

# Codegenerierung für Assoziationen in MOF 2.0

Carsten Amelunxen, Andy Schürr  
Fachgebiet Echtzeitsysteme  
Technische Universität Darmstadt  
D-64283 Darmstadt  
e-mail: [carsten.amelunxen|andy.schuerr]@es.tu-darmstadt.de

Lutz Bichler  
Institut für Softwaretechnologie  
Universität der Bundeswehr München  
D-85577 Neubiberg  
e-mail: lutz.bichler@unibw-muenchen.de

## Abstract:

Die Spezifikation von MOF 2.0 enthält neue Konstrukte zum Ausdrücken von Beziehungen zwischen Assoziationen. Damit diese Konstrukte in modellgetriebenen Entwicklungsprozessen sinnvoll eingesetzt werden können, werden Abbildungen auf objektorientierte Programmiersprachen benötigt. Im Rahmen dieses Papiers stellen wir Möglichkeiten vor, die neuen Konstrukte auf Java-Code abzubilden. Unsere Vorschläge basieren auf dem Java Metadata Interface (JMI) Standard, der die Abbildung von MOF-Modellen auf Java-Schnittstellen festlegt.

## 1 Einleitung

Die Entwicklung von Softwaresystemen hat im Laufe der Jahre nicht nur an Bedeutung gewonnen, sondern ist mit der fortschreitenden technologischen Entwicklung auch immer komplexer geworden. Bestrebungen, das Problem der wachsenden Komplexität zu lösen, führten zu dem Konzept der modellbasierten Entwicklung. Insbesondere in den letzten Jahren hat die modellbasierte Softwareentwicklung deutlich an Bedeutung gewonnen, was zu einem großen Teil auf die Existenz einer einheitlichen Modellierungssprache (UML) [Ob03b] zurückzuführen ist.

Die weitere Zunahme von Akzeptanz und Anwendung der modellbasierten Softwareentwicklung ist jedoch direkt abhängig von den Möglichkeiten der Codegenerierung aus Modellen. Derzeit existierende Vorschläge und Standards ermöglichen oft keine vollständige Abbildung von Modellen auf Programmiersprachen. So bietet z. B. das Java Metadata Interface (JMI) [Di02] lediglich eine Lösung zum Generieren von Schnittstellen aus MOF-Modellen an. Zudem werden nur die Signaturen der Schnittstellen exakt festgelegt, nicht aber das Verhalten einer bestimmten Schnittstelle.

Viele der in MOF enthaltenen Modellierungselemente sind ungenau oder sogar unvollständig beschrieben. Dies führt zu unterschiedlichen Interpretationen der Semantik dieser Modellierungselemente und entsprechend unterschiedlichen Übersetzungen auf Programmiersprachen. So gibt es z.B. unterschiedliche Ansätze zur Abbildung von Assoziationen, deren Anwendung zu unterschiedlichem Verhalten der in JMI definierten Schnittstellen führen würde. Dies zeigt, dass ein Standard zur Abbildung von Modellierungssprachen auf Programmiersprachen auch die Implementierung bzw. das Verhalten von Schnittstellen berücksichtigen muss.

Außerdem enthält die neue Spezifikation der Meta Object Facility (MOF) [ACC<sup>+</sup>03] einige neue Konstrukte zum Ausdruck von Beziehungen zwischen Assoziationen bzw. Assoziationsenden. Diese neuen Beziehungen müssen auf Java-Code abgebildet werden, um den JMI-Standard auch weiterhin zur Abbildung von MOF-Modellen verwenden zu können. Daher bilden die neuen Konstrukte und ihre Umsetzung in Java-Code den Schwerpunkt dieser Arbeit. Es werden Vorschläge angegeben, wie die neuen Konstrukte unter Einhaltung der durch JMI vorgegebenen Schnittstellen in Java implementiert werden können.

Im Folgenden werden in Abschnitt 2 existierende Vorschläge zur Abbildung von Assoziationen auf Programmiersprachen vorgestellt. In Abschnitt 3 werden die neuen Assoziations-Beziehungen der MOF 2.0 Spezifikation genauer dargestellt und in Abschnitt 4 Vorschläge zur Abbildung dieser Konstrukte auf Quellcode gemacht. Abschließend erfolgt in Abschnitt 5 eine Zusammenfassung der Arbeit und ein Ausblick auf weitere Konstrukte der MOF, deren Abbildung auf Java noch untersucht werden muss.

## **2 Verwandte Arbeiten**

Assoziationen und ihre Abbildung auf objektorientierte Programmiersprachen werden in der Literatur häufig diskutiert. Die Diskussion reicht zurück bis in die Anfänge der objektorientierten Softwareentwicklung [Ru87]. Zur Abbildung von Assoziationen auf objektorientierte Programmiersprachen werden im Wesentlichen zwei Konzepte (mit entsprechend vielen Varianten) vorgeschlagen. Entweder werden Assoziationen auf sich wechselseitig referenzierende Attribute abgebildet ([Ru96], [No96], [No97], [GAADC02]), oder auf eigene Klassen, die die Assoziation unabhängig von den beteiligten Klassen repräsentieren ([GBHS97], [HBR00]).

Rumbaugh [Ru96] betrachtet Assoziationen als ein Konzept der objektorientierten Modellierung, das auf derselben Stufe wie die Klasse und das Objekt anzuordnen ist. Da die meisten objektorientierten Programmiersprachen keine Assoziationen unterstützen, müssen diese auf Klassen und Attribute abgebildet werden. Die Art der Abbildung ist von den Eigenschaften der Assoziation abhängig. Rumbaugh unterscheidet logische Eigenschaften, Eigenschaften des Designs und Eigenschaften der Zielsprache. Logische Eigenschaften sind z.B. Assoziationsname, Rollennamen und Multiplizität. Als Design-Eigen-

schaften bezeichnet Rumbaugh Navigierbarkeit, Veränderbarkeit und Sichtbarkeit. Unter Eigenschaften der Zielsprache fallen alle Möglichkeiten, eingebettete Objekte oder Objektpreferenzen in einer der gewählten Zielsprache auszudrücken.

Bei der Abbildung von Assoziationen auf eine Programmiersprache unterscheidet Rumbaugh zwischen qualifizierten und unqualifizierten Assoziationen. Er schlägt vor, unqualifizierte Assoziationen auf Zeiger abzubilden, die in Attributen der beteiligten Klassen gespeichert werden. Für Multiplizitäten, die größer als 1 sind, wird ein Attribut erzeugt, dessen Typ eine Menge von Zeigern ist. Eine qualifizierte Assoziation wird in ähnlicher Weise abgebildet. Der einzige Unterschied ist, dass eine zusätzliche Indirektion erforderlich ist, um das qualifizierende Attribut abzubilden. Zum Zugriff auf die Assoziationsenden werden für Multiplizitäten 0..1 und 1 Methoden zum Setzen und Abfragen, und für Multiplizitäten größer als 1 Methoden zum Einfügen, Abfragen und Löschen vorgeschlagen.

Allerdings behandelt Rumbaugh lediglich die grundlegenden Eigenschaften der Abbildung von Assoziationen auf Programmiersprachen. Besondere Probleme einzelner Eigenschaften, z.B. der Multiplizität, Navigierbarkeit oder Sichtbarkeit, werden nicht behandelt. Diese Lücke schließen Henderson [HBR00] und insbesondere Genova et. al. [GdCL03] durch zusätzliche Abbildungsvorschläge. Navigierbarkeit und Sichtbarkeit können während der Codegenerierung behandelt werden, da sich diese Eigenschaften einer Assoziation zur Laufzeit nicht ändern. Dagegen muß die Multiplizität zur Laufzeit überprüft werden, da sich die Anzahl der zu einem Objekt existierenden Instanzen einer Assoziation während der Laufzeit verändern kann. Insbesondere exakt festgelegte Multiplizitäten, d.h. Multiplizitäten, deren untere und obere Grenzen übereinstimmen, stellen ein Problem für die Einfüge- bzw. Löschoperationen dar. Daher wird vorgeschlagen, die untere Grenze nicht beim Löschen einer Assoziation zu überprüfen, sondern nur beim Abfragen. Dies ermöglicht das Ändern einer Assoziation mit einer festgelegten Multiplizität durch aufeinander folgendes Löschen und Hinzufügen. Allerdings wird die Verantwortung für die Erhaltung der Konsistenz an den Programmierer übertragen, da auch Zustände erzeugt werden können, die nicht der Spezifikation entsprechen.

Weiterhin wird zwischen statischen und dynamischen Assoziationen unterschieden ([Ru96], [St]). Statische Assoziationen modellieren eine strukturelle Beziehung zwischen Objekten, während dynamische Assoziationen zum Aufruf eines bestimmten Verhaltens verwendet werden. Die Abbildung einer dynamischen Assoziation kann durch einen Parameter für eine Operation einer Klasse abgebildet werden. Dynamische Assoziationen sind für MOF-Modelle weniger bedeutend als für UML-Modelle, da in MOF nur zur Modellierung der Struktur eines Systems vorgesehen ist.

Eine weitere Möglichkeit, Assoziationen zu klassifizieren, ist die Aggregation. In der Literatur werden bis zu sieben verschiedene Stufen der Aggregation unterschieden ([KR94], [HS98]). In den bisherigen MOF-Spezifikationen wurden die Aggregationstypen *none*, *shared* und *composite* unterschieden, wobei der Aggregationstyp *shared* aufgrund seiner unklaren Bedeutung nicht unterstützt wurde. Folgerichtig existieren in MOF 2.0 nur noch die Aggregationstypen *none* und *composite*, so dass bei der Codegenerierung nur noch diese berücksichtigt werden müssen. Die Abbildung dieser Aggregationsstufen auf objektorientierte Programmiersprachen erfordert eine geeignete Kopplung der Lebenszyklen

beteiligter Objekte. Dies ist in Java schwieriger als in anderen Sprachen, da Objekte in der Regel nicht explizit gelöscht werden.

Obwohl das Problem der Abbildung von Assoziationen bereits über einen langen Zeitraum behandelt wird, ist bis heute keine allgemein anerkannte „beste“ Lösung gefunden worden. Dies drückt sich unter anderem darin aus, dass Assoziationen von den meisten Werkzeugen nachwievor einfach auf Attribute und wechselseitige Zugriffsoperationen in den beteiligten Klassen abgebildet werden. Im Zusammenhang mit der modellbasierten Entwicklung gewinnt das Thema Codegenerierung in letzter Zeit wieder an Bedeutung. Dies gilt insbesondere im Zusammenhang mit der Model-Driven Architecture (MDA) [MM01], die eine Vorgehensweise zum modellbasierten Entwickeln auf der Basis der Unified Modeling Language (UML) [Ob03a] definiert. Die UML wird durch ein Metamodell definiert, das eine Instanz der MOF [ACC<sup>+</sup>03] ist. Um den Umgang mit MOF-basierten Modellen zu erleichtern und ihre Implementierung zu vereinfachen, wurden mehrere Ansätze entwickelt, die MOF- Modelle implementieren ([Ma03], [EMF02]) und über Schnittstellen zugänglich machen.

Trotz ihrer Nähe zu den Standards MOF und JMI unterscheiden sich die Werkzeuge erheblich. Dies liegt im Wesentlichen an den unterschiedlichen Anwendungsschwerpunkten, die zu verschiedenen Schwerpunkten bzgl. des Funktionsumfangs geführt haben, aber auch an der Unvollständigkeit des JMI-Standards. Aus diesem Grund müssen zukünftige Versionen des JMI-Standards neben der Festlegung der Schnittstellen auch exakte Festlegungen des Verhaltens dieser Schnittstellen enthalten. Zudem bietet die UML Infrastrukturbibliothek neue Mechanismen zur Definition von Beziehungen zwischen Assoziationsenden an. Zur Abbildung dieser Mechanismen werden ebenfalls Abbildungsvorschriften benötigt. Deren Definition bildet den Schwerpunkt der folgenden Abschnitte dieser Arbeit.

### **3 Objektbeziehungen in MOF 2.0**

Objekte sind der Grundbaustein objektorientierter Programmier- und Modellierungssprachen. Die Bestandteile eines Systems werden in objektorientierten Modellen durch Klassen modelliert und durch Objekte, als Instanzen dieser Klassen, repräsentiert. In der Regel sind die Bestandteile eines Systems nicht voneinander unabhängig. Daher müssen Beziehungen zwischen Objekten ausgedrückt werden, um eine vollständige Abbildung eines Systems zu ermöglichen.

In diesem Abschnitt werden die Möglichkeiten zur Darstellung von Assoziationen in der Meta Object Facility (MOF) diskutiert. Die MOF ist die Basis der objektorientierten Programmierungs- und Modellierungsstandards der Object Management Group (OMG). MOF ist eine objektorientierte Modellierungssprache, die zur Definition anderer Modellierungssprachen, insbesondere der UML verwendet wird und wird daher häufig auch als Metamodellierungssprache bezeichnet, da ihr Hauptanwendungsgebiet die Definition von Metamodellen ist.

In den 1.x Versionen von MOF und UML waren diese durch voneinander unabhängige Metamodelle mit ähnlichen, aber häufig in Details voneinander abweichenden Konstruk-

ten definiert. Um die daraus resultierenden Schwierigkeiten in Zukunft zu vermeiden, wurden die Metamodelle der 2.0-Versionen aneinander angeglichen. Die UML wurde in Infrastruktur und Superstruktur unterteilt. Die Infrastruktur stellt die Basiskonzepte der objektorientierten Modellierung bereit und dient insbesondere zur Definition von Modellierungssprachen. MOF ist als Erweiterung der UML Infrastruktur definiert und stellt Mechanismen bereit, die zum Umgang mit Modellen benötigt werden. Die Besonderheit der MOF ist, dass diese Mechanismen für alle Modelle definiert sind, die Instanzen einer Instanz von MOF sind.

### 3.1 Attribute

Objekte werden durch Instanziierungen von Klassen erzeugt, die Struktur und Verhalten einer Klasse festlegen. Die Attribute einer Klasse bestimmen die Eigenschaften der Objekte, und die Methoden beschreiben das Verhalten der Objekte. Der Typ eines Attributes kann entweder ein Datentyp oder eine Klasse sein. Wenn es sich bei dem Typ eines Attributes um eine Klasse handelt, entspricht dies einer gerichteten Beziehung von der Klasse, die das Attribut enthält, zu der Klasse, die der Typ des Attributes ist.

Eine derartige, implizite Beziehung zwischen Klassen beschränkt sich auf das Verwenden von Instanzen anderer Klassen. Eine explizite Semantik bzw. konkrete Eigenschaften von Beziehungen können mit dieser Art von Beziehung nicht modelliert werden. Das MOF 2.0-Metamodell bietet die Möglichkeit, implizite Beziehungen durch sich gegenseitig referenzierende Attribute zu modellieren. Das Prinzip des gegenseitigen Referenzierens limitiert die Beziehung auf einen ungerichteten, binären Zusammenhang. Im Unterschied zu expliziten Assoziationen ist bei der Verwendung von wechselseitig referenzierten Attributen keine Instanziierung weiterer Klassen des MOF-Metamodells notwendig, um eine Beziehung zwischen zwei Klassen abzubilden. Dies ermöglicht eine sehr einfache und effiziente Abbildung der Beziehung in objektorientierten Programmiersprachen.

In Abb. 1 ist ein wechselseitiger Verweis zwischen den Klassen `Mitarbeiter` und `Identifikation` dargestellt. Notationell wird bei der Verwendung von Attributen zur Definition einer Beziehung zwischen Klassen nicht von der Verwendung einer expliziten Assoziation unterschieden. Im Unterschied zu den Attributen `name`, `vorname` und `ident` wird das Attribut `mitarbeiter` der Klasse `Identifikation` daher wie ein Assoziationsende dargestellt.

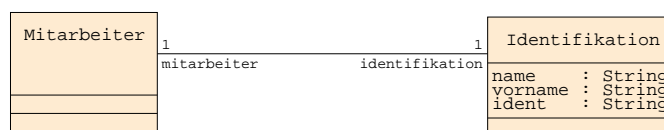


Abbildung 1: Verwendung von Attributen zum Ausdruck einer Beziehung

Aus dieser Notation resultiert das Problem, dass aus der grafischen Darstellung eines Modells nicht auf dessen Repräsentation durch Elemente des MOF-Metamodells geschlossen

werden kann. Da die Codegenerierung aber auf der Repräsentation im Metamodell basiert, kann der Modellierer die Codegenerierung für eine Assoziation nicht beeinflussen. Stattdessen entscheidet das verwendete Werkzeug über die Repräsentation und damit über den entstehenden Code. Um dem Modellierer die Beeinflussung der Codegenerierung zu ermöglichen, müssen den verschiedenen Möglichkeiten zum Ausdruck von Beziehungen zwischen Objekten in MOF auch verschiedene Notationen zugeordnet werden.

Die MOF-Spezifikation erlaubt die Darstellung einer Assoziation durch eine Raute (siehe Abb. 3). Eine mögliche Unterscheidung ist daher, durch Attribute ausgedrückte Beziehungen durch eine einfache Linie und explizite Assoziationen durch eine Linie mit einer Raute darzustellen. Kombiniert mit den Symbolen zur Festlegung der Navigierbarkeit könnten Assoziationen vollständig vom Modellierer spezifiziert werden.

### 3.2 Assoziationen

Beziehungen zwischen Klassen können in MOF 2.0 auch explizit durch Instanziierung der Metaklasse `Association` des MOF-Metamodells realisiert werden. Instanzen dieser Klasse repräsentieren eine explizite Assoziation durch Attribute, die Eigenschaften der Assoziation darstellen, und referenzieren zudem zwei sogenannte Assoziationsenden, die die Verbindungen zu den assoziierten Klassen herstellen. Assoziationsenden kapseln die Eigenschaften der Assoziation bezüglich der zugehörigen Klasse.

In MOF 2.0 können die Assoziationsenden entweder Bestandteil der Assoziation oder der Klasse sein. Der besitzende Classifier (vgl. [Ob03a], Abschnitt 11.3), also Klasse oder Assoziation, ist in der Lage, über das Assoziationsende zu navigieren. Neben Eigenschaften zur Festlegung der Sortierung (`isOrdered`) oder der Eindeutigkeit (`isUnique`) von Assoziationsinstanzen bietet MOF 2.0 verschiedene Möglichkeiten, Beziehungen zwischen den Mengen der Assoziationsinstanzen zu definieren. Diese Beziehungen beeinflussen den Code, der generiert werden muss in entscheidendem Maße. In den Abschnitten 3.2.1 bis 3.2.3 werden die Beziehungen von Assoziationsenden ausführlich dargestellt.

Das folgende Szenario mit zugehöriger Modellierung in Abbildung 2 soll den Einsatz von Beziehungen zwischen Assoziationsenden in MOF 2.0 verdeutlichen und im weiteren Verlauf als Beispiel für eine konkrete Darstellung dienen.

*Die Entwickler eines Systemhauses bearbeiten eine Reihe von Entwicklungsprojekten. Es werden Software- und Hardwareentwicklungsprojekte bearbeitet. Die Softwareentwickler bearbeiten ausschließlich Softwareentwicklungsprojekte und die Hardwareentwickler bearbeiten ausschließlich Hardwareentwicklungsprojekte.*

Das Klassendiagramm in Abbildung 2 beschreibt einen Ausschnitt des Szenarios und verwendet dazu die verschiedenen Definitionsmöglichkeiten für Assoziationsenden. Die genauen Bedeutungen der dadurch ausgedrückten Einschränkungen der Assoziationen werden in den folgenden Abschnitten erläutert.

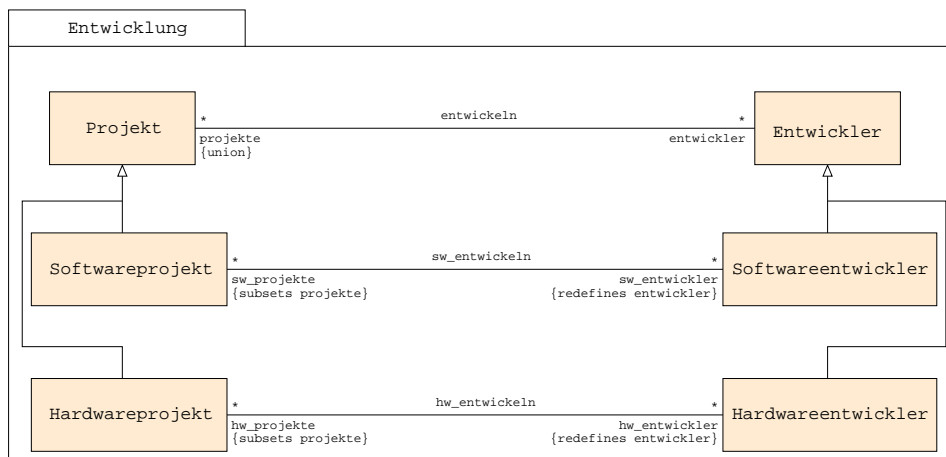


Abbildung 2: Definitionen von Assoziationsenden

### 3.2.1 redefines

Wenn zwei Assoziationsenden verschiedener Assoziationen typkonform sind, kann eine Teilmengenbeziehung zwischen zwei Assoziationsenden beschrieben werden. Die Typkonformität besteht dabei in der Typkonformität der jeweils zugehörigen Klasse. Ein Assoziationsende einer Assoziation kann ein Assoziationsende einer anderen Assoziation redefinieren. Dies wird durch Angabe von **redefines** an einem Assoziationsende notiert. Eine Redefinition eines Assoziationsendes bedeutet eine Einschränkung der möglichen Instanzen der Assoziation, die das redefinierte Assoziationsende enthält. Es sind nur noch Instanzen möglich, die auch Instanzen der Assoziation sind, welche das redefinierende Ende enthält.

Unser Beispiel enthält die Assoziationsenden `sw_entwickler` und `hw_entwickler`, die jeweils das Assoziationsende `entwickler` redefinieren. Dadurch wird festgelegt, dass in einem Hardwareprojekt nur Hardwareentwickler und in einem Softwareprojekt nur Softwareentwickler arbeiten.

### 3.2.2 subsets

Zwischen typkonformen Assoziationsenden kann eine Teilmengenbeziehung ausgedrückt werden, so dass Instanzen eines als Teilmenge deklarierten Assoziationsendes ebenso als Instanzen des als Obermenge fungierenden Assoziationsendes geführt werden. Dazu wird das eine Assoziationsende als **subset** eines anderen Assoziationsendes definiert. In unserem Beispiel sind die Assoziationsenden `sw_projekte` und `hw_projekte` als Teilmengen des Assoziationsendes `projekte` definiert. Dies drückt aus, dass jedes Softwareprojekt und jedes Hardwareprojekt auch in der Menge der Projekte enthalten ist.

### 3.2.3 union

Eine Vereinigungsmengenbeziehung zwischen einer Menge von Assoziationsenden kann durch **union** angegeben werden. In Abb. 2 wird durch die Definition des Assoziationsendes **projekte** als **union** festgelegt, dass die Menge der Instanzen von **projekte** ausschliesslich aus der Vereinigung der Mengen der als **subset** deklarierten Assoziationsenden besteht. In unserem Beispiel wird dadurch ausgedrückt, dass ausschließlich Software- und Hardware entwickelt wird, da andersartige Entwicklungsprojekte durch die **union** ausgeschlossen werden.

### 3.3 Vererbung von Assoziationen

Assoziationen können von anderen Assoziationen erben. Dies wird ähnlich der Vererbung bei Klassen durch einen Pfeil von der erbenden zur geerbten Assoziation dargestellt (vgl. Abb. 3). Vererbung bei Assoziationen ist dann sinnvoll, wenn attributierte Assoziationen verwendet werden, da die Attribute dann auch in der spezialisierenden Assoziation verwendet werden können. Bei einer bloßen Verknüpfung zweier Assoziationen über deren Enden, wie in Abschnitt 3.2, würden diese Informationen verloren gehen.

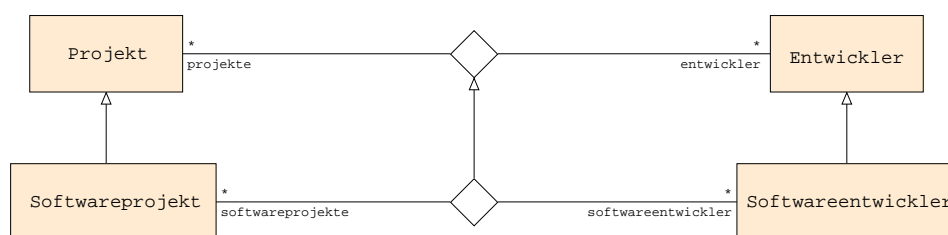


Abbildung 3: Vererbung zwischen Assoziationen

Gemäß der Spezifikation der UML-Infrastrukturbibliothek ist allerdings undefiniert, welche Auswirkungen die Vererbung zwischen Assoziationen auf die Assoziationsenden der erbenden Assoziation hat (siehe [Ob03a], Seite 113). Insbesondere ist unklar, wie sich die Kombination der Vererbung von Assoziation mit den in Abschnitten 3.2.1 bis 3.2.3 beschriebenen neuen Spezifikationsmöglichkeiten für Assoziationsenden verhält.

Eine Diskussion der verschiedenen Möglichkeiten, Vererbung und Definition von Assoziationsenden zu mischen, kann im Rahmen dieses Papiers aus Umfangsgründen nicht erfolgen. Aus diesem Grund werden im folgenden Kapitel auch keine Vorschläge zur Übersetzung von Vererbungsbeziehungen zwischen Assoziationen in Code angegeben.

## 4 Codegenerierung

Ein wesentlicher Aspekt für die praktische Relevanz von MOF und insbesondere der vorgestellten MOF 2.0 Konstrukte ist ihre Abbildung auf Java-Quellcode. Mit der JMI-Spezifikation existiert ein Ansatz, um MOF 1.4-konforme Metamodelle auf Java-Code abzubilden [Di02].

Die JMI-Spezifikation umfasst jedoch lediglich einen Mechanismus zur Generierung von Schnittstellen und definiert das Verhalten einer JMI-konformen Implementierung nur informell. Eine konkrete Abbildung eines MOF-Modells in Implementierungscode enthält die JMI-Spezifikation nicht.

Die Schnittstellen für die Erzeugung und den Zugriff auf die Metadaten teilen sich in vier Kategorien auf:

- Paket-Schnittstellen für den Zugriff auf alle zugehörigen Objekte der im entsprechenden Paket enthaltenen Modellelemente
- Klassenstellvertreter-Schnittstellen für die Erzeugung und Verwaltung von Instanz-Objekten und die Bereitstellung von klassenbezogenen Attributen und Methoden der zugehörigen Klasse
- Instanz-Schnittstellen für die Bereitstellung von objektbezogenen Attributen und Methoden der zugehörigen Klasse
- Assoziations-Schnittstellen für die Erzeugung und Verwaltung von Instanzen der zugehörigen Assoziation

Eine Abbildung von Assoziationen auf einen reinen Referenzierungs-Mechanismus, wie in Abschnitt 4.1 dargestellt, ist schon aufgrund der möglichen Attributierung von Assoziationen nicht möglich. Die JMI-Spezifikation sieht daher eine Schnittstelle pro Assoziation vor. Diese Schnittstelle enthält Methoden um eine Menge von Tupeln zu verwalten. Die verwalteten Tupel von assoziierten Klasseninstanzen werden zur einfacheren Betrachtung im Folgenden als Links bzw. Link-Objekte bezeichnet. Link-Objekte kapseln jeweils eine Instanz der zugehörigen Assoziation. Die Klasse, welche die generierte Schnittstelle implementiert, wird daher nur einmal instanziiert.

Unserer Ansicht nach sollte sich ein Vorschlag für die Abbildung von MOF 2.0 auf Java nicht auf eine Erweiterung der existierenden JMI-Spezifikation beschränken, sondern konkrete Codevorschläge zur Implementierung der definierten Schnittstellen enthalten. Dies würde zum einen verschiedene Interpretationen des Verhaltens einer JMI-konformen Implementierung vermeiden und zum anderen die Implementierung von JMI-konformen Werkzeugen erleichtern. Die ab Abschnitt 4.2 folgenden Abbildungen der in Abschnitt 3 beschriebenen Möglichkeiten zur Definition von Beziehungen zwischen Klassen auf Java beziehen sich also auf die Implementierung der von JMI vorgesehenen Assoziations-Schnittstellen.

## 4.1 Referenzierende Attribute

Die Methode des gegenseitigen Verweises, wie in Abschnitt 3.1 beschrieben, stellt eine Möglichkeit dar, eine Beziehung zwischen zwei Klassen ohne explizite Assoziation zu realisieren. Referenzierende Attribute stellen einen wechselseitigen Verweis dar und können direkt, ohne eine Abbildung auf eine zusätzliche Klasse, implementiert werden. Der JMI-Generierungsmechanismus definiert die Abbildung einer Referenz durch Erzeugen von Accessor- und Mutator-Methoden in den entsprechenden Schnittstellen. Da die gegenseitige Referenzierung von Attributen in MOF 2.0 singularer Art ist, beschränkt sich die Abbildung auf die Implementierung einer get- und einer set-Methode.

Es existieren bereits erprobte Verfahren um eine gegenseitige Referenzierung zu implementieren. Das Graphersetzungswerkzeug FUJABA [FU] verwendet einen Mechanismus, der eine sinnvolle Implementierung des gegenseitigen Verweises darstellen kann. Abbildung 4 zeigt eine mögliche Implementierung eines gegenseitigen Verweises nach der Methode von FUJABA anhand des Beispiels aus Abbildung 1. Diese Methode implementiert die Instanz-Schnittstellen der beherbergenden Klassen der verweisenden Attribute.

```
private Mitarbeiter mitarbeiter;  
  
public Mitarbeiter getMitarbeiter () throws  
    javax.jmi.reflect.JmiException {  
    return this.mitarbeiter;  
}  
  
public void setMitarbeiter (Mitarbeiter value) throws  
    javax.jmi.reflect.JmiException {  
    if (this.mitarbeiter != value)  
    {  
        if (this.mitarbeiter != null)  
        {  
            Mitarbeiter oldValue = this.mitarbeiter;  
            this.mitarbeiter = null;  
            oldValue.setIdentifikation (null);  
        }  
        this.mitarbeiter = value;  
        if (value != null)  
        {  
            value.setIdentifikation (this);  
        }  
    }  
}
```

Abbildung 4: Implementierung eines gegenseitigen Verweises nach der Methode von FUJABA

## 4.2 Assoziationen

Für die Abbildung der Beziehungen zwischen Assoziationsenden in MOF 2.0 müssen drei Fälle unterschieden werden. Die Betrachtungen der einzelnen Fälle erfolgen in den jeweiligen Abschnitten.

Generell erfordert die Generierung von Implementierungscode für Assoziationen in MOF 2.0 die Unterscheidung von navigierbaren und nicht navigierbaren Assoziationsenden. Der Begriff der Navigierbarkeit wird von der MOF 2.0 Spezifikation gegenüber früheren Spezifikationen und dem allgemeinen Verständnis des Begriffs neu definiert. In MOF 2.0 sind alle Assoziationsenden navigierbar in dem Sinne, dass für jede Klasseninstanz alle verlinkten Instanzen ermittelt werden können. Navigierbarkeit im Sinne von MOF 2.0 bedeutet, dass eine Klasse direkt Methoden anbietet, um auf assoziierte Instanzen zugreifen zu können. Die Ermittlung von verlinkten Klasseninstanzen eines nicht navigierbaren Assoziationsendes muss über die Methoden der entsprechenden Assoziationsklassen erfolgen.

Die folgenden Implementierungsvorschläge bilden nicht navigierbare Assoziationsenden ab. Navigierbare Assoziationsenden entsprechen referenzierten Assoziationsenden in den früheren MOF Spezifikationen. Die Abbildungsvorschrift von JMI zur Umsetzung referenzierter Assoziationsenden kann also auf navigierbare Assoziationsenden in MOF 2.0 angewendet werden. Für referenzierte bzw. navigierbare Assoziationsenden wird in JMI die Instanz-Schnittstelle um Accessor- und Mutator-Methoden für die entsprechenden Collections von Link-Objekten der zugehörigen Assoziation erweitert. Die Vorschläge zur Implementierung von Assoziationsbeziehungen beeinflussen lediglich die Verwaltung von Link-Objekten in Assoziationsklassen und sind daher für navigierbare sowie nicht navigierbare Assoziationsenden gleichermaßen anzuwenden.

Das MOF 2.0 Metamodell bietet für Assoziationsenden die Attribute `isOrdered` und `isUnique` an. Das gesetzte Attribut `isOrdered` wirkt sich auf die JMI-Schnittstellen derart aus, dass die Menge der Link-Objekte in einer sortierten Java-Collection (`java.util.List`) verwaltet werden. Die folgenden Implementierungsvorschläge gehen von dem Vorgabewert (`isOrdered = false`) aus.

Das Attribut `isUnique` gibt an, ob zwischen Klasseninstanzen mehrere Instanzen (Links) derselben Assoziation existieren können. Die folgenden Implementierungsvorschläge gehen hier ebenfalls von dem Vorgabewert (`isUnique=true`) aus, da eine Berücksichtigung dieses Attributes eine eindeutige Identifizierung von Assoziationsinstanzen erfordern und somit die derzeitige JMI-Spezifikation verletzen würde.

Zur Demonstration der Quellcodegenerierung und zur weiteren Diskussion wird das Modell aus Abbildung 2 verwendet. Die von der JMI-Spezifikation generierte Schnittstelle für die `entwickeln`-Assoziation ist in Abbildung 5 dargestellt. Die Methoden der Abbildung enthalten folgende Semantik:

- `exists`: Gibt an, ob eine Assoziationsinstanz zwischen den übergebenen Objekten existiert.
- `getProjekte`: Liefert eine Menge von Objekten, die mit dem übergebenen Objekt über eine Assoziationsinstanz verknüpft sind.

- `getEntwickler`: Liefert eine Menge von Objekten, die mit dem übergebenen Objekt über eine Assoziationsinstanz verknüpft sind.
- `add`: Verknüpft die übergebenen Objekte über eine Assoziationsinstanz.
- `remove`: Entfernt die Assoziationsinstanz zwischen den übergebenen Objekten.

```
public interface Entwickeln extends
javax.jmi.reflect.RefAssociation {
    public boolean exists (Projekt projekte, Entwickler entwickler)
        throws javax.jmi.reflect.JmiException;

    public Collection getProjekte (Entwickler entwickler) throws
        javax.jmi.reflect.JmiException;

    public Collection getEntwickler (Projekt projekte) throws
        javax.jmi.reflect.JmiException;

    public boolean add (Projekt projekte, Entwickler entwickler)
        throws javax.jmi.reflect.JmiException,
        javax.jmi.reflect.DuplicateException;

    public boolean remove (Projekt projekte, Entwickler entwickler)
        throws javax.jmi.reflect.JmiException;
}
```

Abbildung 5: JMI Schnittstelle für die `entwickeln`-Assoziation des Modells aus Abbildung 2

#### 4.2.1 redefines

Die Anwendung einer `redefines` Beziehung auf zwei Assoziationsenden ist nur möglich, wenn zwischen der zu dem redefinierenden Assoziationsende gehörenden Klasse und der zu dem redefinierten Assoziationsende gehörenden Klasse eine Vererbungsbeziehung besteht (vgl. Abbildung 2). Die Redefinition eines Assoziationsendes soll in diesem Kontext die geerbte Assoziation der Oberklasse ersetzen.

In dem Beispiel der Abbildung 2 wirkt sich die `redefines`-Beziehung derart aus, dass durch die Redefinition des Assoziationsendes `entwickler`, eine direkte, durch die Vererbung ermöglichte `entwickeln`-Assoziation zwischen einem Objekt der Klasse `Softwareprojekt` und einem Objekt der Klasse `Entwickler` nicht möglich ist. Die Abfrage von Instanzen der `entwickeln`-Assoziation zwischen einem Objekt der Klasse `Softwareentwickler` und einem `Entwickler` liefert die Verknüpfungen der Assoziation `sw_entwickeln`.

Aufgrund der Vererbungsbeziehung zwischen der Klasse `Projekt` und der Klasse `Softwareprojekt` kann der `entwickeln`-Assoziation aus Abbildung 5 auch ein Tupel, bestehend aus einem `Softwareprojekt`- und einem `Entwickler`-Objekt, übergeben werden. Die `redefines`-Beziehung zwischen den Assoziationsenden `entwickler` und `sw_entwickler` soll eine derartige Verknüpfung verhindern.

Die Unterbindung einer solchen Verknüpfung kann nur über die Implementierung der Schnittstelle erfolgen. Abbildung 6 zeigt notwendige Ergänzungen einer möglichen Realisierung der `redefines`-Beziehung zwischen den Assoziationsenden `entwickler` und `sw_entwickler` aus Abbildung 2. Die Methoden der Abbildung 6 enthalten folgende Semantik:

- `exists`: Fängt die durch `redefines` verbotene Verknüpfung durch eine Ausnahme ab. Andere Fälle werden gemäß der vorgesehenen Semantik behandelt.
- `getProjekte`: Keine Veränderung gegenüber der vorgesehenen Semantik
- `getEntwickler`: Die durch `redefines` verbotene Verknüpfung wird aus der redefinierenden Assoziation entnommen. Andere Fälle werden gemäß der vorgesehenen Semantik behandelt.
- `add`: Fängt die durch `redefines` verbotene Verknüpfung durch eine Ausnahme ab. Andere Fälle werden gemäß der vorgesehenen Semantik behandelt.
- `remove`: Fängt die durch `redefines` verbotene Verknüpfung durch eine Ausnahme ab. Andere Fälle werden gemäß der vorgesehenen Semantik behandelt.

In dem Beispiel aus Abbildung 6 werden die durch die `redefines`-Beziehung ausgeschlossenen Fälle explizit abgefangen. Die Implementierung der `sw_entwickeln`-Assoziation zwischen den Klassen `Softwareprojekt` und `Softwareentwickler` bleibt unverändert.

#### 4.2.2 subsets

Eine `subsets`-Beziehung zwischen zwei Assoziationsenden definiert das durch „subsets“ gekennzeichnete Assoziationsende als Untermenge des mit „subsets“ angegebenen Assoziationsendes. Instanzen von Assoziationen, die das als Untermenge gekennzeichnete Assoziationsende beinhalten, sind also sowohl in der Menge der als Untermenge definierten Assoziation als auch in der Menge der als Obermenge fungierenden Assoziation zu führen. Für die Implementierung einer `subsets`-Beziehung bedeutet dies, dass das als `subsets` definierte Assoziationsende betreffende Veränderungen in der entsprechenden Assoziation an die verknüpfte Assoziation propagiert.

Abbildung 8 zeigt notwendige Ergänzungen zur Implementierung einer `subsets`-Beziehung anhand der `sw_entwickeln`-Beziehung aus Abbildung 2. Dabei wird zunächst angenommen, dass `projekte` nicht als `union` definiert sei. Der Code der als Obermenge fungierenden `entwickeln`-Assoziation bleibt bis auf die Änderung in Abbildung 7 unverändert. Die Abbildung 7 enthält eine Ergänzung, die notwendig ist, wenn in der Obermenge Instanzen gelöscht werden, die ebenfalls Teil der Untermenge sind. Die `remove`-Methode leitet den Aufruf an die entsprechende Untermenge weiter und führt eine Löschung auf der eigenen Menge von Instanzen durch, um die Mengen synchron zu halten. Die Instanz der Assoziation wird dabei über das zugehörige Paket ermittelt.

Der Code aus Abbildung 8 hält sich strikt an die JMI-Spezifikation. Die von der JMI-Spezifikation definierte Kapselung von Assoziationsinstanzen in Form von Link-Objekten

```

public class EntwickelnImpl implements Entwickeln {
    public boolean exists (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException
    {
        if((projekte instanceof Softwareprojekt) &&
            !( entwickler instanceof Softwareentwickler ))
            throw new JmiException( "Entwickler has been redefined." );
        else
            // ursprünglich vorgesehener Code
        }

    public Collection getProjekte (Entwickler entwickler) throws
    javax.jmi.reflect.JmiException
    {
        // ursprünglich vorgesehener Code
    }

    public Collection getEntwickler (Projekt projekte) throws
    javax.jmi.reflect.JmiException
    {

        if( projekte instanceof Softwareprojekt )
        {
            Sw_Entwickeln swe =
            EntwicklungPackageImpl.getSw_Entwickeln();
            return swe.getSw_Entwickler((Softwareprojekt)projekte);
        }

        // ursprünglich vorgesehener Code
    }

    public boolean add (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException,
    javax.jmi.reflect.DuplicateException
    {
        if((projekte instanceof Softwareprojekt) &&
            !( entwickler instanceof Softwareentwickler ))
            throw new JmiException( "Entwickler has been redefined." );
        else
            // ursprünglich vorgesehener Code
        }

    public boolean remove (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException
    {
        if((projekte instanceof Softwareprojekt) &&
            !( entwickler instanceof Softwareentwickler ))
            throw new JmiException( "Entwickler has been redefined." );
        else
            // ursprünglich vorgesehener Code
        }
    }
}

```

Abbildung 6: Notwendige Ergänzungen zur Implementierung einer redefines-Beziehung

```

public class EntwickelnImpl implements Entwickeln {

    // ursprünglich vorgesehener Code

    public boolean remove (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException
    {
        if((projekte instanceof Softwareprojekt) &&
            (entwickler instanceof Softwareentwickler))
        {
            Sw_Entwickeln subset =
            EntwicklungPackageImpl.getSw_Entwickeln();
            if(subset.exists((Softwareprojekt)projekte,
                (Softwareentwickler)entwickler))
                subset.remove((Softwareprojekt)projekte,
                    (Softwareentwickler)entwickler);
        }
        // ursprünglich vorgesehener Code
    }
}

```

Abbildung 7: Notwendige Ergänzungen zur Implementierung einer subsets-Beziehung

verursacht eine Gleichsetzung der subsets-Beziehungen von beiden Enden einer Assoziation. Diese Gleichsetzung könnte nur durch eine erhebliche Änderung der JMI-Spezifikation aufgehoben werden. Bei einer neuen Version der JMI-Spezifikation sollte die Verwaltung von Assoziationsinstanzen prinzipiell überdacht werden, zumal die aktuelle Spezifikation auch nicht zwischen mehreren Assoziationsinstanzen zweier gleicher Objekte unterscheiden kann, obwohl diese Unterscheidung bereits von MOF 1.4 gefordert wird. Der Implementierungsvorschlag in Abbildung 8 ist also, analog zu der JMI-Spezifikation, nicht in der Lage zwischen mehreren Assoziationsinstanzen zweier gleicher Objekte zu unterscheiden. Eine derartige Unterscheidung würde die JMI-Spezifikation zu stark verletzen. Die Methoden aus Abbildung 8 enthalten folgende Semantik:

- exists: Keine Veränderung gegenüber der vorgesehenen Semantik
- getSw\_Projekte: Keine Veränderung gegenüber der vorgesehenen Semantik
- getSw\_Entwickler: Keine Veränderung gegenüber der vorgesehenen Semantik
- add: Leitet den Aufruf an die als Obermenge fungierende Assoziation weiter und legt eine eigene Verknüpfung zwischen den übergebenen Objekten an.
- remove: Leitet den Aufruf an die als Obermenge fungierende Assoziation weiter und löscht die Verknüpfung zwischen den übergebenen Objekten aus der eigenen Menge von Assoziationsinstanzen.

```

public class Sw_EntwickelnImpl implements Sw_Entwickeln {
    public boolean exists (Softwareprojekt softwareprojekte,
                          Softwareentwickler softwareentwickler)
    throws javax.jmi.reflect.JmiException
    { // ursprünglich vorgesehener Code    }

    public Collection getSw_Projekte (Softwareentwickler softwareentwickler)
    throws javax.jmi.reflect.JmiException
    { // ursprünglich vorgesehener Code    }

    public Collection getSw_Entwickler (Softwareprojekt softwareprojekte)
    throws javax.jmi.reflect.JmiException
    { // ursprünglich vorgesehener Code    }

    public boolean add (Softwareprojekt softwareprojekte,
                       Softwareentwickler softwareentwickler)
    throws javax.jmi.reflect.JmiException,
           javax.jmi.reflect.DuplicateException
    {
        Entwickeln superset = EntwicklungPackageImpl.getEntwickeln();
        superset.add( softwareprojekte, softwareentwickler );
        // ursprünglich vorgesehener Code
    }

    public boolean remove (Softwareprojekt softwareprojekte,
                           Softwareentwickler softwareentwickler)
    throws javax.jmi.reflect.JmiException
    {
        Entwickeln superset = EntwicklungPackageImpl.getEntwickeln();
        superset.remove( softwareprojekte, softwareentwickler );
        // ursprünglich vorgesehener Code
    }
}

```

Abbildung 8: Notwendige Ergänzungen zur Implementierung einer subsets-Beziehung

### 4.2.3 union

Eine union-Beziehung zwischen Assoziationsenden ist immer eine Erweiterung einer subsets-Beziehung. Das in einer subsets-Beziehung als union gekennzeichnete Assoziationsende ist eine exakte Vereinigung aller Untermengen. Dementsprechend muss die explizite Veränderung der Obermenge generell unterbunden werden. Veränderungen der Obermenge können ausschließlich implizit über die Untermengen vorgenommen werden.

Das Beispiel aus Abbildung 9 zeigt Ergänzungen zur Implementierung einer union-Beziehung anhand des Modells aus Abbildung 2. Die Implementierung der sw\_entwickeln-Beziehung ist identisch mit Abbildung 8. Die Definition des Assoziationsendes projekte als union wirkt sich nur auf die zugehörige entwickeln-Assoziation aus.

Die von der subsets-Beziehung verursachte Veränderung der als Obermenge fungierenden Assoziation entfällt. Im Folgenden wird die Semantik der Methoden aus Abbildung 9 beschrieben:

- exists: Leitet den Aufruf an die entsprechende Assoziation weiter.
- getProjekte: Leitet den Aufruf an die entsprechende Assoziation weiter.
- getEntwickler: Leitet den Aufruf an die entsprechende Assoziation weiter.
- add: Löst eine Ausnahme aus. Explizite Veränderungen einer Vereinigung sind nicht möglich.
- remove: Löst eine Ausnahme aus. Explizite Veränderungen einer Vereinigung sind nicht möglich.

```

public class EntwickelnImpl implements Entwickeln {
    public boolean exists (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException
    {
        if((projekte instanceof Softwareprojekt) &&
            (entwickler instanceof Softwareentwickler))
            return EntwicklungPackageImpl.getSw_Entwickeln()
                .exists((Softwareprojekt)projekte, (Softwareentwickler)entwickler);

        if((projekte instanceof Hardwareprojekt) &&
            (entwickler instanceof Hardwareentwickler))
            return EntwicklungPackageImpl.getHw_Entwickeln()
                .exists((Hardwareprojekt)projekte, (Hardwareentwickler)entwickler);

        return false;
    }

    public Collection getProjekte (Entwickler entwickler) throws
    javax.jmi.reflect.JmiException
    {
        if( entwickler instanceof Softwareentwickler )
            return EntwicklungPackageImpl.getSw_Entwickeln()
                .getSw_Projekte((Softwareentwickler)entwickler);
        if( entwickler instanceof Hardwareentwickler )
            return EntwicklungPackageImpl.getHw_Entwickeln()
                .getHw_Projekte((Hardwareentwickler)entwickler);

        return NULL;
    }

    public Collection getEntwickler (Projekt projekte) throws
    javax.jmi.reflect.JmiException
    {
        if( projekte instanceof Softwareprojekt )
            return EntwicklungPackageImpl.getSw_Entwickeln()
                .getSw_Entwickler((Softwareprojekt)projekte);
        if( projekte instanceof Hardwareprojekt )
            return EntwicklungPackageImpl.getHw_Entwickeln()
                .getHw_Entwickler((Hardwareprojekt)projekte);

        return NULL;
    }

    public boolean add (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException
    {
        throw new JmiException( "Entwickeln is a union of subsets" );
    }

    public boolean remove (Projekt projekte, Entwickler entwickler)
    throws javax.jmi.reflect.JmiException
    {
        throw new JmiException( "Entwickeln is a union of subsets" );
    }
}

```

Abbildung 9: Notwendige Ergänzungen zur Implementierung einer union-Beziehung

## 5 Schlusswort

Die Metamodellierungssprache MOF definiert in der Version 2.0 neue Möglichkeiten zur Spezifikation von Beziehungen zwischen Klassen. Es wird eine Möglichkeit bereitgestellt, eine binäre Beziehung direkt zu spezifizieren, indem zwei Attribute einer Klasse gegenseitige Verweise enthalten. Außerdem lassen sich Beziehungen zwischen den Assoziationsenden verschiedener Assoziationen angeben.

Um diese neuen Möglichkeiten im Rahmen modellgetriebener Softwareentwicklungsprozesse nutzbar machen zu können, werden Abbildungen auf objektorientierte Programmiersprachen benötigt. Diese sind in den zur Zeit existierenden Standards, z.B. JMI, nicht enthalten. In diesem Papier haben wir daher für die wesentlichen neuen Konstrukte der MOF 2.0 Spezifikation Abbildungen auf Java-Code angegeben und ihre Auswirkungen auf die JMI-Schnittstellen beschrieben.

Neben den hier betrachteten Mechanismen verdient insbesondere die Vererbung zwischen Assoziationen eine genauere Betrachtung. Obwohl der Vererbungsmechanismus schon lange in MOF und UML enthalten ist, ist seine genaue Bedeutung nach wie vor unklar. Aus diesem Grund werden wir im nächsten Schritt die genaue Bedeutung der Vererbungsbeziehung zwischen Assoziationen und auch die Kombination von Vererbung und den neuen Spezifikationsmöglichkeiten in MOF 2.0 näher untersuchen.

## Literatur

- [ACC<sup>+</sup>03] Adaptive Ltd, Ceira Technologies Inc., Compuware Corporation, Data Access Technologies Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA Technologies, MetaMatrix, Rational Software, Softeam, Sun Microsystems, Telelogic AB, Unisys, und WebGain: *Meta Object Facility (MOF) 2.0 Core Proposal*. April 2003. ad/2003-04-07.
- [Di02] Dirckze, R.: *Java<sup>TM</sup> Metadata Interface (JMI) Specification, Version 1.0*. Unisys. 1.0. Juni 2002.
- [EMF02] *The Eclipse Modeling Framework (EMF) Overview*. September 2002.
- [FU] FUJABA. <http://www.fujaba.de>.
- [GAADC02] Gueheneuc, Y., Albin-Amiot, H., Douence, R., und Cointe, P. Bridging the gap between modeling and programming languages. 2002.
- [GBHS97] Graham, I., Bischof, J., und Henderson-Sellars, B.: Associations considered a bad thing. *Journal of Object-Oriented Programming*. 9(9):41–48. Februar 1997.
- [GdCL03] Genova, G., del Castillo, C. R., und Llorens, J.: Mapping uml associations into java code. *Journal of Object Technology*. 2(5):135–162. September/Oktober 2003.
- [HBR00] Harrison, W., Barton, C., und Raghavchari, M.: Mapping uml designs to java. In: *Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'2000)*. volume 35. S. 178–187. Minneapolis, Minnesota, USA. Oktober 2000. ACM SIGPLAN Notices. ACM Press.

- [HS98] Henderson-Sellars, B.: Clarifying specialized forms of association in uml and oml. *Journal of Object-Oriented Programming*. 11(2):47–54. Februar 1998.
- [KR94] Kilov, H. und Ross, J.: *Information Modeling: An Object-Oriented Approach*. Prentice Hall. Januar 1994.
- [Ma03] Matula, M.: *NetBeans Metadata Repository*. SUN Microsystems. März 2003.
- [MM01] Miller, J. und Mukerji, J.: *Model Driven Architecture*. Object Management Group. Juli 2001. Document number ormsc/2001-07-01.
- [No96] Noble, J.: Some patterns for relationships. In: *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21)*. Melbourne, Australien. 1996. Prentice Hall.
- [No97] Noble, J.: Basic relationship patterns. In: *Proceedings of the European Conference on Pattern Languages of Program Design (EuroPLOP'97)*. Irsee, Deutschland. 1997.
- [Ob03a] Object Management Group: *Unified Modeling Language: Infrastructure, Version 2.0*. September 2003. ptc/03-09-15.
- [Ob03b] Object Management Group: *Unified Modeling Language, Version 1.5*. März 2003. formal/2003-03-01.
- [Ru87] Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*. S. 466–481. Orlando, Florida, USA. Oktober 1987. ACM Press.
- [Ru96] Rumbaugh, J.: Models for design: Generating code for associations. *Journal of Object-Oriented Programming*. 8(9):13–17. Februar 1996.
- [St] Stevens, P. On the interpretation of binary associations in the unified modelling language.