenahme der objektorientierten Modellierungskonstrukte der UML grafisch zu notieren. Konkret stellt die MOF Modellierungsarchitektur eine Untermenge des Sprachkerns der UML dar. Diese Untermenge besteht im wesentlichen aus den Modellierungskonstrukten:

- Klasse, zum Modellieren der Metaobjekte eines MOF-Metamodells,
- Assoziation, zum Modellieren von binären Relationen zwischen diesen Objekten,
- Datentyp, zum Modellieren von Daten,
- und Package, mit denen die Modelle modularisiert werden können.

3.3.1 MOF-Klasse

Klassen sind die fundamentalen Bausteine eines jeden MOF konformen Metamodells, und damit natürlich auch des MOF Modells selbst. Der im Kontext der MOF benutzte Begriff *Klasse* ist ähnlich dem einer Klasse in der UML. Eine Klasse wird im Metamodell auf der Modellebene M2 definiert und kann Instanzen dieser Klasse, die sogenannten Metaobjekte, auf der Modellebene M1 besitzen. Eine MOF-Klasse ist demnach eine abstrakte Spezifikation eines Metaobjektes einschließlich seines Zustandes, seiner Schnittstellen und seines Verhaltens. Die Spezifikation des Zustandes geschieht mittels Attributen, welche mit einem innerhalb der Klasse eindeutigen Namen und einem Typ versehen sind. Der Typ eines Attributes kann seinerseits wieder eine MOF-Klasse oder ein Datentyp (siehe Abschnitt 3.3.3) sein. Ein MOF-Attribut ist demnach ein Platzhalter für einen Wert mit einem explizit angegebenen Typ. Die aus der objektorientierten Modellierung bekannten Konzepte des Klassenattributes und des Instanzattributes haben auch innerhalb der MOF Gültigkeit. Somit wird ein

Klassenattribut von allen Instanzen dieser Klasse geteilt, während ein Attribut auf der Instanzebene einen separaten Platzhalter für jede Instanz einer Klasse darstellt.

Die Schnittstellen eines Metaobjektes zu anderen Metaobjekten werden durch *Assoziationen* deklariert, welche in Abschnitt 3.3.2 eingeführt werden.

Zur Spezifikation des Verhaltens eines Metaobjektes werden Operationen benutzt. Da das MOF-Modell als PIM technologieneutral ist enthält es keinerlei Methoden, die das Verhalten eines Metaobjektes implementieren. Mit dem Konstrukt der Operationen wird lediglich ein Name und eine Typsignatur deklariert, mit dem eine Methode, welche das gewünschte Verhalten implementieren muß, aufgerufen werden kann. Neben dem Aufruf von Operationen des Metaobjektes sind das Erzeugen und Entfernen von Objekten, das Lesen und Speichern von Attributen und das Abfragen der Relationen zwischen den Objekten typische Beispiele für das Verhalten eines Metaobjektes. Die Implementation eines derartigen Verhaltens wird in einfachen Fällen, bei der Abbildung des Metamodells auf spezielle plattformspezifische Schnittstellen automatisch generiert.

Ein weiteres Konzept der objektorientierten Modellierung, nämlich das der Vererbung, kann unter einigen Einschränkungen auch bei der Modellierung mit dem MOF-Modell benutzt werden. In der Terminologie der MOF, als auch der UML, wird oftmals der Begriff *Generalisierung* anstelle von *Vererbung* verwendet. Wird eine Klasse durch eine andere Klasse generalisiert, so wird erstere mit *Subklasse* bezeichnet und die zweite mit *Superklasse*. Die Subklasse erbt dabei von der Superklasse sämtliche Attribute, Operationen und Referenzen. Um eine Abbildung dieses Konzepts auf verschiedene Implementationstechnologien sicherzustellen, unterliegt die Generalisierung folgenden Restriktionen. Eine Klasse darf sich nicht selbst generalisieren, weder direkt noch indirekt. Des weiteren darf eine Subklasse nicht durch eine Superklasse generalisiert werden wenn diese Modellelemente enthält, welche unter gleichem Namen bereits in der Superklasse definiert sind. Dadurch wird ein *Überladen* von Operationen und Attributen, wie es in vielen objektorientierten Programmiersprachen üblich ist, untersagt. Eine multiple Vererbung, also die Zuordnung von mehreren Superklassen zu einer Subklasse ist nur dann erlaubt, wenn sämtliche in den Superklassen enthaltene Modellelemente bezüglich ihres Namens verschieden sind. Die einzige Ausnahme zu dieser Bestimmung besteht in der *Diamandregel*, welche den Superklassen ein Erben von Namen aus einer gemeinsamen weiteren Klasse erlaubt.

Soll die Deklaration von Superklassen bezüglich einer Klasse unterbunden werden, so kann diese als *Root*-Klasse definiert werden. Ebenso ist eine Notation dieser Klasse

als *Leaf*-Klasse möglich, falls sie selbst keine Subklassen besitzen soll.

Klassen deren einzige Bestimmung es ist als Superklasse zu fungieren, können in Analogie zur UML als *abstrakte* Klassen spezifiziert werden. Eine Instanziierung solcher Klassen ist dann nicht vorgesehen.

3.3.2 MOF-Assoziationen

Um Beziehungen innerhalb eines Metamodells auszudrücken, bietet das MOF-Modell das Konstrukt der *Associations*. Assoziationen definieren die Beziehungen zwischen Instanzpaaren von Klassen und dürfen, anders als in UML, nur binär sein. Neben einem eindeutigen Namen besitzen binäre Assoziationen jeweils zwei Enden. Diese Enden sind innerhalb der Assoziation ebenfalls eindeutig zu benennen und ihr Typ entspricht einer der beiden assoziierten Klassen. Die Anzahl der an der Projektion einer Klasse auf eine andere Klasse beteiligten Instanzen wird durch einen Wertebereich an jedem Assoziationsende in eckigen Klammern notiert. Beispielsweise bedeutet die Angabe von [0..1], daß keine oder eine Verbindung zwischen den Instanzen zweier Klassen besteht. Soll die obere Grenze dieses Wertebereiches nicht vorgegeben werden, kann dies mit einem Stern (*) zum Ausdruck gebracht werden. Die Einordnung der Assoziationen in das MOF-Modell zeigt Abbildung 3.5.

Die Semantik von Assoziationen in einem MOF-Metamodell entspricht der einer Aggregation. Sind zwei Klassen A und B durch eine Aggregationsbeziehung miteinander verbunden, so kann beispielsweise eine Instanz der Klasse A als das „Ganze“ betrachtet werden, welches eine Instanz der Klasse B enthält.

Eine Aggregation kann durch die Eigenschaft „aggregate“ der Assoziationsenden näher charakterisiert werden. Gültige Werte für diese Eigenschaft sind „non-aggregate“, „composite“ und „shared“.

non-aggregate Der Wert „non-aggregate“ gibt an, daß zwischen den zwei betrachteten Instanzen keine eigentliche Aggregation besteht, sondern lediglich eine lose Koppelung. Bei einer solch losen Kopplung haben die beteiligten Instanzen keinen Einfluss auf den Lebenszyklus der jeweils anderen. Wird eine der beiden Instanzen entfernt, kann die andere durchaus weiter bestehen. Bei jeder Assoziation muß mindestens eines der beiden Assoziationsenden der Art „non-aggregate“ sein, da sich anderenfalls die Instanzen beider assozierten Klassen gegenseitig enthalten würden. Eine derartige Semantik ist in MOF-Metamodellen jedoch nicht zulässig.

composite Der Wert „composite“ deklariert eine stark Form der Aggregation. In der UML wird eine Komposition zweier Klassen mit einem schwarzen Diamanten notiert. Eine Klasseninstanz kann nur Bestandteil genau einer anderen Instanz sein. Diese kann ihrerseits ein Aggregat bestehend aus weiteren Instanzen beliebiger Anzahl sein. Beim Entfernen eines Aggregates, werden automatisch alle ihre Bestandteile entfernt. Der Lebenszyklus einer Instanz hängt in einer Komposition also vom Lebenszyklus der ihr übergeordneten Instanzen ab.

shared Der Wert „shared“ ist für die Eigenschaft „aggregate“ eines Assoziationsendes nur in der UML definiert. Im MOF-Modell selbst existiert die „shared“-Semantik nicht. Die „shared“-Aggreagtion ist die schwächere Form der Aggregation und wird in der UML durch einen weißen Diamanten symbolisiert. Der Unterschied zur starken Aggregation besteht darin, daß eine Instanz Bestanteil mehrerer Instanzen sein kann. Die Instanz wird nur dann entfernt, wenn sie zu keinem Aggregat mehr gehört.

Abbildung 3.4 stellt die starke und die schwache Aggregation gegenüber und verdeutlicht die Auswirkung des Entfernens von Instanzen.

Eine weitere Anwendung der Assoziationen in einem MOF-Metamodell ist die Deklaration von Referenzen. Der Typ eines Klassenattributes kann mit Hilfe einer Referenz auf eine andere, diesen Typ repräsentierenden, Klasse angegeben werden. Eine solche Referenz läßt sich als gerichtete, navigierbare Assoziation zwischen diesen beiden Klassen modellieren.

3.3.3 MOF-Datentyp

Das MOF-Modell stellt dem Modellierer sechs technologieneutrale primitive Datentypen zur Verfügung, die jeweils eine Instanz der „PrimitiveType“-Klasse sind. Von diesen sechs Typen „Integer“, „Boolean“, „String“, „Long“, „Float“ und „Double“ werden nur die ersten drei benutzt, um das MOF-Modell selbst zu beschreiben.

Neben den primitiven Datentypen lassen sich auch komplexere Datentypen mittels der Datentypkonstruktoren „EnumerationType“, „StructureType“, „CollectionType“ und „AliasType“ definieren. Der strukturelle Aufbau von Datentypen wird in Abbildung 3.6 dargestellt.

3.3.4 MOF-Packages

MOF-Packages besitzen die Semantik von Containern bzw. Behältern, durch dessen Verwendung sich komplexe Metamodelle aus einfacheren Teilmodellen zusammensetzen lassen. Ebenfalls ist es möglich Teilmodelle in einem Metamodell wiederzuverwenden. Zum Zwecke der Komposition und der Wiederverwenbarkeit von Metamodellen bietet das MOF-Modell die vier verschiedenen Mechanismen:

Generalization Die Generalisierung von Packages verläuft analog zur Generalisierung von Klassen. Ein Package, das Sub-Package, erbt dabei alle Modellelemente des Super-Packages.

Nesting Mit diesem Mechanismus lassen sich Packages ineinander verschachteln. Ein Package kann dabei ein oder mehrere weitere Packages enthalten. Diese Packages selbst dürfen weder vererbt noch von anderen Packages importiert werden. Eine Instanziierung dieser Packages ist ebenfalls nicht möglich.

Import Ein Package ist in der Lage ein anderes Package zu importieren. Damit erhält es die Möglichkeit, alle im importierten Package definierten Modellelemente zu verwenden.

Clustering Das „Clustering“ von Packages ist eine stärkere Ausprägung des Importmechanismus. Das importierende und das importierte Package wird zu einem Cluster mit gemeinsamem Lebenszyklus verbunden. Dieser Sachverhalt bedeutet, daß das Entfernen des importierenden Packages ebenso zu einem Entfernen der importierten Packages führt. Analog dazu führt das Instanziieren eines Packages aus einem Cluster dazu, daß alle in diesem Cluster enthaltenden Packages ebenfalls instanziiert werden.

Das vollständige MOF-Modell ist in Anhang A abgebildet.

3.4 UML Profile for MOF

Das UML-Metamodell und das MOF-Modell weisen eine Reihe von Unterschieden bezüglich der Nomenklatur ihrer Modellelemente und deren Attribute auf. Um Instanzen eines UML-Metamodells auf zum MOF-Modell kompatible Metamodelle abzubilden, wurde das „UML Profile for MOF“ von der OMG vorgeschlagen. Das

„UML Profile for MOF“ (siehe [OMG04]) regelt demnach das Mapping zwischen UML- und MOF-Metamodellen, als auch umgekehrt. Somit ist es möglich, MOF-konforme Metamodelle mit der UML zu erstellen und anschließend nach MOF zu transformieren. Da diese Transformation bidirektional durchführbar ist, kann das „UML Profile for MOF“ ebenso zur Visualisierung von MOF-Metamodellen durch Diagramme der UML genutzt werden.

Die folgende Tabelle ist dem „UML Profile for MOF“ entnommen worden und stellt das Mapping zwischen UML-Elementen und MOF-Elementen dar.

UML-Element	**Stereotype**	**MOF-Element**
Model	«metamodel»	Package
ElementImport		Import
Class		Class
Attribute		Attribute
Attribute	«reference»	Reference
Operation		Operation
Parameter		Parameter
Exception		Exception
Attribute (within an Exception)		Parameter
Association		Association
AssociationEnd		AssociationEnd
DataType		DataType
DataValue		Constant
Constraint		Constraint
Generalization		Generalizes
TaggedValue		Tag

Zu jedem Element existiert zusätzlich eine Tabelle aus der hervorgeht, wie die einzelnen Eigenschaften der MOF-Elemente auf die Eigenschaften der UML Elemente und „Tagged Values“ abgebildet werden. Weitere Bedingungen und Einschränkungen, die bei dieser Abbildung beachtet werden müssen, sind ebenfalls im „UML Profile for MOF„ zu finden. Dieser Sachverhalt soll kurz am Beispiel der Abbildung von einer UML-Klasse auf eine MOF-Klasse dargelegt werden. Aus einer *Class Property Map* werden die Übersetzungsregeln entnommen und die entsprechenden Eigenschaften der MOF-Klasse gesetzt.

MOF Property	UML Property or Value
contents	ownedElement, feature
visibility	visibility
isAbstract	isAbstract
isRoot	isRoot
isLeaf	isLeaf
supertypes	generalization.parent
isSingleton	value of taggedValue with tag = "org.omg.uml2mof.isSingleton"; otherwise false

Neben der Anwendung dieser Übersetzungsregeln muß die Bedingung erfüllt werden, daß die UML Eigenschaft „ownedElement" als geordnete Liste modelliert wird. Analog zum Mapping der Klassen wird bei allen weiteren Elementen verfahren. Anmerkend sei erwähnt, daß sich nicht jedes Element mit all seinen Eigenschaften vollständig transformieren läßt. Die mit dem „UML Profile for MOF" definierten Bedingungen und Einschränkungen müssen demnach beim Aufstellen der Modelle Berücksichtigung finden und könnten in eine für den Entwickler verbindliche Methode bzw. in einer Modellierungsrichtlinie resultieren.

Des weiteren müssen die für das „UML Profile for MOF" benötigten Stereotypes dem jeweils verwendeten Modellierungswerkzeug bekannt gemacht werden. Dies geschieht in der Regel mit einem geeigneten Austauschformat für Metadaten, wie es im nächsten Abschnitt vorgestellt wird.

3.5 Austausch von Modellen

Zur Sicherstellung der Interoperabilität verschiedener Werkzeuge untereinander ist ein gemeinsames Verständnis der zu kommunizierenden Daten und Metadaten unabdingbar. Da bei n miteinander operierenden Werkzeugen bis zu $\frac{n*(n-1)}{2}$ Dokumente zum Austausch der Daten erstellt und gepflegt werden müssen, hat sich die Verwendung einer standardisierten Inhaltssprache als Schnittstelle für den Datenaustausch als vorteilhaft herausgestellt. Im Laufe der letzten Jahre hat sich die „eXtensible Markup Language" (XML), wie sie vom World Wide Web Consortium (W3C) entwickelt wurde, als praxistauglich und effektiv erwiesen. Frühere Sprachen zur Beschreibung von strukturierten Informationen, wie die „Standard Generalized

Markup Language“ (SGML) haben sich aufgrund ihrer hohen Komplexität nicht in der Breite durchsetzen können, dienten jedoch jüngeren Sprachen wie der „Hyper-Text Markup Language“ (HTML) oder der XML als konzeptuelle Vorlage. Somit läßt sich die XML, bezüglich ihrer zugrundeliegenden Architektur, als Untermenge der SGML verstehen.

3.5.1 eXtensible Markup Language

Im Vordergrund der XML steht die Trennung der Daten bzw. des Inhalts von den Metadaten bzw. dem Inhaltsmodell. In einem XML Dokument werden die Daten mittels Paaren von „Tags“ (identifizierbare Markierungen) strukturiert. Diese Tags werden durch die Literale „<“ für startende Tags bzw. „</“ für beendende Tags und dem Literal „>“ kenntlich gemacht. Zudem besitzt jedes Tag einen eindeutigen Namen gemäß dem Element, welches es repräsentiert. Diese Elemente können optional über eine beliebige Menge von Attributen verfügen, weitere Elemente enthalten und auch von anderen Elementen referenziert werden. Referenzen und Kompositionen sind dabei die wesentlichen Konzepte zur Strukturierung der Elemente, welche immer in unterhalb eines gemeinsamen „Rootelementes“ angeordnet sein müssen.

Separat zu dem XML-Dokument kann das darin enthaltene Inhaltsmodell mittels Document Type Declaration's (DTD's) beschrieben werden. Diese DTD's enthalten damit die vollständige Strukturbeschreibung der verwendeten XML-Sprache bzw. des XML-Dialektes. In der Praxis sind die DTD's optional, da ein Parser die Struktur des XML-Dokuments anhand der durchgängigen Verwendung von Start- und Endtags rekonstruieren kann. Wird dagegen die Verwendung von gültigen XML-Dokumenten gefordert, müssen diese anhand ihrer DTD's validiert werden.

Die Beschreibung des Inhaltsmodells läßt sich mit den folgenden vier verschiedenen Deklarationsarten durchführen:

Element Type Declaration Mit der Element Type Declaration werden sämtliche zur Verfügung stehenden Elemente benannt und der von ihnen enthaltene Inhalt definiert.

Attribute List Declaration Die Attribute List Declaration gibt an, welche Elmente Attribute enthalten und welchen Wert diese Attribute besitzen können. Weiterhin können in der Attribute List Declaration auch Vorbelegungen von Attributen mit bestimmten Werten (default values) vorgenommen werden.

Entity Declarations Entities werden in XML benutzt, um zum einen den Inhalt von separaten Dateien in das Dokument mit einzubeziehen und zum anderen, um Aliase von beliebigen Zeichen und Zeichenketten, wie z.B. den Sonderzeichen „<“, „>“ etc., bereitzustellen.

Notation Declaration In der Notation Declaration lassen sich Referenzen auf externe Dateien mit binärem Inhalt angeben. Diese Binärdateien werden nicht geparst sondern direkt an die Anwendung, welche das XML-Dokument einlesen und interpretieren soll, durchgereicht.

Neben der Deklaration des Inhaltsmodells eines XML-Dokumentes anhand von DTDs existiert mittlerweile auch eine weitere Möglichkeit, derartige Dokumente zu beschreiben. XML-Schema ist der offizielle Nachfolger der XML DTD und behebt dessen Schwachstellen bezüglich der Ausdrucksmächtigkeit zur Definition von XML-Sprachen. So haben z.B. eine Reihe zusätzlicher Datentypen in XML Schema Einzug gehalten, die Parametrisierbarkeit der einzelnen Elementdeklarationen wurde erhöht und die Modellierung von Mengeneigenschaften wurde vereinfacht. Einen detaillierten Überblick zu XML, XML DTD's und XML Schema findet man in [W3C04a] bzw. in [W3C04b].

3.5.2 XML Metadata Interchange

Unabhängig davon, ob zur Beschreibung der XML-Sprache ein DTD oder ein Schema definiert wird, bedeutet eine solche manuelle Definition einen beträchtlichen Aufwand auf Seiten der Entwickler. Ebenso müssen die DTDs gewartet und an Änderungen des Metamodells angepaßt werden. Da bei der Verwendung des MDA-Ansatzes Metamodelle genutzt werden, die wiederum Instanzen des MOF-Modells sind und eines der Kernziele dieses Ansatzes die automatisierte Transformation von Modellen ist, wurde anfang 1999 ein Verfahren zur automatischen Abbildung von MOF auf XML von der OMG standardisiert. Dieser Standard heißt XML Metadata Interchange und befindet sich derzeit in der Version 2.0. Die Spezifikation [OMG02b] schlägt ein kompaktes Regelwerk zur Generierung von XML-Sprachen, deren MOF-konforme Metamodelle bekannt sind, vor. XMI ermöglicht somit die Implementation generischer Werkzeuge, die in eine MDA-Architektur integriert werden können und den Austausch von Metadaten mittels XML realisieren. Nebst der Erzeugung der XML DTD zur Beschreibung der abstrakten Syntax der Modelle können auch zu diesen DTDs valide Dokumente generiert werden. Die XMI-Spezifikationen in den Versionen 1.0, 1.1 und 1.2 bestehen aus zwei Kernkomponenten, den *„XML DTD*

Production Rules“ und den *„XML Document Production Rules“*. In der neuesten XMI-Spezifikation — der Version 2.0 — wird zudem ein Regelwerk angegeben, mit Hilfe dessen die Abbildung von MOF auf XML Schema vorgenommen werden kann.

Eines der prominentesten Beispiele für ein MOF-konformes Metamodell, das von Modellierungswerkzeugen genutzt und deren Instanzen mittels XMI ausgetauscht werden können, ist das UML-Metamodell.

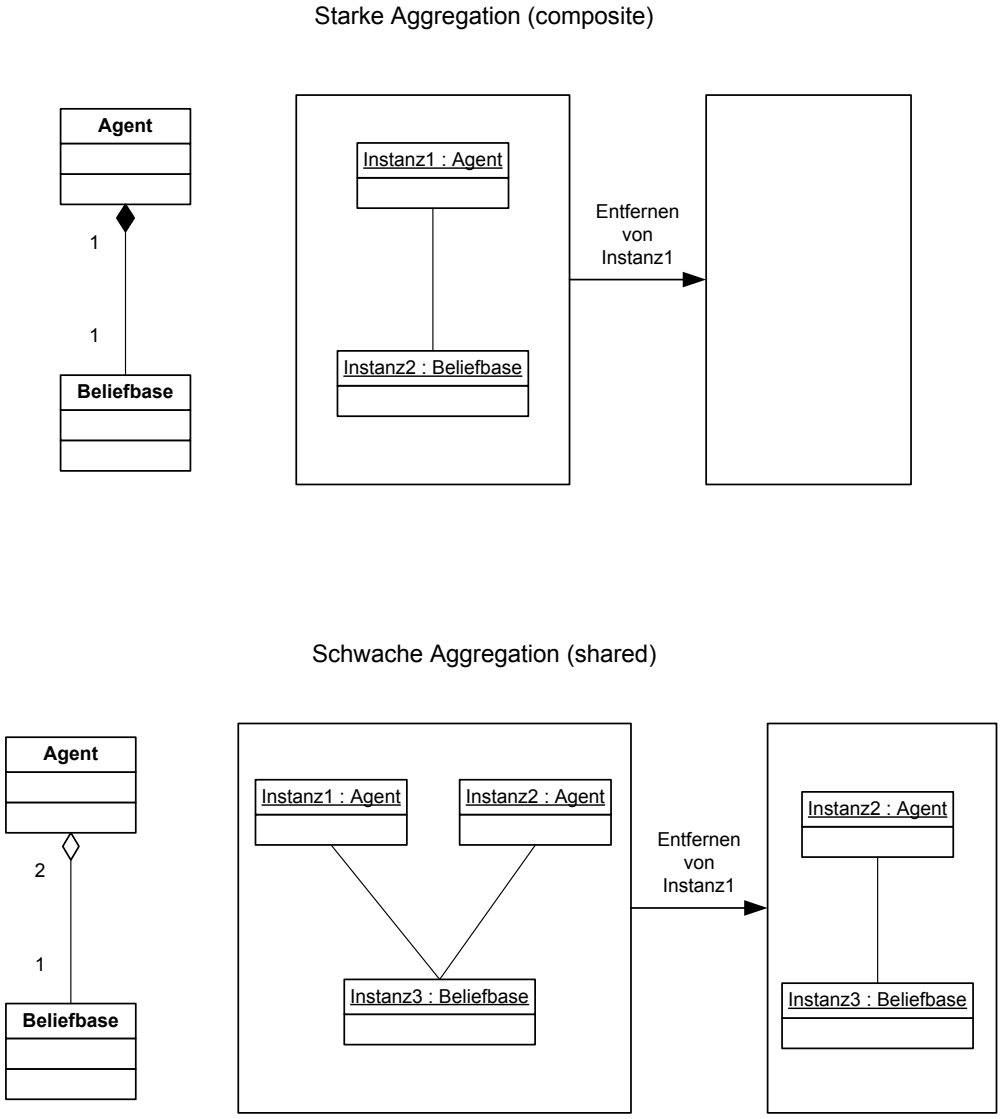


Abbildung 3.4: Arten der Aggregation

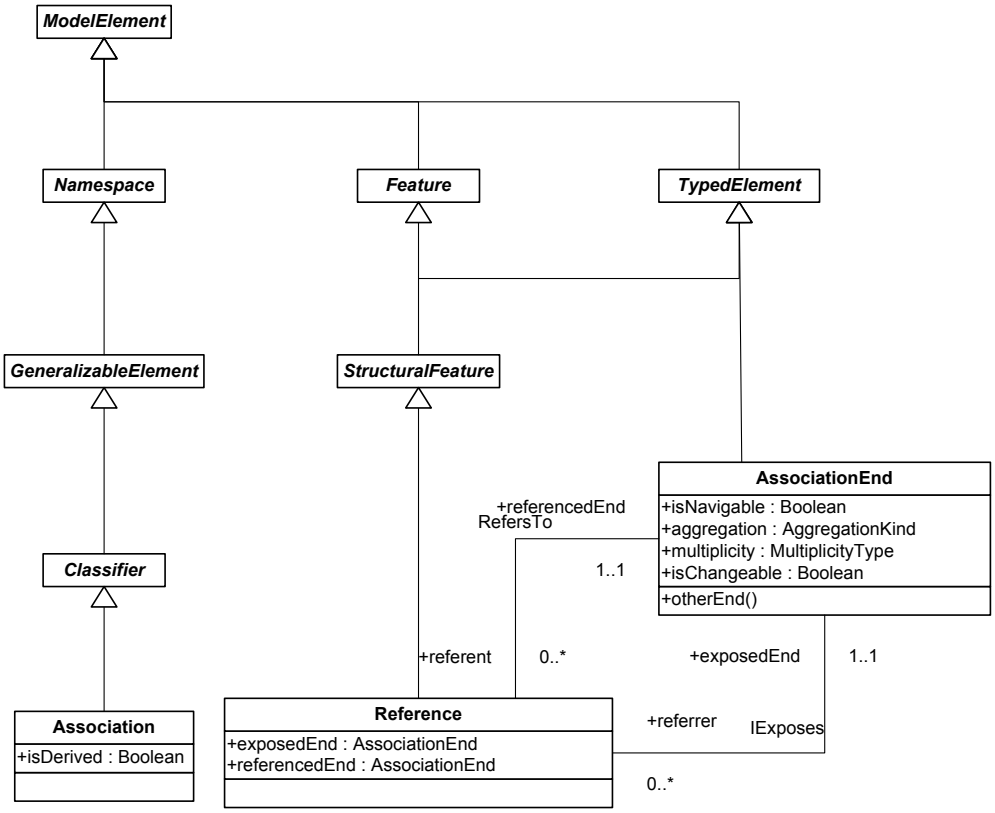


Abbildung 3.5: Assoziationen im MOF-Modell [UC02]

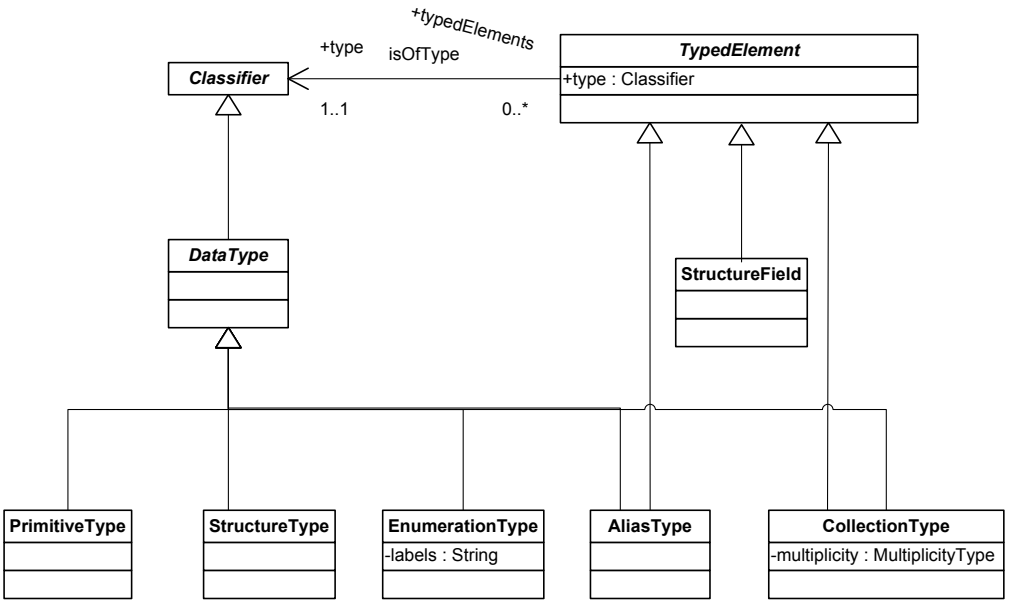


Abbildung 3.6: Datentypen im MOF-Modell [UC02]

4 Analyse und Entwurf der Werkzeugarchitektur

Nachdem in Kapitel 2 die allgemeinen Grundlagen des Agentenparadigmas vorgestellt wurden und Kapitel 3 einen Ansatz zur modellbasierten Anwendungsentwicklung eingeführt hat, wird in diesem Kapitel dargestellt, wie sich diese beiden Bereiche sinnvoll zusammenführen lassen. Ziel dieser Zusammenführung ist die Realisierung eines metamodellbasierten Entwurfswerkzeuges für BDI-Agentensysteme.

4.1 Einordnung des Werkzeuges

Aufgrund der Tatsache, daß bereits einige Werkzeuge existieren, die den Entwurf von Agenten- und Multiagentensystemen ermöglichen bzw. vereinfachen, bleibt vorerst die Frage nach der Notwendigkeit eines weiteren Entwurfswerkzeuges zu klären. Hierfür ist eine Einordnung des Werkzeuges in den Prozeß der Entwicklung von Agentensystemen zweckmäßig. Dieser Prozeß ähnelt dem Prozeß der traditionellen Softwareentwicklung insofern, daß sich auch hier alle Phasen, von der Analyse eines Problembereiches über den Entwurf einer Problemlösung bis hin zur Implementation einer konkreten Lösung, sowie deren Überprüfung mittels einer abschließenden Testphase, wiederfinden sollten. Da das Gebiet der agentenbasierten Softwareentwicklung jedoch noch relativ jung ist, bzw. es noch an in der industriellen Praxis erfolgreich eingesetzten agentenbasierten Softwarelösungen mangelt, ist der bisher gesammelte Erfahrungsschatz im Umgang mit derartigen Systemen eher gering. Ein allgemeingültiger und weit verbreiteter Entwicklungsprozeß hat sich in diesem Bereich noch nicht herauskristallisiert.

Statt dessen findet sich in der Literatur ein breit gefächertes Repertoire an Ansätzen und Vorschlägen, die konkretisieren, wie beim Entwurf eines Agentensystems zu verfahren ist. Derartige Vorschläge resultieren meist in einer Reihe von Entwurfsmethoden. Einen umfangreichen Überblick sowie eine Gegenüberstellung solcher Methoden bietet [Sud04].

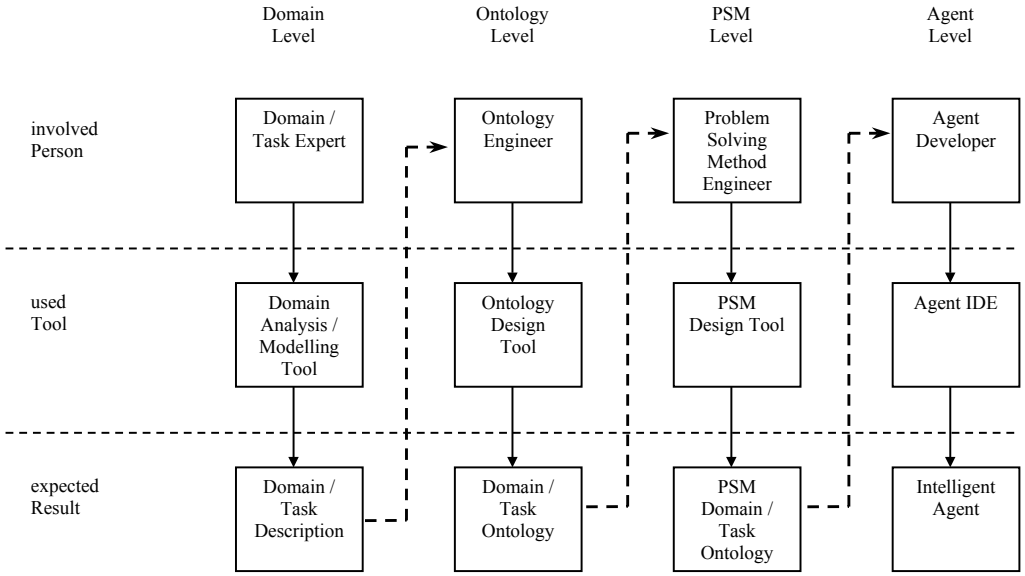


Abbildung 4.1: Entwicklung von Agenten und Agentensystemen (nach [KNB+03])

Neben diesen Methoden existiert auch eine Reihe von Werkzeugen, die dem Anwendungsentwickler eine Unterstützung bei der Umsetzung verschiedener Techniken zur Entwicklung von Agentensystemen bietet. Eine solche Technik ist die des Modellierens einzelner Aspekte von Agenten- und Multiagentensystemen. [KNB+03] stellt einen Entwicklungszyklus vor, der sich in vier Ebenen bzw. Phasen untergliedert. Jede dieser Ebenen umfaßt einen separaten Aspekt eines solchen Systems und nominiert geeignete Werkzeuge für dessen Modellierung. Die Ebenen sind im einzelnen:

- Ebene der Domäne
- Ebene der Ontologien
- Ebene der Problemlösungsmethoden
- Ebene der Agentensysteme

Der Prozess des Modellierens des Gesamtsystems lässt sich somit als eine Aufeinanderfolge von Teilprozessen in den jeweiligen Ebenen veranschaulichen (siehe Abbildung 4.1).

Diese arbeitsteilige Entwicklung von Agentensystemen geht einher mit einer Spezialisierung von Experten in den einzelnen Bereichen. Diese Experten, welche ihr Fachwissen in den einzelnen Ebenen einbringen sollen, müssen durch eine geeignete

Werkzeugunterstützung in die Lage versetzt werden ihre Modelle miteinander zu kommunizieren bzw. zwischen den Ebenen zu transferieren. Eine Kommunikation über die Grenzen einer Ebene hinweg wird zur Zeit noch durch eine fehlende Integration der einzelnen Werkzeuge erschwert. Aus diesem Sachverhalt ergeben sich eine Reihe von Anforderungen, sowohl an die zu erstellenden Modelle als auch an die verwendeten Werkzeuge. Diese Anforderungen werden in den Abschnitten 4.3.1 und 4.3.2 herausgestellt.

Das metamodellbasierte Entwurfswerkzeug dieser Arbeit gehört zur Ebene der Agentensysteme. In dieser Ebene befinden sich neben grafischen Modellierungswerkzeugen zum Entwurf von Agenten auch Frameworks zur Programmierung von Agenten z.B. Jadex (siehe Abschnitt 2.3.2). Des weiteren befindet sich in dieser Ebene auch das Agentenframework JACK ([Age04]). Dieses Framework verfügt über eine integrierte Entwicklungsumgebung, in der das Verhalten von Agenten sowohl programmiert als auch modelliert werden kann.

Bevor sich die folgenden Abschnitte eingehend mit den Anforderungen an das zu entwickelnde Entwurfswerkzeug beschäftigen, sowie die Art und Weise seiner Verwendung und die damit verbundene Grundfunktionalität skizzieren, werden allgemeine Überlegungen bezüglich des Entwurfsprozesses dargelegt.

4.2 Der Entwurfsprozeß

Dem Entwurfsprozeß und der daraus resultierenden Spezifikation eines Agentensystems sollte eine geeignete Methode zugrunde liegen. In der Regel stellen derartige agentenbasierte Entwurfsmethoden eine Erweiterung von etablierten Methoden aus der Objektorientierung dar. Diese Methoden zielen auf eine Dekomposition des Systems in die wesentlichen Objektklassen der Anwendungsdomaine ab, und fokussieren dabei auf deren Verhalten und deren Beziehungen zu anderen Klassen. Um dieses Ziel zu erreichen werden in der Regel drei verschiedene Modelltypen vorgeschlagen, das Objektmodell, ein dynamisches Modell und ein funktionales Modell. Das Objektmodell enthält die Informationen über die Objekte eines Systems, deren Struktur, deren Operationen und ihre Beziehungen untereinander. Eine übliche Repräsentation eines solchen Modells ist das Klassendiagramm der UML. Die dynamischen Aspekte des Systems, also eine Beschreibung des Verhaltens dieser Objekte mittels Zuständen, Zustandsübergängen, Aktionen und Interaktionen werden im dynamischen Modell abgebildet. Die UML bietet für diese Zwecke die Notationsformen der Zustandsdiagramme an. Das funktionale Modell beschreibt dagegen den Datenfluß

sowohl innerhalb als auch zwischen Systemkomponenten. Für diese, die Abläufe innerhalb eines Systems wiedergebende Sicht, haben sich das Kollaborationsdiagramm als auch das Sequenzdiagramm bewährt. Den Kernpunkt dieser objektorientierten Methode bilden demnach die Objektmodelle, welche anhand des dynamischen Modells und des funktionalen Modells verfeinert werden. Diese, für objektbasierte Systeme entwickelte Methode ist für die Spezifikation eines BDI-Agentensystems nicht ausreichend (vgl. [KGR96]), da sie weder eine Hilfestellung bei der Modellierung mentaler Konzepte beinhalten, noch auf die höhere Komplexität von Agenten, so man sie denn als Objekt auffaßt, eingehen.

In dieser Arbeit wird jedoch weder eine Empfehlung für eine konkrete Methode noch ein Vergleich zwischen der Vielzahl bereits existierender Methoden gegeben. Statt dessen orientiert sich das Werkzeug an einer agentenorientierten Methode, welche von [KGR96] vorgeschlagen wurde, und wie sie für den Entwurf von BDI-Agenten genutzt werden kann. In Analogie zur objektorientierten Modellierung von Softwaresystemen unterscheidet man bei dieser Methode zur Modellierung von Agentensystemen zwischen einer externen und einer internen Sichtweise.

Bei der externen Sicht auf einen Agenten betrachtet man in erster Linie die Dienstleistungen, die er seiner Umgebung zur Verfügung stellt. Der Agent nimmt im System also eine Rolle ein und ist für einen bestimmten Aufgabenbereich verantwortlich. Um diese Dienstleistungen zu erbringen, und somit seine Verantwortung wahrzunehmen, kann eine Kooperation mit anderen Agenten eingegangen werden. Die dabei ausgeübte Interaktion zwischen verschiedenen oder gleichen Agenten betrachtet man üblicherweise ebenso in der externen Sicht. Wie der Agent diese Dienstleistungen konkret erbringt, muß für den Außenstehenden (Externen), welcher entweder eine Person oder wiederum ein Agent ist, unerheblich sein. Der Agent wird im Sinne der Systemtheorie also als sogenannte *Black-Box* verstanden. Um die externe Sicht auf Modelle abzubilden, bieten sich ein Agentenmodell und ein Interaktionsmodell an. Das Agentenmodell hat dabei die Aufgabe, die Agentenklassen zu beschreiben und deren Instanzen zu identifizieren. Des weiteren soll im Agentenmodell auch die Beziehung zwischen den Agentenklassen abgebildet werden. Das Interaktionsmodell greift dagegen die Beschreibung der vom Agenten nach außen hin angebotenen Dienstleistungen und Verantwortungsbereiche auf. Dies beinhaltet ebenso die Interaktionen zwischen den Agenten und die Modellierung von Nachrichten, mit denen die Agenten kommunizieren.

Die externe Sicht soll in dieser Arbeit nicht weiter aufgegriffen werden. Das zu realisierende Entwurfswerkzeug wird ausschließlich die Aspekte der internen Sicht berücksichtigen. Um ein detailliertes Modell eines einzelnen Agenten zu erhalten,

bedarf es einer solchen internen Sicht. In dieser Sicht wird die interne Struktur eines Agenten betrachtet. Die Spezifikation beschränkt sich dabei auf Modelle der mentalen Konzepte eines Agenten. Liegt dem Agenten eine BDI-Architektur zugrunde schlägt [KGR96] vor, diese mentalen Konzepte für jede Agentenklasse mittels separater Modelle darzustellen. Hieraus resultieren dann jeweils ein Beliefmodell, ein Planmodell und ein Modell für die Ziele eines Agenten, das Goalmodell. Auf diese interne Strukturbeschreibung soll das Entwurfswerkzeug fokussieren, jedoch auch zukünftige Erweiterungen für die Beschreibung der externen Aspekte unterstützen. In Abschnitt 4.7 wird gezeigt, wie sich diese drei Modelle in ein gemeinsames BDI-Agentenmetamodell einfügen.

4.3 Einsatz des Werkzeuges

Der Einfachheit halber, und um Verwechslungen bei der Benutzung des Begriffes *Werkzeug* bzw. *Entwurfswerkzeug* mit anderen Werkzeugen zu vermeiden, wird von nun das in dieser Arbeit entwickelte metamodellbasierte Entwurfswerkzeug für BDI-Agentensysteme mit seinem Namen *Agent Modeling Kit*, bzw. mit seinem Akronym *AMoK* angesprochen.

Beim Entwurf von Agentensystemen ist im Vorfeld zu untersuchen, welche bereits vorhandenen Infrastrukturen genutzt werden können. Diese Infrastrukturen reichen von Sprachen zur Beschreibung von Agenten, über Entwicklungsumgebungen, bis hin zu Laufzeitumgebungen für Multiagentensysteme, die teils aus dem akademischen, teils aus dem kommerziellen Bereich stammen. [Man02] stellte bereits 2002 eine Auflistung von 36 solcher Softwareprodukte vor.

Das Ziel von AMoK ist es, den Übergang von der Entwurfsphase in die Implementationsphase zu erleichtern. Die mit AMoK erstellten Agentenmodelle sollen demnach automatisch in eine plattformspezifische Repräsentationsform transformiert werden können. Diese Repräsentationsformen können z.B. die *Agent Description Files*, wie sie von *jadex* interpretiert werden, sein. Eine solche Unterstützung sollte jedoch nicht auf nur eine einzelne Plattform für Multiagentensysteme zugeschnitten sein, sondern mit einfachen Mitteln auch auf weitere solche Plattformen ausgeweitet werden können. Zusätzlich ist festzustellen, daß die Multiagentenplattformen selbst auch einer ständigen Weiterentwicklung unterliegen können, was sich auch auf die verwendeten Agentenarchitekturen auswirken kann.

Die sich aus diesen Sachverhalten abgeleitete Erweiterbarkeit von AMoK ist daher

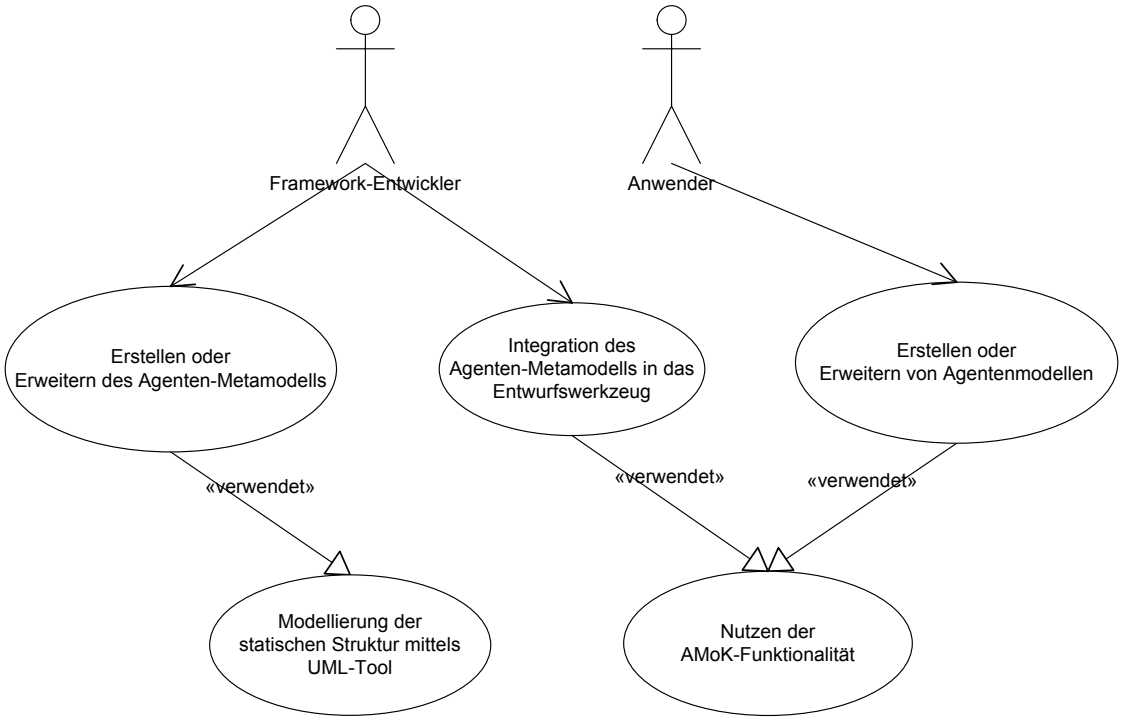


Abbildung 4.2: Anwendungsfalldiagramm - Einsatz von AMoK

eine essentielle Anforderung an das Werkzeug, die sich in dessen gesamter Architektur widerspiegelt. Um diese Erweiterbarkeit zu ermöglichen, soll das dem Werkzeug zugrundeliegende Metamodell für BDI-Agenten anpaßbar, bzw. austauschbar sein.

Beim Entwurf von Agentenspezifikationen mit AMoK lassen sich zwei verschiedene Sichten identifizieren. Die eine Sicht definiert die Rolle des Framework-Entwicklers, während sich die andere Sicht auf den tatsächlichen Einsatz von AMoK als Werkzeug zur agentenbasierten Anwendungsentwicklung bezieht. Das Zusammenspiel, der bei der Verwendung von AMoK beteiligten Akteure und ihre jeweiligen Aufgabenbereiche, ist in Form eines vereinfachten Anwendungsfalldiagramms, siehe Abbildung 4.2, dargestellt und wird im folgenden genauer betrachtet. Ziel dieser Betrachtung ist es, aus der Analyse der Anwendungsfälle konkrete Anforderungen bezüglich der Funktionalität und der Benutzbarkeit an das Entwurfswerkzeug „AMoK“ herzuleiten.

4.3.1 Rolle des Framework-Entwicklers

Die am Entwurfsprozeß des Agentenframeworks beteiligten Entwickler übernehmen die Aufgabe, AMoK an dieses anzupassen. Dazu müssen sie die Struktur der statischen Aspekte ihrer Agentenarchitektur in Form eines umfassenden Metamodells definieren. Das Metamodell muß MOF-konform sein, um anschließend in AMoK integriert werden zu können. Desweiteren muß eine Komponente implementiert werden, die die mit AMoK modellierten Agenten in die von dem Agentenframework vorgesehene Beschreibungsform, z.B. in eine XML-Notation, überführen. Es empfiehlt sich, daß die Integration des Metamodells und das Bereitstellen einer Komponente zur automatisierten Codegenerierung ebenso vom Framework-Entwickler vorgenommen wird, da dieser am ehesten in der Lage sein sollte, das Metamodell zu testen und bei dessen Einsatz auftretende Fehler im Vorfeld zu beheben. Mit dem Wissen über die Semantik aller im Metamodell definierten Modellelemente soll die Benutzeroberfläche von AMoK abschließend an die Vorstellungen der potentiellen Anwender angepaßt werden.

Um die Entwickler des Agentenframeworks bei der Bewältigung obiger Aufgaben zu unterstützen, sollte AMoK über folgende Funktionen verfügen:

Integration von Metamodellen: Das Werkzeug muß die Möglichkeit bieten, MOF-konforme Metamodelle aus einem geeigneten Austauschformat zu importieren. Liegt das Metamodell in Form eines UML-Klassendiagramms vor, so muß eine transparente Transformation in ein MOF-Metamodell vorgenommen werden können. Ein geeigneter Mechanismus sollte den Zugriff auf die Elemente des Metamodells regeln, und allen an der Verarbeitung beteiligten Systemkomponenten zur Verfügung stellen.

Anpassung der Visualisierung von Modellelementen: Um den späteren Benutzern den Einstieg in das Entwurfswerkzeug zu erleichtern und die Unterscheidung der einzelnen Modellelemente voneinander anhand einer geeigneten grafischen Notation zu ermöglichen, muß neben dem Metamodell selbst auch ein Präsentationsmodell integriert werden können. Die Instanzen des Präsentationsmodells beschreiben dann, wie ein Modellelement visualisiert werden soll.

Einbettung von Code-Generatoren: AMoK muß eine Schnittstelle besitzen, mittels derer Komponenten zur Spezifikationserzeugung von den Entwicklern hinzugefügt werden können.

4.3.2 Rolle des Anwenders

Den Entwicklern von Agentensystemen, die aus Sicht des Werkzeuges die Rolle des Anwenders übernehmen, muß AMoK ermöglichen, von den soeben geschilderten technischen Details abstrahieren zu können. Diese sollen mit AMoK ein auf einem korrekten Metamodell basierendes Modellierungswerkzeug vorfinden, und sich ungehindert auf den Entwurf der Agentenmodelle konzentrieren können.

Die Agentenmodelle müssen derart formuliert werden können, daß sie einen Agenten auf einfache und verständliche Art und Weise repräsentieren. Sie erfüllen damit die Aufgaben der Kommunikation, der Visualisierung und der Überprüfbarkeit der einzelnen Agenten, bzw. des Agentensystems.

Unter der Aufgabe der Kommunikation sei in diesem Kontext anlehnend an [GBB03] der gesamte Bereich des Schaffens einer gemeinsamen Verständigungsbasis gemeint. Um konzeptuelle Fehler bei der Konstruktion eines Agentensystems bereits im Vorfeld zu minimieren ist es unerläßlich, daß alle an dessen Entwicklung beteiligten Personen eine gemeinsame Terminologie benutzen und mit Hilfe der erstellten Modelle über das Multiagentensystem kommunizieren können.

Eng mit der Aufgabe der Kommunikation verbunden ist der Aspekt der Visualisierung des Modells durch eine geeignete Notationform. Eine grafische Notation hat sich in der Praxis vorwiegend dort gegenüber einer rein textuellen Beschreibung durchgesetzt, wo das schnelle Erschließen komplexer Zusammenhänge im Vordergrund steht. Des weiteren sind grafische Notationsformen auch aus anderen Modellierungssprachen, wie z.B. der UML oder die um Konstrukte der Agentenkommunikation erweiterte AUML (*Agent UML*) bekannt. Eine Anlehnung an bestehende und bekannte grafische Notationen erleichtert den Einstieg in das Modellieren und ermöglicht einen intuitiveren Umgang mit den Modellen. Abschließend soll das Agentenmodell auch der Verifikation des Agentensystems dienen können. Dazu muß aus ihm hervorgehen, ob alle Aspekte vollständig, korrekt und widerspruchsfrei erfaßt wurden.

Die aus der Anwendersicht abgeleiteten Anforderungen lassen sich nun wie folgt formulieren.

Verwalten der Agentenmodelle: Der Anwender muß auf intuitive Art und Weise Agentenmodelle aus dem Metamodell ableiten können. Diese Agentenmodelle sollen in einem zentralen Repository persistent aufbewahrt werden. Außerdem soll das Werkzeug dem Anwender eine Übersicht über alle im Repository vorhandenen Agentenmodelle zur Verfügung stellen, damit der Anwender bereits vorhandene Modelle auswählen, modifizieren und auch entfernen kann.

Austausch der Agentenmodelle: Die Agentenmodelle sollen zur Wahrung der Interoperabilität mit anderen Werkzeugen, oder auch mit weiteren AMoK-Instanzen, in ein standardisiertes Austauschformat überführt und in einer Datei abgelegt werden können. Analog dazu soll ein Importmechanismus von Agentenmodellen aus derartigen Dateien vorhanden sein.

Bearbeiten der Agentenmodelle: Sämtliche Attribute der Modellelemente sollen über ein Formular editierbar sein, wobei das Formular eine auf die verschiedenen Attributtypen zugeschnittene Eingabemöglichkeit vorsehen kann. Jedes Modellelement soll neben der Repräsentation über derartige Formulare auch grafisch in einem Diagramm dargestellt, und in diesem, zur Verbesserung der Lesbarkeit des Diagramms, beliebig arrangiert und mit Kommentaren versehen werden können.

Verifizieren der Agentenmodelle: Dem Anwender soll optisch signalisiert werden, welche Aspekte bereits modelliert wurden und welche noch einer weiteren Bearbeitung bedürfen.

Generieren von Agentenspezifikationen: Agentenmodelle sollen automatisch in eine für das jeweils zugrundeliegende Agentenframework adäquate Spezifikation transformiert werden können.

4.4 Architektur des Entwurfswerkzeugs

Die an das Entwurfswerkzeug gestellten Anforderungen bezüglich der Funktionalität, der Anpassbarkeit und der Erweiterbarkeit müssen sich in dessen Architektur widerspiegeln. In den nun folgenden Sektionen wird dargelegt, aus welchen Kernkomponenten die Anwendungsarchitektur aufgebaut, und nach welchen Auswahlkriterien diese ermittelt wurden. Im Mittelpunkt der konzeptuellen Überlegungen beim Entwurf dieser prototypischen Architektur stand eine umfangreiche Analyse des Umfelds modellgetriebener Anwendungen, um von bestehenden Standards und Standardisierungsbemühungen profitieren zu können.

Der Kern der vorgeschlagenen Architektur besteht aus einer Modellmanagement-Komponente, einer Editorkomponente, einer Templateengine und aus einem exemplarischen Plugin zur Generierung von Jadex-ADF. Der Zugriff auf diesen Applikationskern durch den Anwender wird innerhalb einer gemeinsamen grafischen Benutzeroberfläche gekapselt. Diese Benutzeroberfläche stellt sich für den Benutzer als

eine homogene Anwendung — dem eigentlichen „Agent Modeling Kit“ dar — deren gesamter Funktionsumfang menügeführt abgerufen werden kann.

Die Modellmanagement-Komponente ihrerseits kapselt eine konkrete Implementation eines Metadaten Repositories, wie es von der OMG angeregt wurde.

Neben den in der Architektur verankerten Technologien ist der in dieser Arbeit vertretene Ansatz von einem externen UML-Modellierungswerkzeug zur Spezifikation des Metamodells abhängig. Diese Abhängigkeit bedingt Wechselwirkungen bei der Auswahl geeigneter Standardimplementationen und wird in den jeweiligen Abschnitten zu den einzelnen Komponenten gesondert angesprochen.

Die in Abbildung 4.3 dargestellte Architektur veranschaulicht das Zusammenspiel der einzelnen Komponenten. Mithilfe eines UML-Modellierungswerkzeuges, dem sogenannten CASE-Tool, wird ein MOF-konformes Metamodell erstellt. Eine Umsetzung des Standards „UML-Profile for MOF“ (siehe Abschnitt 3.4) benutzt den Erweiterungsmechanismus der „Unified Modeling Language“ und erweitert die Konstrukte des UML-Klassendiagramms um eine Menge von „Stereotypes“. Innerhalb dieser Erweiterung kann die statische Struktur der Agenteninterna als Klassendiagramm abgebildet und mittels Metadaten-Austauschformat (vgl. Abschnitt 3.5.2) exportiert werden.

Die Modellmanagement-Komponente ist in der Lage, das in der UML notierte Metamodell mit Hilfe eines generischen XMI-Readers in das Metadaten Repository zu importieren und in ein MOF-konformes Metamodell zu transformieren. Diese Abbildung von UML nach MOF wird in Abschnitt 5.3.1 erörtert. Ein anschließender Export des nach MOF transformierten Metamodells ist optional. Das Gleiche gilt für die mit AMoK erstellten Agentenmodelle.

Eine ebenfalls generisch arbeitende Implementation des Standards JSR 040, der sogenannten JMI „Java Metadata Interfaces“, stellt eine Abbildung der Meta-Metadaten als auch der Metadaten auf Java Schnittstellen her. Parallel zu diesen Schnittstellen, welche in Abschnitt 5.1 besprochen werden, kommt eine reflektive API zum Einsatz. Die reflektive API wird von den Komponenten benutzt, die kein explizites Wissen über die Semantik der einzelnen Modellelemente haben. Zu diesen Komponenten gehören das Metadaten Repository und die Editorkomponente, die jeweils vollständig auf generischen Algorithmen basieren. Die Vorgabe reflektive Mechanismen zur Introspektion und Manipulation der Metadaten einzusetzen, ergibt sich direkt aus den Flexibilitätsbedingungen der Anforderungsanalyse. Demnach soll das Entwurfswerkzeug auf einem beliebigen, nicht im Vorwege feststehenden, Metamodell arbeiten können.

Die Editorkomponente ermöglicht die Agentenmodelle textuell, unterstützt durch einen prototypischen Formulargenerator, zu editieren. Der Diagrammeditor arbeitet ebenfalls generisch und stellt die Elemente, sowie ihre Relationen zueinander, grafisch dar. Um dem Anwender eine erwartungskonforme Haptik und einen aus anderen grafischen Modellierungswerkzeugen bekannten Komfort bei der Handhabung der Strukturdiagramme zu bieten — und dennoch den Implementationsaufwand für diese Komponente so gering wie möglich zu halten — wurde auf das Framework JGraph (siehe Abschnitt 5.6.1) zurückgegriffen.

Für die Generierung der Agentenspezifikationen, gemäß der Agentenmodelle, zeichnen die Templateengine Velocity von Apache sowie eine proprietäre Plugin-Schnittstelle verantwortlich. Ein bereits implementiertes Plugin zur Erzeugung eines ADF veranschaulicht, wie die einzelnen Elemente der Agentenmodelle mittels Java-Interfaces abgefragt werden können. Die aus diesem Vorgang resultierenden Daten können mit einem für ADFs vordefinierten Template zusammengeführt werden. Den Vorgang des Zusammenführens der Agentendetails und des Templates übernimmt die Templateengine.

4.5 Die Modellmanagement-Komponente

Der zentrale Bestandteil der soeben vorgeschlagenen Architektur ist die Modellmanagement-Komponente. Für diese Komponente wurde zuerst untersucht, ob schon wiederverwendbare Softwarelösungen existieren, die den folgenden Kriterien genügen:

Bereitstellung der Kernfunktionalität Die jeweilige Implementation sollte den Import und Export von Modellen mittels XMI-Reader bzw. XMI-Writer erlauben. Weiterhin sollten Mechanismen vorhanden sein, die eine Abbildung der Modelle auf Java-Schnittstellen ermöglichen.

Persistenz der Metadaten Die Metadaten sollten persistent vorgehalten werden.

Offener Code Die existierenden Lösungen sollten möglichst in Java codiert und als Open Source-Projekt realisiert sein, um die Integration in AMoK zu erleichtern.

Verwendung offener Standards Um die Interoperabilität des Entwurfswerkzeuges auch zukünftig gewährleisten zu können, sollte die in diesem Kontext beson-

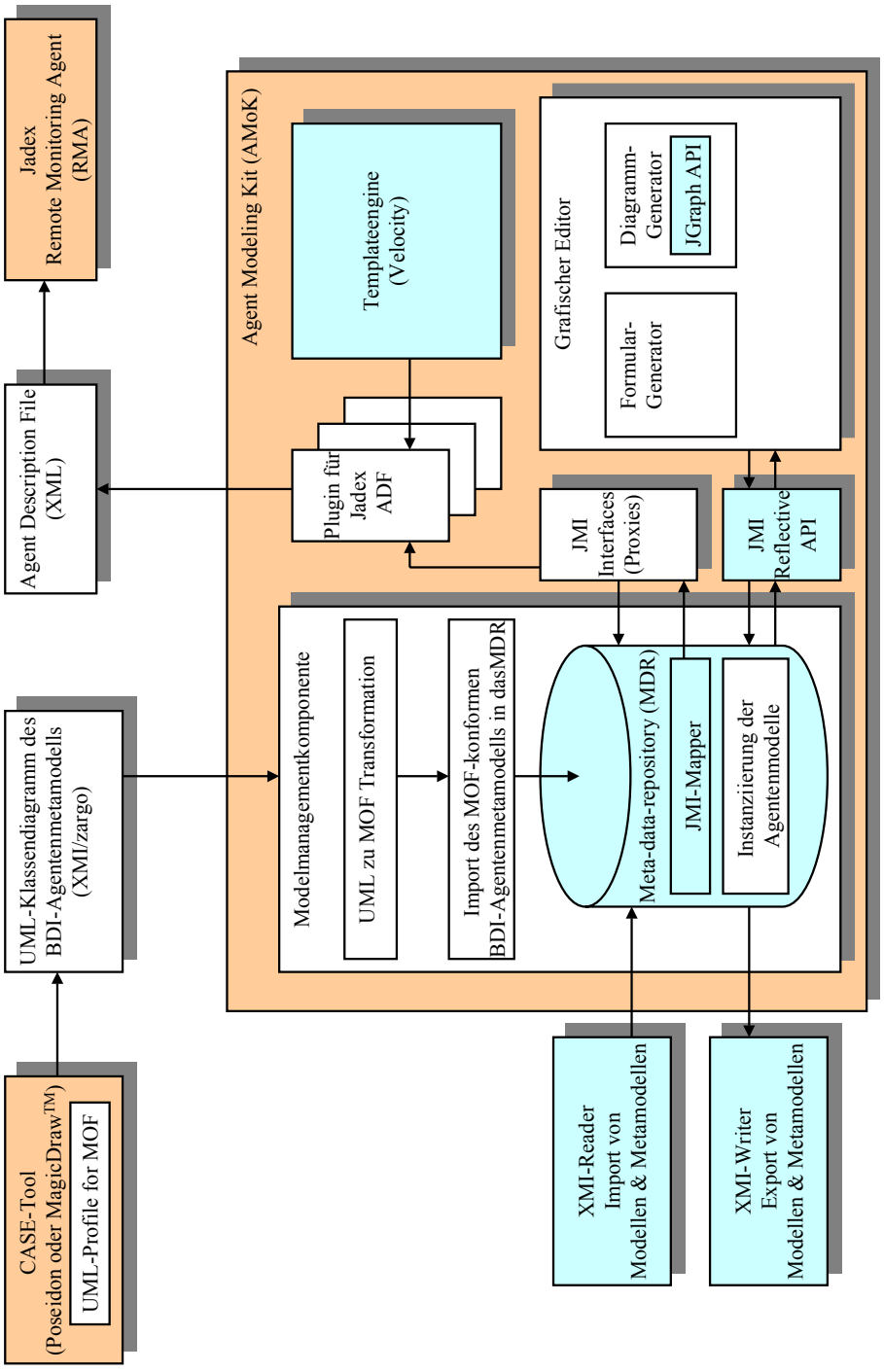


Abbildung 4.3: AMoK Architektur

ders essentielle Modellmanagement-Komponente auf offenen und etablierten Standards aufsetzen können.

Dokumentation und Support Ein ebenfalls nicht zu vernachlässigendes Kriterium ist das der Unterstützung bei möglichen Problemfällen und das Vorhandensein einer Dokumentation des vorliegenden Quellcodes.

Stand der Entwicklung Zweckmäßigerweise sollten für AMoK nur Implementationen in Betracht gezogen werden, die bereits in Referenzprojekten zum Einsatz kommen, oder zumindest ein hohes Maß an Vollständigkeit und Fehlerresistenz aufweisen.

Aus dieser Untersuchung resultieren zwei grundsätzlich verwendbare Softwarelösungen, nämlich das „Metadata Repository“ (MDR) der Firma „Sun Microsystems“ und das „Eclipse Modeling Framework“ (EMF) der „Eclipse Foundation“. Beide Implementationen unterstützen den Prozeß des „Model Driven Development“, unterscheiden sich jedoch in den Details ihrer Architektur.

4.5.1 Eclipse Modeling Framework (EMF)

Das Eclipse Modeling Framework ist ein relativ junger Bestandteil der Eclipse Plattform. Ziel der Eclipse Plattform ist es, die Integration verschiedener Werkzeuge aus dem Bereich der Softwareentwicklung zu vereinfachen. Erreicht wird dieses Ziel durch ein stark ausgeprägtes Plug-in Konzept der Eclipse Plattform selbst. Umfangreiche Informationen zur Eclipse Plattform findet man im Internet unter der Addresse www.eclipse.org. Das für diese Arbeit besonders interessante Plug-in ist die EMF. Neben der Integration des EMFs in Eclipse kann dieses auch in Standalone-Anwendungen genutzt werden.

Das EMF ermöglicht, ausgehend von einem Metamodell, Modelle zu erstellen und zugehörige Java-Implementationen zu generieren. Die abstrakte Syntax des verwendeten Metamodells selbst wird durch das EMF-Modell „Ecore“, siehe Abbildung 4.4 beschrieben. Ecore ist demnach das der EMF zugrundeliegende Meta-Metamodell und unterscheidet sich vom MOF-Modell insofern, daß es nur die wesentlichsten Konstrukte zur Definition von auf Klassen basierenden Modellen beinhaltet. Ecore kann als Untermenge des MOF-Modells, ähnlich dem „Essentiell MOF“ EMOF (siehe [OMG03]), aufgefasst werden. Serialisierungen des EMOF-Modells können mit dem EMF eingelesen und transparent auf das Ecore-Modell abgebildet werden.

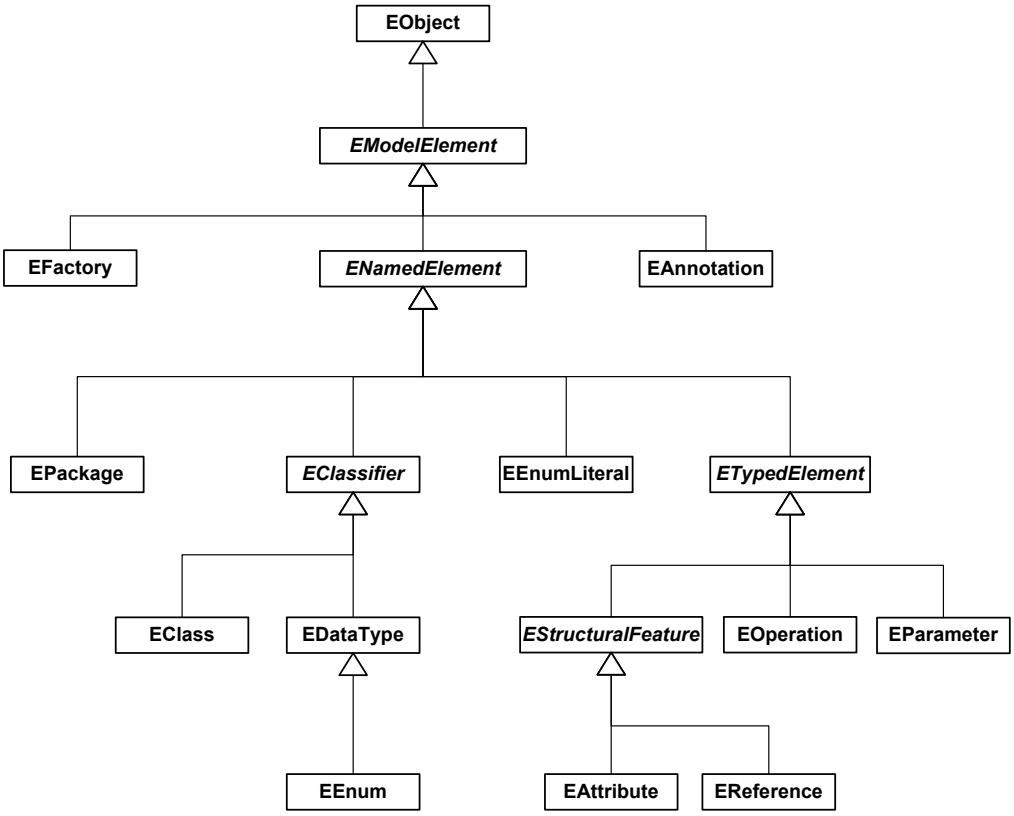


Abbildung 4.4: Ecore-Modell

Die Beschreibung des Ecore-Modells kann selbst wiederum mit dem Ecore-Modell vorgenommen werden. Ecore ist demnach selbstbeschreibend und läßt sich genau wie das MOF-Modell in die Ebene M3 der in Kapitel 3 erwähnten Modellhierarchie einordnen.

Zum Austausch von EMF-konformen Modellen zwischen auf EMF-basierenden Anwendungen wird eine Implementation des XML Metadata Interchange Standards 2.0 benutzt. Weitere Serialisierungsformen für die Modelle sind jedoch vorgesehen. Für die Erzeugung der Modelle werden drei alternative Wege vorgeschlagen. Ausgehend von einem zum Ecore-Modell konformen XMI-Dokument kann ein Modell mittels XML-Editor bearbeitet werden. Ist ein solches XMI-Dokument nicht vorhanden kann alternativ das Modellierungswerkzeug Rational Rose genutzt werden, um die statische Struktur des Modells mit UML-Klassendiagrammen zu beschreiben. Rational Rose implementiert ebenfalls die XMI Version 2.0 und ist somit in der Lage, das UML-Modell als XMI-Dokument zu exportieren. Die dritte Möglichkeit ist das Beschreiben der Modelle in der Programmiersprache Java. Dieser Weg ist

besonders für Programmierer interessant, die dennoch das EMF als Persistenzdienst oder zum Austausch der Modelle einsetzen wollen.

Die Abbildung der Ecore-konformen Modelle auf Javaklassen übernimmt ein EMF-eigener Code-Generator. Dieser stellt eine Parallelentwicklung zu der JMI-Spezifikation dar. JMI selbst kann nicht für diese Abbildung benutzt werden, da diese Spezifikation speziell das Mapping von zum Standard MOF 1.4 konformen Modellen auf Java regelt, EMF jedoch mit einem eigenen Meta-Metamodell aufwartet.

4.5.2 Meta Data Repository (MDR)

Das MDR ist die Zweite der untersuchten Softwarelösungen, die auch tatsächlich im Entwurfswerkzeug AMoK als Modellmanagement-Komponente zum Einsatz kommt. Beim MDR handelt es sich um ein Open Source Werkzeug der Firma Sun, das erstmals im Frühjahr 2002 vorgestellt wurde. Hauptsächlich eingesetzt wird dieses Metadaten Repository in der Netbeans-Plattform. Zwischen der Netbeans-Plattform und dem bereits vorgestellten Eclipse-Projekt existieren eine ganze Reihe von Gemeinsamkeiten. Beide Projekte zielen auf eine bessere Integration von Werkzeugen zur Softwareentwicklung ab und setzen dabei auf den MDA-Ansatz. Im Gegensatz zum EMF wird der MDA-Ansatz beim MDR jedoch stärker in den Vordergrund gestellt, da ausschließlich die offenen Standards MOF, JMI und XMI umgesetzt wurden.

Die Netbeans IDE (Integrated Development Environment) ist stark modular aufgebaut. Das MDR-Modul enthält die Modelle aller für die Softwareentwicklung zur Verfügung stehenden Sprachen wie z.B. XML, Java, EJB etc. und versorgt wiederum andere Module, wie einen Projektexplorer oder ein Editormodul mit den benötigten Metadaten. Die Verwendung von Metadaten und Meta-Metadaten in integrierten Entwicklungsumgebungen ist im Prinzip nicht neu und findet in vielen modernen IDEs statt. Das MDR bietet jedoch den Vorteil, daß die Metadaten persistent vorgehalten werden und somit auch kompliziertere Abfragen möglich sind, anstatt diese Metadaten bei jeder Anfrage aus den jeweiligen Dokumenten, z.B. Quellcodes, extrahieren zu müssen.

Für die Verwaltung und die Manipulation der Metadaten stellt die Netbeans IDE einen MDR-Explorer, siehe Abbildung 4.5, zur Verfügung. Dieser setzt auf dem MDR auf und bietet dem Anwender eine einfache Darstellung der im Repository abgelegten Metamodelle, sowie den Zugriff auf die Grundfunktionalitäten des MDR. Die Möglichkeit diesen MDR-Explorer auch für AMoK nutzbar zu machen, wurde zwar

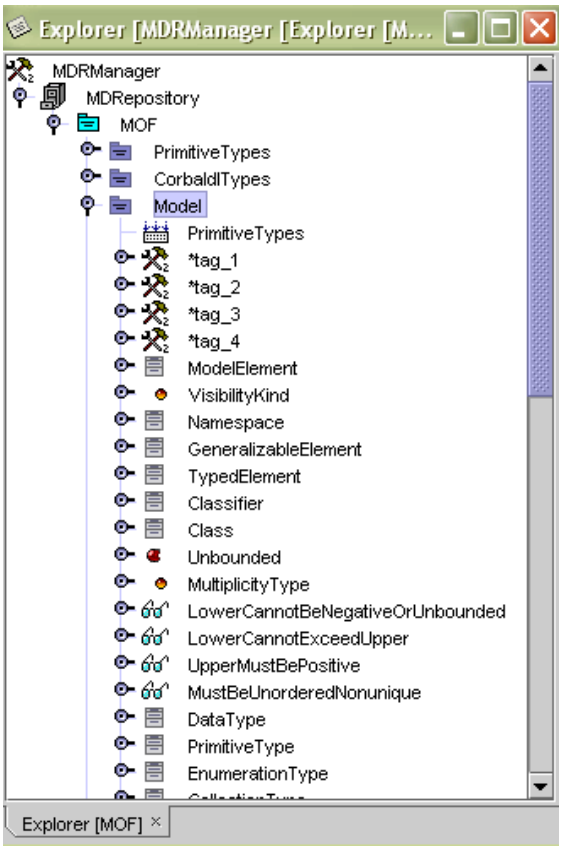


Abbildung 4.5: Screenshot - MDR-Explorer

in Erwägung gezogen, jedoch aufgrund der komplexen Verankerung des Explorers in das mit weit über 50.000 Klassen sehr umfangreiche Netbeans-Projekt nicht realisiert. Statt dessen wurde für AMoK mit recht geringem Aufwand eine ähnliche Komponente implementiert, die auf einer Standalone-Version des MDR basiert.

Die Standalone-Version des MDR wurde nachträglich von Sun zur Verfügung gestellt, um auch *Model Driven Development* außerhalb des Netbeans Projektes zu vereinfachen und wird bereits in kommerziellen Produkten wie „Poseidon for UML“ der „Gentleware AG“ oder „MetaBoss“ der Firma „Softaris“ eingesetzt.

Das MDR benutzt eine Implementation des MOF-Modells der Version 1.4, welches in Kapitel 3.3 ausführlich beschrieben ist. Alle für den Zugriff auf dieses Meta-Metamodell benötigten Interfaces wurden bereits generiert und über eine „MOF-API“ zur Verfügung gestellt. Darüber hinaus können auch zum älteren MOF-Modell Version 1.3 konforme Metamodelle mit dem MDR genutzt werden. Diese werden während ihres Importvorgangs in das MDR transparent in MOF 1.4 konforme Metamodelle transformiert.

Neben der MOF API stellt das MDR auch eine JMI API und eine MDR API bereit. Die JMI API enthält generelle reflektive JMI Interfaces, wie sie durch die JMI Spezifikation JSR 040 definiert wurden und wird durch die MDR API um für Repository spezifische, generische APIs erweitert. Diese APIs bieten z.B. Schnittstellen zu einem Eventhandler und zu einem Storagehandler an. Kern der MDR Implementation ist die sogenannte MDR Engine. Die MDR Engine basiert auf der MDR API und der JMI API, um ein MOF 1.4 - Repository als Standalone-Version einzurichten und zu initialisieren.

Für den Import und Export von Modellen und Metamodellen stehen die generischen Werkzeuge XMI-Reader, XMI-Writer, welche sowohl den Standard XMI 1.1 als auch XMI 1.2 unterstützen, zur Verfügung. Ebenfalls vorhanden ist ein JMI-Generator mit dem eine Abbildung aller Modellelemente auf entsprechende Javaklassen zum Zwecke der Anbindung externer Anwendungen auf die Metadaten realisiert wird. Sowohl XMI-Reader, XMI-Writer als auch der JMI-Generator sind nicht integraler Bestandteil der MDR-Engine, sondern gehören zur Netbeans-Plattform selbst, und stehen damit allen Modulen dieser offenen Entwicklungsumgebung zur Verfügung. Da sie allein auf den reflektiven JMI-Interfaces — der JMI API — arbeiten, lassen sie sich auch für andere JMI-konforme Metadaten Repositories, außer dem MDR, verwenden.

Einen Überblick über die Architektur der MDR-Engine bietet die Abbildung 4.6. Werkzeuge wie AMoK setzen oberhalb der MDR-Architektur auf und benutzen einen

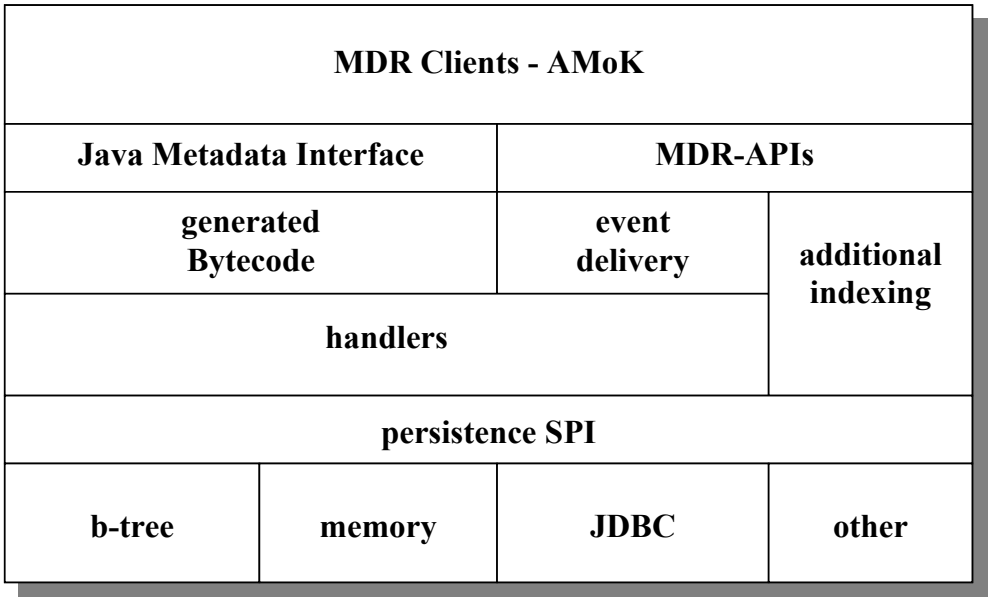


Abbildung 4.6: MDR-Architektur

MDR-Manager der MDR-API um ein neues Repository anzulegen, oder um den Zugang zu einem bereits vorhandenem Repository zu bekommen. Beim Importieren als auch beim Instanziieren von Metamodellen werden „Extents“, virtuelle Subrepositories, unterhalb dieses Repositories angelegt und JMI-Interfaces generiert. Ein Bytecode Generator erzeugt dynamisch zur Laufzeit eine Standardimplementation zu jedem vorhandenem JMI-Interface.

Um die Repräsentation der Metadaten mit dem Repository zu synchronisieren, kann jedes Extent auf etwaige Veränderungen überwacht werden. Tritt eine solche Veränderung auf, wird ein entsprechendes Ereignisobjekt (*Event*) erzeugt und von einem *Eventhandler* an alle Interessenten weitergeleitet. Der in der Architekturskizze aufgeführte zusätzliche Indexdienst wurde zur Steigerung der Performanz bei der Abfrage beliebiger Metadaten konzipiert. Um häufig abgefragte Attribute eines Metamodells in den zusätzlichen Index aufnehmen zu lassen, kann es, mittels eines MDR spezifischen *Index-Tags*, als indizierbar deklariert werden. Der zusätzliche Indexdienst wurde beim Entwurf von AMoK nicht weiter berücksichtigt, da die zu verwaltenden Metamodelle nicht vorweg bekannt sind, und somit eine Entscheidung über die Einteilung der Modellattribute nach bestimmten Prioritäten nicht getroffen werden kann.

Unterhalb der Ereignisbehandlung und der Bytecode-Erzeugung ist die Persistenze-

bene angesiedelt. Die Client-Ebene kann vom Aspekt der Datenpersistenz abstrahieren, indem Dienste der Persistenzebene in Anspruch genommen werden. Sämtliche Manipulationen an den Metadaten resultieren in Anfragen, welche wiederum von spezielle Storagehandlern bearbeitet werden. Die Storagehandler stellen dann die Persistenz der Metadaten und ihrer Änderungen sicher, indem sie entsprechende Methoden einer Storage-Implementation aufrufen und die Metadaten an diese übergeben. Derzeit verfügt die MDR-Engine über zwei verschiedene Speichermechanismen. Die Standardimplementation des MDR-Storage ist eine eigene Datenbank, dem „BTree“. Die BTree-Datenbank besteht im wesentlichen aus einer Data-Datei zur Aufnahme von serialisierten Objekten und aus einer Index-Datei. Sämtliche Metadaten werden serialisiert, mit einem eindeutigem Schlüssel versehen und in die Data-Datei geschrieben. Dieser eindeutige Schlüssel wird anschließend, zusammen mit dem Offset des serialisierten Objektes innerhalb der Data-Datei, in der Index-Datei abgelegt.

Neben dem BTree besteht auch die Möglichkeit, die Metadaten nur im Speicher vorhalten zu lassen. Derartige transiente Repositories sind besonders für Testzwecke interessant, können aber auch die Performanz des Systems erhöhen, wenn sie in Kombination mit dem BTree eingesetzt werden. Diese Kombination läßt sich erreichen indem Metadaten, die einen transienten Charakter aufweisen, mit einem *transient-Tag* versehen werden. Grundsätzlich ist es ebenfalls möglich noch weitere Storage-Implementationen in das MDR zu integrieren. So sehen die Netbeans-Entwickler auch eine JDBC-Anbindung des MDR an relationale Datenbanken für die nahe Zukunft vor.

4.6 Modellierungswerkzeuge

Wie Eingangs beschrieben soll der Framework-Entwickler in die Lage versetzt werden, das von AMoK verwendete BDI-Agentenmetamodell auf einfache Weise mit einem UML-Modellierungswerkzeug zu erstellen oder zu modifizieren. Die in [Jec] vorgenommene Gegenüberstellung von 100 solcher Werkzeuge kann vermuten lassen, daß die Auswahl in diesem Bereich sehr groß ist. Allerdings eignen sich nur Werkzeuge, die über die folgende Funktionalität verfügen:

Erstellen von Klassendiagrammen gemäß UML 1.3 oder 1.4

Aus folgender Überlegung sollte der Frameworkentwickler das in der AMoK Architektur verwendete Metamodell des BDI-Agentensystems in der UML der Version

1.3 bzw. 1.4 erstellen. Das Metamodell der UML 2.0 wurde mit dem *Diagram Interchange (DI)* um Informationen zur Präsentation der einzelnen Diagrammelemente, wie z.B. Größe und Position eines Klassenelementes, erweitert. Die ebenfalls am Fachbereich Informatik der Universität Hamburg entwickelte Technik des Diagram Interchange [FM03] wurde von der OMG in den Standard UML 2.0 aufgenommen und in deren XMI DTD integriert. Mit dem Diagram Interchange wird sichergestellt, daß bei der Übertragung von UML-Diagrammen zwischen zwei oder mehreren UML-Entwurfswerkzeugen kein Informationsverlust bezüglich ihrer Darstellung entsteht. Diese Informationen sind bei der Beschreibung der statischen Struktur eines MOF-Metamodells mittels UML-Klassendiagrammen jedoch nicht von Belang. Um Metamodelle, die mit Klassendiagrammen der UML 2.0 erstellt werden, in Metamodelle der MOF 1.4 zu transformieren, müßten vorher sämtliche Präsentationsinformationen entfernt werden.

Import und Export der Diagramme mittels XMI

Ein Viertel der in [Jec] verglichenen Werkzeuge bietet die Möglichkeit des Diagrammaustausches mittels XMI. Da die Modellmanagementkomponente von AMoK XMI 1.1 und XMI 1.2 lesen und schreiben kann, sollte das verwendete Modellierungswerkzeug ebenfalls mindestens eine der beiden Versionen unterstützen. Anmerkend sei erwähnt, daß sich diese Unterstützung in der Regel auf das Vorhandensein eines XMI DTD der UML beschränkt, und zu diesem DTD valide Dokumente eingelesen und geschrieben werden können. XMI selbst ermöglicht jedoch eine Abbildung beliebiger MOF-Metamodelle auf XML.

Unterstützung des „UML Profile for MOF“

Um die Transformation der UML-Modelle in MOF-Modelle durchführen zu können, soll der in Abschnitt 3.4 erörterte Standard „UML Profile for MOF“ unterstützt werden. Dazu müssen die benötigten *Stereotypes* und *Tagged Values* dem Modellierungswerkzeug bekannt gemacht werden. Templates, die diese Aufgabe übernehmen, stehen derzeit nur für „Poseidon for UML“ ab Version 1.5 und für „MagicDraw“ ab Version 6.0 zur Verfügung.

Im Rahmen dieser Arbeit wurde das Werkzeug „Poseidon for UML 1.6“ der Gentleware AG eingesetzt, da aktuellere Poseidon-Versionen ausschließlich auf UML 2.0 basieren.

4.7 Entwurf des BDI-Agentenmetamodells

Die Grundlage des Entwurfswerkzeuges AMoK ist das BDI-Agentenmetamodell. Die Herangehensweise beim Entwurf eines solchen Metamodells soll nun skizziert werden. Die theoretischen Möglichkeiten und Einschränkungen beim Modellieren eines MOF-konformen Metamodells mit der Modellierungssprache UML wurden in Kapitel 3 bereits angesprochen. Bei der praktischen Umsetzung müssen folgende Regelungen beachtet werden:

Dekomposition binärer Assoziationen

Da in MOF nur binäre Assoziationen zulässig sind, müssen n-äre Assoziationen durch Dekomposition in binäre Assoziationen überführt werden. Dieser Vorgang wird in Abbildung 4.7 dargestellt.

Auszeichnung sämtlicher Modellelemente

Während es den Entwicklern von UML-Diagrammen freigestellt bleibt, ob und wie sie ihre Modellelemente benennen, ist eine eindeutige Bezeichnung dieser Elemente beim Entwurf eines MOF-konformen Metamodells, wie es in dieser Arbeit benötigt wird, unabdingbar. Diese Tatsache begründet sich darauf, daß für jedes Element ein gültiges, kompilierbares Interface generiert werden muß. Bei den in dieser Arbeit abgebildeten Metamodellen wurde lediglich aus Gründen der Übersichtlichkeit auf die Bezeichung von Assoziationen und deren Enden verzichtet. Weiterhin gilt zu beachten, daß reservierte Schlüsselworte der Zielsprache, in diesem Fall Java, nicht verwendet werden dürfen. So würde beispielsweise ein Modellelement namens „class“ in das ungültige Interface:

```
public interface classClass extends javax.jmi.reflect.RefClass {
     public class createClass();
}
```

übersetzt.

Vollständigkeit des Metamodells

Damit eine surjektive Abbildung des Metamodells auf das Code-Modell der jeweiligen, zu unterstützenden Zielplattform möglich ist, muß die statische Struktur des Agenten vollständig beschrieben werden. Die Modellierung zusätzlicher Elemente kann jedoch durchaus sinnvoll sein, wenn sie sich dafür eignen, die Modelle besser zu strukturieren, und damit deren Lesbarkeit erhöhen.

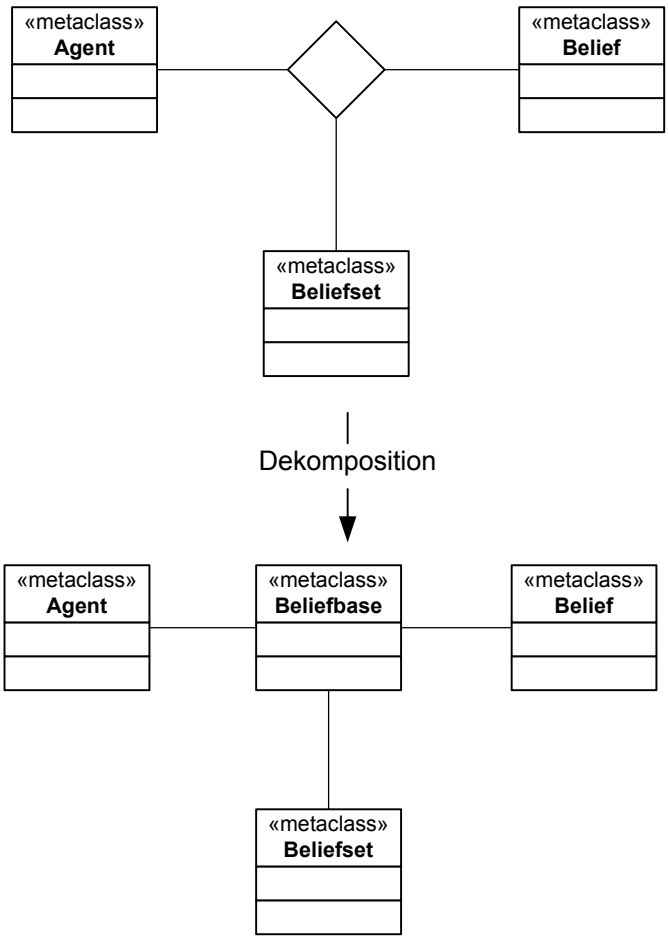


Abbildung 4.7: Dekomposition n-ärer Assoziationen

Ein Jadex-spezifischer BDI-Agent besitzt die Attribute *name*, *description* und *type* (siehe Abbildung 4.8), um sich von weiteren Agenten zu unterscheiden. Weitere Attribute wie *class*, *package* und *propertyfile* können optional angegeben werden. Das Attribut *package* gibt vor, wo die entsprechende Java-Klasse, die den Agenten implementiert, angelegt wurde. Diese Klasse wird mit dem Attribut *class* angegeben. Das Attribut *class* wird mit dem Wert „`jadex.runtime.JadeWrapperAgent`“ vorbelegt, der auf eine bereits vorhandene Agentenimplementierung verweist. Zur Angabe von speziellen Laufzeiteigenschaften des Agenten kann durch die Belegung des Attributes *propertyfile* außerdem eine externe Datei genutzt werden.

Der Agent selbst wird als Root-Element modelliert. So kann dem Metamodell später entnommen werden, daß es sich bei dem Element „Agent“ um das Element handelt, welches als erstes instanziiert werden muß. Weitere Komponenten eines BDI-Agenten werden als „composite“-Aggregation modelliert, wobei deren Deklaration auf M1-Ebene optional ist. Zu diesen Komponenten gehören im einzelnen:

Imports Sämtliche Java-Klassen, die der Agent verwendet, werden mit dem Attribut *importstatement* der Klasse „Import“ definiert.

Capabilities Wird das durch die Jadex-Architektur bereitgestellte Konzept der „Capability“ verwendet, können diese — ebenfalls mittels ADF spezifizierten — Capabilities in der gleichnamigen Komponente „Capabilities“ als Referenz auf ein solches ADF angegeben werden. Zu diesem Zweck stellt die „Capability-Reference“ das Attribute *file* bereit. Die Tatsache, daß ein Agent durch eine beliebige Anzahl von Capabilities modularisiert werden kann, findet sich in diesem Metamodellelement wieder.

Membranebase Die Modellierung der „Membranebase“ ermöglicht eine Abgrenzung der Capabilities innerhalb eines Agenten oder einer weiteren Capability. Es kann deklariert werden, welche Ereignisse (Event-Elemente) und welche Ziele (Goal-Elemente) die Membran passieren dürfen.

Beliefbase Das Metamodellelement „Beliefbase“ aggregiert alle dem Agenten bekannten Beliefs. Ein „Belief“ selbst kann sowohl einzeln, als auch als Menge von Beliefs, dem „Beliefset“, modelliert werden. Die Metamodellelemente „Belief“ und „Beliefset“ besitzen zusätzliche Attribute wie *class*, *updaterate*, *propagate*, *ref* und *type*, um das Objekt, welches das Belief repräsentiert, genauer zu charakterisieren. Neben diesen Attributen enthält ein Belief auch ein sogenanntes „Fact“ bzw. „Facts“ im Fall von Beliefsets, dessen Struktur durch das Metamodellelement „Expression“ beschrieben werden kann. Diese

„Expression“-Elemente werden dann auf M1-Ebene die Informationen zur Initialisierung von Belief-Objekten kapseln.

Goalbase Alle vom Agenten anzustrebenden Ziele werden durch das Aggregat „Goalbase“ repräsentiert. Das „Goalbase“-Element setzt sich dabei zusammen aus beliebig vielen Zielen der Kategorien „perform“, „achieve“, „query“, „maintain“ und „initial“, deren Struktur wiederum mit zugehörigen Metamodellelementen deklariert wird. Obwohl sich die Semantik der einzelnen Zielarten stark voneinander unterscheidet, zeichnen sie sich durch ein hohes Maß an Strukturähnlichkeit aus. Daher können die strukturellen Gemeinsamkeiten durch eine Superklasse „GoalType“ repräsentiert werden, die der Generalisierung der spezifischeren Ziele dient (siehe Abbildung 4.10).

Planbase Sämtliche dem Agenten zur Verfügung stehenden Pläne lassen sich durch den Container „Planbase“ zusammenfassen. Die Eigenschaften *instant*, *priority* und *type* sind auf M2-Ebene als Attribute des Elements „Plan“ modelliert und können auf Ebene M1 mit Informationen zur Abarbeitung der Pläne durch den Agenten versehen werden. Dabei spezifiziert die als primitiver Datentyp „Boolean“ modellierte Eigenschaft *instant*, ob ein Plan einmalig beim Startvorgang des Agenten zur Ausführung gelangt, oder erst später instanziiert wird. Um einem Plan Vorrang vor einem anderen Plan zu geben, kann man ihn durch die als „Integer“-Datentyp modellierte Eigenschaft *priority* mit einer entsprechenden Priorität versehen. Mit dem Attribute *type* wird auf Ebene M1 definiert, welche technischen Details — z.B. die Zuweisung eines eigenen Threads — bei der Abarbeitung dieses Planes relevant sind. Das Metamodellelement Plan besteht seinerseits wiederum aus weiteren Elementen wie „Constructor“ zur Strukturbeschreibung des Konstruktors der entsprechenden Planimplementation oder Elementen zur Deklaration von Filtern und Bedingungen, die für diesen Plan relevant sind. Diese Elemente sind im einzelnen: „Constructor“, „Filter“, „Waitqueuefilter“, „Triggercondition“, „Precondition“ und „Contextcondition“. Das Metamodellelement Plan ist in Abbildung 4.9 dargestellt.

Eventbase Die Beschreibungen der vom Agenten zu verarbeitenden Ereignisse werden durch das Element „Event“ vorgenommen. Dieses Metamodellelement besteht lediglich aus einer Klasse mit den Attributen *name* und *description*. Der Besitzer aller „Event“-Elemente ist das Metamodellelement „Eventbase“.

Expressionbase Das Metamodellelement „Expression“, das der Beschreibung von Ausdrücken innerhalb des Jadex-Frameworkes dient, wird meistens an entsprechender Stelle als Bestandteil anderer Modellelemente verwendet, kann aber

ebenso zum Zwecke der Wiederverwendbarkeit im Metamodellelement „Expressionbase“ gesammelt werden. Dies ist besonders sinnvoll, wenn diese Expressions Ausdrücke der Jadex-Abfragesprache OQL (Object Query Language) beschreiben, beispielsweise zum Beziehen von Beliefs. Solche Ausdrücke können dann einfach referenziert werden, da sie in einem gemeinsamen Container, der „Expressionbase“, vorgehalten werden.

Propertybase Analog zu der Ablage von Laufzeiteigenschaften des Agenten in einer Property-Datei, welche dem Agenten bekannt gemacht werden kann, lassen sich solche Eigenschaften, zu denen z.B. das Verwenden des Loggermodules gehört, auch direkt in den „Property“-Metamodellelementen deklarieren, die wiederum in dem Metamodellelement „Propertybase“ zusammengefasst werden.

Languagebase Die vom Agenten generierten und empfangenen Nachrichten werden durch Sprach-Klassen des Agentenframeworks kodiert, deren Instanziierungsvorgang durch die zum Metamodellelement „Languagebase“ gehörenden „Expression“-Elemente detaillierter beschrieben werden kann.

Ontologybase In Analogie zur „Languagebase“ aggregiert das Metamodellelement „Ontologybase“ ebenfalls Elemente des komplexen Typs „Expression“, deren Instanzen Informationen zur Instanziierung JADE-spezifischer Ontologieklassen bereitstellen.

ServiceDescriptionbase Metamodellelemente zur Spezifikation der Syntax von Dienstbeschreibungen werden zum Element „ServiceDescriptionbase“ zusammengefaßt. Diese Dienstbeschreibungen deklarieren die Funktionalität, die ein Agent seiner Umgebung, insbesondere anderen Agenten, zur Verfügung stellt.

Agentdescriptions Das Metamodellelement „Agentdescriptions“ setzt sich aus einer beliebigen Anzahl von Metamodellelementen des Typs „Agentdescription“ zusammen. Der Nutzen einer „Agentdiscription“ ist die formale Beschreibung eines Agenten, wie sie von anderen Agenten zum Zwecke der Kooperation abgefragt werden kann. Die Metaklasse „Agentdescription“ besteht aus Mengen von „Language“-, „Ontology“-, „Protocol“- und „Service“-Elementen. Somit lassen sich auf der Ebene M1 Informationen zur Beschreibung der verwendeten Sprache, der eingesetzten Ontologie, der benutzten Protokolle und der angebotenen Dienste modellieren.

Parameters Der letzte Bestandteil eines Jadex-konformen BDI-Agentenmetamodells ist die Metaklasse „Parameters“. Für die Parame-

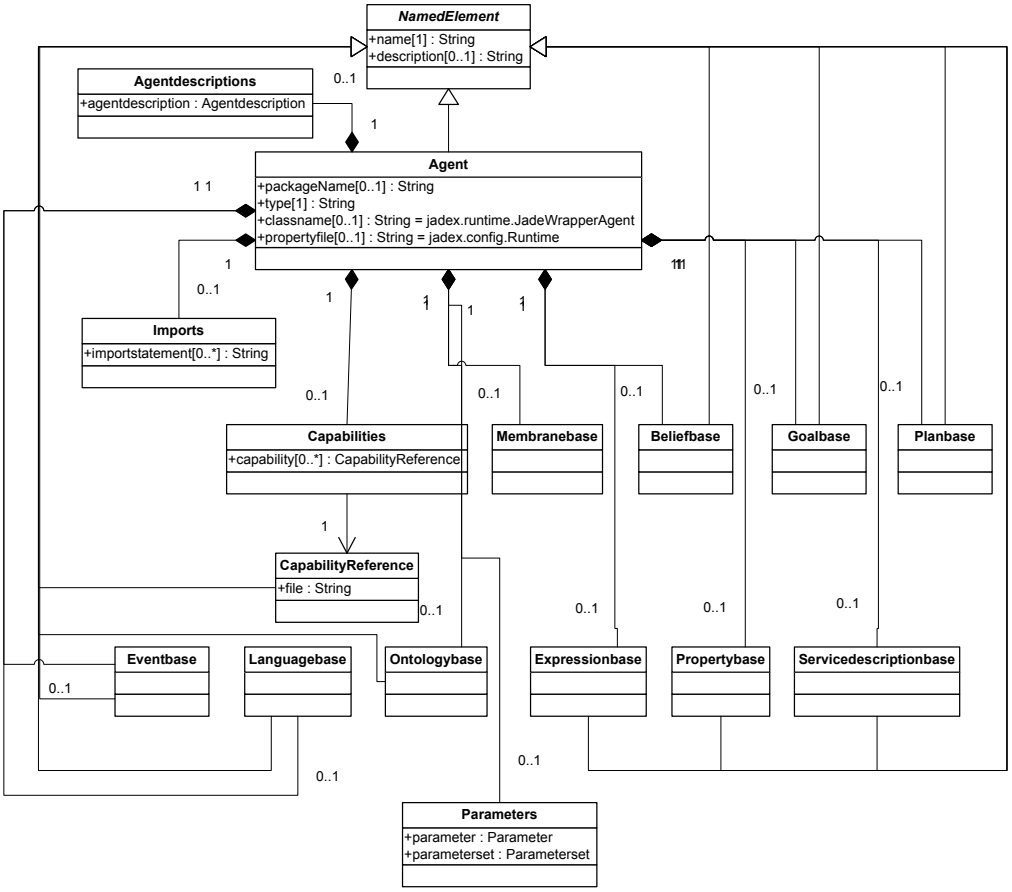


Abbildung 4.8: Jadex BDI-Agent

trisierung eines Agenten stehen sowohl „Parameter“ als auch „Parametersets“ zur Verfügung. Beide Metamodellelemente bestehen aus den Metamodellelementen „Constraint“ und „Value“, bzw. einer Menge von „Value“-Elementen als Bestandteil des „Parameterset“-Elementes, und werden als Typ des Attributes *parameter* bzw. *parameterset* des Metamodellelementes „Parameters“ referenziert.

Alle Modellelemente, die sich durch einen Namen und eine Beschreibung auszeichnen, werden als Subklassen der abstrakten Klasse „NamedElement“ modelliert.

Das vollständige UML-Klassendiagramm dieses Metamodells wird in „Poseidon for UML“ erstellt. Dazu wird im ersten Schritt das Template für das „UML Profile for MOF“ geladen und somit der Namensraum „MOF-Metamodel“ verfügbar. In diesem

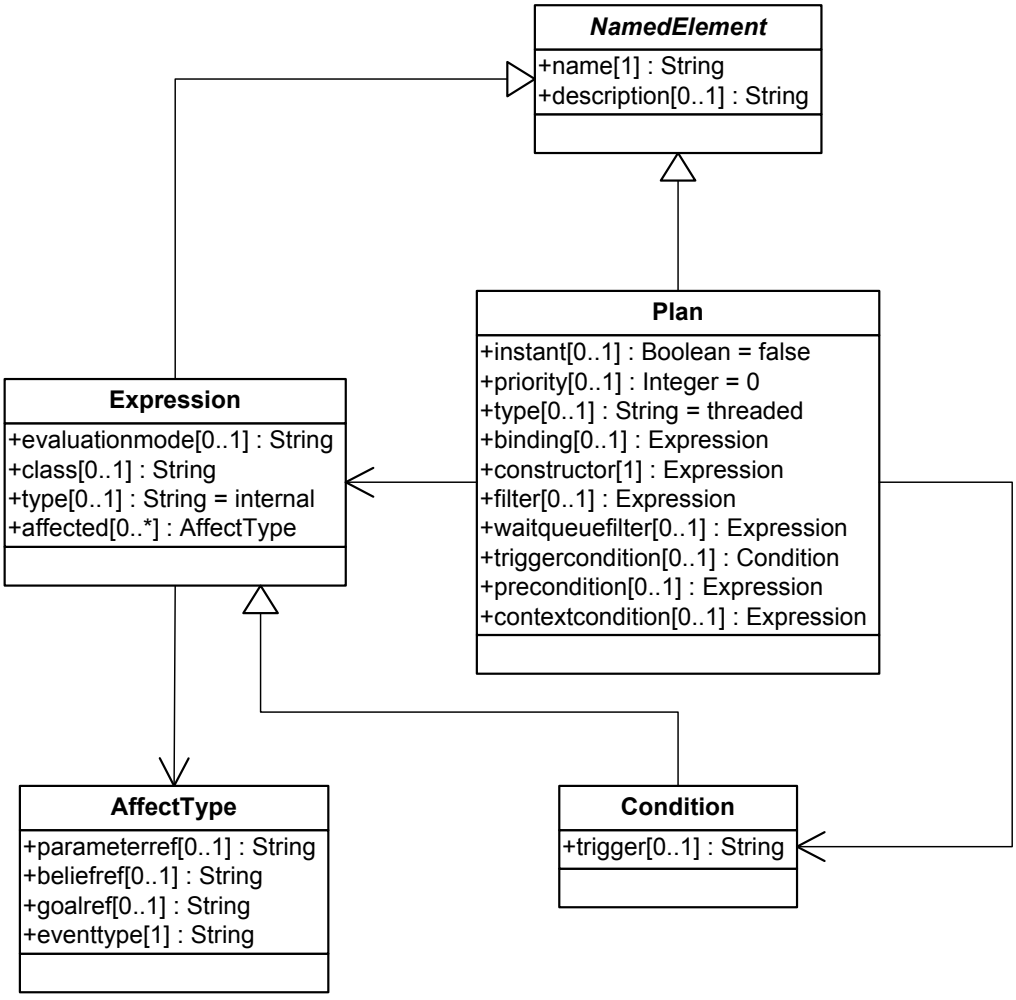


Abbildung 4.9: Metamodellelement Plan

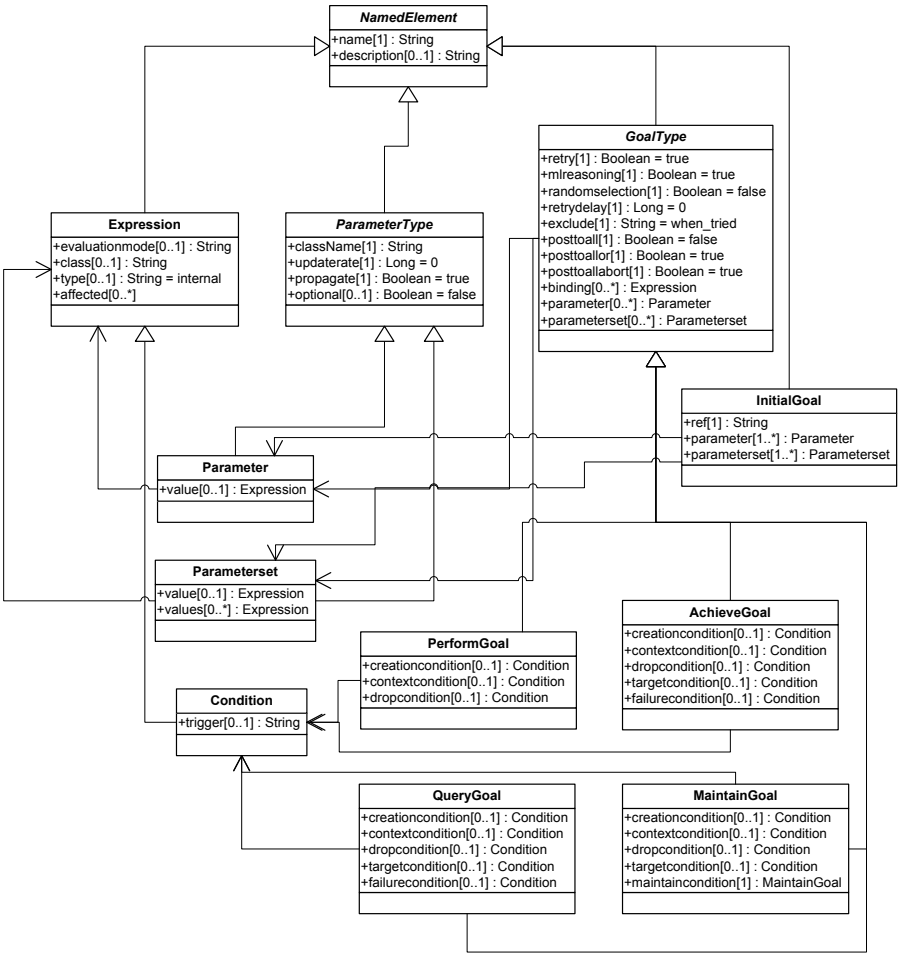


Abbildung 4.10: Metamodellelement GoalType

Namensraum wird ein Package „bdiagent“ erstellt, innerhalb dessen sämtliche Metamodellelemente erstellt werden. Das Metamodell wird abschließend abgespeichert und steht damit dem Werkzeug AMoK zur Verfügung.

5 Implementation und Einsatz des Entwurfswerkzeuges

In diesem Kapitel wird dargelegt, wie die in Kapitel 4 vorgestellte AMoK-Architektur in Form einer prototypischen Implementierung umgesetzt wurde. Dabei wird ebenfalls auf darauf eingegangen, welchen Zweck die einzelnen Komponenten erfüllen und wie sie eingesetzt werden. In Anhang B ist abgebildet, wie alle realisierten Komponenten unter einer gemeinsamen Benutzeroberfläche des AMoK-Werkzeug integriert wurden.

5.1 Java Metadata Interface (JMI)

Die „Java Metadata Interface“ - Spezifikation nennt sich „JSR 040“ [UC02] und wurde im Rahmen des „Java Community Process“ erstmals im Juni 2002 in der Version 1.0 verabschiedet. Ziel dieser Spezifikation ist es, eine genormte Abbildung von der MOF auf Java-Interfaces zu ermöglichen. Anmerkend sei allerdings erwähnt, daß derartige Abbildungen auf einer abstrakteren Ebene in der MOF-Spezifikation selbst bereits vorgesehen und fest verankert sind. Der in Abschnitt 3.5.2 vorgestellte XML Metadata Interchange ist ein treffendes Beispiel für eine solche technologieneutrale Abbildung.

Um die Erzeugung, die Speicherung, den Zugriff und den Austausch von Metadaten mittels Java-Interfaces bereitzustellen, baut die JMI-Spezifikation direkt auf dem MOF-Standard bis Version 1.4 auf und bietet ein reflektives Programmiermodell für die Java-Plattform an. Dieses Programmiermodell wird sowohl von dem Metadaten Repository (MDR), als auch von den einzelnen Komponenten der AMoK-Architektur selbst genutzt.

JMI unterscheidet vier Arten von Metaobjekten, für die definiert wurde, nach welchem Muster sie durch ein Java-Interface repräsentiert werden können. Diese vier

Arten sind die „instance“-, die „class proxy“-, die „association“- und die „package“-Objekte und werden anhand des Metamodells eines Plans, wie in Abbildung 4.9 dargestellt, erläutert.

Jedes Metamodell ist innerhalb eines *Packages* angesiedelt. Dieses *Package* wird auf genau ein Interface abgebildet, welches vom Typ `javax.jmi.reflect.RefPackage` ist und den Zugriff auf alle in diesem Metamodell enthaltenen Klassen und Assoziationen durch Rückgabe eines „class proxy“-Objektes ermöglicht. Dazu wird für jede Klasse bzw. Assoziation ein Interface für ein Stellvertreterobjekt, den „class proxy“ bzw. „association proxy“ generiert. Anschließend wird im Package-Interface zu diesen „proxy“-Objekten eine zugehörige `get()`-Methode angelegt. Daraus ergibt sich für den Plan das folgende Interface:

```
package bdiagent;
public interface BdiagentPackage extends javax.jmi.reflect.RefPackage {
  public bdiagent.NamedElementClass getNamedElement();
    public bdiagent.PlanClass getPlan();
    public bdiagent.AffectTypeClass getAffectType();
    public bdiagent.ConditionClass getCondition();
    public bdiagent.ExpressionClass getExpression();

    public bdiagent.AssociationPlan2Expression
     getAssociationPlan2Expression();
    public bdiagent.AssociationPlan2Condition
     getAssociationPlan2Condition();
    public bdiagent.AssociationExpression2AffectedType
     getAssociationExpression2AffectedType();
}
```

Ein „class proxy“-Interface repräsentiert eine Klasse mit zwei Konstruktoren zum Erstellen eines „instance“-Objektes und ist dabei vom Typ `javax.jmi.reflect.RefClass`. Mit dem ersten Konstruktor kann man ein „instance“-Objekt erzeugen und anschließend initialisieren. Alternativ kann man die zur Initialisierung benötigten Parameter auch dem zweiten Konstruktor übergeben. Das Interface für den „class proxy“ für das Element „NamedElement“ besitzt natürlich keine Konstruktoren, da es als abstrakte Klasse definiert wurde. Das „class proxy“-Interface für die Klasse Plan sieht folgendermaßen aus:

```
package bdiagent;
```

```
public interface PlanClass extends javax.jmi.reflect.RefClass {
    public Plan createPlan();
    public Plan createPlan(java.lang.String name,
      java.lang.String description,
      boolean instant,
      java.lang.Integer priority,
      java.lang.String type,
      bdiagent.Expression binding,
      bdiagent.Expression constructor ...
}
```

Für das mit dem obigen Interface erzeugbare „instance“-Objekt für einen Plan wird ebenfalls ein eigenes Interface generiert. Mit diesem Interface ist es dann möglich, auf die Attribute des „instance“-Objektes sowohl lesend (`get()`-Methode) als auch schreibend (`set()`-Methode) zuzugreifen. Eine Planinstanz kann somit anhand des folgenden, verkürzt dargestellten Interfaces modelliert werden.

```
package bdiagent;
public interface Plan extends bdiagent.NamedElement {
    public boolean isInstant();
    public void setInstant(boolean newValue);
    public java.lang.Integer getPriority();
    public void setPriority(java.lang.Integer newValue);
    public java.lang.String getType();
    public void setType(java.lang.String newValue);
...
}
```

Zur Behandlung der Abhängigkeiten zwischen Modellelementen wird für jede Assoziation ein „association“-Interface erstellt. Diese Interfaces verfügen über Methoden zum Erzeugen und Löschen von Assoziationen sowie zur Prüfung auf Existenz einer Assoziation. Ebenfalls wird der Zugriff auf beide an einer Assoziation beteiligten Instanzen ermöglicht. Ein „association“-Interface für die Assoziation zwischen einem Plan und einer Expression stellt sich somit folgendermaßen dar.

```
package bdiagent;
public interface AssociationPlan2Expression extends javax.jmi.reflect.RefAssociation {
    public boolean exists(bdiagent.Plan planEnd, bdiagent.Expression expressionEnd);
    public bdiagent.Plan getPlanEnd(bdiagent.Expression expressionEnd);
```

```
    public bdiagent.Expression getExpressionEnd(bdiagent.Plan planEnd);
    public boolean add(bdiagent.Plan planEnd, bdiagent.Expression expressionEnd);
    public boolean remove(bdiagent.Plan planEnd, bdiagent.Expression expressionEnd);
}
```

Alle bisher dargestellten Interfaces werden von dem in AMoK integrierten Netbeans-JMIMapper generiert und anschließend kompiliert. In ihrer Gesamtheit bilden sie eine Java-API zur Verwaltung des BDI-Agentenmetamodells und werden innerhalb des AMoK-Werkzeuges von dem ADF-Exporter verwendet. Für alle anderen generisch arbeitenden Komponenten wird das reflektive Programmiermodell verwendet, das in Abbildung 5.1 dargestellt ist. Dieses Modell ermöglicht es, Algorithmen zu entwickeln, die in der Lage sind, auf beliebigen Metamodellen zu operieren, ohne daß dazu eine explizite Kenntnis der Struktur und Semantik dieser Metamodelle vorhanden sein muß. In den Code-Beispielen der folgenden Abschnitte wird der Einsatz reflektiver Interfaces skizziert.

5.2 Initialisieren des Repository's

Beim ersten Start des Werkzeuges AMoK wird die BTree-Datenbank erstellt, die Repository-Struktur angelegt und mit dem MOF-Modell initialisiert. Es gibt zu jedem Zeitpunkt nur ein Repository im System, für dessen Verwaltung der als Singleton implementierte MDR-Manager verantwortlich ist.

```
private static MDRepository rep;
try {
    rep = MDRManager.getDefault().getDefaultRepository();
}
catch (Exception repError) {...}
```

Bei jeder weiteren Benutzung von AMoK wird dieses Repository verwendet. Alternativ kann AMoK auch der Name einer anderen, schon bestehenden oder zu erstellenden Datenbank beim Start als Parameter übergeben werden. Nachdem das Repository initialisiert ist, muß das Metamodell für die interne Struktur der BDI-Agenten geladen werden.

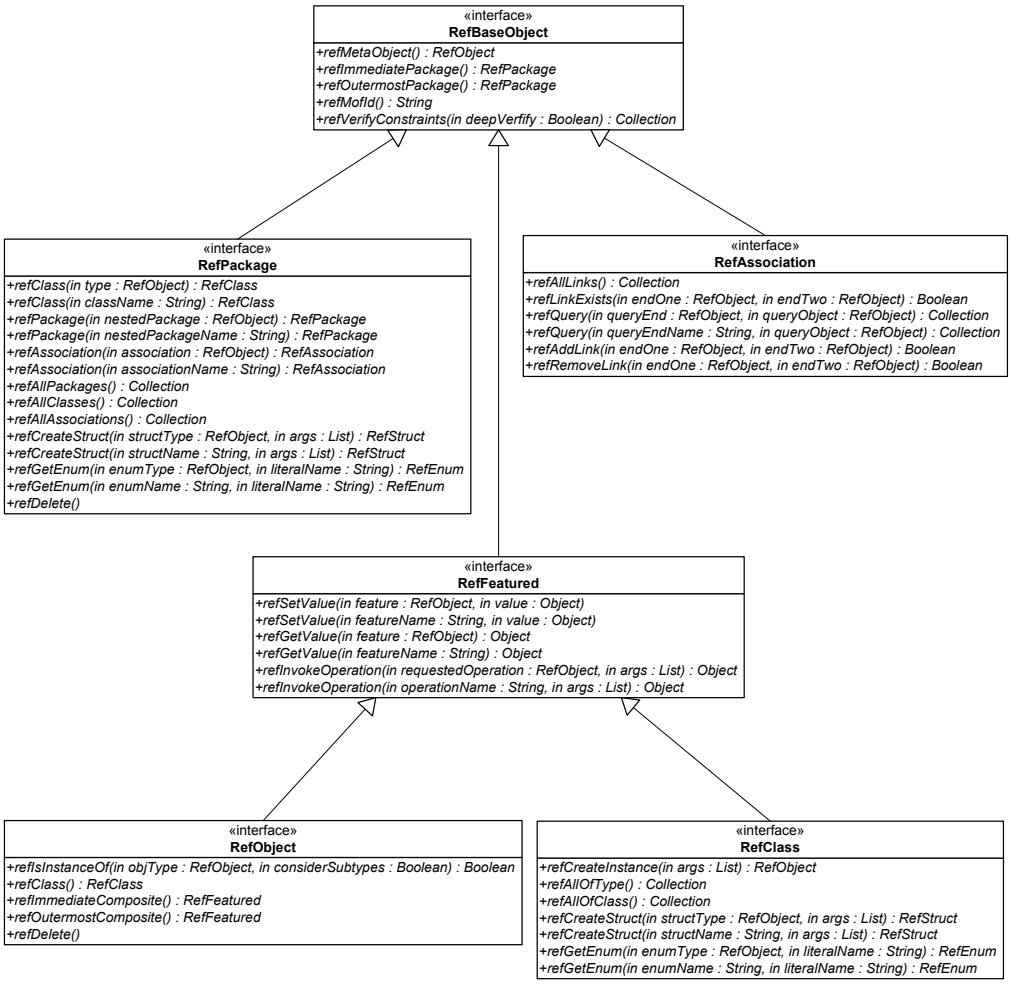


Abbildung 5.1: Reflektion-Mechanismus der JMI

5.3 Import von Metamodellen

In Abschnitt 4.7 wurde dargelegt, wie das Metamodell für BDI-Agenten mit dem Modellierungswerkzeug „Poseidon for UML“ erstellt werden kann. Poseidon legt das Klassendiagramm in Form einer XMI-Datei zusammen mit einer PGML-Datei und einer ARGO-Datei in einem ZIP-komprimierten Archiv — der ZARGO-Projektdatei — ab. Die ARGO-Datei ist ein XML-Dokument und enthält die für Poseidon relevanten Metainformationen zu dem gespeicherten Projekt. Alle Informationen, die die grafische Anordnung der Diagrammelemente beschreiben, sind in der PGML-Datei, ebenfalls ein XML-Dokument, kodiert.

Der Anwender von AMoK hat zwei Möglichkeiten, das Metamodell über ein Menü der grafischen Benutzerschnittstelle zu importieren. Zum einen kann ein bereits MOF-konformes Metamodell direkt geladen werden. Zum anderen besteht auch die Möglichkeit, falls dieses in Form eines UML-Klassendiagrammes vorliegt, entweder die entsprechende XMI-Datei oder aber eine ZARGO-Projektdatei auszuwählen. Im letzteren Fall wird das UML-Klassendiagramm dem im folgenden beschriebenen UML zu MOF Transformator zugeführt.

Zum Zwecke der Transformation von UML-Modellen in MOF-Modelle haben die Entwickler der Netbeans-IDE ein Kommandozeilen-basiertes Werkzeug namens „UML2MOF“ implementiert. Die darin enthaltene Klasse `org.netbeans.lib.jmi.uml2mof.Transformer` wird von dem in AMoK integrierten Transformator benutzt.

5.3.1 Der Transformationsprozeß

Der Transformationsprozeß verläuft in drei Schritten. Als erstes wird im Repository ein *Extent* (`MOFInstance`) zur Aufnahme einer Instanz des MOF-Modells erstellt. Extents sind in der MOF-Spezifikation als logische Domänen für Modellinstanzen definiert. Sie besitzen die Semantik eines Containers bzw. Packages. Auf alle in einem Extent enthaltenen Modellelemente wie Klassen, Assoziationen u.s.w. kann nur innerhalb dieses Extents zugegriffen werden. Das Konzept der Modularisierung komplexer Metamodelle durch „Nesting“, „Clustering“, „Generalization“ und „Import“ von Packages wird durch die verwendete MOF-Implementation MDR direkt auf Extents abgebildet.

Weiterhin wird ein Extent (`UMLInstance`) zur Aufnahme einer Instanz des UML-Metamodells erstellt. Das eigentliche UML 1.4 Metamodell kann jedoch nicht di-

rekt in das Repository importiert werden, da UML-Konstrukte wie „Class“ auch in der Sprache Java enthalten sind und eine Abbildung solcher Konstrukte auf diese Zielsprache somit zu fehlerhaften Java-Interfaces führen würde (siehe auch Abschnitt 4.7). Zur Lösung dieses Problems muß eine von Sun bereitgestellte XMI-Differenzdatei importiert werden, die das bestehende UML-Metamodell unter anderem um die Tags: „UmlClass“, „UmlPackage“, „UmlAssociation“ und „UmlException“ erweitert. Das UML Metamodell der Version 1.4 kann als XMI-Dokument unter der Internetaddresse „http://www.omg.org/cgi-bin/doc?ad/01-02-15“ bezogen werden.

Anschließend wird das Klassendiagramm des BDI-Agentenmetamodells in das Extent `UMLInstance` geladen. Das Einlesen der Metamodelle geschieht unter Zuhilfenahme des generischen Werkzeugs XmiReader.

Im letzten Schritt werden beide Extents der Methode `execute()` der Klasse `org.netbeans.lib.jmi.uml2mof.Transformer` übergeben und der Transformationsprozeß initiiert. Gemäß den Regelungen der „UML Profile for MOF“-Spezifikation wird nun sequentiell für jedes UML-Element ein entsprechendes MOF-Element erzeugt und im Extent `MOFInstance` abgelegt. Das transformierte, nun MOF-konforme Metamodell wird außerdem mit dem Präfix „MOF“ versehen und mittels XmiWriter im Dateisystem für eine mögliche spätere Verwendung abgelegt.

5.4 Erstellen von Agentenmodellen

Nachdem das Metamodell für die BDI-Agenten in das AMoK-Werkzeug eingepflegt wurde, steht es für das Erstellen von BDI-Agenten zur Verfügung. Der Anwender hat dabei zwei Sichten auf die im Repository enthaltenen Metadaten. Die erste Sicht ist die „Default-View“. Dargestellt durch einen Baum bietet sie einen Überblick über alle im Repository vorhandenen Agentenmodelle. Innerhalb der Knoten für die Agentenmodelle werden bereits modellierte Elemente wie z.B. Beliefbase, Planbase etc. aufgelistet. Der Anwender kann diese Elemente auswählen und weiter bearbeiten. Die Erstellung eines Agentenmodells beginnt in dieser Sicht zwangsläufig mit dem im Metamodell deklarierten „Root“-Element, dem „Agent“.

Die zweite Sicht, die „Expert-View“, ist für Benutzer konzipiert, die bereits über Erfahrungen mit der Modellierung in AMoK verfügen. Diese Sicht schränkt den Anwender nicht künstlich ein und gibt einen vollständigen Überblick über sämtliche im Repository vorhandenen Modelle und Metamodelle sowie über deren vollständige Struktur. Importierte Metamodelle werden in einem dafür vorgesehenen Knoten

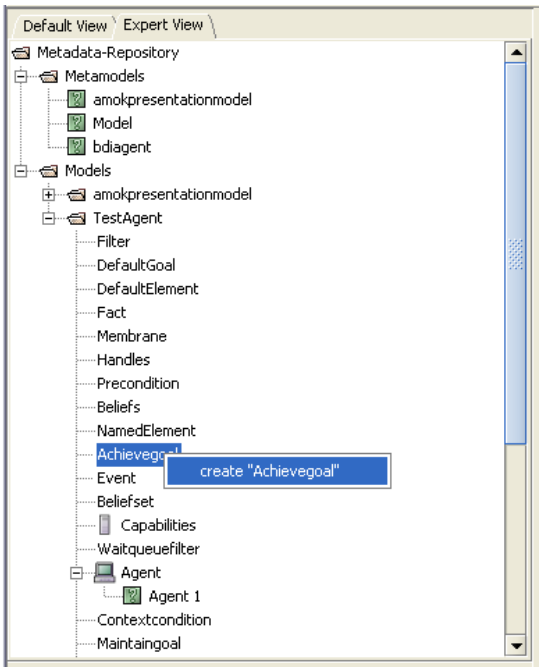


Abbildung 5.2: Screenshot - Expert View

„Metamodels“ aufgelistet und stellen in einem Kontextmenü erweiterte Funktionen, wie das Generieren von statischen JMI-Interfaces zur Anbindung externer Anwendungen auf das Repository, zur Verfügung. Die Expert-View ist in Abbildung 5.2 dargestellt.

Beide Sichten wurden gemäß dem Entwurfsmuster „Model-View-Controller“ implementiert und verwenden ausschließlich die reflektiven JMI-Interfaces zum Generieren der Baummodelle.

5.5 Der Formulargenerator

Der Formulargenerator ist ebenfalls eine rein generisch arbeitende Komponente, die dem Modellierer das Ausgestalten der Modellelemente ermöglicht. Mit Hilfe der erzeugten Formulare können die Eigenschaften des Agenten und seiner Bestandteile spezifiziert werden. Außerdem werden die Beziehungen zwischen den Modellelementen, wie Aggregation, Vererbung und Typreferenz, bei der Erzeugung der Formulare berücksichtigt.

Die primitiven Datentypen „Integer“, „String“ und „Boolean“ werden auf entsprechende visuelle Komponenten, wie `JTextField` bzw. `JCheckBox` der Java-Swing-Bibliothek abgebildet. Für die Repräsentation komplexer Datentypen wurde die Containerkomponente `JList` verwendet. Das Erstellen von Elementen, welche mit dem aktuell editierten Element assoziiert sind, erfolgt über die Schaltfläche `JButton`.

Die Verwendung der JMI-API zur Abbildung primitiver Datentypen auf ihre Formularelemente wird im folgenden, vereinfachten Codebeispiel erläutert:

```
RefObject instance;
RefClass rClass = instance.refClass();
MofClass metaObject = (MofClass) rClass.refMetaObject();

for (Iterator it = metaObject.getContents().iterator(); it.hasNext(); ) {
      ModelElement element = (ModelElement) it.next();
      if (element instanceof javax.jmi.model.Attribute) {
         if ( ( ( (javax.jmi.model.Attribute) element).getType() instanceof
                 javax.jmi.model.PrimitiveType)) {
               type = ((javax.jmi.model.Attribute) element).getType().getName();
               addVisuellComponent ( rClass.refOutermostPackage().refClass(type) );
          }
      }
  }
```

Zu einer beliebigen Objektinstanz `instance` wird die entsprechende Klasse `rClass` abgefragt. Danach wird das Meta-Objekt `metaObject` zu dieser Klasse betrachtet. Das Meta-Objekt einer Klasse ist immer vom Typ `MofClass` deren Attributtyp anschließend untersucht wird. Handelt es sich bei dem Attributtyp um einen bekannten primitiven Datentyp, wird die entsprechende visuelle Komponente dem Formular hinzugefügt. Analog zu diesem Vorgang werden komplexere Strukturen und Assoziationen anhand des Metamodells ermittelt und ausgewertet.

Die Abbildung 5.3 zeigt ein Formular, das die Modellkomponente „Plan“ repräsentiert.

5.6 Der Diagrammeditor

Die Darstellung der Agentenmodelle übernimmt der Diagrammgenerator. Die einzelnen Diagrammelemente können vom Anwender beliebig arrangiert und beschrif-

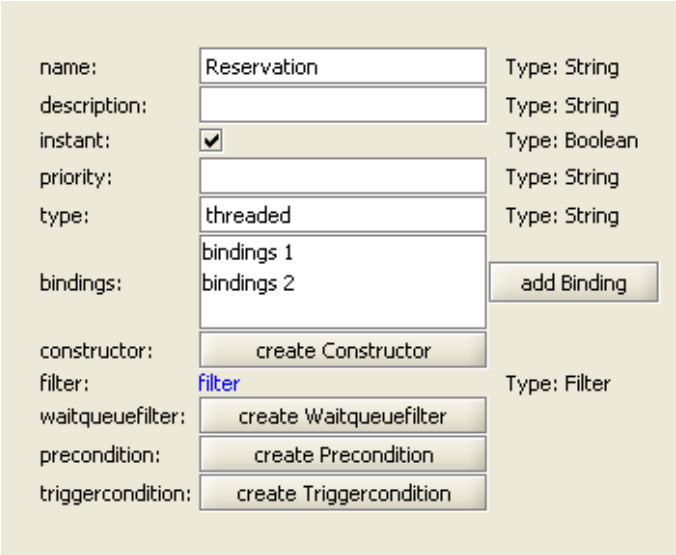


Abbildung 5.3: Screenshot - Formular eines Planes

tet werden. Darüberhinaus lassen die Diagramme auch Rückschlüsse betreffend der Vollständigkeit bzw. Gültigkeit der Agentenmodelle zu. Dazu werden die einzelnen Diagrammelemente verschiedenfarbig eingefärbt, so daß auf einen Blick erkennbar ist, welche Modellelemente hinreichend modelliert wurden und an welchen Stellen noch weitere Informationen benötigt werden. Die Navigation durch das Agentenmodell ist ebenfalls über die Diagrammkomponente möglich. Bei der Implementaion des Diagrammeditors wurde darauf geachtet, daß die Bedienung der Diagramme möglichst kompfortabel und intuitiv vollzogen werden kann. Als grafisches Framework zur Darstellung der Diagramme wurde das JGraph-Framework benutzt.

5.6.1 Das JGraph-Framework

Das JGraph-Framework ist ein Open-Source Projekt, das bei „SourceForge.net“ bezogen werden kann (http://sourceforge.net/projects/jgraph/). Dieses Framework ist komplett in der Sprache Java geschrieben, vollständig kompatibel zur Java User-Interface-Bibliothek „Swing“ und zeichnet sich durch eine klare übersichtliche und wohl dokumentierte Architektur (siehe [Ald03]) aus.

Die Model-View-Controller Architektur von JGraph ist in Abbildung 5.4 dargestellt und besitzt ein hohes Maß an Gemeinsamkeiten mit der Swing-Komponente „JTree“. Ein Graph besteht, wie aus der Graphentheorie bekannt, aus einer Menge

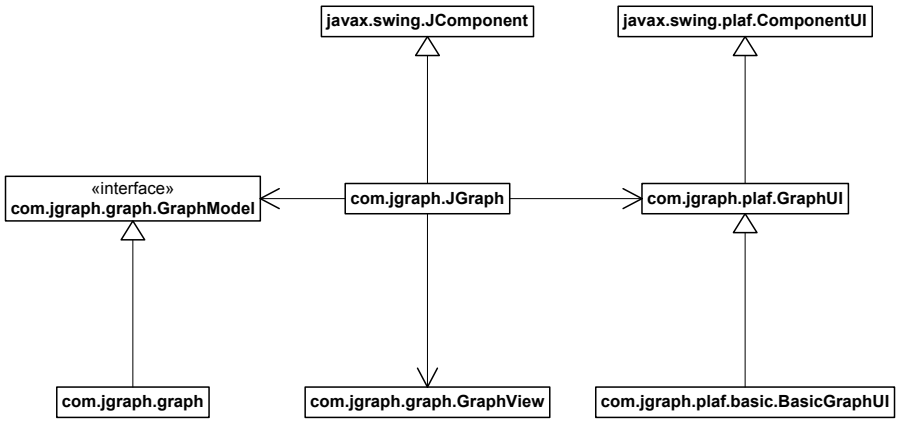


Abbildung 5.4: JGraph - Model View Controller [Ald03]

von Knoten und einer Menge von Kanten. Der gesamte Graph wird durch die Klasse JGraph vertreten, der von der Swing-Klasse „JComponent“ erbt und somit die Rolle eines Controllers einnimmnt. Die Klasse JGraph besitzt eine Referenz auf das Modell des Graphen (`GraphModel`) und eine Referenz auf eine Sicht des Graphen, der `GraphView`. Ebenfalls wird der Mechanismus des „Plugable Look and Feel“ der Swing-Bibliothek unterstützt, indem die Verwaltung des User-Interfaces für einen JGraph delegiert wird.

Das Modell des Graphen kapselt alle Daten für diesen Graph und seiner Bestandteile den „Cells“. Diese Cells sind entweder Knoten, in der Nomenklatur von JGraph „Vertices“, Kanten („Edges“) und „Ports“. Die Ports enthalten dabei die Verbindungsinformation der Kanten, also Informationen darüber welcher Knoten Quelle bzw. Ziel einer Kante ist.

Die Cells werden durch das Interface `GraphCell` repräsentiert, welches seinerseits das Interface `MutableTreeNode` implementiert. Dieses Interface wird dann wiederum von den Interfaces `Edge` und `Port` implementiert. Für den Diagrammeditor des AMoK-Werkzeuges war es ausreichend die ebenfalls im JGraph-Framework vorhanden Standardimplementationen (`DefaultGraphCell`, `DefaultEdge` und `DefaultPort`) dieser Interfaces zu benutzen, deren Einordnung in eine gemeinsame Schnittstellen-Hierarchie in Abbildung 5.5 dargestellt ist.

Bei der Implementation des Diagrammeditor brauchte demnach nur eine Abbildung der Modellelemente des Agentenmodells auf die Modellelemente „GraphCell“, „Edge“ und „Port“ des Graphenmodells „GraphModel“ vorgenommen werden. Da die

Klasse `DefaultGraphCell` eine Sub-Klasse der Klasse `DefaultMutableTreeNode` stand außerdem das Selektionsmodell der Komponente „JTree“ zur Verfügung. Anhand dieses Selektionsmodells kann die Auswahl eines Diagrammelementes auf eine Auswahl eines Modellelementes abgebildet werden, um die Navigation durch das Agentenmodell auch mit Hilfe der grafischen Darstellung zu ermöglichen.

Das folgende Codebeispiel skizziert die Verwendung des JGraph-Frameworks bei der Erzeugung eines Diagrammelementes mit dem Namen des Typs des zugehörigen Agentenmodellelementes:

```
RefObject refObj = refClass.refCreateInstance(new Vector());
String name = refObj.refClass().toString();

GraphModel graphModel = new DefaultGraphModel();
JGraph graph = new JGraph(model);

Map attributes = new Hashtable();
DefaultGraphCell graphCell = new DefaultGraphCell(name);
AttributeMap attrib = graphModel.createAttributes();
attributes.put(graphCell, attrib);
graphModel.insert(cell, attributes);

JPanel diagrammPanel = new JPanel(graph);
```

In Abbildung 5.6 ist ein mit dem AMoK-Diagrammeditor erstelltes Agentenmodell abgebildet.

5.6.2 Das Präsentations-Modell

Zusätzlich zu dem gundlegenden BDI-Agentenmetamodell wurde ein weiteres Metamodell, das Präsentations-Metamodell, in das AMoK-Werkzeug integriert, mit dem ein Modell zur Anpaßung des Werkzeugs in einen bestimmten Anwendungskontext erstellt werden kann. Dieses Metamodell dient einzig und allein dem Zweck den Wiedererkennungswert der zu modellierenden Komponenten zu erhöhen, indem diese mit geeigneten Grafiken bzw. Symbolen versehen werden können. Das Präsentationsmetamodell liegt als XMI-Dokument vor und wird beim Startvorgang des Werkzeugs geladen. Beim Importieren von Metamodellen wird ein zugehöriges Präsentationsmodell instanziiert, das dann optional an das Metamodell angepaßt werden kann.

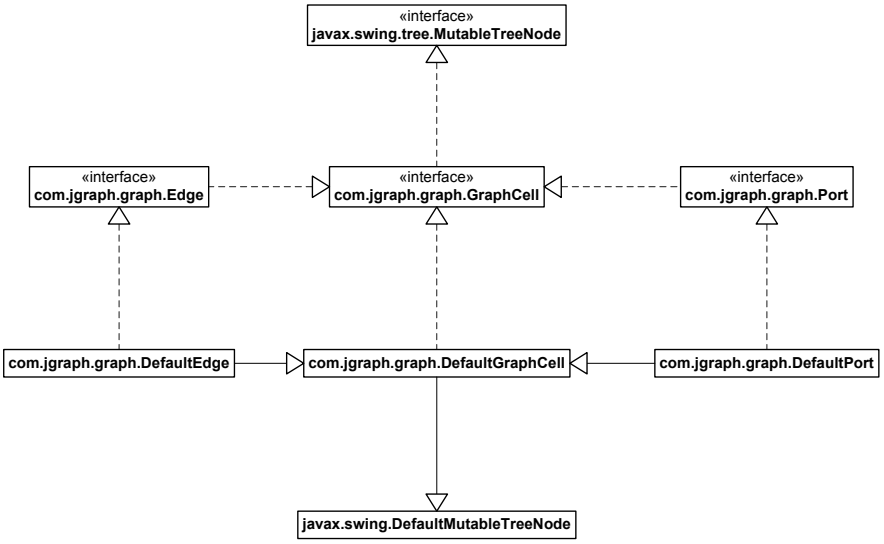


Abbildung 5.5: GraphCell Schnittstellenhierarchie [Ald03]

Die Struktur des Präsentationsmetamodelles ist in Abbildung 5.7 dargestellt. Zu jedem Element des importierten Metamodells wird ein Metaobjekt „Entry“ erzeugt und dem Container „Presentation“ über eine Element-Id zugeordnet. Über einen Dialog kann der Anwender die einzelnen Metamodellelemente mit Grafikdateien versehen, deren Namen und Pfadangaben in den „Entry“-Objekten abgelegt werden. Die angegebenen Grafikdateien werden dann zur Illustration der Agentenmodellelemente sowohl vom Diagrammeditor als auch von den Baumansichten der „Expertview“ und der „Defaultview“ benutzt.

5.7 Export von Modellen

Die mit AMoK erstellten Modelle können sowohl als XMI-Dokumente exportiert werden, als auch in eine für das Agentenframework adäquate Repräsentationsform transformiert werden. Für den Export mittels XMI wurde die Netbeans-Implementation des XMIWriter in das Werkzeug integriert. Somit ist es möglich sowohl die Agentenmodelle als auch das zugehörige Agentenmetamodell anderen MDA-Anwendungen zur weiteren Verarbeitung zu überlassen.

Für die sonstige Erzeugung von Agentenspezifikationen aus deren Modellen kommt der im folgenden Abschnitt vorgestellte Code-Generator zum Einsatz.

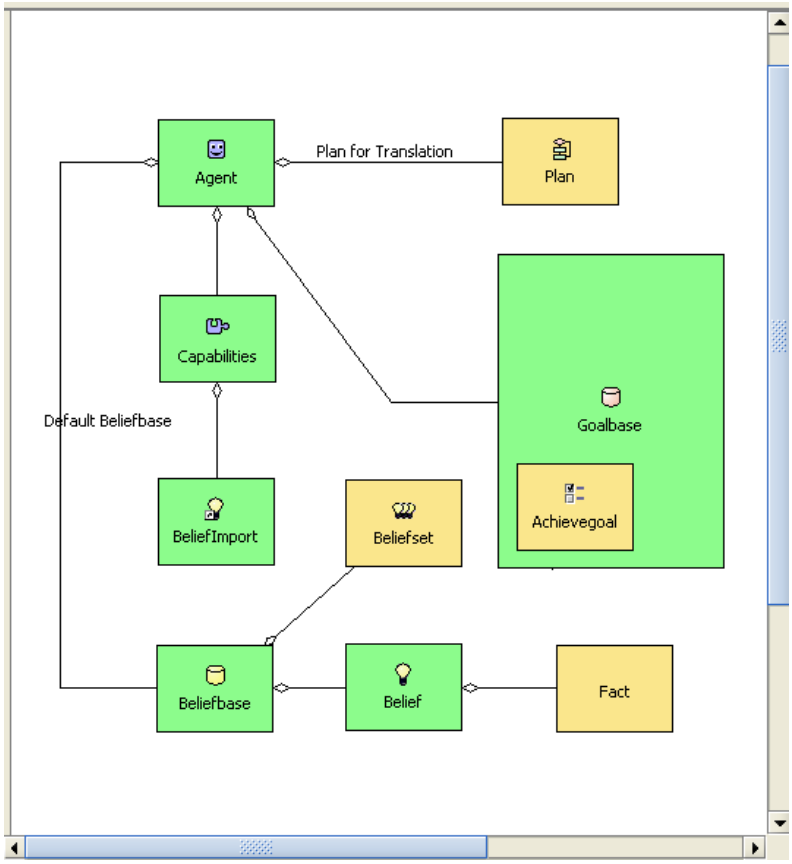


Abbildung 5.6: Screenshot - Diagramm eines Agentenmodells

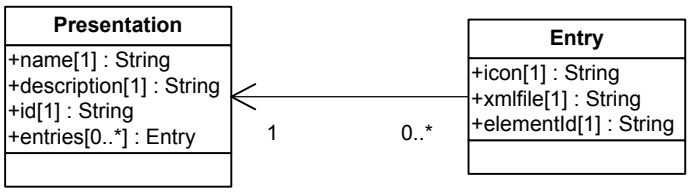


Abbildung 5.7: Präsentations-Metamodell

5.8 Der Code-Generator

Beim Entwurf und bei der Implementation des Code-Generators mußte berücksichtigt werden, daß die möglichen Agenten-Repräsentationsformen bezüglich der verwendeten Syntax sehr heterogen sein können. So bietet Jadex die Möglichkeit Agenten mittels XML-Dokumenten zu definieren, andere Plattformen wie JAM dagegen benutzen fünf separate Text-Dateien, deren Grammatik in einer erweiterten Backus-Naur-Form beschrieben ist, zur Definition der Agenteninterna. Wiederum andere Frameworks wie beispielsweise das kommerzielle JACK bieten eine Java-Spracherweiterung inklusive Präcompiler an, um die interne Architektur eines Agenten in einer eigenen Agentenbeschreibungssprache zu formulieren.

Diese Tatsache resultierte in der Verwendung einer Template-Engine. Für jedes Framework muß ein eigenes Template erstellt werden, aus dem die Struktur des Agenten hervorgeht. Dieses Template wird anschließend von einem Export-Plugin ausformuliert. Für AMoK wurde ein bereits ein solches Export-Plugin (ADF-Exporter) entwickelt, welches auf der Template-Engine „Velocity“ beruht.

5.8.1 Die Velocity Template-Engine

Die Velocity Template-Engine (siehe [Apa04]) ist eine Open Source Entwicklung des Apache Jakarta Projektes und kann in der derzeit aktuellen Version 1.4 unter http://jakarta.apache.org/velocity/index.html bezogen werden. Zur Beschreibung der Templates wird die Velocity eigene Template Language (VTL) benutzt. Diese definiert eine Reihe von Statements zur Verwendung von Variablen, Referenzen und Direktiven aus denen sich ein Template zusammensetzt.

Die API der Velocity stellt einen Container (`java.util.Map`) zur Aufnahme von Schlüssel-Werte Paaren bereit. Dieser Container, das sogenannte Context-Objekt, wird vom ADF-Exporter initialisiert und mit sämtlichen Schlüssel-Wert Paaren gefüllt. Die Schlüssel entsprechen dabei den im Template verwendeten Platzhaltern, die durch die zugehörigen Werte, die vom ADF-Exporter mittels der generierten Metamodell API abgefragt werden, zu ersetzen sind. Dieses Merging von Context-Objekt und Template übernimmt die Template-Engine. Das Ergebnis dieses Merging-Prozesses wird dann vom ADF-Exporter in eine vom Anwender angegebene Ausgabedatei geschrieben.

Das folgende, gekürzte Code-Beispiel aus dem ADF-Exporter soll den soeben beschrieben Vorgang demonstrieren:

```
Velocity.init();
Template template = Velocity.getTemplate("template.vm");
StringWriter stringWriter = new StringWriter();

VelocityContext context = new VelocityContext();

bdiagent.Agent agent = getRootObject();
if (agent != null){
   context.put("agentName", agent.getName());
   context.put("agentClass", agent.getClassName());
}
template.merge(context, stringWriter);
```

Ein zugehöriges Template könnte dann so aussehen:

```
<agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://jadex.sourceforge.net/jadex.xsd"
name="$agentName"' class="$agentClass">
</agent>
```

Analog zu diesem Beispiel wurde das komplette Template erstellt, so daß der ADF-Exporter in der Lage ist, Jadex-ADFs zu generieren. Ein solches ADF enthält, das Vorhandensein eines entsprechenden Agentenmodelles vorausgesetzt, alle Informationen die Jadex benötigt um den betreffenden Agenten zu initialisieren und auszuführen.

6 Zusammenfassung und Ausblick

Ziel der vorliegenden Arbeit war es, den Prozeß des Modellierens von auf mentalen Konzepten basierenden Agenten zu unterstützen. Um dieses Ziel zu erfüllen, wurde nach eingehender Betrachtung der BDI-Agentendomäne nach einer adäquaten Architektur gesucht, die den gestellten Anforderungen an das in dieser Arbeit entwickelte Entwurfswerkzeug „AMoK“ genügt. Da der Bereich der Agentensysteme und somit auch der Agentenframeworks zur Umsetzung solcher Systeme ständigen Neu- und Weiterentwicklungen unterliegt, muß sich dieses Entwurfswerkzeug mit möglichst geringem Aufwand an solche neuen Situationen anpassen lassen. Ein Prototyp eines derartigen Modellierungswerkzeuges, sowie seine zugrundeliegende Architektur wurde in dieser Arbeit entworfen, implementiert und vorgestellt.

Dieses Werkzeug überbrückt die Kluft zwischen Entwurf und Implementation eines Agenten dadurch, daß es in der Lage ist, eine abstrakte Beschreibung der internen Architektur eines solchen Agenten zu benutzen, um das Erstellen konkreter Agentenmodelle zu ermöglichen, und diese anschließend in eine geeignete Agentenspezifikation zu transformieren. Aufbau und Struktur solcher Agentenspezifikationen sind zwar für konkrete Frameworks wie Jadex bekannt, deren automatische Erzeugung soll aber ebenfalls flexibel und erweiterbar unterstützt werden. Ein auf Metamodellen basierendes Modellierungswerkzeug, das eine automatische Code-Generierung unterstützt, ermöglicht den Prozeß des „Model Driven Development“ und bedingt eine damit assoziierte „Model Driven Architecture“ (MDA).

Ansätze für eine solche „Model Driven Architecture“ wurden bereits von der „Object Management Group“ (OMG) vorgeschlagen und resultieren in einer Reihe von Standards und deren Spezifikationen. Diese Spezifikationen ermöglichen erstmals die Realisierung von standardisierten Implementationen zum Austausch, zur Verwaltung und zur Transformation von Metadaten und Metametadaten. Das Potential von in einer grafischen Modellierungssprache erstellten Modellen als formales Beschreibungsmittel komplexer Strukturen und Zusammenhänge kann in solch einer Architektur besser als bisher ausgeschöpft werden, da deren Informationsgehalt direkt auf ein Code-Modell abgebildet wird. Dieser Sachverhalt erfordert jedoch eine

exakte, klare und formale Semantikdefinition der verwendeten Modellierungssprache.

Das BDI-Agentenmetamodell für das Entwurfswerkzeug „AMoK“ wurde mit der Notationsform der Klassendiagramme deklariert, welche durch die Modellierungssprache UML bereitgestellt werden. Die Tatsache, daß die abstrakte Syntax dieser Modellierungssprache durch die ebenfalls von der OMG standardisierte „Meta Object Facility“ (MOF) beschrieben werden kann, ließ vermuten, daß sich Instanzen beider Modelle problemlos aufeinander abbilden lassen.

In dieser Arbeit wurde festgestellt, daß dies in der Praxis jedoch nicht problemlos möglich ist. Weitere normative Spezifikationen wie die des „UML Profile for MOF“ sowie konzeptuelle Überlegungen bei der Erstellung des Metamodells für BDI-Agenten mußten die Diskrepanz zwischen UML und MOF schließen. Die Evolution beider OMG-Entwicklungen gibt jedoch Grund zu der Annahme, daß sich zukünftige Versionen von MOF und UML noch stärker als bisher aneinander orientieren werden, und sich der Kern der UML im Kontext der MDA besser als bisher, als grafisches Beschreibungsmittel von MOF-konformen Metamodellen, eignen wird.

Die Spezifikation JSR 040 des Java Metadata Interchange, welche ein Mapping von Metadaten auf Java-Interfaces ermöglicht, vereinfacht die Implementierung generischer, auf beliebigen Metamodellen basierender Komponenten, wie Metadaten-Explorer, Formulargeneratoren, Diagrammgeneratoren und Modellimport- bzw. Exportkomponenten, aus denen sich das Werkzeug „AMoK“ zusammensetzt. Dieser Prototyp zeigt, daß die benutzte Architektur im Kontext des „Model Driven Development“ tragfähig ist und eine kompfortable Modellierung von BDI-Agenten zuläßt.

Dennoch impliziert dieses Werkzeug keinen konkreten Anwendungsbezug und ließe sich nach entsprechender Modifikation des Metamodells sowie der Velocity-Templates auch in anderen Bereichen, wie der Generierung von SQL-Befehlen aus Modellen eines Datenbankschemas u.ä. einsetzen. Die Fokussierung auf den Kontext der BDI-Agentensysteme begründet sich dadurch, daß dieses Themengebiet sehr aktuell ist und die werkzeuggestützte Modellierung von Agentenarchitekturen dementsprechend unterrepräsentiert ist.

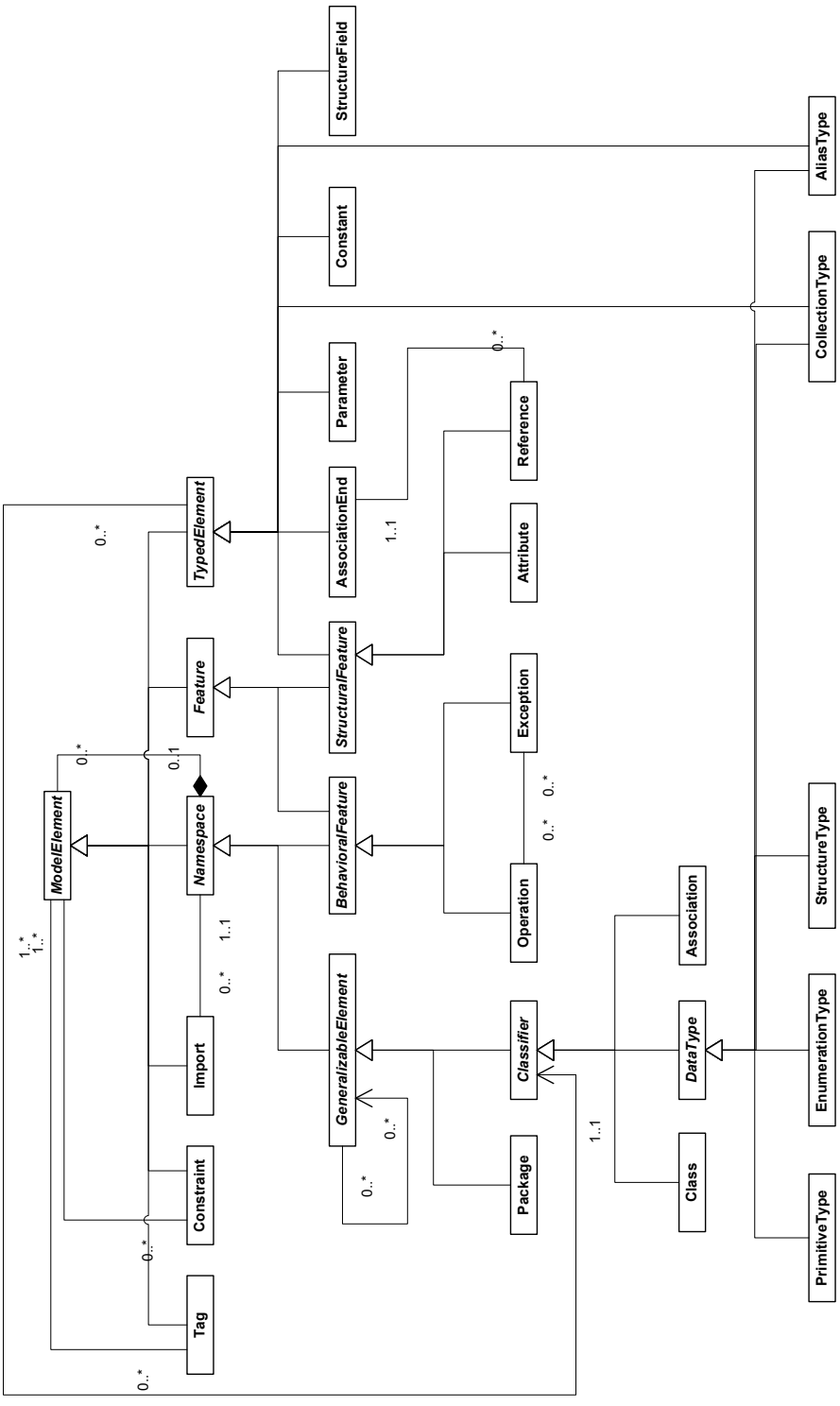
Die Frage nach der Akzeptanz eines solchen Werkzeugs in der Praxis kann nicht eindeutig beantwortet werden. Es wird vermutet, daß diese stark von der Rolle des Anwenders und der Anwendungsdomäne abhängt. Läßt sich diese Rolle mit der Rolle eines Systementwicklers vereinbaren, führen die gebotenen Vorteile der Flexibilität und Adaptionsfähigkeit dieses Werkzeugs zu einem unverkennbaren Mehrwert in

Form von Einsparung von Arbeitzeit, da bei einer solchen Adaption keine Eingriffe in den Programm-Code nötig sind.

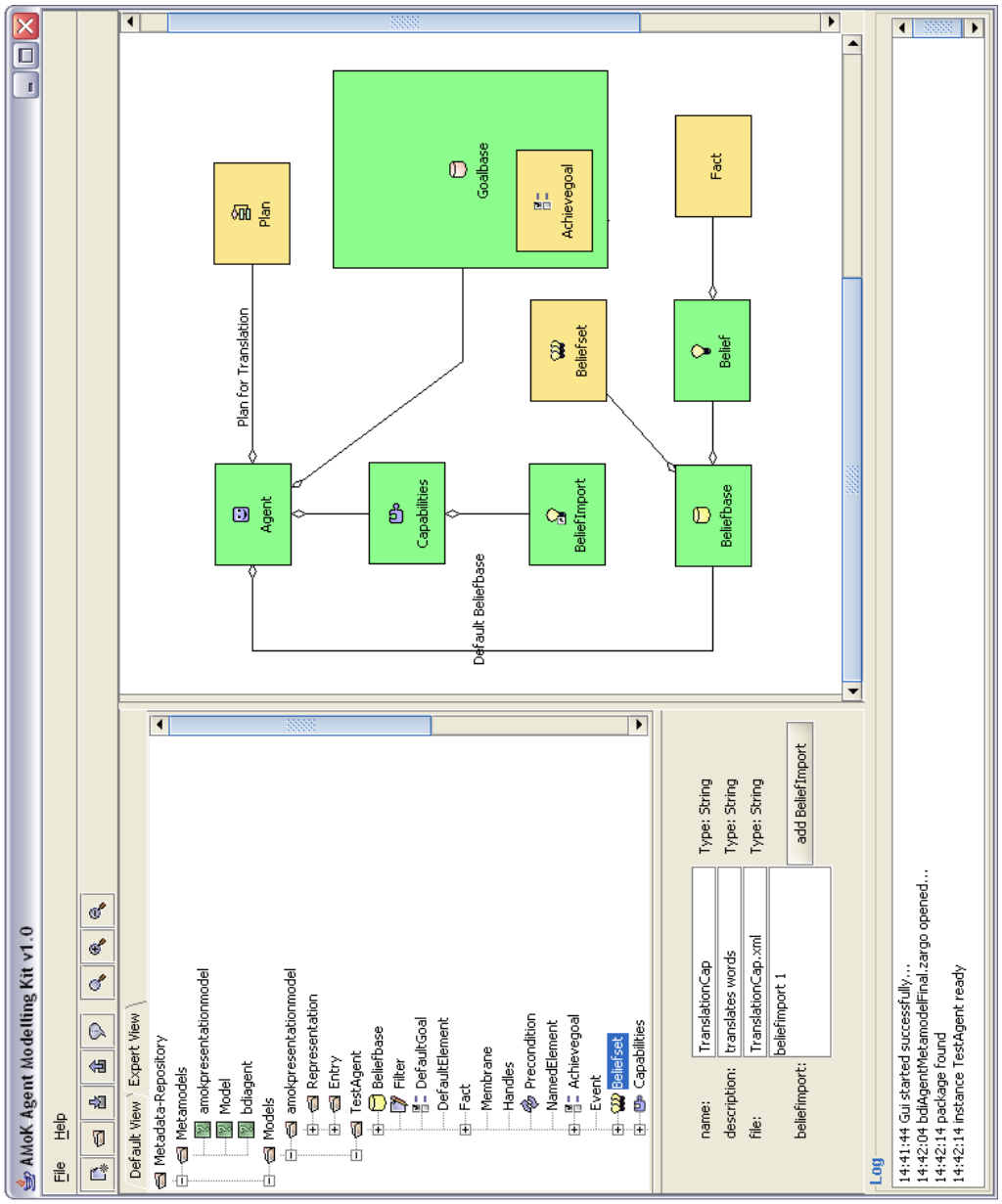
Stehen dagegen Aspekte wie Hilfestellung und Anwenderfreundlichkeit bei der Modellierung des entsprechenden Systems im Vordergrund, bedarf es eines klar erkennbaren und vom Entwurfswerkzeug durchgesetzten Modellierungsprozesses gemäß der vertretenen Methodologie. Diese bedingt nicht nur Wissen um die strukturelle Beschaffenheit, sondern auch um die Bedeutung der zu erstellenden Modelle und kann nicht von „AMoK“ unterstützt werden. Diese These wird durch den derzeitigen Mangel an ähnlich gearteten Referenzprojekten gestützt.

Abschließend läßt sich feststellen, daß die Konzentration eines Modellierungswerkzeuges allein auf die rein statischen Aspekte einer Agentenarchitektur nicht ausreichend erscheint. Mechanismen zur prozeßorientierten Gestaltung von Agentenplänen, Kooperationsmodellen oder Verhandlungsprotokollen könnten das AMoK-Werkzeug zukünftig erweitern. Solche dynamischen Aspekte bedingen jedoch völlig andere Anforderungen an die Systemarchitektur, die mit der derzeitigen MDA-Architektur nicht erfüllt werden können.

A MOF-Modell


StructureField
Constant
Parameter
AssociationEnd
Reference
Attribute
StructuralFeature
Exception
BehavioralFeature
Operation
GeneralizableElement
Classifier
Package
Association
DataType
Class
AliasType
CollectionType
StructureType
EnumerationType
PrimitiveType
TypedElement
Feature
ModelElement
Namespace
Import
Constraint
Tag
0..*
1..1
1..*

B Das Agent Modeling Kit



Abbildungsverzeichnis

Literaturverzeichnis

[Age04] Agent Oriented Software: *JACK Intelligent Agents - Agent Manual*. 2004

[Ald03] Alder, Gaudenz: Design and Implementation of the JGraph Swing Component. `http://prdownloads.sourceforge.net/jgraph/paper1.1.pdf?download`, 2003. – Forschungsbericht

[And03] Andresen, Andreas: *Komponentenbasierte Softwareentwicklung mit MDA, UML und XML*. Carl Hanser Verlag München Wien, 2003

[Apa04] Apache Jakarta Project. *Velocity User's Guide*. `http://jakarta.apache.org/velocity/user-guide.html`. 2004

[BCT03] Bellifemine, Fabio ; Caire, Giovanni ; Trucco, Tiziana: JADE Administrator's Guide / Telecom Italia. `http://jade.tilab.com/doc/administratorsguide.pdf`, Dezember 2003. – Forschungsbericht

[BPL04] Braubach, Lars ; Pokahr, Alexander ; Lamersdorf, Winfried: Jadex: A Short Overview. In: *Main Conference Net.ObjectDays 2004*, 2004

[Bra87] Bratman, Michael E.: *Intention, Plans, and Practical Reason*. Harvard University Press, 1987

[Bru91] Brustoloni, Jose C. *Autonomous Agents: Characterization and Requirements*. 1991

[Bur97] Burkhardt, Rainer: *UML-Unified Modeling Language: Objektorientierte Modellierung für die Praxis*. Addison Wesley, 1997

[DL89] Durfee, E.H. ; Lesser, V.R: Negotiating Task Decomposition and Allocation Using Partial Global Planning. In: *Distributed Artificial Intelligence* 2 (1989), S. 229–244

[FG96] Franklin, Stan ; Graesser, Art: Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, Springer Verlag, 1196 (Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages)

[FIP02] FIPA: FIPA Abstract Architecture Specification / Foundation for Intelligent Physical Agents. `http://www.fipa.org/specs/fipa00001/XC00001K.pdf`, November 2002 (XC00001K). – Forschungsbericht

[FM03] Fransson, Jens ; Müller, Stefan: *UML-Diagramme - Austausch und Interaktion*, Universität Hamburg Fachbereich Informatik, Diplomarbeit, Juli 2003

[GBB03] Gräßle, Patrick ; Baumann, Henriette ; Baumann, Philippe: *UML projektorientiert*. Galileo Press, 2003

[GL87] Georgeff, M. P. ; Lansky, A. L.: Reactive Reasoning and Planning. In: *Proceedings of the Sixth National Conference on Artificial Intelligence*, 1987

[Her96] Hermans, Björn: *Intelligent Software Agents on the Internet: an inventory of currently offered functionality in the information society & a prediction of (near-)future developments*, Universität Tilburg, Holland, Diplomarbeit, Juli 1996

[HSN97] H. S. Nwana, D.T. N.: An Introduction to Agent Technology. In: Hyacinth S. Nwana, Nader A. (Hrsg.): *Software Agents and Soft Computing* Bd. 1198, Springer Verlag, 1997

[Hub00] Huber, Marcus J.: JAM: A BDI-theoretic Mobile Agent Architecture. In: *AgentLink News* (2000), Mai

[Jec] Jeckle, Mario. *100 UML-Tools*. `http://www.jeckle.de/umltools.html`

[Jec00] Jeckle, Mario: Konzepte der Metamodellierung - Zum Begriff Metamodell. In: *Softwaretechnik Trends, Bd. 20, Heft 2* (2000), Mai

[KGR96] Kinny, David ; Georgeff, Michael ; Rao, Anand: A Methodology and Modelling Technique for Systems of BDI Agents. In: *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96* Bd. 1038, Springer Verlag, 1996

[Küh01] Kühnel, Ralf: *Agentenbasierte Softwareentwicklung.* Addison-Wesley, 2001

[KNB+03] Krempels, K.-H. ; Nimis, J. ; Braubach, L. ; Pokahr, A. ; Herrler, R. ; Scholz, T.: Entwicklung intelligenter Multi-Multiagentensysteme - Werkzeugunterstützung, Lösungen und offene Fragen. In: K. Dittrich, A. Oberweis K. Rannenberg W. W. (Hrsg.): *Informatik 2003 - 33. Jahrestagung der GI* Bd. P-34 Gesellschaft für Informatik e.V., Köllen Druck+Verlag GmbH, 9 2003, S. 31–46

[KWB03] Kleppe, Anneke ; Warmer, Jos ; Bast, Wim: *MDA explained - the model driven architecture: practice and promise.* Addison-Wesley, April 2003 (The Addison-Wesley Object Technology Series)

[Man02] Mangina, Eleni: Review of Software Products for Multi-Agent Systems. In: *AgentLink* (2002), Juni

[Mül96] Müller, Jörg P.: *The design of intelligent agents - A layered approach.* Springer Verlag, 1996

[MW95] Michael Wooldridge, Nicholas R. J.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review* (1995), Januar

[OMG02a] OMG: Meta Object Facility (MOF) Specification v1.4 / Object Management Group, Inc. OMG. `http://www.omg.org/technology/documents/formal/mof.htm`, April 2002. – Technischer Bericht

[OMG02b] OMG: OMG XML Metadata Interchange (XMI) Specification v1.2 / Object Management Group, Inc. OMG. `http://www.omg.org/technology/documents/formal/xmi.htm`, Januar 2002. – Technischer Bericht

[OMG03] OMG: Meta Object Facility (MOF) Specification v2.0 / Object Management Group, Inc. OMG. `http://www.omg.org/technology/documents/formal/mof.htm`, April 2003. – Technischer Bericht

[OMG04] OMG: UML Profile for Metaobject Facility (MOF) Specification v1.0 / Object Management Group, Inc. OMG. `http://www.omg.org/docs/formal/04-02-06.pdf`, Februar 2004. – Technischer Bericht

[RG95] Rao, Anand S. ; Georgeff, Michael P.: BDI Agents: From Theory to Practice. (1995)

[SPVG03] Sycara, Katia ; Paolucci, Massimo ; Velsen, Martin V. ; Giampapa, Joseph: The RETSINA MAS Infrastructure. In: *Autonomous Agents and Multi-Agent Systems* 7 (2003), Nr. 1-2

[SR95] S.J. Russel, P. N.: Artificial Intelligence: A Modern Approach. (1995)

[Str98] Strahringer, Susanne: Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips. In: *GI-Workshop Münster* (1998), März

[Sud04] Sudeikat, Jan: *Betrachtung und Auswahl der Methoden zur Entwicklung von Agentensystemen*, Hochschule für Angewandte Wissenschaften Hamburg, Diplomarbeit, 2004

[UC02] Unisys Corporation, Ravi D.: Java Metadata Interface (JMI) Specification v1.0 / Java Community Process. `http://java.sun.com/products/jmi/`, Juni 2002. – Technischer Bericht

[W3C04a] W3C, 2004: Extensible Markup Language (XML) 1.1 / World Wide Web Consortium (W3C). `http://www.w3.org/TR/2004/REC-xml11-20040204/`, April 2004. – W3C Recommendation

[W3C04b] W3C, 2004: XML Schema Part 0: Primer Second Edition / World Wide Web Consortium. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`, Oktober 2004. – W3C Recommendation