

**Prüfer:** Prof. Dr. rer. nat. Jochen Ludewig

**Betreuer:** Dipl. Inform. Ralf Reißing

**Beginn am:** 10.07.2001

**Beendet am:** 23.01.2002

**CR-Klassifikation:** D.1.5, D.2.2, D.2.8

Diplomarbeit Nr. 1952

**Konzeption und Realisierung  
eines Metrikenwerkzeugs für die  
Unified Modeling Language**

Christoph Schmider

Institut für Informatik  
Universität Stuttgart  
Breitwiesenstr. 20 - 22  
D-70565 Stuttgart

# Abkürzungen und Akronyme

CSV	Comma Seperated Values
DBMS	Datenbankmanagementsystem
MOF	Meta Object Facility
MOOSE	Metrikenwerkzeug für den ObjektOrientierten Entwurf
ODEM	Object-oriented DDesign Model
OMG	Object Management Group
QOOD	Quality of Object-Oriented Design
SQL	Structured Query Language
UML	Unified Modeling Language
XML	eXtensible Markup Language
XMI	XML Metadata Interchange
W3C	World Wide Web Consortium

# Metriken

DITC	depth of inheritance tree of a class
DNHP	depth in the nesting hierarchy of a package
NAC	number of attributes of a class
NACP	number of afferently coupled packages of a package
NADC	number of afferent dependencies of a class
NADP	number of afferent dependencies of a package
NAEC <sub>1</sub>	number of local afferent extends relationships of a class
NAS	number of attributes in the system
NCP	number of classes in a package
NCS	number of classes in the system
NECP	number of efferently coupled packages of a package
NEDC	number of efferent dependencies of a class
NEDP	number of efferent dependencies of a package
NIP	number of interfaces in a package
NIS	number of interfaces in the system
NOC	number of operations of a class
NOS	number of operations in the system
NPP	number of packages in a package
NPS	number of packages in the system

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b> .....	<b>6</b>
1.1	Aufgabenstellung .....	6
1.2	Aufbau der Arbeit .....	7
<b>2</b>	<b>Grundlagen</b> .....	<b>8</b>
2.1	Object-oriented DDesign Model (ODEM) .....	8
2.1.1	UML-Metamodell .....	8
2.1.2	Basiselemente .....	10
2.1.3	Zusätzliche Elemente .....	11
2.1.4	Metriken mit ODEM .....	12
2.2	Quality of Object-Oriented Design (QOOD) .....	14
2.2.1	Metriken in QOOD .....	14
2.3	XML Metadata Interchange (XMI) .....	15
2.3.1	eXtensible Markup Language (XML) .....	15
2.3.2	XMI und das UML-Metamodell .....	16
2.3.3	XMI in der Praxis .....	20
<b>3</b>	<b>Anforderungen</b> .....	<b>24</b>
3.1	Abbildung nach ODEM .....	24
3.1.1	Spezialfälle aufgrund von ODEM .....	24
3.1.2	Korrektur von Fehlern .....	26
3.2	Erhebung von Metriken .....	27
3.3	Präsentation der Messwerte .....	27
3.3.1	Allgemeine Anforderungen .....	27
3.3.2	Reports für QOOD .....	29
3.4	Nicht-funktionale Anforderungen .....	30
<b>4</b>	<b>Konzeption</b> .....	<b>33</b>
4.1	Konfiguration der Metriken .....	33
4.1.1	Der Plug-In-Ansatz .....	33
4.1.2	Der Datenbank-Ansatz .....	35
4.1.3	Vergleich der Alternativen .....	37
4.1.4	Formulierung der Anfragen .....	39
4.1.5	Eingabe der Funktionen .....	42
4.2	Abbildung nach ODEM .....	42
4.3	Ausgabe der Messergebnisse .....	45
4.4	Bedienung des Werkzeugs .....	47
<b>5</b>	<b>Realisierung</b> .....	<b>49</b>
5.1	Rahmenbedingungen .....	49
5.2	Systemüberblick .....	49
5.3	Der Parser .....	50
5.3.1	Auswahl der XML-Parserschnittstelle .....	50
5.3.2	Analyse der Daten .....	51
5.4	Die Filter .....	52

5.5	Die Datenbank .....	53
5.5.1	Speichern mehrerer Entwürfe .....	53
5.5.2	Zusätzliche Relationen .....	54
5.5.3	Aggregatbeziehungen .....	55
5.5.4	Behandlung von Sonderfällen .....	55
5.5.5	Der Datenbank-Adapter .....	55
5.6	Der Evaluator .....	56
5.7	Der Reportgenerator .....	56
5.8	Bewertung der Implementierung .....	58
5.8.1	Größe .....	58
5.8.2	Laufzeit .....	58
5.8.3	Speicherplatzbedarf .....	59
5.9	Erprobung mit QOOD .....	60
<b>6</b>	<b>Zusammenfassung und Ausblick.....</b>	<b>64</b>
6.1	Projektverlauf .....	64
6.1.1	Tätigkeiten .....	64
6.1.2	Termine und Aufwand .....	66
6.2	Ergebnisse .....	67
6.3	Ausblick .....	69
6.4	Fazit .....	70

# 1 Einleitung

Die Entwicklung von Software setzt sich aus mehreren Tätigkeiten zusammen. Eine wichtige Rolle spielt dabei der Entwurf. Er gibt die Struktur der Software vor. Die Qualität des Entwurfs wirkt sich auf die Kosten für die Realisierung und die Wartung der Software aus. Eine Änderung des Entwurfs während späterer Phasen ist sehr aufwendig. Es ist daher wichtig, von Anfang an eine hohe Entwurfsqualität zu haben. Für die Qualitätssicherung ist es notwendig, Entwurfsbewertungen durchzuführen.

Metriken können bei der Bewertung der Entwurfsqualität helfen. Bei großen Entwürfen ist es sehr aufwendig, Messungen "von Hand" vorzunehmen. Die Erhebung der Metriken durch ein Werkzeug kann daher die Effizienz des Bewertungsprozesses wesentlich steigern.

Werkzeuge zur Erhebung von Metriken auf Quellcode gibt es einige. Diese können jedoch erst eingesetzt werden, wenn zumindest eine rudimentäre Implementierung vorliegt. Dies ist in der Entwurfsphase jedoch in der Regel nicht der Fall. Der Entwurf wird normalerweise grafisch, in Form von Diagrammen dargestellt. In der objektorientierten Softwareentwicklung ist die UML (Unified Modeling Language) die am meisten verbreitete Notation für den Entwurf. Es gibt eine Vielzahl von Werkzeugen, die den Entwurf mit der UML unterstützen. Ein Austausch von Modellinformationen zwischen verschiedenen Werkzeugen wird durch XMI (XML Metadata Interchange) ermöglicht. XMI (OMG, 2000b) erzeugt eine textuelle Darstellung von UML-Modellen. Viele der UML-Werkzeuge können Modelle als XMI-Dokumente exportieren.<sup>1</sup> Die Suche nach einem Werkzeug, das Metriken auf einer Entwurfsbeschreibung in XMI erheben kann, gestaltet sich jedoch schwierig.

## 1.1 Aufgabenstellung

Im Rahmen dieser Arbeit soll ein Werkzeug zur Erhebung von Metriken auf UML-Entwurfsbeschreibungen konzipiert und prototypisch realisiert werden.

Das formale Modell ODEM (Reißing 2000b) ermöglicht die formale Definition von Metriken für UML-Modelle. Das Metrikenwerkzeug soll mit beliebigen Metriken, die auf der Basis von ODEM definiert sind, konfiguriert werden können. Es erhält als Eingabe eine UML-Entwurfsbeschreibung als XMI-Datei und führt darauf die gewünschten Messungen aus.

Zur Ausgabe sollen zwei Optionen zur Verfügung stehen. Zum einen sollen die Messergebnisse in ein HTML-Dokument eingebunden werden können. Zum anderen sollen die Ergebnisse so ausgegeben werden können, dass sie von einem Tabellenkalkulationsprogramm importiert werden können. In der Tabellenkalkulation können dann die Messwerte weiterverarbeitet werden, z.B. durch statistische Auswertungen. Außerdem kann die Diagrammfunktion der Tabellenkalkulation zur grafischen Darstellung der

---

1. Eine umfangreiche Liste mit UML-Werkzeugen, in der auch XMI-Unterstützung vermerkt ist, findet man z.B. unter <http://www.jeckle.de/umltools.htm>

Messwerte genutzt werden. Die Anordnung der Messergebnisse soll sowohl in den HTML-Reports als auch in der Tabellenkalkulation möglichst flexibel vorgenommen werden können.

Das Metrikenwerkzeug soll erprobt werden, indem es benutzt wird, um die Metriken für das Qualitätsmodell QOOD (Quality of Object-Oriented Design, Reißing 2001a) zu erheben.

## **1.2 Aufbau der Arbeit**

Zur Lösung der Aufgabe wurde im Rahmen dieser Arbeit das Werkzeug MOOSE (Metrikenwerkzeug für den ObjektOrientierten SoftwareEntwurf) entwickelt. Dieser Bericht beschreibt die Konzeption und die Realisierung des Werkzeugs. Kapitel 2 vermittelt einige Grundlagenkenntnisse über ODEM, QOOD und XMI. In Kapitel 3 werden die Anforderungen an MOOSE aufgeführt. Kapitel 4 befasst sich mit der Konzeption des Werkzeugs MOOSE. Kapitel 5 beschreibt die Realisierung des Werkzeugs. In Kapitel 6 folgt schließlich ein Rückblick auf den Projektverlauf, eine Bewertung der Ergebnisse und ein Ausblick auf die mögliche Weiterentwicklung des Metrikenwerkzeugs.

## 2 Grundlagen

Hier werden einige Grundlagenkenntnisse zum besseren Verständnis der Aufgabe und ihrer Lösung vermittelt. Kapitel 2.1 beschreibt das formale Modell ODEM. In Kapitel 2.2 wird eine Übersicht über QOOD gegeben. In Kapitel 2.3 wird die Darstellung von UML-Modellen als XMI-Dokumente erläutert. Dabei wird auch auf Fehler eingegangen, die bei der XMI-Generierung durch bestimmte UML-Werkzeuge auftreten.

### 2.1 Object-oriented Design Model (ODEM)

ODEM ist ein formales Modell für den objektorientierten Entwurf. Es wurde konzipiert, um eine möglichst einfache formale Definition von Metriken zu ermöglichen. ODEM basiert auf dem UML-Metamodell. Das UML-Metamodell ist Teil der UML-Spezifikation (OMG, 2000c) und dient zur formalen Definition der UML. Das UML-Metamodell ist sehr komplex. Metrikendefinitionen auf der Basis des UML-Metamodells sind deshalb schwer zu formulieren und zu verstehen. Um die Definition der Metriken zu erleichtern, wird in ODEM eine zusätzliche Abstraktionsschicht eingeführt. Diese besteht aus Mengen, Relationen und Attributen.

ODEM deckt nur einen Teil des UML-Metamodells ab. Es beschränkt sich auf statische Struktur eines Modells. Aber auch hier werden Modellelemente oder Teile von Modellelementen weggelassen, um das Modell möglichst einfach zu halten. Weggelassen werden z.B. Templates, Subsysteme und Multiplizitäten von Attributen und Assoziationen.

In Kapitel 2.1.1 wird der von ODEM abgedeckte Ausschnitt des UML-Metamodells beschrieben. In den Kapiteln 2.1.2 und 2.1.3 werden die Basiselemente und zusätzliche Elemente von ODEM vorgestellt. Kapitel 2.1.4 enthält einige Beispiele für die Definition von Metriken mit ODEM.

#### 2.1.1 UML-Metamodell

Zur Beschreibung des UML-Metamodells werden UML-Klassendiagramme verwendet. Abbildung 1 zeigt das UML-Klassendiagramm für die Modellelemente in ODEM. *ModelElement* ist Oberklasse aller Modellelemente. Jedes *ModelElement* hat das Attribut *name*. Stereotypen dienen dazu, Modellelemente mit einer bestimmten Bedeutung zu versehen. Mit Stereotypen können Instanzen einer Klasse des Metamodells noch feiner klassifiziert werden.

*NameSpace* dient zur hierarchischen Strukturierung des Modells. Ein Namensraum (*NameSpace*) kann beliebige andere Modellelemente enthalten. Dies wird dargestellt durch eine Komposition zwischen *NameSpace* und *ModelElement*. Das Attribut *visibility* der Assoziationsklasse *ElementOwnership* legt dabei die Sichtbarkeit des Modellelements außerhalb des Namensraums fest. Mögliche Werte sind 'public', 'protected' und 'private'.



Pakete sind Unterklassen von *Namespace* und können daher andere Modellelemente enthalten. Ein Modell ist ein spezielles Paket, das alle anderen Modellelemente mittelbar oder unmittelbar enthält.

*Class*, *Interface* und *DataType* sind Unterklassen von *Classifier*. *Classifier* enthalten *Features*. *Classifier* ist eine Unterklasse von *GeneralizableElement*. *Classifier* können daher *Features* von anderen *Classifiern* erben. Die Klasse *Classifier* erbt von *GeneralizableElement* das Attribut *isAbstract*. Das Attribut *isAbstract* gibt an, ob ein Modellelement instanziiert werden kann.

Attribute und Operationen sind *Features*. Sie erben daher die Attribute *ownerScope* und *visibility*. Mögliche Werte für *ownerScope* sind 'classifier' für Klassenattribute (bzw. -operationen) und 'instance' für Objektattribute (bzw. -operationen).

*Attribute* ist Subklasse von *StructuralFeature*. Attribute haben daher eine Assoziation zu *Classifier*, durch die der Typ eines Attributs festgelegt wird.

Operationen können *Parameter* haben. Das Attribut *kind* der Klasse *Parameter* legt fest, ob es sich um einen Ein- oder Ausgabeparameter handelt. Mögliche Werte sind 'in', 'inout', 'out'. Zusätzlich gibt es den Wert 'return' zur Kennzeichnung von Rückgabeparametern.

Auch Beziehungen werden im UML-Metamodell als Klassen modelliert (vgl. Abbildung 2). Oberklasse aller Beziehungen ist *Relationship*, die selbst wiederum Unterklasse von *ModelElement* ist. *Relationship* hat die Unterklassen *Generalization*, *Association* und *Dependency*. *Usage* und *Abstraction* sind Unterklassen von *Dependency*.

Ein Modellelement hat in einer Beziehung eine von zwei möglichen Rollen (z.B. *client* oder *supplier* bei einer *Dependency*). Eine Ausnahme sind hierbei Assoziationen. An einer Assoziation können beliebig viele Modellelemente beteiligt sein. Diese können dabei unterschiedliche Rollen einnehmen. Zur Modellierung von Assoziationen wird deshalb die zusätzliche Klasse *AssociationEnd* verwendet. Die Attribute von *AssociationEnd* legen die Rolle eines Modellelements in der Assoziation fest. Das Attribut *isNavigable* gibt an, ob ein Element von anderen Beteiligten der Assoziation aus erreichbar ist. *Aggregation* gibt an, ob das Element die anderen an der Assoziation beteiligten Elemente enthält. Mögliche Werte sind 'none', 'aggregate' und 'composite'. Die stärkste Aggregationsart ist 'composite'. Bei 'composite' gehören die Teile genau einem Element. Bei 'aggregate' kann der Teil auch noch in anderen Elementen enthalten sein. Bei 'none' handelt es sich nicht um eine Aggregation. Das Attribut *name* (geerbt von *ModelElement*) enthält den Rollennamen des Modellelements, mit dem das Assoziationsende verbunden ist.

## 2.1.2 Basiselemente

Abbildung 3 zeigt die Basiselemente von ODEM. Modellelemente in ODEM sind Pakete, Klassen, Interfaces, Attribute, Operationen und Parameter.

Die Attribute der Modellelemente sind aus dem UML-Metamodell übernommen. Die Modellelemente *Attribute* und *Parameter* erhalten zusätzlich das Attribut *type*. Dieses ist aus ihren Assoziationen zu *Classifier* im UML-Metamodell hergeleitet.

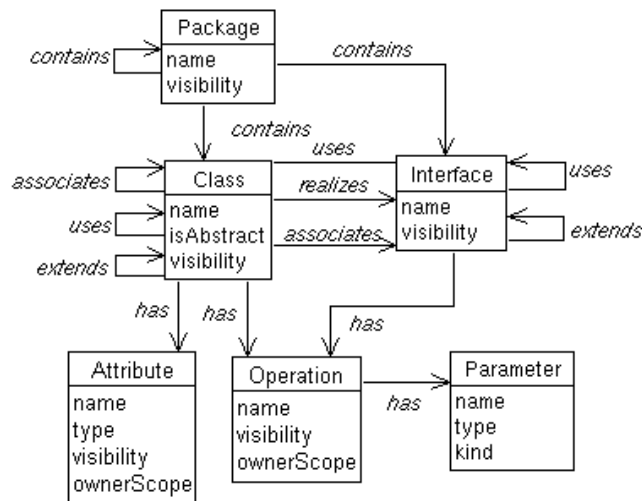


Abbildung 3: Basiselemente von ODEM

Im UML-Metamodell besitzt die Komposition von Modellelementen in ihrem Namensraum das Attribut *visibility*. In ODEM geht dieses Attribut auf die Elemente Paket, Klasse und Interface über. Pakete können Pakete, Klassen und Interfaces enthalten (*contains*). Klassen und Interfaces haben Operationen. Klassen haben zusätzlich Attribute. Operationen haben Parameter (*has*).

Vererbung zwischen Klassen und zwischen Interfaces wird durch die *extends*-Relation ausgedrückt. Diese ergibt sich aus *Generalizations* im UML-Metamodell. Die *uses*-Relation zwischen Klassen/Interfaces ergibt sich aus *Usage*-Beziehungen im UML-Metamodell. Die *realizes*-Relation modelliert die Realisierung von Klassen durch Interfaces. Im UML-Metamodell wird dies durch *Abstractions* mit Stereotyp «realize» dargestellt. Klassen können mit anderen Klassen oder mit Interfaces assoziiert sein. In die *associates*-Relation von ODEM werden nur navigierbare Assoziationen übernommen.

### 2.1.3 Zusätzliche Elemente

Als zusätzliche Elemente gibt es in ODEM abgeleitete Relationen und erweiterte Relationen. Außerdem haben Relationen auch Attribute.

#### Abgeleitete Relationen

Allgemeine Abhängigkeiten werden in ODEM durch die Relation *depends\_on* modelliert, die von den Relationen *associates*, *extends*, *realizes* und *uses* abgeleitet ist:

$$depends\_on(x,y) \Leftrightarrow associates(x,y) \vee extends(x,y) \vee realizes(x,y) \vee uses(x,y)$$

Zusätzlich gibt es eine Relation *depends\_on* für die Abhängigkeit zwischen Paketen:

$$depends\_on(p,q) \Leftrightarrow \exists c,c' \in C \cup I: contains(p,c) \wedge contains(q,c') \wedge depends\_on(c,c')$$

## Erweiterte Relationen

Bei den bisher vorgestellten Relationen waren geerbte Beziehungen von Modellelementen nicht enthalten. Bei den erweiterten Relationen wird auch die Vererbung berücksichtigt. Hierfür wird *extends\** als transitive Hülle der *extends*-Relation definiert. In den Relationen *associates\**, *has\**, *realizes\** und *uses\** sind auch geerbte Beziehungen enthalten.

Die *contains*-Relation erfasst nur die unmittelbar in einem Paket enthaltenen Modellelemente. Um eine Relation zu haben, bei der auch mittelbar in einem Paket enthaltene Modellelemente erfasst werden, wird die Relation *contains\** als die transitive Hülle von *contains* definiert.

## Attribute von Relationen

Bisher geben die Relationen nur Auskunft darüber, ob Beziehungen eines Typs zwischen zwei Modellelementen bestehen. Es ist jedoch keine Aussage über die Anzahl der Beziehungen möglich. Alle Relationen in ODEM erhalten deshalb ein Gewicht (*weight*) als Attribut. Das Gewicht einer Relation gibt die Anzahl der Beziehungen an.

Als einzige Relation besitzt *associates* noch ein weiteres Attribut. Das Attribut *aggregation* gibt die Aggregationsart an. Gibt es mehrere Assoziationen mit unterschiedlicher Aggregation zwischen zwei Modellelementen, so wird der Wert des Attributs durch die stärkste Aggregationsart bestimmt.

### 2.1.4 Metriken mit ODEM

Tabelle 1 enthält eine Übersicht der formalen Bezeichner der Elemente in ODEM mit ihrer Bedeutung.

Mit diesen Bezeichnern können System-, Paket- und Klassenmetriken formuliert werden. Im Folgenden einige Beispiele aus QOOD:

- Durch die Metrik NCS (number of classes in the system) wird die Anzahl der Klassen im gesamten System ermittelt:  

$$\text{NCS}(S) = |C|$$
- Man kann jedoch zwischen konkreten und abstrakten Klassen unterscheiden. Die Anzahl der konkreten Klassen, die unmittelbar in einem Paket enthalten sind, erhält man mit der Metrik  $\text{NCP}_c$  (number of concrete classes in a package):  

$$\text{NCP}_c(p) = |\{c \in C: \text{contains}(p,c) \wedge \neg c.\text{isAbstract}\}|$$
- Die Metrik NAC (number of attributes of a class) dient zum Zählen der Attribute einer Klasse. Durch die Verwendung von *has\** werden auch die geerbten Attribute mitgezählt:  

$$\text{NAC}(c) = |\{a \in A: \text{has}^*(c,a)\}|$$
- Will man wissen, wie stark eine Klasse von anderen Klassen abhängt, so kann man die von ihr ausgehenden (efferenten) Beziehungen messen. Bei der Metrik NEDC (number of efferent dependencies of a class) werden die Gewichte addiert und somit auch die Anzahl der Beziehungen zwischen zwei Klassen berücksichtigt:  

$$\text{NEDC}(c) = \sum_{c' \in C \cup I} \text{depends\_on}^*(c,c').\text{weight}$$

Bezeichner	Bedeutung
A	Menge aller Attribute in S
C	Menge aller Klassen in S
I	Menge aller Interfaces in S
M	Menge aller Parameter in S
O	Menge aller Operationen in S
P	Menge aller Pakete in S (inklusive S)
S	Das System (Paket, das alle anderen Modellelemente enthält)
associates	Relation für Assoziationen zwischen Klassen/Interfaces
associates*	Erweiterte <i>associates</i> -Relation (enthält auch geerbte Assoziationen)
contains	Relation für die Aggregation von Modellelementen in Paketen
contains*	Erweiterte <i>contains</i> -Relation (Transitive Hülle von contains)
depends_on	Aggregierte Relation für alle Abhängigkeiten zwischen Klassen/Interfaces oder zwischen Paketen.
depends_on*	Erweiterte <i>depends_on</i> -Relation (inklusive geerbter Abhängigkeiten)
extends	Relation für die Vererbung von Features
extends*	Erweiterte Vererbungsrelation (Transitive Hülle von extends)
has	Relation für die Aggregation von Features in Klassen/Interfaces
has*	Erweiterte <i>has</i> -Relation (inklusive geerbter Features)
realizes	Relation für die Realisierung eines Interface durch eine Klasse
realizes*	Erweiterte <i>realizes</i> -Relation (inklusive geerbten Beziehungen)
uses	Relation für Benutzungsbeziehungen zwischen Klassen/Interfaces
uses*	Erweiterte <i>uses</i> -Relation (inklusive geerbten Beziehungen)

Tabelle 1: Formale Bezeichner in ODEM

- Die Metrik NADC (number of afferent dependencies of a class) ermittelt hingegen die in eine Klasse hineingehenden (affarenten) Beziehungen.  

$$NADC(c) = \sum_{c' \in C \cup I} depends\_on^*(c', c).weight$$

## 2.2 Quality of Object-Oriented Design (QOOD)

Metriken können bei der Bewertung eines Entwurfs helfen. Bloße Messwerte für sich allein geben aber noch keinen Aufschluss über die Qualität eines Entwurfs. Um den Messwerten eine Bedeutung zu geben, müssen sie im Zusammenhang mit einem Qualitätsmodell betrachtet werden. Das Qualitätsmodell bestimmt, wie die Messwerte im Bezug auf die Entwurfsqualität zu interpretieren sind, und legt somit den Qualitätsbegriff fest (Reiing, 2001a). Fur die Bewertung von objektorientierten Entwurfen wurde von Reiing das Qualitätsmodell QOOD entwickelt. Die Qualitt eines Entwurfs wird dabei an der Qualitt der daraus entstehenden Software festgemacht. Die Qualitt der Software wird durch verschiedene Faktoren wie z.B. Wartbarkeit, Wiederverwendbarkeit und Brauchbarkeit bestimmt. In QOOD werden Entwurfskriterien identifiziert, die sich positiv auf die Faktoren der Softwarequalitt auswirken. Fur den Faktor Wartbarkeit sind dies z.B. die Kriterien Knappheit, Strukturiertheit, Entkopplung, Zusammenhalt, Einheitlichkeit, Dokumentierung und Verfolgbarkeit.

Viele dieser Kriterien (z.B. Verstndlichkeit) sind schwer an messbaren Groen festzumachen. Hier muss eine subjektive Wertung vorgenommen werden. Diese Bewertung wird in QOOD anhand von Fragebogen vorgenommen.

Andere Kriterien konnen mit Hilfe von messbaren Werten (objektiven Metriken) bestimmt werden. In QOOD sind das die Kriterien Knappheit, Strukturiertheit und Entkopplung des Faktors Wartbarkeit. Allerdings erfassen solche Metriken nicht unbedingt alle Aspekte eines Kriteriums. Die abschlieende Wertung fur diese Kriterien erfolgt in QOOD daher ebenfalls durch einen Bewerter. Dieser wird dabei durch die Messergebnisse unterstutzt.

### 2.2.1 Metriken in QOOD

Alle objektiven Metriken in QOOD sind auf der Basis von ODEM definiert. Die Metriken in QOOD sind klassifiziert nach ihrem Bereich. Es gibt die Bereiche System, Paket und Klasse, wobei Klassenmetriken sowohl fur Klassen als auch fur Interfaces erhoben werden.

QOOD bietet die Auswahl zwischen unterschiedlich detaillierten Metriken. In manchen Fallen konnen einfache Metriken wie z.B. NAC (number of attributes of a class) ausreichen. Bei Bedarf konnen die Messwerte nach bestimmten Gesichtspunkten verfeinert werden. So kann man zwischen geerbten und lokal in einer Klasse definierten Attributen unterscheiden. Auch eine Unterscheidung nach dem Sichtbarkeitsbereich ist moglich. Selbstverstndlich konnen alle moglichen Verfeinerungen auch kombiniert werden.

Die folgenden drei Abschnitte geben einen berblick daruber, was bei QOOD gemessen wird und welche Verfeinerungen dabei moglich sind.

#### **Knappheit (*Conciseness*)**

Auf Klassenebene werden die Anzahl der Attribute und Operationen ermittelt. Bei Paketen werden enthaltene Klassen, Interfaces und Pakete gezhlt. Bei Systemmetriken wird die Anzahl der verschiedenen Modellelemente im gesamten System bestimmt. Als Ver-

feinerung kann bei allen Modellelementen der Sichtbarkeitsbereich betrachtet werden. Attribute und Operationen können zusätzlich danach unterschieden werden, ob sie geerbt oder lokal definiert sind und ob sie für jedes Objekt (Objektattribut bzw. -operation) oder nur einmal für die Klasse (Klassenattribut bzw. -operation) existieren. Klassen werden in konkrete und abstrakte Klassen unterteilt.

### **Strukturiertheit (*Structuredness*)**

Hier werden Vererbungshierarchie und die Schachtelungshierarchie von Paketen betrachtet. Für Klassen wird die Tiefe im Vererbungsbaum und die Anzahl der Kindklassen ermittelt. Bei Paketen misst man die Schachtelungstiefe und die Anzahl der enthaltenen Pakete. Für die Metriken zum Kriterium Strukturiertheit gibt es in QOOD keine Verfeinerungen.

### **Entkopplung (*Decoupling*)**

Es werden für Klassen und Pakete die ausgehenden (efferenten) und hineingehenden (afferenten) Abhängigkeiten ermittelt. Hierfür werden die Gewichte der Relationen addiert. Bei Paketen wird zusätzlich die Anzahl der abhängigen Pakete (bzw. der Pakete zu denen Abhängigkeiten bestehen) ermittelt. Im einfachsten Fall wird nicht nach Art der Abhängigkeit unterschieden. Bei Abhängigkeiten zwischen Klassen können als Verfeinerung die verschiedenen Arten der Beziehungen einzeln betrachtet werden. Bei Assoziationen wird hierbei noch nach Aggregationsart unterschieden. Außerdem kann unterschieden werden zwischen Abhängigkeiten von abstrakten und konkreten Klassen, lokal definierten und geerbten, reflexiven und nicht reflexiven, paketinternen und -externen Abhängigkeiten. Bei Abhängigkeiten zwischen Paketen werden keine Verfeinerungen betrachtet.

## **2.3 XML Metadata Interchange (XMI)**

Die UML ist eine grafische Notation. Diese ist zwar für den Menschen interpretierbar, für den Rechner enthält ein Diagramm jedoch nur die Bildinformation. Modellinformationen werden intern anders dargestellt. Um einen Austausch von UML-Modellen zwischen verschiedenen Werkzeugen zu ermöglichen, wird eine standardisierte, maschinenlesbare Darstellung der Modellinformationen benötigt. XMI bietet hier eine Lösung auf der Basis der *eXtensible Markup Language XML* (W3C, 2000b).

Das folgende Kapitel soll die für das Verständnis von XMI notwendigen Kenntnisse über XML liefern. In Kapitel 2.3.2 wird erläutert, was der XMI-Standard eigentlich leistet. Während dieses Projekts wurden verschiedenen Werkzeuge zur Generierung von XMI verwendet. In Kapitel 2.3.3 werden Fehler aufgezeigt, die bei der Generierung von XMI durch diese Werkzeuge auftraten.

### **2.3.1 eXtensible Markup Language (XML)**

XML ist "eine Metasprache zur Beschreibung und Instanziierung von Auszeichnungssprachen für allgemeine Dokumente" (Duden, 2001). Wie bei HTML werden auch XML-Dokumente durch Anfang- und Ende-*Tags* in Bereiche, sogenannte Elemente, unterteilt.

Diese Elemente können auch Attribute besitzen. Ein Element kann einfachen Text und andere Elemente enthalten. Die Struktur von XML-Dokumenten kann durch *Document Type Declarations* (DTD) festgelegt werden. Mit Hilfe von DTDs ist es somit möglich, Auszeichnungssprachen wie z.B. HTML zu definieren.

Abbildung 4a zeigt ein XML-Dokument, das Informationen über ein Buch enthält. Abbildung 4b zeigt die dazu passende DTD. Ein Buch besteht aus Autor und Titel. Autor und Titel sind jeweils eine beliebige Zeichenfolge. Ein Buch besitzt als Attribut seine ISBN-Nummer.

```
<Buch ISBN = '0-330-25864-8'>
  <Titel>
    The Hitchhiker's Guide to the Galaxy
  </Titel>
  <Autor>
    Douglas Adams
  </Autor>
</Buch>
```

#### (a) Buch als XML Dokument

```
<!ELEMENT Buch (Titel, Autor)>
<!ELEMENT Titel (#PCDATA)>
<!ELEMENT Autor (#PCDATA)>
<!ATTLIST Buch ISBN CDATA>
```

#### (b) XML-DTD für Buch

### Abbildung 4: Beispiel XML

## 2.3.2 XMI und das UML-Metamodell

XMI ist ein Standard zum Austausch von Metadaten (Modellen) zwischen verschiedenen Anwendungen. Voraussetzung für die Anwendung von XMI ist, dass der Typ des Modells, also sein Metamodell, durch UML-Klassendiagramme dargestellt werden kann. Da das UML-Metamodell mit UML-Klassendiagrammen definiert wird, kann XMI für UML-Modelle verwendet werden. XMI kann aber auch für andere Modelle verwendet werden. So können beispielsweise auch Datenbank-Schemata mit XMI dargestellt werden (vgl. Brodsky, 1999).

Der XMI-Standard legt fest, wie aus einem mit der UML dargestellten Metamodell (z.B. dem UML-Metamodell) eine auf XML basierende Auszeichnungssprache herzuleiten ist. XMI besteht aus zwei Teilen. Zum einen wird festgelegt, wie aus der Beschreibung des Metamodells eine XML-DTD hergeleitet wird.<sup>1</sup> Hierzu sind Regeln festgelegt, wie die in der Beschreibung eines Metamodells vorkommenden Elemente der UML in Elemente einer XML-DTD zu überführen sind.

1. Diese DTD liefert jedoch keine Beschreibung des Metamodells. Sie liefert zwar eine Struktur, die es zulässt, alle dem Metamodell entsprechenden Modelle zu übertragen, sie lässt aber auch Dokumente zu, die Konsistenzbedingungen des Metamodells verletzen.

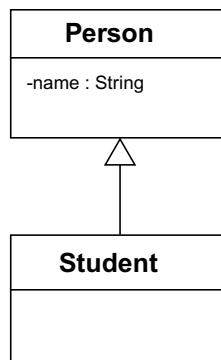
Zum anderen bestimmt XMI, wie eine Instanz des Metamodells, also ein Modell (z.B. ein UML-Modell), als XML-Dokument darzustellen ist. Dies geschieht durch Regeln, die festlegen, wie Instanzen der verschiedenen Elemente des Metamodells als Elemente eines XML-Dokuments dargestellt werden.

Die Diagramme, die zur Definition eines Metamodells verwendet werden, müssen bestimmte Voraussetzungen erfüllen, um für XMI geeignet zu sein.<sup>1</sup> So sind z.B. die zugelassenen Modellelemente auf Klassen und Pakete beschränkt. Als Beziehungen zwischen den Modellelementen sind Assoziationen und Generalisierungen zugelassen. Bei den Assoziationen darf es sich um normale Assoziationen oder Kompositionen handeln. Es sind z.B. keine Assoziationsklassen erlaubt. Außerdem müssen alle Assoziationsenden Rollennamen besitzen.

Nicht alle Klassendiagramme des UML-Metamodells erfüllen die Voraussetzungen für die Verwendung von XMI. Daher müssen leichte Veränderungen vorgenommen werden. In Abbildung 2.3, Seite 15 entfällt beispielsweise die Assoziationsklasse für *ownedElement*. Dafür erhält *ModelElement* das zusätzliche Attribut *visibility*. Die modifizierten Klassendiagramme sind, ebenso wie die daraus hergeleitete DTD, Teil der UML-Spezifikation.

### Darstellung eines UML-Modelles durch XMI

Die Darstellung eines UML-Modells in XMI wird hier anhand eines Beispiels erläutert. Abbildung 5 zeigt ein einfaches UML-Klassendiagramm. Die Klasse *Person* besitzt das private Attribut *name* mit Typ *String*. *Student* ist Unterklasse von *Person*.



**Abbildung 5: Beispiel für ein UML-Klassendiagramm**

Abbildung 6 zeigt das XMI-Dokument für das Modell. Die Zeilennummern darin sind nicht Teil von XMI, sondern wurden zur besseren Übersicht eingefügt. Es wurden alle Modellinformationen, die nicht durch ODEM abgedeckt werden, weggelassen. Die vollständige Beschreibung des dargestellten Modells in XMI ist ca. doppelt so lang. Die Dar-

1. Es sind gerade die Konstrukte erlaubt, die auch von der MOF (Meta Object Facility, OMG, 2000a) benutzt werden. Die MOF ist wie XMI und UML ein OMG-Standard und dient dazu, Metadaten zu definieren und als CORBA-Objekte darzustellen.

stellung des Modells in XMI erfolgt anhand des UML-Metamodells. Zum besseren Verständnis kann man die Klassendiagramme des UML-Metamodells (Abbildungen 1 und 2, Seite 9) heranziehen.

Die beiden ersten Zeilen des XMI-Dokuments enthalten allgemeine XML-Informationen. Sie geben den Zeichensatz und die DTD des Dokuments an.

Die Zeilen 3 bis 6 sind XMI-spezifisch. Hier wird die XMI-Version und der Name und die Version des Metamodells angegeben, dem die Daten entsprechen. Das XML-Element `XMI.content` (7-58) enthält die Modellbeschreibung.

Die eigentliche Beschreibung des Modells beginnt in Zeile 8 und endet in Zeile 57. Wie jedes Modellelement erhält auch das Modell selbst eine XMI-ID (`xmi.id = 'txmiid2'`). Diese dient als im ganzen Dokument eindeutiger Bezeichner eines Modellelements und wird für Referenzen auf das Modellelement benutzt. Die Paketstruktur des Metamodells spiegelt sich in den Namen der *Tags* wieder. Die Klasse *Model* ist im UML-Metamodell Teil des Pakets *Model\_Management*<sup>1</sup>. Deshalb hat das Modell im XMI-Dokument den *Tag*-Namen `'Model_Management.Model'`.

Jedes Modellelement ist Instanz einer Klasse des UML-Metamodells. Eine Klasse des UML-Metamodells (auch Metaklasse genannt) kann Attribute besitzen und an Assoziationen beteiligt sein. "Teil von"-Beziehungen von Modellelementen werden im Metamodell durch Kompositionen dargestellt. Jedes Modellelement ist Instanz einer Metaklasse. Bei einer Instanz werden die Attribute mit Werten versehen, aus Assoziationen zwischen Metaklassen werden Beziehungen zwischen Modellelementen. Handelt es sich bei einer Assoziation um eine Komposition, so wird aus ihr eine "Teil-von"-Beziehung. Für jedes Modellelement werden in der XMI-Datei zuerst die Werte der Attribute seiner Metaklasse angegeben. Anschließend folgen seine Beziehungen und zuletzt die in ihm enthaltenen Modellelemente.

Die Metaklasse *Model* hat das Attribut *name*. Der Name des Modells ('Beispiel', Zeile 9) ist im Diagramm nicht zu sehen. Da die Assoziationsklasse *ElementOwnership* für XMI entfernt wird und das Attribut *visibility* auf *ModelElement* übergeht, hat auch das Modell einen Sichtbarkeitsbereich (Zeile 10). Das Modell hat keine Beziehungen. Da *Model* Unterklasse der Metaklasse *Namespace* ist, enthält es andere Modellelemente. Die Auflistung der enthaltenen Modellelemente beginnt mit dem *Tag* `<Foundation.Core.Namespace.ownedElement>`. `'Foundation.Core.Namespace'` ist der vollständig qualifizierte Name der Metaklasse, von der die Komposition geerbt wird. `'ownedElement'` ist der Rollenname von *ModelElement* bei der Komposition zwischen *Namespace* und *ModelElement*.

Im Modell enthalten ist die Klasse *Person* (Zeilen 12-29). Die Klasse *Person* ist eine Instanz der Metaklasse *Class*. *Class* hat die Attribute *name*, *visibility* (von *ModelElement*) und *isAbstract* (von *GeneralizableElement*) (13-15). Attribute von einem Aufzählungstyp werden als leere XML-Elemente dargestellt (z.B. *visibility* mit den möglichen Werten 'public', 'protected' und 'private', Zeile 14). Hierbei wird der Attributwert nicht als

---

1. Der eigentliche Name im UML-Metamodell 'Model Mngement' beinhaltet ein Leerzeichen. Da dies für Elementnamen in XML nicht erlaubt ist, werden in der für XMI angepassten Variante des UML-Metamodells alle Leerzeichen in Namen durch Unterstriche ersetzt.

```

1  <?xml version = '1.0' encoding = 'ISO-8859-1' ?>
2  <!DOCTYPE XMI SYSTEM 'UMLX13.dtd' >
3  <XMI xmi.version = '1.0'>
4  <XMI.header>
5  <XMI.metamodel xmi.name = 'UML' xmi.version = '1.3' />
6  </XMI.header>
7  <XMI.content>
8  <Model_Management.Model xmi.id = 'txmiid1' >
9  <Foundation.Core.ModelElement.name>Beispiel</Foundation.Core.ModelElement.name>
10 <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
11 <Foundation.Core.Namespace.ownedElement>
12 <Foundation.Core.Class xmi.id = 'txmiid2' >
13 <Foundation.Core.ModelElement.name>Person</Foundation.Core.ModelElement.name>
14 <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
15 <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false' />
16 <Foundation.Core.ModelElement.namespace>
17 <Model_Management.Model xmi.idref = 'txmiid1' />
18 </Foundation.Core.ModelElement.namespace>
19 <Foundation.Core.Classifier.feature>
20 <Foundation.Core.Attribute xmi.id = 'txmiid3' >
21 <Foundation.Core.ModelElement.name>name</Foundation.Core.ModelElement.name>
22 <Foundation.Core.ModelElement.visibility xmi.value = 'private' />
23 <Foundation.Core.Feature.ownerScope xmi.value = 'classifier' />
24 <Foundation.Core.StructuralFeature.type>
25 <Foundation.Core.DataType xmi.idref = 'txmiid4' />
26 </Foundation.Core.StructuralFeature.type>
27 </Foundation.Core.Attribute>
28 </Foundation.Core.Classifier.feature>
29 </Foundation.Core.Class>
30 <Foundation.Core.Class xmi.id = 'txmiid5' >
31 <Foundation.Core.ModelElement.name>
32 Student
33 </Foundation.Core.ModelElement.name>
34 <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
35 <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false' />
36 <Foundation.Core.ModelElement.namespace>
37 <Model_Management.Model xmi.idref = 'txmiid1' />
38 </Foundation.Core.ModelElement.namespace>
39 <Foundation.Core.GeneralizableElement.generalization>
40 <Foundation.Core.Generalization xmi.idref = 'StudentPerson' />
41 </Foundation.Core.GeneralizableElement.generalization>
42 </Foundation.Core.Class>
43 <Foundation.Core.Generalization xmi.id = 'StudentPerson' >
44 <Foundation.Core.ModelElement.name></Foundation.Core.ModelElement.name>
45 <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
46 <Foundation.Core.Generalization.child>
47 <Foundation.Core.Class xmi.idref = 'txmiid5' />
48 </Foundation.Core.Generalization.child>
49 <Foundation.Core.Generalization.parent>
50 <Foundation.Core.Class xmi.idref = 'txmiid2' />
51 </Foundation.Core.Generalization.parent>
52 </Foundation.Core.Generalization>
53 <Foundation.Core.DataType xmi.id = 'txmiid4' >
54 <Foundation.Core.ModelElement.name>String</Foundation.Core.ModelElement.name>
55 </Foundation.Core.DataType>
56 </Foundation.Core.Namespace.ownedElement>
57 </Model_Management.Model xmi.id = 'txmiid1' >
58 </XMI.content>
59 </XMI>

```

Abbildung 6: Darstellung eines UML-Modells in XMI

Inhalt, sondern als Attribut des Elements dargestellt. Leere Elemente können in XML als Einfache *Tags*, die am Ende mit einem Schrägstrich markiert sind, dargestellt werden.

Person ist Teil des Modells (16-18). In XMI entspricht eine Komposition im Metamodell aus Sicht des enthaltenen Modellelements einer normalen Assoziation. Diese wird dargestellt durch ein XML-Element. Der Name des Elements ist der Rollenname der assoziierten Metaklasse (`<Foundation.Core.ModelElement.namespace>`). In dem XML-Element steht eine XMI-Referenz (`<Model_Management.Model xmi.idref = 'txmiid1'/>`). Eine XMI-Referenz ist ein leeres XML-Element. Der *Tag*-Name entspricht dem Namen der Metaklasse des Modellelements, auf das referenziert wird. Als XML-Attribut enthält die Referenz die XMI-ID des Modellelements, auf das sie verweist.

Außerdem wird von Person die Generalisierungsbeziehung zu Student referenziert (19-21). Die Generalisierung zwischen Person und Student ist selbst ein Modellelement (43-52).

Anschließend folgen die enthaltenen Modellelemente von Person. Die Klasse Person enthält das Attribut Name (18-25). Weitere Modellelemente in *Beispiel* sind die Klasse Student (30-42) und der Datentyp String (53-55).

### 2.3.3 XMI in der Praxis

In diesem Projekt wurden die folgenden UML-Werkzeuge mit XMI-Unterstützung zur Erprobung und zum Test des Metrikenwerkzeugs verwendet:

- Together ControlCenter<sup>1</sup> (Version 5.02)
- ArgoUML<sup>2</sup> (Version 0.8)
- Poseidon Community Edition<sup>3</sup> (Version 1.0)

*ArgoUML* und *Poseidon* exportieren nicht nur XMI, sondern benutzen dieses Format zur Speicherung von Modellen. *Poseidon* ist eine kommerzielle Weiterentwicklung des Open Source Werkzeugs *ArgoUML* und diesem daher sehr ähnlich. Aussagen, die im Folgenden über *ArgoUML* gemacht werden, gelten auch für *Poseidon*, wenn nicht explizit auf Unterschiede hingewiesen wird. Bei der Generierung von XMI durch diese Werkzeuge fielen einige Fehler und Probleme auf. Diese sind auf unterschiedliche Ursachen zurückzuführen:

- Für die Arbeit mit UML-Diagrammen kann von Details des UML-Metamodells abstrahiert werden. Bei einem XMI-Export muss jedoch das vollständige UML-Metamodell zugrundegelegt werden. Es ist davon auszugehen, dass diese Fehler auch bei anderen CASE-Tools auftreten.
- Alle drei verwendeten Werkzeuge sind in der Lage, Java-Quelltext aus Klassendiagrammen zu erzeugen. Die Werkzeuge halten sich daher nicht genau an das UML-Metamodell, sondern erlauben einige Java-spezifische Ausnahmen.
- Bei *Together* fiel eine Reihe weiterer Fehler auf.

---

1. <http://www.togethersoft.com>

2. <http://argouml.tigris.org>

3. <http://www.gentleware.com>



- Bei *ArgoUML* wird die spezielle Notation der Beziehung nicht benutzt. Im Diagramm und in der XMI-Datei steht eine *Dependency* mit Stereotyp «realize».
- *Poseidon* stellt Realisierungsbeziehungen auch in der XMI-Datei korrekt als *Abstraction* dar.

### Probleme durch Java

Alle drei verwendeten Tools sind in der Lage, aus UML-Diagrammen Java Quelltext zu generieren. Sie benutzen daher eine an Java angepasste Variante der UML. Dies führt zu Abweichungen vom UML-Metamodells oder zumindest zu Problemen bei der Auswertung Messergebnisse. Aufgefallen sind folgende Punkte:

- In Java kennzeichnet das Schlüsselwort *void*, wie in C, Operationen ohne Rückgabeparameter. Alle drei Werkzeuge behandeln *void* jedoch wie einen normalen Datentyp. Dies ist im Sinne der UML kein Fehler, da *void* in der UML keine besondere Bedeutung hat. Will man jedoch Operationen mit und ohne Rückgabewert unterscheiden, so muss dieser Umstand beachtet werden. Nur *Poseidon* bietet die Möglichkeit, Rückgabeparameter explizit zu löschen.
- Im UML-Metamodell haben Pakete, Klassen, Interfaces, Operationen und Attribute einen Sichtbarkeitsbereich. Für diesen gibt es mit abnehmender Sichtbarkeit die Werte 'public', 'protected' und 'private'. Bei Java kann bei Paketen kein Sichtbarkeitsbereich angegeben werden. Für Klassen gibt es die Sichtbarkeit 'public'. Wird für eine Klasse keine Sichtbarkeit angegeben, so ist sie nur innerhalb des Pakets sichtbar. Für Eigenschaften (Attribute und Operationen) gibt es die drei Sichtbarkeitsbereiche aus der UML. Wird kein Sichtbarkeitsbereich angegeben, so ist die Eigenschaft innerhalb des Pakets sichtbar. Die drei UML-Werkzeuge gehen beim XMI-Export unterschiedlich mit nicht angegebenen Sichtbarkeitsbereichen um. Bei *ArgoUML* und *Poseidon* fehlt in solchen Fällen das Attribut *visibility*. Dies ist eine Abweichung vom UML-Metamodell. Bei *Poseidon* muss allerdings für Eigenschaften ein Sichtbarkeitsbereich angegeben werden. *Together* setzt bei Klassen, bei denen keine Wert angegeben wird, 'private' als *visibility* ein. Dies entspricht auch der tatsächlichen Sichtbarkeit von Java-Klassen innerhalb eines Pakets. Bei Paketen wird aber ebenfalls 'private' eingesetzt. Da Pakete in Java aber für alle anderen Pakete sichtbar sind, ist dies ein Fehler. Bei Eigenschaften ist die Sichtbarkeit im Paket in Java zwischen 'protected' und 'private' einzuordnen.<sup>1</sup> Bei *Together* wird dafür jedoch 'public' angegeben.
- In Java gibt es die Möglichkeit, geerbte Attribute zu verstecken, indem ein neues Attribut mit gleichem Namen definiert wird. Die Werkzeuge erlauben daher eine Redefinition von geerbten Attributen. In der UML ist eine Redefinition von Attributen nicht erlaubt.

---

1. In Java sind Features bei Angabe von 'protected' innerhalb des Pakets und in allen Unterklassen sichtbar (vgl. Gosling et al., 2000).

## Fehler bei Together

- In einer von *Together* generierten XMI-Datei ist der Wert von *ownerScope* für Attribute immer 'classifier' und bei Operationen immer 'instance'. Die Werte sind daher unbrauchbar.
- Assoziationen sind bei *Together* standardmäßig in beide Richtungen nicht navigierbar. Es kann jedoch eingestellt werden, dass alle Assoziationen als Pfeil (und somit als navigierbar) dargestellt werden. Bei einem XMI-Export bleibt jedoch weiterhin die Standardeinstellung gültig. In diesem Fall stimmt das Diagramm nicht mit der XMI-Datei überein. Erst wenn die Navigierbarkeit für jede Assoziation explizit angegeben wird, wird dies auch in der XMI-Datei vermerkt. Eine Möglichkeit, in beide Richtungen navigierbare Assoziationen zu definieren, gibt es bei *Together* nicht.
- Parameter (außer return-Parameter) haben für *kind* stets den Wert 'inout'. Aus UML-Sicht ist dies falsch, da der Standardwert für *kind* 'in' ist und in den Diagrammen keine andere Angabe gemacht wird. Die Parameterübergabe erfolgt in Java stets als *call-by-value*. Auch aus dieser Sicht muss hier zumindest bei den primitiven Datentypen 'in' stehen. Auch bei Referenztypen<sup>1</sup> ist 'inout' als Standard zumindest eine unglückliche Wahl.
- Zur Implementierung von Assoziationen werden durch *Together* automatisch Attribute generiert. Diese werden auch in die XMI-Datei eingetragen. Dadurch werden die Messwerte verfälscht, da die Assoziationen doppelt ins Gewicht fallen.
- In XMI wird der Typ eines Attributs oder Parameters als Referenz auf den entsprechenden *Classifier* dargestellt. *Together* generiert für alle vorkommenden Typen einen Datentyp (*DataType*) mit entsprechendem Namen und verweist auf diesen, auch wenn es sich beim Typ um eine Klasse oder ein Interface handelt. Hierdurch wird zwar der Name des Typs repräsentiert, der direkte Zusammenhang zum eigentlichen Typ geht jedoch verloren. Die generierten Datentypen sind unmittelbar im Modell enthalten. Ist einer der "echten" Typen ebenfalls direkt im Modell enthalten, so führt das zu einer Verletzung des UML-Metamodells, das keine zwei Modellelemente mit gleichem Namen im gleichen Namensraum erlaubt.
- *Together* erzeugt nicht immer wohlgeformtes XML. Zeichen, die in XML eine besondere Bedeutung haben (z.B. '<'), werden von *Together* nicht immer geschützt. Dies hat zur Folge, dass schon beim Parsen der XMI-Datei ein Fehler auftritt.

---

1. Referenztypen sind in Java Klassen, Interfaces und Arrays. Bei Referenztypen enthalten Variablen immer eine Referenz auf das eigentliche Objekt. Somit können auch Objekte, die als Parameter übergeben werden, durch eine Operation verändert werden.

## 3 Anforderungen

In der Aufgabenstellung (Kapitel 1) wurden die Anforderungen an das Werkzeug MOOSE schon grob umrissen. In diesem Kapitel folgt nun eine detailliertere Beschreibung. In Kapitel 3.1 wird beschrieben, wie ein UML-Modell auf eine Instanz von ODEM abgebildet werden soll. In Kapitel 3.2 wird erläutert, welche Metriken erhoben werden sollen. In Kapitel 3.3 wird darauf eingegangen, wie die Ausgabe der Messwerte erfolgen soll. Kapitel 3.4 widmet sich den nicht-funktionalen Anforderungen.

### 3.1 Abbildung nach ODEM

Das Werkzeug erhält als Eingabe eine UML-Entwurfsbeschreibung in Form eines XMI-Dokuments. Da die Metriken auf der Basis von ODEM definiert sind, muss zunächst eine Abbildung des UML-Modells auf eine Instanz von ODEM vorgenommen werden.

Die Beschreibung des formalen Modells ODEM (Reißing, 2000b) umfasst auch die Abbildungsvorschrift, wie ein UML-Modell nach ODEM überführt wird. Es wird daher nicht weiter auf die Herleitung jedes einzelnen Elements eingegangen. Allerdings müssen für die praktische Umsetzung der Abbildung einige Spezialfälle beachtet werden. Diese Spezialfälle gehen zum Teil auf UML-Modellelemente zurück, die nicht in ODEM berücksichtigt werden. Teilweise müssen Fehler bei der XMI-Generierung von UML-Werkzeugen korrigiert werden.

#### 3.1.1 Spezialfälle aufgrund von ODEM

Wie in Kapitel 2.1 beschrieben, sind in ODEM nicht alle Modellelemente enthalten, die die UML zur statischen Beschreibung eines Entwurfs zur Verfügung stellt. Es stellt sich jedoch die Frage, wie solche Elemente behandelt werden sollen, wenn sie in einem UML-Modell vorkommen. Man kann diese Modellelemente einfach weglassen und den damit verbundenen Informationsverlust in Kauf nehmen. Andererseits kann man versuchen, solche Konstrukte auf Elemente abzubilden, die in ODEM enthalten sind. Hier geht jedoch die Information über speziellen Eigenschaften des Elements verloren. Es gibt noch einen weiteren Aspekt, der beachtet werden muss, wenn man das gesamte UML-Metamodell betrachtet. Manche Modellelemente in ODEM können eine besondere Bedeutung erhalten, wenn sie in Verbindung mit anderen Modellelementen auftreten. Das ist beispielsweise bei Templateparametern der Fall (s.u.). Im Folgenden wird die Behandlung von nicht in ODEM liegenden Modellelementen beschrieben. Auf Elemente, die nicht der Beschreibung der statischen Struktur eines Entwurfs dienen, wird dabei nicht eingegangen:

- In der UML können Modellelemente *Constraints* haben. Attribute können durch den *Constraint* {frozen} als Konstanten gekennzeichnet werden. *Constraints* werden nicht beachtet. Konstanten werden wie normale Attribute behandelt.
- Multiplizitäten werden ignoriert.
- Allgemeine Abhängigkeiten (*Dependencies*) werden in die *uses*-Relation übernommen. Dies entspricht auch der Interpretation von Booch, Rumbaugh, Jacobsen (1999,

p.137 ff.). Diese bezeichnen eine *Dependency* als “using relationship“, die ausreicht, um die Benutzung eines Modellelements durch ein anderes zu kennzeichnen. Der Stereotyp «use» kann demnach verwendet werden, um eine explizite Abhebung von anderen Abhängigkeiten zu erreichen.<sup>1</sup>

- In der UML können Klassen andere Klassen oder Interfaces enthalten. In ODEM ist dies nicht möglich. Klassen und Interfaces werden daher gegebenenfalls in der Schachtelungshierarchie nach oben verschoben, bis sie direkt in einem Paket enthalten sind.
- Assoziationsklassen sind eine Mischung aus Klasse und Assoziation und dienen dazu, Assoziationen darzustellen, die Attribute, Operationen oder auch Assoziationen haben. Sie werden wie normale Assoziationen behandelt, die zusätzlichen Elemente werden ignoriert.
- Subsysteme vereinigen die Eigenschaften von Klassen und Paketen. Wie Pakete dienen Subsysteme dazu, andere Modellelemente zusammenzufassen. Ein Subsystem stellt jedoch nicht nur eine strukturelle Einheit dar, sondern weist auch ein bestimmtes Verhalten auf. Subsysteme haben daher Operationen und könne optional auch instanziiert sein. Subsysteme werden hier als Pakete behandelt. Verhaltensspezifische Aspekte gehen dabei verloren.
- Parametrisierte Modellelemente (Templates) werden weggelassen. Templates und ihre Auswirkungen auf den Entwurf sind zu komplex, um ihnen mit einer einfachen Abbildung auf andere Elemente, die in ODEM vorhanden sind, gerecht zu werden. Im UML-Metamodell ist ein Modellelement ein Template, sobald es mindestens einen Templateparameter besitzt. Für einen Templateparameter dürfen nur Modellelemente einer bestimmten Metaklasse eingesetzt werden. Diese wird bestimmt durch ein Dummy-Element der Metaklasse, das mit dem Templateparameter assoziiert ist. Ist in einem Modell ein Templateparameter enthalten, so muss bei der Abbildung nach ODEM das parametrisierte Modellelement und das Dummy-Element entfernt werden.
- Im UML-Metamodell wird zwischen Operationen und Methoden unterschieden. Bei Operationen handelt es sich um einen Dienst, der von einer Klasse angeboten wird und der aufgerufen werden kann. Bei einer Methode handelt es sich um die Implementierung einer Operation. Eine Redefinition einer Operation in einer Unterklasse ist in der UML nicht möglich. Eine Operation kann jedoch in einer Unterklasse erneut deklariert werden. Hierdurch wird eine Redefinition der dazugehörigen Methode angezeigt. In ODEM werden nur Operationen, nicht jedoch Methoden betrachtet. Operationen, die mit der gleichen Signatur schon in einer Oberklasse vorkommen, müssen also ignoriert werden.
- Konstruktoren werden nicht nach ODEM übernommen. Operationen können laut der UML-Spezifikation der OMG durch den Stereotyp «create» als Konstruktoren gekennzeichnet werden. Rumbaugh, Jacobson, Booch (1999) sehen hierfür den Stereotyp «constructor» vor. Operationen mit einem der beiden Stereotypen werden nicht

---

1. Booch, Rumbaugh und Jacobson, die “Erfinder” der UML, unterscheiden in ihrem ‘UML User Guide’ Abhängigkeiten nicht nach verschiedenen Unterklassen wie im UML-Metamodell. Bei ihnen ist nur von verschiedenen Stereotypen der Abhängigkeiten die Rede.

nach ODEM abgebildet. Optional sollen auch Operationen aussortiert werden können, die den gleichen Namen haben wie die Klasse, zu der sie gehören. Bei Entwürfen für Programmiersprachen wie C++ oder Java, bei denen Konstruktoren immer den Klassennamen erhalten, kann somit auf eine explizite Kennzeichnung per Stereotyp verzichtet werden.

### 3.1.2 Korrektur von Fehlern

Bei der Generierung von XMI durch UML-Werkzeuge können Verletzungen des UML-Metamodells auftreten (vgl. 2.3.3). Diese sollen bei der Abbildung nach ODEM soweit wie möglich korrigiert werden.

#### Allgemeine Korrekturen

Einige Fehler wurden bei mehreren Werkzeugen festgestellt. Die Maßnahmen zu ihrer Korrektur bleiben ohne Effekt, wenn das UML-Modell korrekt in XMI dargestellt ist. Diese Maßnahmen können daher unabhängig vom Werkzeug, durch das die XMI-Datei generiert wurde, vorgenommen werden.

- Die Unterklassen von *Dependency* im UML-Metamodell werden nicht von allen UML-Werkzeugen unterstützt. Sie werden durch Abhängigkeiten mit verschiedenen Stereotypen dargestellt. Instanzen von *Dependency* mit den Stereotypen «use», «call», «create» und «instantiate» werden in ODEM Beziehungen der *uses*-Relation. Der Stereotyp «send» kann außer Acht gelassen werden, da dieser nur für Abhängigkeiten zwischen den Modellelementen Operation und Signal verwendet wird. Signale sind für ODEM nicht relevant.
- Analog zum vorherigen Punkt werden *Dependencies* mit Stereotyp «realize» in die *realizes*-Relation in ODEM übernommen.
- Manche UML-Werkzeuge erlauben die Redefinition von geerbten Attributen in Unterklassen. In der UML sind Redefinitionen von Attributen nicht erlaubt und sollen auch in ODEM nicht vorkommen. Enthält eine Klasse ein Attribut, das den gleichen Namen wie ein geerbtes Attribut besitzt, so wird die erneute Definition ignoriert.

#### Together ControlCenter

Die Entwürfe, mit denen MOOSE erprobt werden soll, wurden mit *Together* erstellt. Deshalb sollen auch Fehler, die speziell bei der XMI-Generierung durch *Together* auftreten, korrigiert werden können. Diese Maßnahmen müssen natürlich optional durchgeführt werden können.

- Da der Wert für den *ownerScope* bei Operationen und Attributen unbrauchbar ist, wird er generell auf 'instance' gesetzt. So erhält man zumindest einen einheitlichen Wert. Da es sich bei den meisten Attributen und Operationen um Objektelemente handelt, erhält man bei 'instance' eine höhere Wahrscheinlichkeit als bei 'classifier', mit dem richtigen Wert zu arbeiten.

- Bei *Together* wird die Realisierung einer Klasse durch ein Interface in der XMI-Datei als Generalisierung dargestellt. "Erbt" eine Klasse von einem Interface, so wird dies als *realizes*-Beziehung interpretiert.
- Bei *Together* ist in XMI normalerweise keines der Enden einer Assoziation navigierbar (*navigable*). Da nur navigierbare Assoziationen nach ODEM übernommen werden, gehen so alle Assoziationen verloren. Daher sollen Assoziationen unabhängig von ihrer Navigierbarkeit nach ODEM übernommen werden können. Dabei erhält man aber zu viele Daten (und somit einen Informationsverlust). Das Fehlverhalten bei *Together* kann umgangen werden, indem für jede Assoziation explizit ihre Navigierbarkeit eingestellt wird. In diesem Fall ist eine Sonderbehandlung der Assoziationen nicht erwünscht. Die Korrektur sollte also optional und unabhängig von den anderen Korrekturen für *Together* erfolgen.

## 3.2 Erhebung von Metriken

Die Menge der Metriken, die durch MOOSE erhoben werden können, soll nicht fest implementiert, sondern konfigurierbar und erweiterbar sein. Bei der Konfiguration sollen beliebige, auf der Basis von ODEM definierte Metriken angegeben werden können. Wie die Konfiguration erfolgen soll, ist nicht vorgegeben.

MOOSE soll erprobt werden, indem mit ihm die Metriken des Qualitätsmodells QOOD erhoben werden. Es muss daher mindestens die Metriken in QOOD unterstützen. Zur Erprobung liegen verschiedene mit *Together* erstellte Entwürfe vor.

## 3.3 Präsentation der Messwerte

Die Ausgabe der Ergebnisse soll in Form von Reports erfolgen. Ein Report ist ein Dokument, das Messwerte enthält. Die Reports sollen möglichst flexibel gestaltet werden können. In Abschnitt 3.3.1 werden die allgemeinen Anforderungen für die Ausgabe von Reports beschrieben. In Abschnitt Abb. 3.3.2 werden die Reports für die Ausgabe der Messergebnisse für QOOD beschrieben.

### 3.3.1 Allgemeine Anforderungen

Es sollen verschiedene Arten von Reports ausgegeben werden können. Zum einen sollen die Messergebnisse so ausgegeben werden, dass sie durch ein Tabellenkalkulationsprogramm, z.B. StarCalc aus dem Star-Office-Paket der Firma SUN, importiert werden können. Als weitere Option sollen die Messwerte in HTML-Dokumenten ausgegeben werden können. Die verschiedenen Arten zur Darstellung von Messwerten in Reports, die im Folgenden beschrieben werden, beziehen sich sowohl auf die Tabellenkalkulation, als auch auf HTML. Die Reports sollen auch unterschiedliche Bereiche umfassen: Es sollen System-, Paket- und Klassenreports ausgegeben werden können.

Ein Systemreport kann System-, Paket-, und Klassenmetriken enthalten. Abbildung 8 zeigt als Beispiel einen Systemreport aus QOOD. Die Bedeutung der darin vorkommenden Akronyme findet man auf Seite 3. Interfaces, Klassen und Pakete werden zeilen-

	Conciseness					Structuredness		Decoupling			
	NAC	NOC				DITC	NAEC <sub>1</sub>	NEDC	NADC		
<b>Interfaces</b>											
mtodem.parse.handler.ElementHandler	0	3				0	0	0	0		
mtodem.parse.handler.OdemRepository	0	14				0	0	0	0		
<b>Classes</b>											
mtodem.Start	1	2				0	0	0	0		
mtodem.basic.MessageHandler	3	8				0	0	0	0		
mtodem.database.OdemDB	2	6				1	0	1	0		
mtodem.database.ModelDB	8	21				1	0	1	0		
mtodem.database.OdemDBConnection	2	2				0	2	0	2		
mtodem.parse.filter.DefaultElementFilter	15	15				1	0	1	0		
mtodem.parse.filter.ElementFilter	1	15				0	2	0	2		
mtodem.parse.filter.TogetherElementFilter	15	15				1	0	1	0		
mtodem.parse.handler.AssociationHandler	4	7				1	0	1	0		
mtodem.parse.handler.AttributeHandler	7	6				1	0	1	0		
mtodem.parse.handler.ClassHandler	5	6				1	0	1	0		
mtodem.parse.handler.DataTypeHandler	3	6				1	0	1	0		
mtodem.parse.handler.InterfaceHandler	9	6				1	0	1	0		
mtodem.parse.handler.ModelElementHandler	0	6				0	12	0	12		
mtodem.parse.handler.NameHandler	1	4				0	0	0	0		
mtodem.parse.handler.OperationHandler	5	6				0	0	0	0		
mtodem.parse.handler.OwnedElemHandler	2	8				1	0	1	0		
mtodem.parse.handler.PackageHandler	4	7				1	0	1	0		
mtodem.parse.handler.ParameterHandler	6	6				1	0	1	0		
mtodem.parse.handler.ParameterHandler	7	12				1	0	1	0		
mtodem.parse.handler.StereotypeHandler	4	6				0	0	0	0		
<b>Packages</b>											
mtodem	1	0	3			0		0	0	0	0
mtodem.basic	1	0	3			0		0	0	0	0
mtodem.database	1	0	0			1		0	0	0	0
mtodem.parse	0	0	2			1		0	0	0	0
mtodem.parse.filter	3	0	0			2		0	0	0	0
mtodem.parse.handler	17	3	0			2		0	0	0	0
System	0	0	1			-1		0	0	0	0
<b>System</b>											
moose	108	99	25	2	6						

Abbildung 8: Systemübersicht bei QOOD

weise aufgeführt, jeweils alphabetisch sortiert. Es wird ihr vollständig qualifizierter Name verwendet. Der vollständige Name besteht aus dem vollständigen Namen des Pakets, in dem das Element enthalten ist und dem Namen des Elements. Zwischen Paketname und dem eigentlichen Elementnamen steht ein Punkt. Für jedes Element werden Spaltenweise seine Messergebnisse aufgeführt. Die Systemmetriken werden ebenfalls spaltenweise aufgeführt. Durch Überschriften können Metriken gruppiert werden. In diesem Beispiel sind die Metriken nach den drei Kriterien Knappheit (*Conciseness*), Strukturiertheit (*Structuredness*) und Entkopplung (*Decoupling*) unterteilt.

Ein Paketreport enthält die Messwerte für ein bestimmtes Paket. Es wird für jedes Paket ein Report erzeugt. In einem Paketreport sollen Text und Messergebnisse beliebig platziert werden können. Außer den Messwerten sollen in Paketreports noch weitere Informationen über das Paket ausgegeben werden können, beispielsweise eine Liste aller

Klassen im Paket. Das Trennzeichen für die einzelnen Werte einer Liste soll frei wählbar sein. Analog zu den Metriken sollen nicht nur bestimmte, sondern beliebige Zusatzinformationen, die in ODEM enthalten sind, ermittelt werden können. Wie bei den Metriken ist die harte Anforderung hier, dass zumindest die Zusatzinformationen für QOOD dargestellt werden können.

Dem Paketreport entspricht auf Klassenebene der Klassenreport. Für jede Klasse wird ein Klassenreport ausgegeben, in dem Text, Messwerte und Zusatzinformationen beliebig angeordnet werden können. Analog zu Paket- und Klassenreports sollen auch Systemreports ausgegeben werden können, bei denen Systemmetriken, Text und Zusatzinformationen beliebig angeordnet sind.

### 3.3.2 Reports für QOOD

Für QOOD sollen verschiedene Reports zum Import in eine Tabellenkalkulationsprogramm generiert werden. Die Metriken in QOOD lassen sich nach verschiedenen Gesichtspunkten verfeinern (vgl. 2.2). Nach den Verfeinerungen der Metriken, die darin enthalten sind, lassen sich die Reports für QOOD in drei verschiedene Verfeinerungsstufen einteilen.

#### 1. Stufe:

Hier werden alle Metriken erhoben, bei denen keine Verfeinerung berücksichtigt wird. Die Messwerte für alle Klassen, Interfaces, Pakete und die Systemmetriken werden in einem Systemreport ausgegeben. Abbildung 8 zeigt ein Beispiel für einen Systemreport der ersten Verfeinerungsstufe für QOOD.

#### 2. Stufe:

Hier werden alle Metriken erhoben, bei denen höchstens eine Verfeinerung berücksichtigt wird. Die Ergebnisse werden in Systemreports ausgegeben. Für jedes der Kriterien Knappheit (*Conciseness*), Strukturiertheit (*Structuredness*) und Entkopplung (*Decoupling*) wird ein eigener Systemreport ausgegeben.

#### 3. Stufe:

In der letzten Verfeinerungsstufe werden zusätzlich Kombinationen der verschiedenen Verfeinerungen berücksichtigt. Beim Kriterium Knappheit werden die Metriken für alle möglichen Kombinationen der Verfeinerungen ermittelt. Bei der Entkopplung wird für jeden Beziehungstyp der aggregierte Wert, der Wert für lokale (nicht geerbte), reflexive, paketinterne und abstrakte Abhängigkeiten ausgegeben.<sup>1</sup>

Weiterhin sollen hierbei noch eine Reihe zusätzlicher Informationen ausgegeben werden. Das sind bei Klassen:

- Jeweils eine Liste der Attribute und der Operationen. Die Notation orientiert sich an der Darstellung von Attributen und Operationen in der UML. Wie in der UML wird der Sichtbarkeitsbereich durch "+", "#" oder "-" für "public", "protected" oder "private" angegeben. Eine Kennzeichnung, ob es sich um ein Klassen- oder Objektattri-

but (bzw. -operation) handelt und ob das Attribut (Operation) geerbt oder lokal definiert ist, erfolgt nicht. Beispiele:

- Attribut: -alter : integer
- Operation: +erhoeheAlter(jahre : int) : int

- Für jede der verschiedenen Relationen und bei Assoziationen zusätzlich für jede Aggregationsart soll eine Liste der Klassen ausgegeben werden, zu denen eine Abhängigkeit besteht. Weitere Verfeinerungen (wie z.B. Reflexivität) werden bei diesen Listen jedoch nicht berücksichtigt. Analog dazu wird für afferente Beziehungen jeweils eine Liste der abhängigen Klassen ausgegeben.

Bei Paketen werden folgende Zusatzinformationen ausgegeben:

- Jeweils eine Liste aller enthaltenen Interfaces, Klassen und Pakete. Für jedes dieser Modellelemente wird auch sein Sichtbarkeitsbereich ausgegeben.
- Eine Liste aller Klassen außerhalb des Pakets, zu denen Abhängigkeiten bestehen.
- Eine Liste aller anderen Pakete, zu denen Abhängigkeiten bestehen.
- Eine Liste aller Klassen innerhalb des Pakets, von denen Klassen in anderen Paketen abhängig sind.

Um die Fülle der hierbei anfallenden Informationen übersichtlich darstellen zu können, wird für jede Klasse und jedes Interface ein Klassenreport und für jedes Paket ein Paketreport generiert. Zusätzlich wird eine Systemreport für die System-Metriken erzeugt. Abbildung 9 zeigt ein Beispiel für einen Klassenreport. Eine Beschreibung der hier nicht gezeigten Reports findet man bei Schmider (2000b).

## 3.4 Nicht-funktionale Anforderungen

### Plattform

Das Werkzeug muss auf den Rechnern der Abteilung Software Engineering (SUN SPARC Station) unter dem Betriebssystem Solaris lauffähig sein.

### Benutzungsschnittstelle

Es wird keine grafische Benutzungsschnittstelle verlangt, der Aufruf über Kommandozeile genügt.

- 
1. In ODEM gibt es inklusive der allgemeinen *depends\_on*-Relation fünf verschiedene Relationen. Nimmt man noch die drei verschiedenen Aggregationsarten für Assoziationen hinzu, erhält man acht unterschiedliche Metriken für Abhängigkeiten. Betrachtet man bei den anderen vier Verfeinerungen für Entkopplung nach Paketzugehörigkeit, Reflexivität, Abstraktheit der Zielklasse und Vererbung jeweils beide Verfeinerungsmöglichkeiten und den aggregierten Wert, so erhält man  $8 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 648$  Kombinationen. Werden die Metriken für efferente und afferente Beziehungen erhoben, verdoppelt sich dies auf 1296. Auch wenn hier einige Kombinationen wegfallen, wie z.B. paketexterne, reflexive Beziehungen oder realizes-Beziehungen zu nicht-abstrakten Klassen, so erscheint die Präsentation aller dieser Werte nicht mehr sinnvoll.

Class mtodem.parse.handler.AttributeHandler									
<b>Conciseness:</b>									
Attributes: -id:String -name:NameHandler -ownerScope:String -visibility:String -type:String -owner:ModelElementHandler -inTypeRef:boolean									
NAC	total		public		protected		private		
	object	class	object	class	object	class	object	class	
total	7		0		0		7		
	7	0	0	0	7	0	7	0	
local	7		0		0		7		
	7	0	0	0	0	0	7	0	
inherited	0		0		0		0		
	0	0	0	0	0	0	0	0	
Operations: +setOwner(handler:ModelElementHandler) : void +characters(ch:char[], start:int, end:int) : void +endElement(namespaceURI:String, localName:String, rawName:String) : boolean +getOwner() : ModelElementHandler +getID() : String +startElement(namespaceURI:String, localName:String, rawName:String, atts:Attributes) : void									
NOC	total		public		protected		private		
	object	class	object	class	object	class	object	class	
total	6		6		0		0		
	6	0	6	0	0	0	0	0	
local	0		0		0		0		
	0	0	0	0	0	0	0	0	
inherited	6		6		0		0		
	6	0	6	0	6	0	6	0	
<b>Structuredness</b>									
DITC (depth of inheritance tree of a class):					1				
NAEC <sub>1</sub> (number of local affarent extends relationships of a class):					0				
<b>Decoupling</b>									
Efferent dependencies	total	local	reflexive	abstract	package-internal	Classes/Interfaces depending on			
depends_on	1	1	0	1	1	mtodem.parse.handler.ModelElementHandler			
extends	1	1	0	1	1	mtodem.parse.handler.ModelElementHandler			
realizes	0	0	0	0	0				
associates	0	0	0	0	0				
none	0	0	0	0	0				
aggregate	0	0	0	0	0				
composite	0	0	0	0	0				
uses	0	0	0	0	0				

Abbildung 9: Klassenreport bei QOOD

Afferent dependencies	total	local	reflexive	abstract	package-internal	dependent Classes/Interfaces
depends_on	0	0	0	0	0	
extends	0	0	0	0	0	
realizes	0	0	0	0	0	
associates	0	0	0	0	0	
none	0	0	0	0	0	
aggregate	0	0	0	0	0	
composite	0	0	0	0	0	
uses	0	0	0	0	0	

**Abbildung 9: Klassenreport bei QOOD**

### Qualitätsanforderungen

Das Werkzeug MOOSE soll als prototypische Implementierung vor allem die Realisierbarkeit der Konzepte zeigen. Explizite Qualitätsanforderungen wurden daher nicht formuliert. Das Werkzeug sollte jedoch vorführbar sein, was gewisse (wenn auch nicht sehr scharfe) Mindestanforderungen hinsichtlich Zuverlässigkeit und Effizienz impliziert.

# 4 Konzeption

In diesem Kapitel werden die Konzepte beschrieben, mit denen die Anforderungen aus Kapitel 3 umgesetzt werden. Im Mittelpunkt stehen dabei die folgenden Fragen:

- Wie und in welcher Form sind die Metriken einzugeben, die erhoben werden sollen?
- Wie können bei der Abbildung nach ODEM Fehler bei der XMI-Generierung bestimmter Tools korrigiert werden?
- Wie wird spezifiziert, wie die Messergebnisse ausgegeben werden?

Die Kapitel 4.1 bis 4.3 behandeln die Antworten auf diese Fragen. Kapitel 4.4 beschreibt die Benutzung des Werkzeugs.

## 4.1 Konfiguration der Metriken

Das Werkzeug soll möglichst beliebige, mit Hilfe von ODEM formulierte Metriken erheben können. Es muss also eine Möglichkeit geben, die Metriken zu konfigurieren. Hierzu müssen die Metrikdefinitionen durch das Werkzeug eingelesen werden können. Eine mögliche Lösung wäre die Eingabe mittels einer Skriptsprache, die alle in ODEM enthaltenen Bezeichner sowie geeignete Operatoren zu ihrer Verknüpfung enthält. Die Spezifikation einer Skriptsprache und die Realisierung eines passenden Interpreters oder Compilers wäre jedoch zu aufwendig, um im Rahmen dieser Diplomarbeit durchgeführt werden zu können. Im Folgenden werden daher alternativ zwei Lösungen vorgestellt, bei denen die Definition neuer Metriken weniger komfortabel ist, die aber einfacher zu realisieren sind. Die ausgewählte Lösung wird anschließend genauer beschrieben.

### 4.1.1 Der Plug-In-Ansatz

Eine Möglichkeit zur Definition der Metriken ist, diese in einer Programmiersprache zu implementieren. Aber gerade eine Beschränkung auf bestimmte, fest implementierte Metriken soll ja vermieden werden. Um beliebige Metriken erheben zu können, muss es also möglich sein, zur Laufzeit Module, die beim Compilieren noch nicht bekannt sind, als sogenannte Plug-Ins zu laden. Lediglich die Schnittstelle, über die auf diese Module zugegriffen werden kann, muss bekannt sein. Bei der im Folgenden vorgestellten Lösung wird die Programmiersprache Java genutzt. Java bietet die Möglichkeit, Klassen dynamisch zur Laufzeit zu laden und zu instanziiieren.

Die Metriken werden in Form von Klassen realisiert. Das Werkzeug MOOSE definiert ein Interface, das von jeder Metrik implementiert werden muss. Dieses Interface enthält eine Operation zur Abfrage der Messergebnisse. Diese wird beim Generieren eines Reports aufgerufen (Abb. 10a).

Um die Ergebnisse berechnen zu können, müssen die Plug-Ins auf das Entwurfsmodell zugreifen können. Ein Ansatz hierzu beruht auf der Idee, die Messungen direkt beim Parsen vorzunehmen, ohne eine explizite interne Darstellung des Modells zu erzeugen. Hierzu wird das Interface der Metriken erweitert. Hinzu kommen Operationen, mit denen ODEM-Modellelemente an die Plug-Ins weitergegeben werden können. Sobald

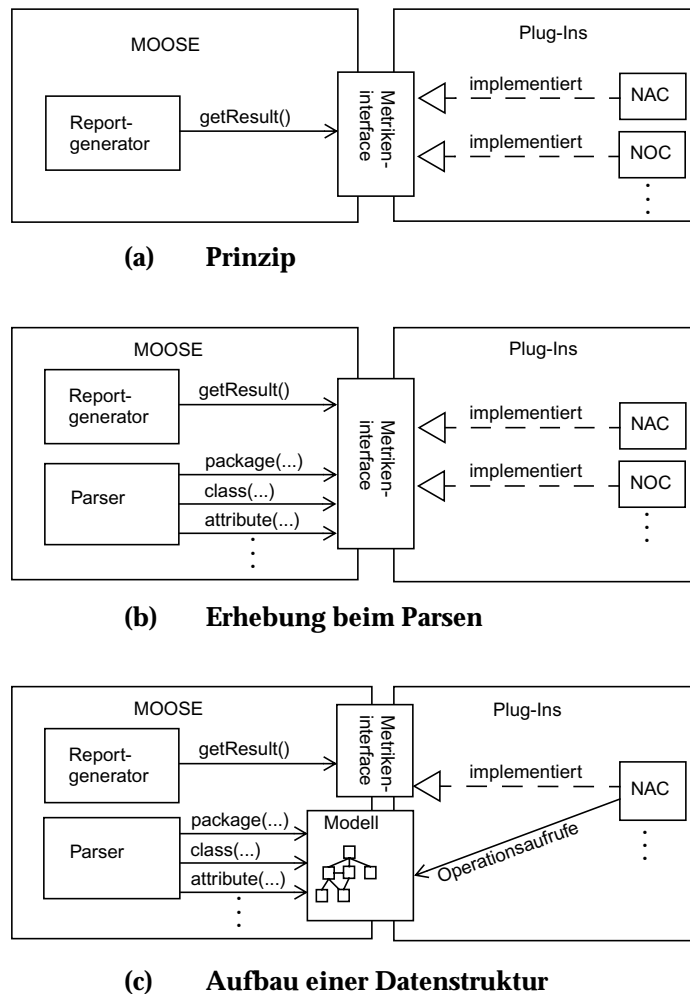


Abbildung 10: Plug-In-Ansatz

der Parser ein Modellelement erkannt hat, wird dieses über die entsprechenden Operationen an die Plug-Ins gemeldet (Abb. 10b). Diese Vorgehensweise hat jedoch einige Nachteile:

- Geparste Modellelemente werden an alle Plug-Ins gemeldet, unabhängig davon, welche Informationen diese tatsächlich benötigen.
- Relevant für die Metriken sind Elemente des formalen Modells ODEM. Um diese herzuleiten, sind jedoch oft mehrere Elemente des UML-Modells notwendig, die sich an unterschiedlichen Stellen in der XMI-Datei befinden können. Ein Beispiel hierfür ist die Berücksichtigung von Dependencies in ODEM-Relationen in Abhängigkeit von ihrem Stereotyp (vgl. 3.1.2). Ein weiteres Beispiel sind die erweiterten Relationen in ODEM. So werden zur vollständigen Berechnung der *extends*-Relation alle *extends*-Beziehungen des Modells benötigt. Das bedeutet, dass die Elemente des UML-Modells, aber auch schon erzeugte Elemente der ODEM-Instanz, im Parser

solange gespeichert werden müssen, bis feststeht, dass sie nicht mehr für die Herleitung weiterer ODEM-Elemente benötigt werden.

- Auch innerhalb der Plug-Ins müssen Modellelemente zwischengespeichert werden. Deutlich wird dies z.B. anhand der Metrik  $NEAC_a$  (Number of abstract efferent Associations of a Class). Diese ist folgendermaßen definiert:

$$NEAC_a(c) = \sum_{c' \in C \cup I: c'.isAbstract} associates^*(c, c').weight$$

Sieht man einmal von dem Sonderfall  $c = c'$  ab, so werden immer drei Elemente der ODEM-Instanz benötigt, um einen Messwert zu aktualisieren: die Klasse, für die die Metrik erhoben wird, eine Assoziation dieser Klasse und die andere an der Assoziation beteiligte Klasse.

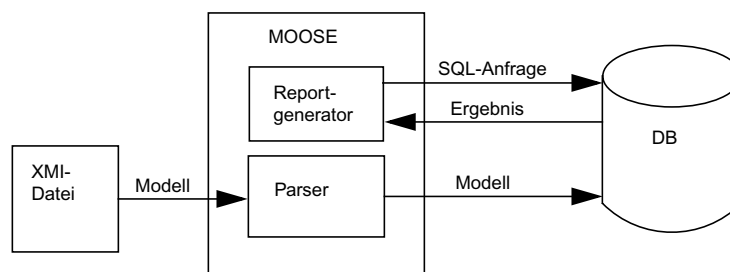
Durch die teilweise Speicherung des Modells innerhalb der Plug-Ins wird der Implementierungsaufwand für diese erhöht. Hinzu kommt, dass Informationen redundant gespeichert werden, z. B. in mehreren Plug-Ins und eventuell zusätzlich noch im Parser.

Diese Probleme können umgangen werden, indem beim Parsen der XMI-Datei eine Datenstruktur aufgebaut wird, die eine explizite Darstellung des Modells enthält. Die Plug-Ins greifen zur Erhebung der Metriken über eine geeignete Schnittstelle auf diese Datenstruktur zu (Abb. 10c).

### 4.1.2 Der Datenbank-Ansatz

Eine andere Möglichkeit ist die Formulierung der Metriken mit einer Anfragesprache. Um den Aufwand für die Spezifikation dieser Sprache und der Implementierung eines Interpreters zu vermeiden, kann dafür eine bereits bestehende Komponente verwendet werden. Hierfür kommt z.B. die Standardanfragesprache für relationale Datenbanken SQL (Structured Query Language) in Frage.<sup>1</sup>

Bei diesem Ansatz wird zuerst die XMI-Datei geparkt und das Modell in einer relationalen Datenbank auf der Basis von ODEM gespeichert. Anschließend können die Metriken als SQL-Anfragen auf dieser Datenbank erhoben werden (Abb. 11).

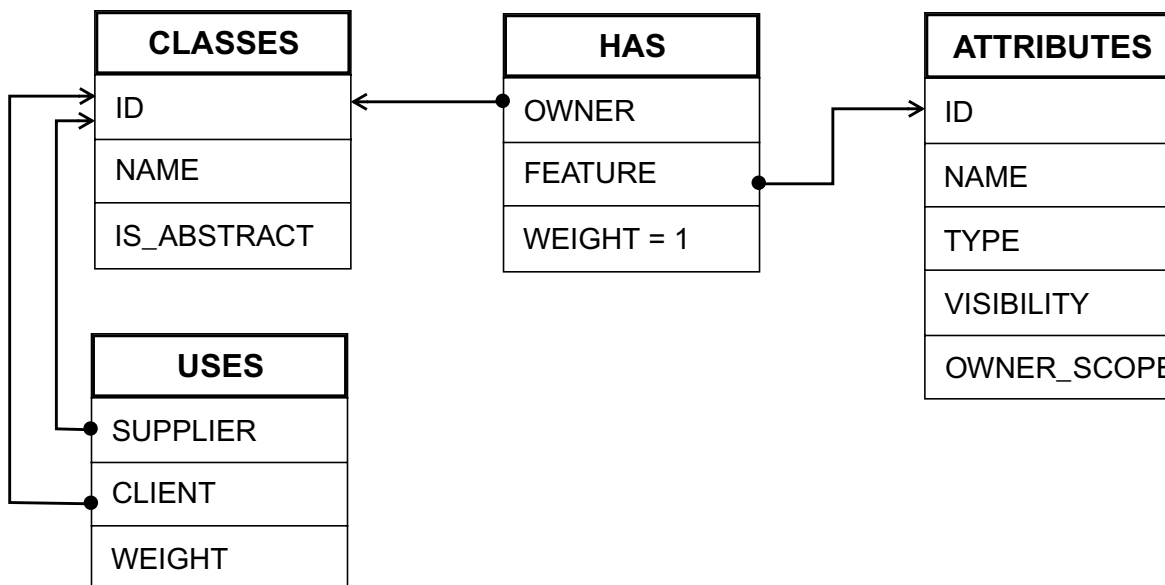


**Abbildung 11: Datenbank-Ansatz**

1. Informationen zu relationalen Datenbanken und SQL liefert das Buch von Date (2000). Einen weniger umfangreichen Einblick in Deutsch erhält man bei Kemper und Eickler (2001). Neben anderen DBMS wird von Kevin und Daniel Kline (2001) auch *postgreSQL* berücksichtigt.

Wie im Entity-Relationship-Modell (ER-Modell) sind die Bestandteile von ODEM Entitäten und Beziehungen zwischen diesen Entitäten. Die Eigenschaften der Entitäten und der Beziehungen werden durch Attribute beschrieben. In ODEM sind Entitäten eines Typs in Mengen zusammengefasst, Beziehungen eines Typs ergeben eine Relation. Dies entspricht Entitätstypen und Beziehungstypen im ER-Modell. Das ER-Modell fordert jedoch, dass jeder Entitätstyp einen Schlüssel besitzt. Dabei handelt es sich um ein oder mehrere Attribute, anhand derer sich eine Entität des Typs eindeutig identifizieren lässt. In ODEM ist dies nicht der Fall. Parameter werden z.B. erst eindeutig identifiziert durch ihre Beziehung zu einer Operation. Durch Hinzufügen eines künstlichen Schlüssels kann ODEM jedoch im ER-Modell dargestellt werden. Im ER-Modell beschriebene Daten können leicht in ein Relationenschema und somit in Tabellen einer relationalen Datenbank überführt werden.<sup>1</sup>

Die Repräsentation eines Entwurfs in der Datenbank spiegelt daher recht genau die Struktur von ODEM wider. Die Formulierung von auf ODEM basierenden Metriken in SQL sollte daher recht einfach sein. Auch umgekehrt lässt eine Datenbankabfrage leicht die ursprüngliche Form der Metrik erkennen. Abbildung 12 zeigt einen Ausschnitt eines möglichen Datenbankschemas zur Speicherung einer ODEM-Instanz. Die Modellelemente (hier Klassen und Attribute) erhalten in der Datenbank eine ID zur eindeutigen Identifikation. Tabelleneinträge für eine Beziehungen bestehen aus den beiden IDs der beteiligten Modellelemente und dem Gewicht der Beziehung (*Weight*). Bei manchen Beziehungen (z.B. *has*) ist das Gewicht immer eins.



**Abbildung 12: Schema der ODEM-Datenbank (Ausschnitt)**

1. Es gibt auch Werkzeuge, die aus ER-Diagrammen CREATE TABLE Statements generieren. Ein Beispiel hierfür ist das Werkzeug Together.

## Das Datenbankmanagementsystem

Um den Datenbankansatz überhaupt realisieren zu können, muss zunächst geklärt werden, ob ein geeignetes Datenbankmanagementsystem (DBMS) erhältlich ist. Geeignet bedeutet hier, dass es keine Kosten für Lizenzgebühren verursachen sollte und trotzdem weitestgehend den SQL92-Standard erfüllen sollte. Außerdem sollte die Installation und Administration möglichst einfach sein, da der Schwerpunkt der Diplomarbeit nicht auf der Einrichtung eines Datenbanksystems liegen sollte.

In die nähere Auswahl kamen zwei Kandidaten. Beide DBMS können im Quelltext aus dem Internet bezogen und kostenlos genutzt werden. Bei *mySQL* handelt es sich um ein schlankes, effizientes DBMS. *PostgreSQL* glänzt mit einem grossen Funktionsumfang, ist aber langsamer als *mySQL*. Den Ausschlag zugunsten von *postgreSQL* gab letztendlich der größere Funktionsumfang. Besonders schwer wog dabei, dass *mySQL* keine Subqueries und keine Sichten (*Views*) unterstützt.

### 4.1.3 Vergleich der Alternativen

Der Vergleich der beiden Ansätze erfolgt hier anhand der Vor- und Nachteile des Datenbankansatzes gegenüber dem Plug-In-Ansatz:

#### Vorteile des Datenbank-Ansatzes

- Die Repräsentation der Daten in der Datenbank ist ähnlich der Beschreibung von ODEM. Somit ist eine einfache Formulierung der Metriken in SQL-Anfragen möglich.
- Die Metriken können einfach eingegeben werden, ohne wie beim Plug-In-Ansatz vorher kompiliert werden zu müssen. Daher stehen die SQL-Anfragen (und somit die Definition der Metrik) auch immer im Quelltext zur Verfügung.
- Die Formulierung einer SQL-Anfrage erfordert weniger Aufwand als die Implementierung einer Klasse. Dies ist von Bedeutung für die Definition von Metriken allgemein, insbesondere aber für die Erprobung des Werkzeugs mit den Metriken für QOOD. Hierfür müssen über 200 Metriken eingegeben werden.
- Die notwendigen SQL-Kenntnisse zur Formulierung der Metriken sind einfacher zu erlernen als eine Programmiersprache.
- Bei der Entwicklung des Werkzeugs wurde eine iterative Vorgehensweise gewählt (vgl. 6.1). Als erste Entwicklungsstufe sollte hierbei das Einlesen einer XMI-Datei und die Abbildung des UML-Modells nach ODEM ermöglicht werden. Bei einer Speicherung in einer Datenbank kann dann sofort, z.B. für den Test, mittels Ad-hoc-Anfragen auf das Modell zugegriffen werden. Beim Plug-In-Ansatz müssten zuerst geeignete Ausgabeoperationen implementiert werden. Beim Datenbank-Ansatz wird auch das Risiko minimiert, bei zu langsamem Projektfortschritt am Ende ohne brauchbares Ergebnis dazustehen. Ist die Übertragung eines UML-Modells in die Datenbank erst einmal realisiert, können die Metriken direkt auf der Datenbank erhoben werden.

- Ein einmal in die Datenbank eingelesenes Modell bleibt gespeichert, bis es explizit gelöscht wird. Beim Plug-In-Ansatz muss die XMI-Datei bei jedem Zugriff erneut eingelesen werden.

### Nachteile des Datenbankansatzes

Mit der Hinzunahme eines DBMS entstehen zusätzliche Kosten. Da *postgreSQL* kostenlos genutzt und aus dem Internet bezogen werden kann, können Kosten für die Anschaffung vernachlässigt werden. Zu betrachten sind jedoch noch Kosten, die entstehen durch:

- Speicherplatzbedarf für Sekundärspeicher. Laut der Dokumentation benötigt *postgreSQL* selbst einen Speicherplatz von ca. 5 Megabyte. Hinzu kommen ca. 1 MB für eine leere Datenbank. Die Daten selbst benötigen etwa den fünffachen Speicherplatz wie bei einer Speicherung in einer Textdatei. Meine eigenen Erfahrungen ergaben einen Speicherplatzbedarf von etwa 30 MB für *postgreSQL* (inklusive der Dokumentation und einer Log-Datei) und 1,5 MB für eine leere Datenbank. Bei Preisen für Magnetplattenspeicher von derzeit unter 1 Cent pro Megabyte sollten diese Kosten jedoch eine untergeordnete Rolle spielen.
- Administration. Zum einen muss *postgreSQL* installiert werden, zum anderen muss entweder dafür gesorgt werden, dass der Datenbank-Server immer läuft, oder er muss vor der Benutzung des Werkzeugs gestartet werden.

Es gibt auch noch weitere Nachteile:

- SQL92 ist nicht berechenbarkeitsvollständig. Daher können nicht alle möglichen Metriken einfach als SQL-Anfrage formuliert werden. Betroffen hiervon sind z.B. die in QOOD rekursiv definierten Metriken DNHP (depth in the nesting hierarchy of a package) und DITC (depth of inheritance tree of a class) enthalten. DNHP ist beispielsweise folgendermaßen definiert:

$$\text{DNHP}(p) = \text{DNHP}(p': \text{contains}(p', p)) + 1$$

$$\text{DNHP}(S) = -1$$

*PostgreSQL* verfügt jedoch über eine prozedurale Programmiersprache (PL/pgSQL<sup>1</sup>), die die Anfragesprache berechenbarkeitsvollständig macht. Allerdings kann diese nicht direkt in SQL-Anfragen verwendet werden. Mit ihrer Hilfe können aber Funktionen definiert werden, die von den Anfragen benutzt werden. Allerdings hat dies eine Reihe von Nachteilen gegenüber einfachen Anfragen. Erstens müssen die Funktionen “von Hand” auf der Datenbank definiert werden, bevor sie durch Anfragen zur Metrikenerhebung benutzt werden kann. Zweitens ist die Definition der Metrik nicht mehr anhand der SQL-Anfrage ersichtlich. Drittens müssen in den Funktionen SQL-Anfragen verwendet werden, um auf die Tabellen zuzugreifen. Normalerweise reicht eine Anfrage, um die Messwerte einer Metrik für alle Elemente zu erheben (vgl. 4.1.4). Wird jedoch in einer Anfrage für jedes Element eine Funktion aufgerufen, um die Metrik zu berechnen, führt jeder Funktionsaufruf zu weiteren Datenbankabfragen. Das macht die Auswertung sehr ineffizient im Vergleich zu “normalen” Metriken.

---

1. PL/pgSQL ist Oracles PL/SQL sehr ähnlich (vgl. Dicken, Hipper, Müßig-Trapp 2000).

- Die Messungen beim Plug-In-Ansatz können effizienter erfolgen als die Auswertung über die Datenbank.

Ich habe hier der Datenbank-Lösung den Vorzug gegeben. Die einfache Eingabe der Metriken und der leicht herzustellende Bezug zwischen der ursprünglichen Definition der Metrik und einer Datenbankabfrage schienen mir einen sehr großen Vorteil gegenüber dem Plug-In-Ansatz darzustellen. Metriken, für deren Berechnung Datenbankfunktionen benötigt werden sind seltene Sonderfälle. Wichtig ist das auch diese Metriken erhoben werden können. Ein echter Nachteil gegenüber dem Plug-In-Ansatz ist hierbei ohnehin nur die mangelnde Effizienz.

Auch die Vorteile, die der Datenbankansatz für die Durchführung dieses Projekts bietet, wurden als wichtige Pluspunkte gewertet. Die Nachteile, die sich aus der Administration des DBMS ergaben, wurden für die prototypische Implementierung als nicht besonders ausschlaggebend bewertet. Der Aufwand hierfür war gering im Vergleich zur Zeitersparnis bei der Metrikendefinition. Für ein fertiges Gebrauchsprodukt könnte diese Bewertung jedoch anders ausfallen. Eine Lösung mit vordefinierten Metriken (z.B. den Metriken aus QOOD), die bei Bedarf per Plug-In erweitert werden können, erscheint mir für viele Anwender attraktiver.

#### 4.1.4 Formulierung der Anfragen

In diesem Abschnitt wird erläutert, wie die Datenbankabfragen zur Ermittlung der Metriken aufgebaut sind und wie die Eingabe der Metriken in das Werkzeug erfolgt.

##### Metriken

Wie oben beschrieben, werden die Metriken als SQL-Anfragen formuliert. SQL-Anfragen liefern als Ergebnis eine Tabelle. Diese kann beliebig viele Spalten und Zeilen haben. Sollen die Ergebnisse durch das Werkzeug weiterverarbeitet werden, so muss festgelegt werden, welches Format die Ergebnistabellen haben müssen und wie ihr Inhalt zu interpretieren ist.

Das Datenbankschema zu den hier aufgeführten Beispielanfragen ist in Abbildung 12 (Seite 36) dargestellt. Abbildung 13 zeigt das UML-Modell, das den Ergebnissen der Anfragen zugrundeliegt.

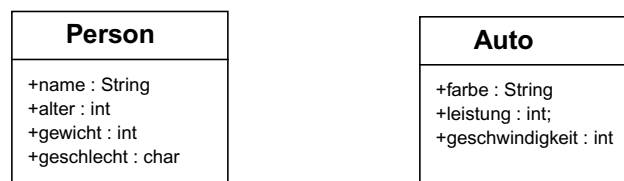


Abbildung 13: UML-Klassendiagramm

Eine SQL-Anfrage für eine Klassenmetrik muss eine zweispaltige Tabelle als Ergebnis liefern. In der ersten Spalte muss der qualifizierte Name der Klasse stehen, die zweite Spalte enthält die jeweiligen Messwerte. Abbildung 14 zeigt eine mögliche SQL-Anfrage für die Metrik NAC mit einem Beispiel für ein Ergebnis der Anfrage.

**Definition:**

NAC (number of attributes of a class)

$$\text{NAC}(c) = |\{a \in A: \text{has}^*(c,a)\}|$$

**Anfrage:**

```
SELECT c.qname, COUNT(a.id) AS NAC
FROM classes c
LEFT JOIN has_ext h ON c.id = h.owner
LEFT JOIN attributes a ON h.feature = a.id
GROUP BY c.qname;
```

**Ergebnis:**

qname	NAC
Auto	3
Person	4

**Abbildung 14: SQL-Anfrage für die Metrik NAC**

Nur zwei Bemerkungen zur obigen Anfrage:

- Die Left-Joins werden verwendet, damit in der ersten Spalte der Tabelle alle Klassen enthalten sind, unabhängig davon, ob sie Attribute besitzen oder nicht. Das Zählen der Attribut-IDs statt der Tupel in der ersten Zeile bewirkt, dass für diese Klassen auch tatsächlich der Wert null und nicht eins ausgegeben wird.
- Auf die Angabe des Metriknamens in der ersten Zeile kann verzichtet werden.

Weitere Informationen zur Formulierung von Datenbankabfragen für mit ODEM definierte Metriken findet man bei Schmider (2002b).

Analog zu Klassenmetriken muss das Ergebnis einer Anfrage für eine Paketmetrik ebenfalls eine zweispaltige Tabelle sein. Die erste Spalte enthält die qualifizierten Namen der Pakete, die zweite Spalte die entsprechenden Messwerte.

Bei Systemmetriken ist das Ergebnis nur ein einzelner Wert, also eine Tabelle mit einer Spalte und einer Zeile. Abbildung 15 zeigt eine mögliche SQL-Anfrage für die Metrik NAS.

**Zusatzinformationen**

Zusätzlich zu den Messergebnissen sollen auch andere Informationen ermittelt werden können, die aus einer Entwurfsbeschreibung auf der Basis von ODEM herleitbar sind

**Definition:**

NAS (number of attributes in the system)

$$\text{NAS}(S) = |A|$$

**Anfrage:**

```
SELECT COUNT(*) AS NAS
FROM attributes;
```

**Ergebnis:**

```
NAS
-----
7
```

**Abbildung 15: SQL-Anfrage für die Metrik NAS**

(vgl. 3.3). Diese Informationen können ebenfalls per SQL aus der Modelldatenbank ermittelt werden.

Zusatzinformationen für ein Modellelement (oder das Modell) können einfache Werte (z.B. der Sichtbarkeitsbereich einer Klasse) oder eine Liste von Werten (z.B. alle Attribute einer Klasse) sein. Ein Wert wird dargestellt als Zeichenkette. Die Zusatzinformationen können so als Funktionen betrachtet werden. Hierbei wird unterschieden zwischen Funktionen, die ein Modellelement auf eine Zeichenkette abbilden und Funktionen, die ein Element auf eine Liste von Zeichenketten abbilden. Wie bei Metriken wird auch hier zwischen Klassen-, Paket- und Systemfunktionen unterschieden.

Bei Funktionen, die nur eine einzelne Zeichenkette als Ergebnis liefern, können die SQL-Anfragen die Ergebnisse in der gleichen Form wie bei Metriken liefern. Sollen jedoch Listen ermittelt werden, so ist es erlaubt, dass der Name eines Modellelements in der ersten Spalte der Ergebnistabelle mehrmals vorkommt. Aus den dazugehörigen Werten in der zweiten Spalte wird durch das Werkzeug die Ergebnisliste für das jeweilige Element erzeugt. Abbildung 16 zeigt eine Anfrage zur Ermittlung der Liste aller Attribute einer Klasse und ein mögliches Ergebnis der Anfrage. Bei Systemmetriken kann die einspaltige Ergebnistabelle mehrere Zeilen besitzen.

Mit den für die Metriken ermittelten Messergebnissen werden keine weiteren Berechnungen durchgeführt. Für die Ausgabe können sie daher wie die Zusatzinformationen als Strings behandelt werden. Unter dem Begriff Funktionen werden daher Metriken und Anfragen für Zusatzinformationen zusammengefasst. Für MOOSE genügt die Unterscheidung zwischen Funktionen, die als Ergebnis einen einzelnen Wert, und Funktionen, die ein Liste von Werten liefern.

**Anfrage:**

```
SELECT c.qname, a.name AS _attributes
FROM classes c
LEFT JOIN has_ext h ON c.id = h.owner
LEFT JOIN attributes a ON h.feature = a.id;
```

**Ergebnis:**

qname	_attributes
Auto	farbe
Auto	leistung
Auto	geschwindigkeit
Person	alter
Person	geschlecht
Person	gewicht
Person	name

**Abbildung 16: SQL-Anfrage für die Liste der Attribute einer Klasse**

### 4.1.5 Eingabe der Funktionen

Die Definitionen der verschiedenen Funktionen werden, wie die Modelle, ebenfalls in der Datenbank gespeichert. Auf die Funktionen kann dann über ihren Namen zugegriffen werden. Ein Eintrag für eine Funktion enthält die folgenden Informationen:

- Name
- SQL-Anfrage
- Beschreibung
- Bereich (System, Paket oder Klasse)
- Art des Ergebnisses (einzelner Wert oder Liste)

Die beiden letzten Punkte werden benötigt, um die Ergebnisse bei der Auswertung richtig zu interpretieren. Bei Klassenmetriken wird zusätzlich eine zweite SQL-Anfrage für Interfaces gespeichert. Die Eingabe der Funktionen erfolgt direkt in die Datenbank. Bei *postgresql* kann die Eingabe z.B. über den grafischen Client *pgaccess* erfolgen. Eine weitere Möglichkeit ist der Import der Daten aus einer Textdatei.

## 4.2 Abbildung nach ODEM

ODEM umfasst nur einen Teil des UML-Metamodells. Außer den Elementen, die aus dem UML-Metamodell übernommen werden, enthält ODEM noch zusätzliche Relationen. Diese werden aus anderen Relationen abgeleitet und dienen der einfacheren Formulierung von Metriken (vgl 2.1).

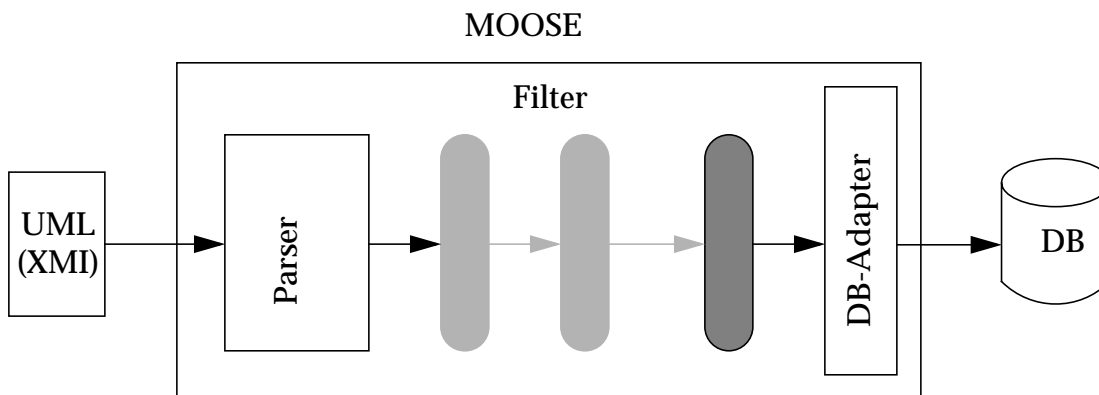
Um aus einem UML-Modell eine explizite Darstellung des Entwurfs als Instanz von ODEM zu erzeugen, kann also prinzipiell in zwei Schritten vorgegangen werden:

1. Parsen der XMI-Datei. Modellelemente, die in der ODEM-Instanz vorkommen, werden in die Datenbank geschrieben.
2. Erzeugen der Tabellen für die erweiterten Modellelemente.

Beim Parsen müssen jedoch folgende Punkte beachtet werden:

- Bei manchen Elementen des UML-Modells kann nur aufgrund anderer Modellelemente entschieden werden, ob und in welcher Form sie nach ODEM übernommen werden. Diese Elemente können aber an einer ganz anderen Stelle in der XMI-Datei stehen. Beispiele:
  - *Dependencies* werden in Abhängigkeit von ihrem Stereotyp in die *uses*- oder die *realizes*-Relation oder gar nicht übernommen.
  - Beziehungen werden nur nach ODEM übernommen, wenn beide beteiligten Modellelemente ebenfalls in ODEM vorkommen.
- Bevor die Elemente in die Datenbank eingetragen werden, müssen Fehler, die sich durch die XMI-Generierung bestimmter Werkzeuge, wie in Kapitel 3.1.2 beschrieben, korrigiert werden können.

Der Parser gibt deshalb alle für ODEM relevanten Elemente an einen Filter weiter (vgl. Abb. 17). Relevant sind alle Modellelemente, die vielleicht (oder sicher) nach ODEM übernommen werden, und solche, die Informationen über die Behandlung anderer Elemente liefern.



**Abbildung 17: Erzeugen einer ODEM-Instanz aus einem UML-Modell**

Der Filter gibt die Modellelemente an den Datenbank-Adapter weiter. Die Ausgabe eines Modellelements durch einen Filter kann auf verschiedene Weise erfolgen:

- Es wird unverändert weitergeleitet.
- Es wird ausgefiltert.
- Es wird in ein Element (oder mehrere) eines anderen Typs umgewandelt.

Modellelemente können im Filter gepuffert werden. Außerdem können im Filter Informationen über bereits weitergeleitete Modellelemente gespeichert werden.

Es gibt einen Standard-Filter, der bei einer korrekten Darstellung des Entwurfs in der XMI-Datei die richtigen Modellelemente an den Datenbank-Adapter weitergibt. Der Datenbank-Adapter übernimmt die Verbindung zum DBMS. Er erzeugt die Tabellen, in denen das Modell gespeichert wird, und generiert aus den Modellelementen, die übergeben werden, die passenden Datenbankeinträge. Die erweiterten Relationen werden ebenfalls vom Datenbank-Adapter generiert.

Die Korrektur der Fehler im UML-Modell (bzw. dessen XMI-Darstellung) ist abhängig vom Werkzeug, mit dem der Entwurf erstellt wurde. Hinzu kommen eventuell weitere Korrekturen, die von anderen Faktoren abhängen. Wurde der Entwurf mit *Together* erstellt, so müssen die Fehler in der XMI-Generierung dieses Werkzeugs korrigiert werden. Hierbei muss aber unterschieden werden, ob auch nicht navigierbare Assoziationen nach ODEM übernommen werden sollen. Dies hängt davon ab, ob die Navigierbarkeit für jede Assoziation beim Entwurf explizit eingestellt wurde (vgl. 3.1.2).

Für die Korrektur von Fehlern in der XMI-Datei stehen folgende Möglichkeiten zur Verfügung:

- Vor den Standard Filter werden ein oder mehrere Filter eingesetzt. In diesen werden die Daten so bearbeitet, dass sie beim Standard-Filter so ankommen, als sei eine korrekte XMI-Datei geparkt worden.
- Ersetzen des Standard-Filters durch einen anderen Filter, der zusätzlich zur Arbeit des Standardfilters die Korrektur der Daten übernimmt.

Die erste Variante ermöglicht mit wenigen Filtern eine sehr flexible Konfiguration. Um redundante Arbeiten in mehreren Filtern zu vermeiden, kann aus Effizienzgründen der zweite Ansatz von Vorteil sein.

Für die in Kapitel 3 beschriebenen Korrekturen werden folgende Filter zur Verfügung gestellt:

- Ein Filter für *Together*. Dieser gibt für Generalisierungsbeziehungen zwischen Klassen und Interfaces als Abstraktionen mit Stereotyp «realize» weiter. (Hierfür muss eine Instanz der Metaklasse *Stereotype* mit dem Namen 'realize' erzeugt werden.) Außerdem wird der *ownerScope* bei allen Attributen und Operationen auf 'instance' gesetzt.
- Ein Filter, der alle Assoziationen als navigierbar kennzeichnet.
- Ein Filter, der alle Operationen, die den gleichen Namen haben wie die Klasse, zu der sie gehören, mit dem Stereotyp «create» als Konstruktor kennzeichnet.

Beim Aufruf des Werkzeugs werden die Klassennamen der benötigten Filter angegeben. Diese werden dynamisch geladen. Deshalb können leicht Filter für andere Korrekturen implementiert und verwendet werden. Voraussetzung hierfür ist, dass vom Filter nur die Daten benötigt werden, die der Parser liefert.

## 4.3 Ausgabe der Messergebnisse

Die Messergebnisse sollen als HTML-Reports oder für den Import durch ein Tabellenkalkulationsprogramm ausgegeben werden können (vgl. 3.3). Für den Import in eine Tabellenkalkulation können die Messwerte als CSV (Comma Separated Values) ausgegeben werden. Manche Tabellenkalkulationsprogramme (z.B. Microsoft Excel und StarCalc von SUN) können auch HTML-Tabellen importieren. Mit dem HTML-Import besteht auch die Möglichkeit, Formatierungen der Tabellen vorzunehmen.

Bei beiden Report-Arten (CSV und HTML) müssen die Messwerte in eine Textdatei eingefügt werden. Die Reports sollen möglichst flexibel gestaltet werden können. Die Form der auszugebenden Reports kann deshalb über Vorlagen definiert werden. Bei einer Vorlage handelt es sich um eine Textdatei, die Platzhalter für verschiedene Informationen enthält. Beim Erzeugen eines Reports werden diese Platzhalter durch die entsprechenden Ergebnisse ersetzt. Auf diese Weise können die Messergebnisse nicht nur als CSV oder HTML-Dokumente ausgegeben werden, sondern in beliebige textbasierte Dokumente eingebunden werden.

Alle Platzhalter werden durch ein Dollarzeichen “\$” eingeleitet. Sollen Dollarzeichen oder andere Zeichen, die sonst als Teil eines Platzhalters interpretiert würden, im Report vorkommen, so können sie durch Voranstellen eines Backslashes “\” geschützt werden. Folgende Informationen könne in alle drei Arten von Reports eingesetzt werden:

- **Elementname:** Für die Zeichenkette `$(name)` wird bei Klassen- bzw. Paketreports der qualifizierte Name der Klasse bzw. des Pakets eingesetzt. Bei Systemreports wird der Name des Modells eingesetzt.
- **Ein einfacher Funktionswert** für das entsprechende Element. Hierbei steht der Funktionsname als Platzhalter. Funktionsnamen werden in der Form `${<Name>}` angegeben, z.B. `${NAC}` für die Metrik NAC. Folgt dem Platzhalter ein Zeichen, das nicht in Funktionsnamen vorkommen darf (z. B. ein Leerzeichen), so können die geschweiften Klammern entfallen.
- **Eine Liste von Werten.** Bei Funktionen, die eine Liste von Zeichenketten als Funktionswert liefern (vgl. 4.1.4), muss zusätzlich angegeben werden, wie die einzelnen Werte voneinander zu trennen sind. Platzhalter für eine Liste von Werten haben die Form `$(list){<Funktion>}`. Innerhalb der geschweiften Klammern wird angegeben, welche Liste eingesetzt wird und wie die einzelnen Werte auszugeben sind. Hierbei gibt es zwei Möglichkeiten:
  1. Einbetten der einzelnen Werte in einen Text. Innerhalb der geschweiften Klammern steht eine Zeichenkette, in der an der gewünschten Stelle der Funktionsname als Platzhalter steht. Diese Zeichenkette wird für jeden Wert in der Liste einmal ausgegeben, wobei das Listenelement für den Funktionsnamen eingesetzt wird. Eine Ausgabe der Attribute einer Klasse (angenommen, diese werden durch die Funktion `_attributes` geliefert) als Listenpunkte in einem HTML-Dokument erreicht man durch folgende Vorlage:

```
<p>
  <ul>$(list){
    <li>${_attributes}</li>
```

```

    </ul>
</p>

```

Liefert `_attributes` die Werte 'alter', 'gewicht' und 'name', so steht im Report:

```

<p>
  <ul>
    <li>alter</li>
    <li>gewicht</li>
    <li>name</li>
  </ul>
</p>

```

2. Trennen der einzelnen Werte durch eine bestimmte Zeichenkette oder ein einzelnes Zeichen. Ein Trenner kann in der Form `$(list){$<Funktionsname> $<Trenner>}` definiert werden. Sollen die Werte aus dem obigen Beispiel nur durch Kommas getrennt werden, so kann folgender Text in die Vorlage eingefügt werden:

```

<p>
  $(list){$_attributes$,}
</p>

```

Für die Werte 'alter', 'gewicht' und 'name' wird hieraus folgende Ausgabe erzeugt:

```

<p>
  alter, gewicht, name
</p>

```

Die bisher aufgezählten Mechanismen erlauben in Systemreports nur die Ausgabe von Systemfunktionen. Ein Systemreport soll aber alle Messwerte, also auch Klassen- und Paketfunktionen, enthalten können. Hierzu besteht die Möglichkeit, Klassen- oder Paketreports als in Systemreports einzubetten. Klassenreports können auch in Paketreports eingebettet werden. In der Vorlage für den Systemreport wird die Vorlage für den Element-Report als Text eingefügt. Ein eingebetteter Klassenreport kann in einer Vorlage folgendermaßen notiert werden:

```
$(class){<Klassenreport>}
```

Für jede Klasse wird ein Klassenreport erzeugt. Als Vorlage dient die Zeichenkette zwischen den geschweiften Klammern. Die Reports werden nach den Klassennamen alphabetisch geordnet und hintereinander im Systemreport ausgegeben. Durch die folgende Vorlage wird ein Systemreport erzeugt, der für alle Klassen die Ergebnisse für die Metriken NAC und NOC enthält:

```

Name, NAC, NOC
$(class){$(name), $NAC, $NOC}

```

Ein damit erzeugter Report kann z.B. folgendermaßen aussehen:

```
Name, NAC, NOC
Frau, 6, 9
Mann, 5, 8
Person, 3, 6
```

Analog dazu können Klassenreports für alle Interfaces eingesetzt werden:

```
$(interface){<Klassenreport>}
```

Paketreports werden ebenfalls nach den Namen sortiert ausgegeben. Sie werden in der Vorlage folgendermaßen angegeben:

```
$(package){<Paketreport>}
```

## 4.4 Bedienung des Werkzeugs

Reports werden in zwei Schritten generiert:

1. Einlesen eines Modells in die Datenbank
2. Generieren der Reports

Hierfür gibt es die beiden Kommandos 'moosin' und 'moosout'. Zusätzlich gibt es das Kommando 'moose', das beide Schritte auf einmal durchführt. Durch 'moose' können verschiedene Reports mit einem Aufruf generiert werden. Im Folgenden werden die möglichen Aufrufe beschrieben:

Einlesen der Modelle:

```
moosin Modellname XMI-Datei [ -o ] [ -FFiltername ] { -fFiltername }
```

```
moosin -d Modellname
```

```
moosin -l
```

Modellname: Name des Modells in der Datenbank. Bei den Modellnamen wird nicht zwischen Groß- und Kleinschreibung unterschieden.

XMI-Datei: XMI-Datei, die das UML-Modell enthält.

-FFiltername: Der angegebene Filter ersetzt den Standardfilter. Angegeben wird hierbei der Klassenname eines Filters.

-fFiltername: Der angegebene Filter wird als erster Filter der Filter-Pipeline dazugeschaltet. Auch hier wird der Klassenname angegeben.

-o: 'Forced **o**verwrite'. Falls schon ein Modell mit dem angegebenen Namen existiert, werden die alten Daten gelöscht und das neue Modell eingelesen.

-d: Löschen des Modells aus der Datenbank.

-l: Auflistung der gespeicherten Modelle.



# 5 Realisierung

## 5.1 Rahmenbedingungen

### Plattform

Die Entwicklung erfolgte auf einer SUN SPARC Station unter dem Betriebssystem Solaris 5.7.

### Programmiersprache

Die Implementierung erfolgte in der Programmiersprache Java. Die wesentlichen Vorteile von Java für dieses Projekt sind:

- Viele XML-Parser stehen als Java-Pakete kostenlos zur Verfügung.
- Mittels JDBC ist eine einfache Anbindung der Datenbank möglich.
- Java bietet die Möglichkeit, Klassen dynamisch zur Laufzeit zu laden. Dies wird für die flexible Erweiterung der Filter genutzt (vgl. 4.4, Optionen -F und -f).

### Werkzeuge

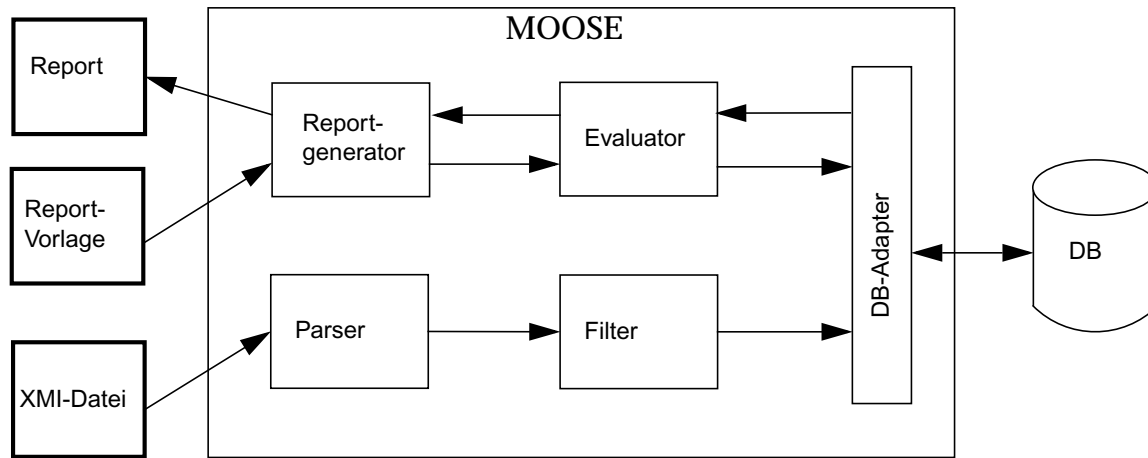
Bei der Realisierung wurden folgende Werkzeuge eingesetzt:

- Als Editor für den Quelltext diente der **GNU Emacs**. Dieser unterstützt Syntaxhervorhebung für Java.
- Für die Versionsverwaltung des Codes und der Dokumentation wurde das **Concurrent Versions System (CVS)** eingesetzt.
- Das **Java 2 Software Development Kit (SDK)** enthält neben der Laufzeitumgebung und einem Compiler weitere Werkzeuge für die Entwicklung mit Java.
- Für die Dokumentation wurde das Textverarbeitungsprogramm **FrameMaker 6.0** verwendet.

## 5.2 Systemüberblick

Die Generierung von Reports erfolgt in zwei Schritten. Zuerst wird ein Modell in die Datenbank eingelesen. Dann wird aus einer Vorlage ein oder mehrere Reports generiert. Hierzu werden die Daten in der Datenbank ausgewertet. Abbildung 18 zeigt die Grobstruktur des Systems und das Zusammenwirken der einzelnen Subsysteme.

Beim Einlesen eines Modells wird die XMI-Datei geparkt. Die geparkten Daten durchlaufen eine Filterpipeline. In der Filterpipeline können werkzeugspezifische Fehler bei der XMI-Generation korrigiert werden (vgl. Kap. 4.2). Von der Filterpipeline gelangen die Daten in den Datenbank-Adapter. Dieser erzeugt aus den Daten Tabelleneinträge in der relationalen Datenbank.



**Abbildung 18: Systemüberblick**

Beim Erzeugen von Reports wird dem Reportgenerator eine Report-Vorlage übergeben. Diese wird geparkt. Für Funktionen, die in der Vorlage vorkommen, wird des Ergebnis vom Evaluator angefordert. Dieser ermittelt die Ergebnistabelle der Funktion, indem er die entsprechende SQL-Anfrage an den Datenbankadapter schickt. Der sendet die SQL-Anfrage an die Datenbank und gibt das Ergebnis an den Evaluator zurück. Vom Evaluator wird das Ergebnis an den Reportgenerator weitergegeben. Von diesem werden die einzelnen Ergebnisse in die Reports eingefügt. Im Folgenden werden die einzelnen Module genauer beschrieben.

## 5.3 Der Parser

Der Parser liest eine XMI-Datei ein und gibt die darin enthaltenen für ODEM relevanten Modellelemente an die Filterpipeline weiter. Dies erfolgt in zwei Schritten:

- Zum einen müssen die lexikalischen Elemente von XMI wie *Tags*, Text usw. erkannt werden. Da XMI auf der Auszeichnungssprache XML basiert, kann hierfür einer der frei verfügbaren XML-Parser verwendet werden.
- Zum anderen müssen anhand der XMI-Sprachelemente die verschiedenen Modellelemente erkannt werden.

### 5.3.1 Auswahl der XML-Parserschnittstelle

Die meisten für Java erhältlichen XML-Parser unterstützen die beiden Standardschnittstellen DOM (Document Object Model, W3C, 2000a) und SAX (Simple API for XML)<sup>1</sup>.

1. <http://www.saxproject.org>

Bei DOM handelt es sich um eine baumbasierte Schnittstelle. Das gesamte XML-Dokument wird zunächst geparkt und in einer internen Baumstruktur abgelegt. DOM definiert die Schnittstelle zum Zugriff auf diesen Baum.

SAX ist eine ereignisbasierte Schnittstelle. Der Parse-Vorgang wird hierbei als eine Folge von Ereignissen betrachtet. Solche Ereignisse können das Lesen von Text, Start- oder Ende-*Tags* usw. sein. SAX definiert die Operationen, die vom Parser bei Eintreten eines Ereignisses aufgerufen werden. Der Zugriff auf die geparkten Daten erfolgt nun durch Implementierung dieser Operationen. Mehr Informationen zu SAX und DOM liefert z.B. McLaughlin (2000).

ODEM umfasst nur einen Teil des UML-Metamodells. Von der komplizierten Darstellung eines UML-Modells in XMI werden daher für ODEM nur wenige Informationen benötigt. Während bei Verwendung von SAX nicht relevante Informationen einfach ignoriert werden können, werden diese bei Verwendung von DOM zu einem großen Teil trotzdem benötigt, um durch den Baum navigieren zu können.

Eine Verwendung von SAX erschien mir daher weniger Implementierungsaufwand zu verursachen. Verwendet wurde die SAX 2.0 Schnittstelle des Xerxes2-Parsers des Apache-XML-Projekts<sup>1</sup>.

### 5.3.2 Analyse der Daten

Um die SAX-Schnittstelle eines Parsers zu verwenden, muss eine Ereignis-Handler implementiert und beim Parser angemeldet werden. Der Ereignis-Handler muss das SAX *ContentHandler*-Interface realisieren. Der Parser ruft dann für jedes Ereignis die entsprechende Operation des Ereignis-Handlers auf.

Das *ContentHandler*-Interface wird bei MOOSE durch die Klasse *ParseController* implementiert. Die Analyse der Daten erfolgt jedoch nicht im *ParseController* selbst. Hierfür werden die Ereignisse an sogenannte Element-Handler weitergemeldet. Es gibt verschiedenen Klassen von Element-Handlern. Ein Element-Handler kann für die Bearbeitung eines Modellelements oder von Teilen eines Modellelements zuständig sein.<sup>2</sup> Alle Element-Handler-Klassen müssen das *ElementHandler*-Interface realisieren. Beim *ElementHandler* handelt es sich um eine abgespeckte Variante des *ContentHandler*-Interface. Es werden nur noch die Ereignisse, die für die Bearbeitung des Modells notwendig sind, weitergegeben. Dies sind:

- Element-Anfang-*Tags*. Hier werden neben dem Namen des *Tags* auch die Attribute weitergegeben.
- Element-Ende-*Tags*
- Einfacher Text

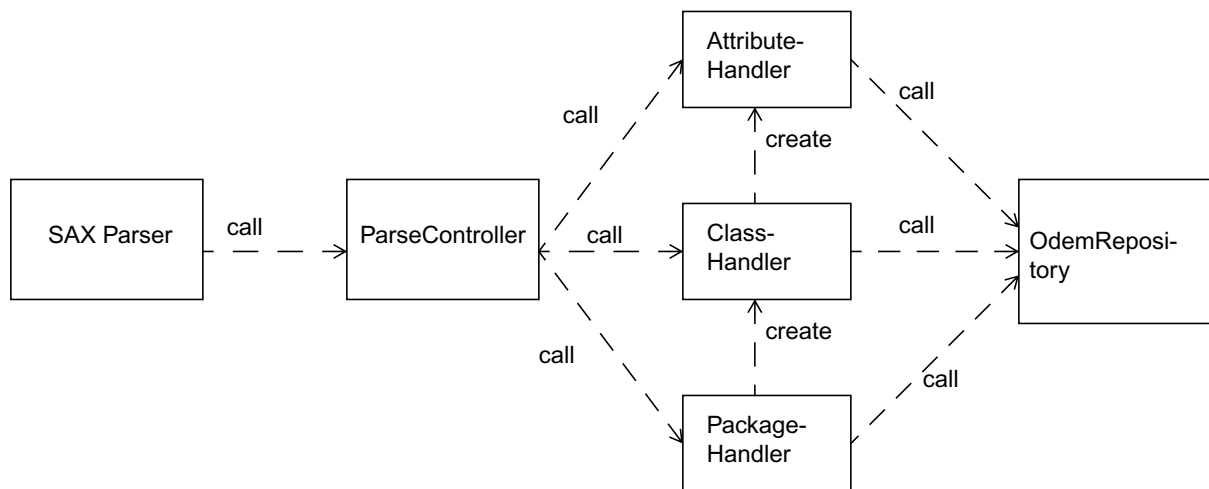
Die Elemente in einem UML-Modell (und somit auch in der XMI-Datei) sind hierarchisch angeordnet. D.h. an oberster Stelle steht das Modell, dieses enthält andere Elemente, die wiederum weitere Elemente enthalten usw. Ein Element-Handler kann daher

---

1. <http://xml.apache.org>

2. Der Begriff Element-Handler bezieht sich hier auf ein XML-Element. In der XMI-Datei sind auch Attribute und Beziehungen von Modellelementen als XML-Elemente beschrieben.

zur Bearbeitung enthaltener Elemente, neue Element-Handler instanziiieren und diese beim *ParseController* anmelden. Der *ParseController* legt dann den alten Handler auf einem Stack ab und meldet die weiteren Ereignisse an den neuen Handler, bis dieser wiederum einen neuen Handler anmeldet oder seine Arbeit beendet hat. Hat ein Element-Handler seine Arbeit beendet, wird der oberste Handler des Stacks zum aktuellen Handler. Als einziger Element-Handler wird der *PackageHandler* vom *ParseController* selbst erzeugt, und zwar wenn beim Parsen der Modellanfang erreicht wird.<sup>1</sup> Abbildung 19 skizziert die Arbeitsweise des Parsers.



**Abbildung 19: Arbeitsweise des Parsers**

Alle Modellelemente, die für ODEM relevant sind, werden an ein *OdemRepository* weitergegeben. *OdemRepository* ist ein Interface, das Operationen zur Weiterleitung aller für ODEM relevanten Informationen besitzt. Außerdem hat *OdemRepository* eine Operation, mit der das Ende des Parse-Vorgangs weitergemeldet wird. In dieser können abschließende Arbeiten durchgeführt werden. Alle Filter und der Datenbankadapter realisieren dieses Interface.

## 5.4 Die Filter

Ein Filter ist eine Klasse, die ein *OdemRepository*-Interface implementiert und die Daten wiederum in ein *OdemRepository* schreibt. Hierbei können Informationen entfernt oder verändert werden. Zusätzlich zu den Operationen von *OdemRepository* haben Filter noch

1. Nach dem UML-Metamodell ist ein Modell selbst ein Paket!

eine weitere Operation, mit der das *OdemRepository* festgelegt wird, in das die Daten ausgegeben werden.

Alle Filter sind Unterklassen von *ElementFilter*. In dieser Klasse sind alle Operationen so implementiert, dass die Daten unverändert weitergeleitet werden. In Unterklassen müssen daher nur Operationen neu implementiert werden, deren Daten im Filter ausgefiltert, verändert oder gespeichert werden.

## 5.5 Die Datenbank

In Kapitel 4.1.2 (Abb. 12, Seite 36) wurde schon das Schema für eine Datenbank zum Speichern einer Entwurfsbeschreibung auf der Basis von ODEM skizziert. Diese Schema wurde so konzipiert, dass eine Übertragung der Metriken in SQL-Anfragen möglichst einfach und nachvollziehbar erfolgen kann. Für jeden Entitätstyp und jede Relation in ODEM wird dabei eine eigene Tabelle angelegt. Die Tabellen für die Entitäten enthalten zusätzlich zu den Attributen aus ODEM noch eine ID als Schlüssel. Hierzu wird die XMI-ID verwendet. Diese dient als eindeutiger Bezeichner eines Modellelements in einer XMI-Datei (vgl. 2.3.2). In der XMI-Datei werden die IDs für Referenzen von Beziehungen auf die beteiligten Modellelemente verwendet. Diese Referenzen können jetzt einfach in die Tabellen für die ODEM-Relationen übernommen werden. Eine genaue Beschreibung der Datenbank liefert Schmider (2002a).

Bisher wurde noch nicht berücksichtigt, dass mehrere Entwürfe in der Datenbank gespeichert werden sollen. Außerdem muss noch geklärt werden, wie die Daten, die nicht direkt aus dem UML-Modell übernommen werden können, beispielsweise die "Stern"-Relationen, realisiert werden.

### 5.5.1 Speichern mehrerer Entwürfe

Da es möglich sein soll, mehrere Entwurfsbeschreibungen gleichzeitig zu speichern, muss für jedes gespeicherte Modellelement festgehalten werden, zu welchem Entwurf es gehört. Hierfür in jeder Tabelle eine extra Spalte einzurichten erscheint nicht sehr praktikabel, da für jede Anfrage die Daten aller Entwürfe durchsucht werden müssen. Das kann verhindert werden, indem für jeden Entwurf eigene Tabellen angelegt werden. Die Kennzeichnung, zu welchem Modell eine Tabelle gehört, erfolgt dabei im Tabellennamen. Allerdings sollen die Anfragen durch den Benutzer nicht für einen bestimmten Entwurf formuliert werden. Dies kann ermöglicht werden, indem das externe Schema der Datenbank komplett durch Sichten realisiert wird, die immer an den gerade zu untersuchenden Entwurf angepasst werden. Diese Anpassung der Sichten kann durch das Werkzeug einfacher durchgeführt werden als eine Anpassung der Anfragen an ein konkretes Modell. Allerdings steht aufgrund der inkrementellen Vorgehensweise bei der Entwicklung (vgl. 6.1) zunächst einmal nur der Zugriff auf die Datenbank "von Hand" zur Verfügung. Hierbei ist das Ändern aller Sichten beim Wechseln des Modells sehr umständlich. Besser wäre daher eine Speicherung jedes Entwurfmodells in einem eigenen Namensraum.

*PostgreSQL* bietet die Möglichkeit, mehrere Datenbanken zu verwalten, wobei jede Datenbank einen eigenen Namensraum besitzt.<sup>1</sup> Mehrere Datenbanken bilden Datenbank-Cluster. Die "Datenbank" für MOOSE wird daher durch mehrere Datenbanken realisiert. Hierbei gibt es eine Hauptdatenbank. Diese enthält Informationen über die gespeicherten Entwürfe. Außerdem werden hier die Definitionen der Funktionen gespeichert. Zusätzlich wird für jeden gespeicherten Entwurf eine so genannte Modell-datenbank angelegt, die das Entwurfsmodell selbst enthält.

### 5.5.2 Zusätzliche Relationen

Zur leichten Definition von Metriken enthält ODEM zusätzliche Relationen. Diese sind von anderen Relationen abgeleitet und enthalten somit keine neuen Informationen. Trotzdem müssen Tabellen für diese Relationen zur Verfügung gestellt werden, um sie bei der Definition der Metriken verwenden zu können. Zur Realisierung dieser Tabellen stehen drei Möglichkeiten zur Auswahl:

- Für die erweiterten Relationen werden echte Tabellen angelegt. Dies führt zur redundanten Speicherung von Daten und somit zu höherem Speicherplatzverbrauch.<sup>2</sup> Allerdings müssen die zusätzlichen Relationen nur ein einziges Mal berechnet werden.
- Realisierung der Tabellen als Sichten (*Views*). Sichten werden für jede Anfrage, in denen sie benutzt werden, neu berechnet und belegen daher keinen zusätzlichen Speicher. Die Bearbeitung der Anfragen ist aber weniger effizient. Zur Berechnung der *extends\**- und der *contains\**-Relation ist ein rekursives Durchlaufen der *extends*- bzw. der *contains*-Tabelle notwendig. Das ist in SQL-Anfragen von *postgreSQL* nicht möglich. Diese beiden Tabellen müssten daher erst auf andere Weise berechnet werden. Alle anderen erweiterten Relationen könnten anschließend als *Views* realisiert werden.
- Ein Mittelweg ist das Anlegen von temporären Tabellen. Diese können bei jedem Start des Werkzeugs einmal automatisch erzeugt werden und bleiben bis zum Programmende erhalten. Zusätzlicher Speicher wird nur während der Ausführung des Werkzeugs benötigt und die Tabellen müssen nicht für jede Anfrage erneut berechnet werden.

Für die Reportgenerierung durch MOOSE erscheint mir die Lösung mit temporären Tabellen am geeignetsten. Allerdings sollen Metriken auch per Ad-Hoc-Anfrage auf der Datenbank erhoben werden können. Hier müssten zuerst die temporären Tabellen "von Hand" erzeugt werden. Um den zusätzlichen Aufwand für die Benutzer zu vermeiden, wird eine andere Lösung gewählt. Die *contains\**- und die *extends\**-Relation werden dauerhaft gespeichert, die anderen erweiterten Relationen werden als Sichten realisiert.

---

1. Eine Datenbank bei *postgreSQL* entspricht einem Katalog des SQL-Standards.

2. Update-Anomalien müssen hier nicht beachtet werden, da ein Modell nicht mehr verändert wird, nachdem es in die Datenbank eingelesen wurde.

### 5.5.3 Aggregatbeziehungen

Die *has*- und *contains*-Relationen in ODEM entsprechen Aggregationen im UML-Meta-modell, d.h. ein Element ist im anderen enthalten. Hierbei handelt es sich um 1:N-Beziehungen, jedes Element ist in nur einem anderen Element enthalten. Es würde daher genügen, für jedes Element den Besitzer in einer zusätzlichen Tabelle zu speichern. Damit würde bei Zugriff auf ein Element über diese Relation einen Verbund (*Join*) gespart. Dies ist zum einen effizienter, zum anderen ist die Anfrage einfacher zu formulieren. Allerdings sollten auf Basis von ODEM formulierte Metriken unter Verwendung der darin vorkommenden Relationen in ein SQL-Anfrage transformiert werden können. Dieser direkte Zusammenhang geht dann jedoch verloren. Es wird daher folgende Lösung gewählt: Aggregatbeziehungen werden als zusätzliche Spalte in den Entitätstabellen realisiert. Zusätzlich werden die expliziten Tabellen für die entsprechenden Relationen als Sichten realisiert. Für die Formulierung der Anfragen stehen beide Konstrukte zur Verfügung.

### 5.5.4 Behandlung von Sonderfällen

Nicht alle Metriken können mit einfache SQL-Anfragen ermittelt werden (vgl. 4.1.3). Für solche Fälle muss zunächst eine Funktion in PL/pgSQL zur Berechnung der Metrik definiert werden. Diese kann dann von SQL-Anfragen verwendet werden, um die Ergebnisse zu ermitteln. Bei MOOSE ist jedoch für die Erhebung immer nur ein Datenbankkommando (nämlich die Anfrage) vorgesehen. Die Datenbankfunktion muss also vorher "von Hand" auf der Datenbank definiert werden.

Bei *postgreSQL* besteht die Möglichkeit, Datenbanken als Kopien von *Templates* zu erzeugen. Dabei werden Tabellen, Datenbankfunktionen usw. kopiert. Für die Modelldatenbanken von MOOSE wurde ein *Template* angelegt. Wird auf diesem Template eine Funktion definiert, so steht sie bei allen anschließend erzeugten Modelldatenbanken zur Verfügung.

### 5.5.5 Der Datenbank-Adapter

Der Datenbank-Adapter besteht aus drei Klassen. In der Klasse *OdemDBConnection* ist die Verbindung zum *postgreSQL* Datenbankserver über JDBC implementiert. Sie besitzt zwei Unterklassen.

Die Klasse *OdemDB* dient zum Zugriff auf die Hauptdatenbank. Über sie wird auf die gespeicherten Funktionen für Metriken und Zusatzinformationen zugegriffen. Außerdem dient sie zur Verwaltung der Modelldatenbanken. Über sie erfolgt das Erzeugen und Löschen von Modelldatenbanken. Indem sie Instanzen der Klasse *ModelDB* erzeugt, regelt sie den Zugriff auf Modelldatenbanken.

Die Klasse *ModelDB* dient zum Zugriff auf Modelldatenbanken. Sie implementiert das *OdemRepository*-Interface. Sie erzeugt aus den übergebenen Modellelementen Einträge in die Modelldatenbank. Die Berechnung der *extends*\*- und der *contains*\*-Relation erfolgt ebenfalls durch *ModelDB*.

## 5.6 Der Evaluator

Der Evaluator dient zur Ermittlung der Funktionsergebnisse für Metriken und Zusatzinformationen. Eine Funktion wird repräsentiert durch ein Funktionsobjekt. Diese Objekte sind klassifiziert nach ihrem Bereich (Klasse, Paket, System) und nach ihrem Rückgabewert (einzelner Wert, Liste). Hieraus ergeben sich sechs verschiedene Klassen von Funktionen. Gemeinsame Oberklasse aller Funktionen ist die Klasse *Function*. Auf die einzelnen Funktionsobjekte im Evaluator wird über die Funktionsnamen zugegriffen. Wird ein Funktionsobjekt angefordert, so greift der Evaluator über den Datenbank-Adapter auf die Funktionsdefinitionen in der Hauptdatenbank zu und erzeugt ein neues Funktionsobjekt. Die Objekte werden im Evaluator gespeichert, so dass bei einem zweiten Zugriff auf eine Funktion nicht mehr auf die Datenbank zugegriffen werden muss.

Jedes Funktionsobjekt besitzt eine Operation zur Ermittlung des Ergebnisses. Diese Operation erhält als Parameter den Namen des Modells, auf dem das Ergebnis ermittelt werden soll. Zur Ermittlung der Ergebnisse greift das Objekt über den Datenbank-Adapter auf die entsprechende Modelldatenbank zu. Bei Paket- und Klassenfunktionen wird das Ergebnis in Form einer Hashtabelle geliefert, bei der über den Namen eines Elements auf dessen Funktionswert zugegriffen werden kann. Bei Systemfunktionen wird ein String oder eine Liste als Ergebnis geliefert.

## 5.7 Der Reportgenerator

Der Reportgenerator erhält als Eingabe eine Report-Vorlage und erzeugt daraus einen Systemreport oder mehrere Paket- oder Klassenreports. Als Schnittstelle zur Erzeugung von Reports dient die Klasse *ReportGenerator*. Ein *ReportGenerator* besitzt drei Operationen zur Ausgabe der drei verschiedenen Report-Arten (Klasse, Paket, System). Diese erhalten als Parameter jeweils den Namen eines Entwurfsmodells, den Dateinamen der Vorlage und den Pfad für die Ausgabe. Bei Klassen- und Paketreports wird für jede Klasse bzw. jedes Paket ein Report ausgegeben. Der Dateiname eines Reports entspricht dem Klassen- bzw. dem Paketnamen.

Der Reportgenerator kann als Interpret arbeiten. Die Vorlage wird geparkt, einfacher Text wird dabei sofort in den Report geschrieben. Für die Platzhalter wird das Ergebnis mit Hilfe des Evaluators ermittelt und dann ebenfalls sofort in den Report geschrieben. Hierbei muss aber z.B. eine Vorlage für einen Klassenreport für jede Klasse erneut interpretiert werden. (Eine gleichzeitige Ausgabe aller Reports ist nicht möglich, wenn diese in einen Systemreport eingebettet sind.) Es bietet sich daher an, eine einmal bekannte Struktur einer Vorlage zu speichern. Eine Report-Vorlage wird daher zuerst geparkt und anschließend ausgegeben.

Aus einer Report-Vorlage wird beim Parsen eine Art Syntaxbaum aufgebaut. Funktionsergebnisse werden beim Parsen über den Evaluators ermittelt und in der Baumstruktur gespeichert. Nach Beendigung des Parsens stößt der *ReportGenerator* die Ausgabe des Baums an.

Die Wurzel des Baums für einen Systemreport ist ein Objekt der Klasse *SystemReportWriter* (vgl. Abb. 20). Ein *SystemReportWriter* besitzt die Operation *write*. Diese erhält als Parameter ein Dateiojekt, in das der Systemreport ausgegeben wird.

Eine Report-Vorlage besteht im Prinzip abwechselnd aus "normalen" Zeichenketten und Platzhaltern. Im *SystemReportWriter* werden die Platzhalter durch *SystemFunctionWriter* repräsentiert. Ein *SystemFunctionWriter* hat eine Operation *write*, mit der der für den Platzhalter generierte Text ausgegeben wird. Als Parameter erhält *write* die Ausgabedatei. Ein *SystemReportWriter* gibt einen Systemreport aus, indem er abwechselnd Text in den Report schreibt und die Operation *write* eines *SystemFunctionWriters* aufruft.

*SystemFunctionWriter* ist ein Interface. Hinter einem *SystemFunctionWriter* können sich Objekte zur Ausgabe eines einzelnen Wertes, einer Liste oder von Klassen- oder Paketreports verbergen.

Wurzel für Klassen- und Paketreports ist ein Objekt der Klasse *ElementReportWriter*. Bei einem *ElementReportWriter* wird der Operation *write* zusätzlich zur Ausgabedatei der Name des Elements (Klasse oder Paket) übergeben, für das der Report ausgegeben werden soll. Die Klassen für die Platzhalter in Element-Reports implementieren das *ElementFunctionWriter*-Interface. Analog zum *ElementReportWriter* wird auch bei ihnen der Operation *write* ein Elementname übergeben. Sollen nur Klassen- oder Paketreports generiert werden, so können nun die Reports für verschiedene Klassen bzw. Pakete in verschiedenen Dateien ausgegeben werden.

Um Klassen- und Paketreports auch in einen Systemreport einbinden zu können, gibt es die Klasse *ElementReportWrapper* (vgl. Abb. 20). *ElementReportWrapper* implementiert das *SystemFunctionWriter*-Interface. Ein *ElementReportWrapper* ist mit einem *ElementReportWriter* assoziiert und besitzt eine Liste mit Elementen. Wird seine Operation *write* aufgerufen, so gibt er für jedes Element in der Liste einen Report in die angegebene Datei aus.

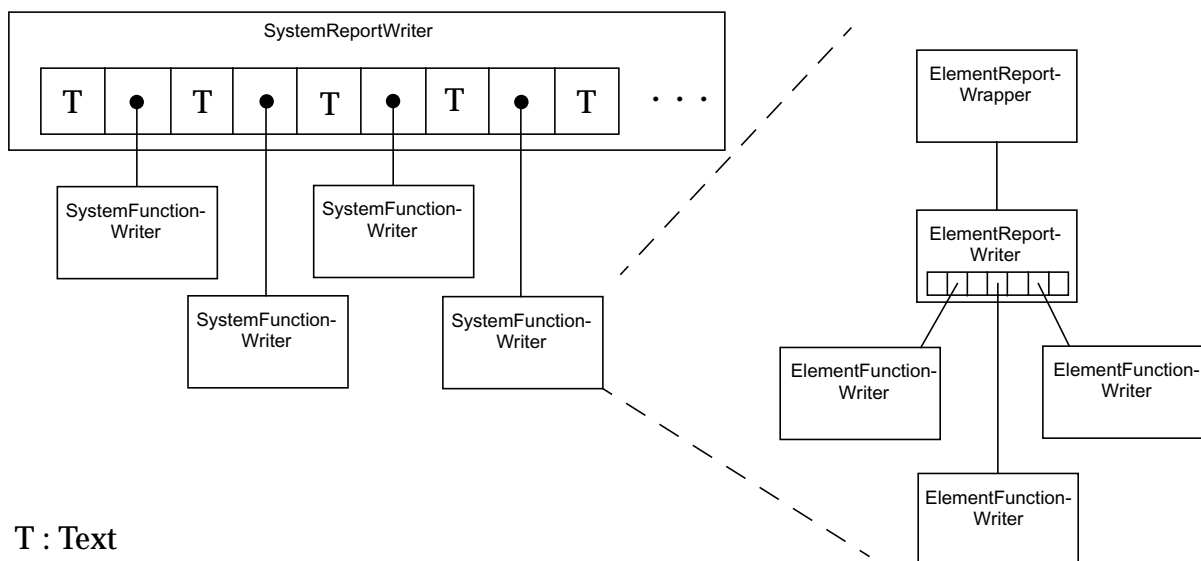


Abbildung 20: Baumdarstellung eines Systemreports

## 5.8 Bewertung der Implementierung

### 5.8.1 Größe

Tabelle 2 gibt einen Überblick über die Größe der Implementierung von MOOSE.

	Klassen	Inter- faces	Attribute	Operati- onen	Code- Zeilen	Gesamt- Zeilen
Parser	17	3	71	49	1140	2166
Filter	5	0	28	15	399	716
DB-Adapter	3	0	17	50	1581	2337
Evaluator	10	0	16	35	385	651
Reportgenerator	12	2	42	46	1003	1512
Sonstige	3	0	3	10	181	268
<b>Gesamt</b>	<b>50</b>	<b>5</b>	<b>177</b>	<b>205</b>	<b>4689</b>	<b>7650</b>

**Tabelle 2: Größe der Implementierung**

Gezählt werden die Anzahl der Klassen, Interfaces, Attribute, Operationen und die Anzahl der Zeilen der einzelnen Subsysteme. Unter Sonstige sind die Start-Klassen für die beiden Einzelkommandos und eine Klasse, die die Ausgabe von Meldungen kontrolliert, zusammengefasst. Vererbte Operationen wurden nur einmal gezählt, auch wenn sie in der Unterklasse neu implementiert wurden. Bei der Anzahl der Code-Zeilen wurden alle Zeilen gezählt, die nicht nur Kommentare oder Leerzeichen enthielten. Bei der Gesamtzahl der Zeilen wurden sowohl Kommentarzeilen als auch Leerzeilen berücksichtigt.

Um für die folgenden Messungen größere Entwürfe zu erhalten, wurden diese aus mehreren Kopien des Entwurfs von MOOSE zusammengesetzt. Der Entwurf E1 entspricht dem Entwurf von MOOSE, der Entwurf E2 enthält diesen Entwurf zweimal usw. Bei E0 handelt es sich um einen Entwurf ohne Modellelemente.

### 5.8.2 Laufzeit

Tabelle 3 zeigt die Laufzeit von MOOSE für verschiedene Aufgaben in Minuten. Da die Messwerte für das Einlesen eines Entwurfs in die Datenbank stark durch die Ausgabe von Informationen auf den Bildschirm beeinflusst wird, wurden diese bei einer zweiten Messung in eine Datei umgeleitet. Weiterhin wurde die Dauer für die Generierung eines Systemreports aus der ersten Verfeinerungsstufe von QOOD und eines Klassenreports aus QOOD gemessen. In einem Klassenreport wird jeweils eine Liste der Attribute und der Operationen im UML-Stil ausgegeben (vgl. 3.3.2). Die einzelnen Listeneinträge werden mit Hilfe von PL/pgSQL-Funktionen erzeugt. Da bei jedem Funktionsaufruf mit einer SQL-Anfrage auf Tabellen zugegriffen werden muss, verbrauchen diese Funktio-

nen viele Ressourcen. Zum Vergleich wurden noch Klassenreports ohne Ausgabe dieser Listen erstellt.

	E0	E1	E2	E4	E8
Einlesen in DB	0.04	0.18	0.25	0.46	1.34
Einlesen in DB (Log in Datei)	0.04	0.12	0.18	0.32	1.06
Systemreport	0.03	0.07	0.12	0.19	0.38
Klassenreport	0.08	0.46	1.42	4.33	13.28
Klassenreport (ohne DB-Funk- tionen)	0.08	0.32	0.59	1.56	4.04

**Tabelle 3: Laufzeit von MOOSE**

### 5.8.3 Speicherplatzbedarf

#### Dynamisch

Tabelle 4 zeigt den Hauptspeicherplatzbedarf zur Ermittlung der Resultate beim Gene-

	E0	E1	E2	E4	E8
Systemreport	5,8	7,9	8,2	10	14
Klassenreport	7	8	9,4	12	15
Klassenreport (ohne DB-Funk- tionen)	6,6	7,9	8,3	10	14

**Tabelle 4: Hauptspeicherbedarf für Zugriff auf Modelldatenbank**

rieren der Reports in Megabyte. Aufgeführt ist hier nur der Speicherplatzbedarf für die Verbindung zur Modelldatenbank. Die Verbindung zur Hauptdatenbank benötigt weitere 5,8 Megabyte. Hinzu kommen 41 MB für MOOSE selbst. Allerdings werden 40 MB schon von der *Java Virtual Machine* verbraucht, unabhängig vom Speicherplatz des Programms selbst. Für das Füllen der Datenbank wurden für alle Entwürfe 43 MB Hauptspeicher für MOOSE und 5,8 MB für den Datenbankserver ermittelt.

## Statisch

Tabelle 5 zeigt den Speicherplatzverbrauch der verschiedenen Entwürfe in der Datenbank.

Entwurf	E0	E1	E2	E4	E8
Speicherplatz in Kilobyte	1971	2251	2443	2867	3699

**Tabelle 5: Speicherplatzverbrauch der Modelldatenbank**

## 5.9 Erprobung mit QOOD

Die Messergebnisse für QOOD können in drei verschiedenen Verfeinerungsstufen betrachtet werden (vgl. 3.3.2). Die Ergebnisse der ersten Verfeinerungsstufe werden alle in einem Systemreport ausgegeben. In der zweiten Verfeinerungsstufe wird für jedes der Kriterien Knappheit (*Conciseness*), Strukturiertheit (*Structuredness*) und Entkopplung (*Decoupling*) ein separater Systemreport ausgegeben. In der dritten Stufe wird für jede Klasse und jedes Paket ein Report ausgegeben. Für die Systemmetriken gibt es zusätzlich einen Systemreport. Insgesamt müssen für QOOD 227 verschiedene Metriken und 26 Zusatzinformationen ermittelt werden (vgl. Tabelle 6).

Bereich	Metriken	Zusatz- informationen
Klasse	154	19
Paket	29	7
System	44	0

**Tabelle 6: Anzahl der Funktionen für QOOD**

Die Ergebnisse wurden zunächst als CSV ausgegeben und durch die Tabellenkalkulation StarCalc von SUN importiert. Leider gibt es bei StarCalc nicht die Möglichkeit, Daten aus einer CSV-Datei in eine speziell formatierte Vorlage zu importieren. Abbildung 21 zeigt einen Klassenreport nach dem Import als CSV. Sollen die Ergebnisse formatiert dargestellt werden, so muss jeder einzelne Report in der Tabellenkalkulation nachbearbeitet werden. StarCalc kann aber auch HTML-Tabellen importieren. Somit ist es möglich, die Darstellung in der Tabelle durch die Report-Vorlage zu bestimmen. Die Reports wurden deshalb auch als HTML-Tabellen ausgegeben. Abbildung 22 zeigt einen Klassenreport als HTML-Tabelle. Allerdings wurden manche Werte nach dem Import durch StarCalc in den falschen Zellen dargestellt (vgl. Abb. 23). Da ich bei Betrachtung des HTML-Quellcodes der Reports keinen Grund für diesen Fehler feststellen konnte, wurden die Reports zum Vergleich auch durch Microsoft Excel importiert. In Excel wurden die Tabellen ohne Fehler dargestellt (vgl. Abb. 24).



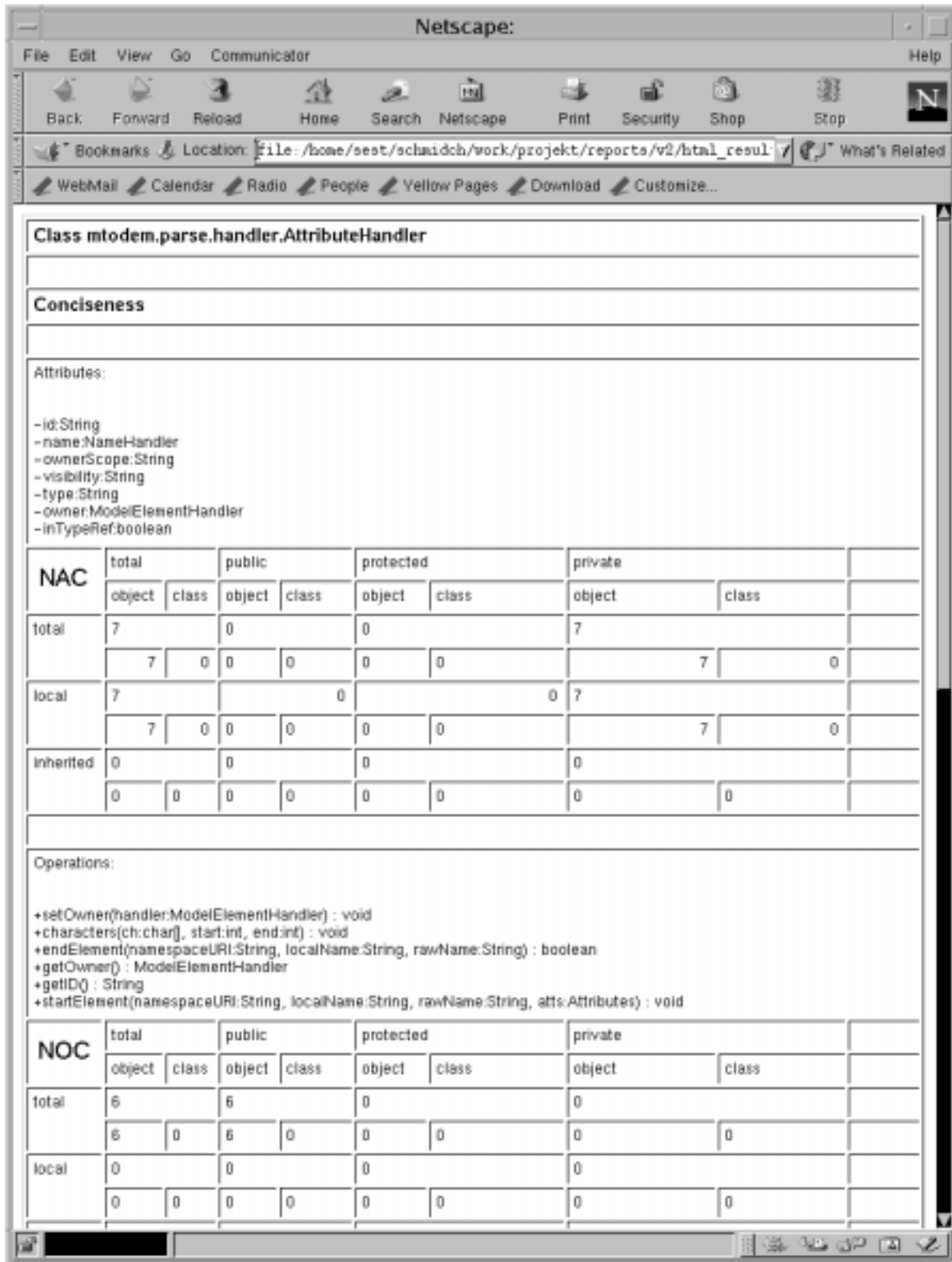


Abbildung 22: Report als HTML-Tabelle

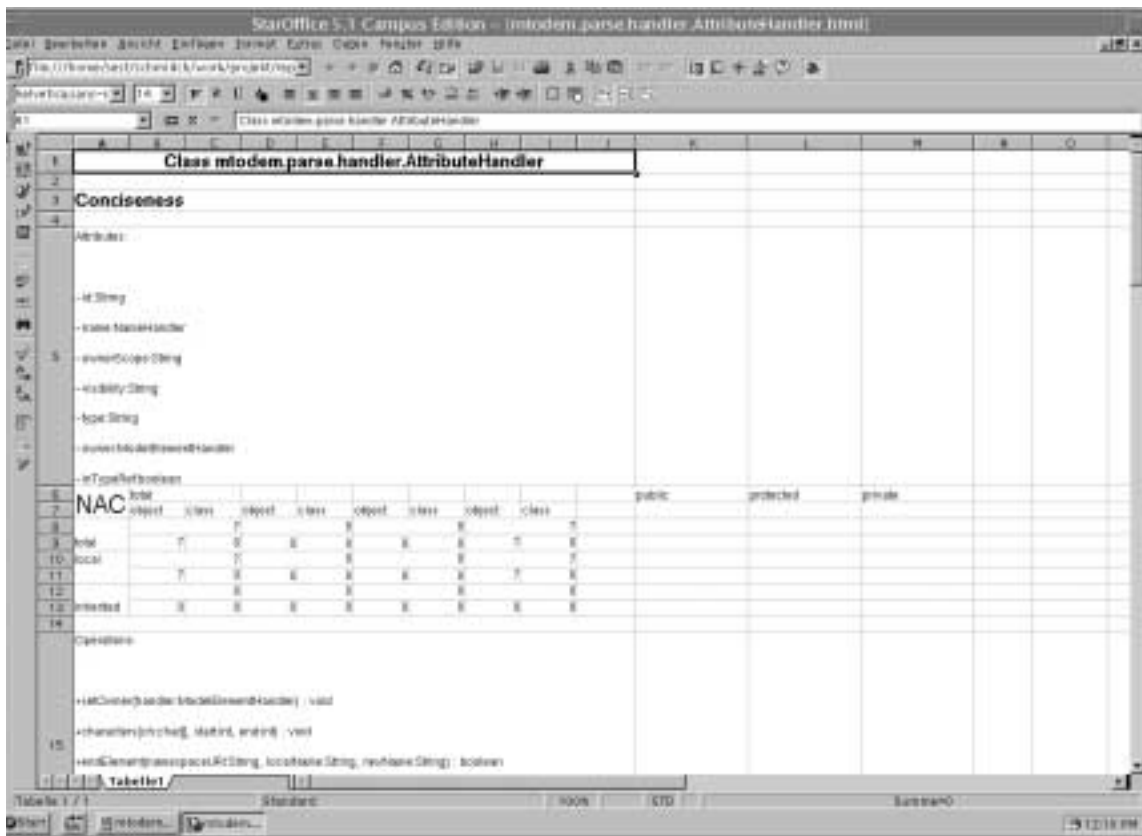


Abbildung 23: Import einer HTML-Tabelle durch StarCalc

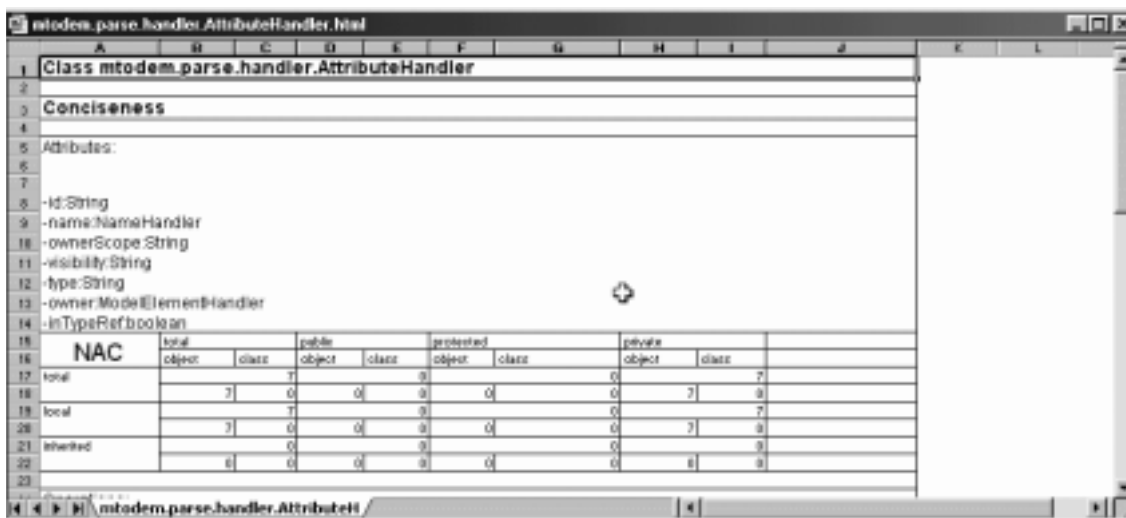


Abbildung 24: Import einer HTML-Tabelle durch Microsoft Excel

# 6 Zusammenfassung und Ausblick

Diese Kapitel enthält einen Rückblick auf den Projektverlauf, eine Zusammenfassung der Ergebnisse und einen Ausblick auf die mögliche weitere Entwicklung des Werkzeugs MOOSE.

## 6.1 Projektverlauf

Das Projekt “Konzeption und Realisierung eines Metrikenwerkzeugs für die UML“ wurde in die groben Phasen

- Projektplanung
- Einarbeitung
- Entwicklung
- Bericht

eingeteilt. Die Anforderungen an das Metrikenwerkzeug waren zu Projektbeginn noch nicht vollständig klar. Mit der Entwicklung von MOOSE sollte erst geklärt werden, ob und inwieweit ein Werkzeug mit den geforderten Eigenschaften überhaupt realisierbar ist. Deshalb wurde bei der Entwicklung inkrementell vorgegangen. Zuerst wurde ein funktionsfähiges System mit eingeschränktem Leistungsumfang geschaffen. Dies wurde in weiteren Entwicklungsphasen schrittweise zu seinem vollen Leistungsumfang erweitert. Die Anforderungen und die Zeitplanung für jede der Entwicklungsphasen wurden jeweils am Ende der vorgehenden Phase festgelegt. Durch diese Vorgehensweise war es möglich, während einer Entwicklungsphase gewonnene Erfahrungen bei der weiteren Planung zu berücksichtigen.

### 6.1.1 Tätigkeiten

#### Projektplanung

Hier wurden die grobe Zeitplanung und die Rahmenbedingungen für das Projekt festgelegt. Eine Zeitplanung für die einzelnen Entwicklungsphasen wurde erst zu Beginn der jeweiligen Phase vorgenommen. Die Ergebnisse der Planung wurden in einem Projektplan gemäß dem IEEE-Standard (IEEE, 1987) festgehalten.

#### Einarbeitung

Die Einarbeitungsphase umfasste die Einarbeitung in ODEM, das UML-Metamodell und XMI. Außerdem wurde in dieser Phase nach Werkzeugen mit XMI-Unterstützung recherchiert. Zum einen interessierte, welche Werkzeuge XMI generieren können. Zum anderen wurde nach Werkzeugen gesucht, die Metriken auf UML-Modellen in XMI-Darstellung erheben können. Ebenfalls Bestandteil dieser Phase war die Suche nach wiederverwertbaren Komponenten, wie z.B. einem XML-Parser.

## Entwicklung

Die Entwicklung wurde in drei Phasen durchgeführt:

1. **Einlesen einer UML-Modellbeschreibung in die Datenbank.** In dieser Phase wurde die Datenbank entworfen, in der die Modellbeschreibungen gespeichert werden. Außerdem wurde das Werkzeug MOOSE so weit entwickelt, dass es eine XMI-Datei für ein UML-Modell parsen und die für ODEM relevanten Daten in der Datenbank speichern kann.
2. **Formulierung der SQL-Anfragen für QOOD.** In dieser Phase wurden die SQL-Anfragen für die Metriken in QOOD formuliert.
3. **Generieren der Reports.** In dieser Phase wurde MOOSE mit der Fähigkeit Reports zu generieren erweitert. Außerdem wurden in dieser Phase die Report-Vorlagen für QOOD erstellt und SQL-Anfragen formuliert, um die Zusatzinformationen für QOOD zu ermitteln.

Während der ersten Entwicklungsphase wurden die folgenden Dokumente erstellt und in den darauf folgenden Phasen gepflegt:

- Spezifikation
- Entwurfsbeschreibung
- Beschreibung der Datenbank

Während der zweiten und dritten Phase wurde ein weiteres Dokument erstellt, das die Konfiguration von MOOSE für QOOD beschreibt.

Die einzelnen Entwicklungsphasen umfassten die Teilschritte:

- Klärung der Anforderungen
- Entwurf
- Implementierung
- Test

Da das Werkzeug prototypisch implementiert werden sollte, wurde nur sehr eingeschränkt getestet. Anders wäre die gewünschte Funktionalität des Prototyps in der gegebenen Zeit nicht realisierbar gewesen. In der ersten Entwicklungsphase wurde dennoch systematisch getestet. In der zweiten Phase beschränkte sich die Prüfung auf ein "Ausprobieren" aller Anfragen. Allerdings sind die SQL-Anfragen auch nicht Teil des Werkzeugs, sondern der Konfiguration für QOOD. Die Tests und die Ergebnisse in der dritten Phase wurden nicht festgehalten. Es wurden lediglich Wegwerftests durchgeführt. Alle bei den Tests festgestellten Fehler wurden behoben.

## Bericht

Nach der Realisierung wurde dieser Bericht verfasst. Er enthält eine Zusammenfassung der Ergebnisse dieser Arbeit.

### 6.1.2 Termine und Aufwand

Tabelle 7 zeigt die einzelnen Tätigkeiten mit ihren geplanten und den tatsächlichen End-

Tätigkeit	Soll-Termin	Ist-Termin
Projektplanung	17.07.01	13.07.01
Einarbeitung	31.07.01	27.07.01
Entwicklung des Werkzeugs	02.11.01	02.11.01
Datenbank	24.08.01	14.09.01
Metriken für QOOD	02.10.01	05.10.01
Reportgenerator	02.11.01	02.11.01
Erstellen des Berichts	21.12.01	23.01.02
Vorbereitung des Zwischen- vortrags	02.10.01	02.10.01
Vorbereitung des Abschluss- vortrags	noch unbekannt	noch unbekannt

**Tabelle 7: Terminplanung**

terminen. Projektbeginn war am 10.07.01. Vom 30.07.01 bis 03.08.01 wurde eine Woche Urlaub eingeplant. Der Gesamtaufwand des Projekts wurde mit 23 Wochen veranschlagt, wobei der Aufwand für eine Woche mit 35 Stunden festgelegt wurde. Daraus ergibt sich ein Gesamtaufwand für das Projekt von 805 Stunden. Abbildung 25 zeigt den Soll- und den Ist-Aufwand für die einzelnen Tätigkeiten in Stunden. Der Abschlussvortrag (Aufwand: 1 Woche) ist dort nicht erfasst. Deshalb wird nur ein Gesamtaufwand von 770 Stunden ausgewiesen. Im Diagramm fällt auf, dass die Summe der Soll-Stunden für die einzelnen Entwicklungsschritte nicht dem Gesamtsoll für die Entwicklung entsprechen. Dies liegt daran, dass der Inhalt und Aufwand jeder Entwicklungsphase erst am Ende der vorhergehenden Phase geplant wurde. So wurden Terminüberschreitungen der vorhergehenden Phase in der weiteren Planung berücksichtigt. Bei der Betrachtung von Tabelle 7 fallen zwei starke Terminüberschreitungen auf:

1. Bei der ersten Entwicklungsphase wurde der Termin um drei Wochen überschritten. Dies macht sich auch beim Aufwand bemerkbar. Es wurden 183% des geplanten Aufwands benötigt. Dies hatte mehrere Ursachen. Zum einen wurde der zusätzliche Aufwand durch die Behandlung von fehlerhaften XMI-Dateien verursacht. Zum anderen wurde die Schätzung für diese Phase auch zu optimistisch vorgenommen. Allerdings wurde am Anfang der Entwicklung auch versucht, eher zu optimistisch als zu pessimistisch zu schätzen, da Terminüberschreitungen bei späteren Entwicklungsphasen noch ausgeglichen werden konnten. Wird ein Termin jedoch zu spät

angesetzt, so besteht die Gefahr, dass die Zeit trotzdem verbraucht wird und später fehlt.

- Das Schreiben des Berichts dauerte über einen Monat länger als geplant. Auch für den Bericht war der Aufwand höher als eingeplant, wurde jedoch nur um 15% überschritten. Der wesentliche Grund für die Terminüberschreitung lag hier im privaten Bereich. Ich hatte die Geburt meiner Tochter Lena nicht ausreichend in der Terminplanung meiner Diplomarbeit berücksichtigt.

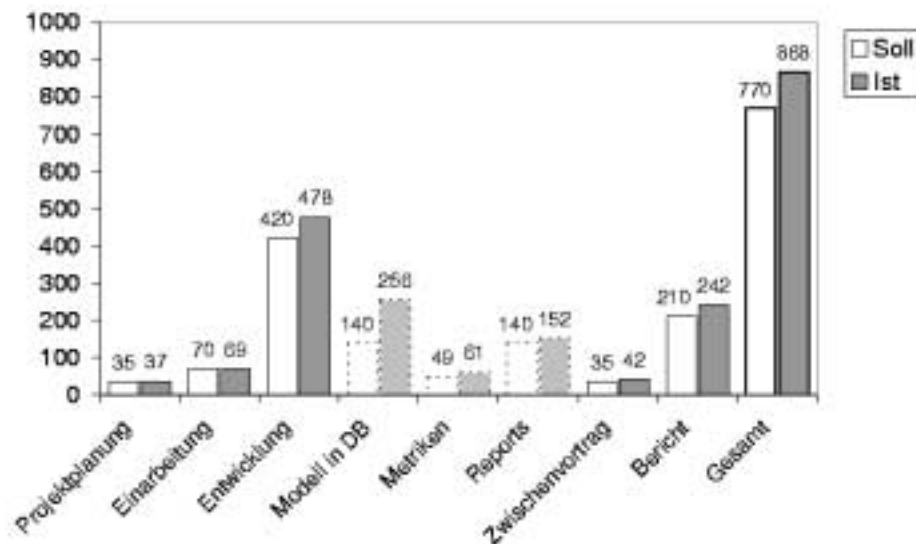


Abbildung 25: Aufwand

## 6.2 Ergebnisse

Mit dem Werkzeug MOOSE ist es möglich, Messungen schon auf Entwurfsbeschreibungen durchzuführen. Da die Eingabe des UML-Modells in Form eines XMI-Dokuments erfolgt, ist MOOSE unabhängig von einem bestimmten UML-Werkzeug. Durch die mögliche Konfiguration der Metriken, die erhoben werden können, kann es auch an individuelle Qualitätsvorstellungen angepasst werden. Für die Ausgabe der Metriken bietet MOOSE die Möglichkeit, die Messergebnisse flexibel in textbasierte Dokumente (z.B. ein HTML-Dokument) einzubinden. Dadurch kann MOOSE für verschiedene Anwendungen konfiguriert werden.

In der Aufgabenstellung wurde verlangt, dass das Werkzeug beliebige Metriken, die auf der Basis von ODEM definiert sind, erheben kann. Mit *postgreSQL* können zwar nicht alle Metriken durch SQL-Anfragen realisiert werden. *PostgreSQL* stellt mit *PL/pgSQL* eine berechenbarkeitsvollständige Programmiersprache zur Verfügung. Diese kann MOOSE zur Definition von Funktionen für solche Sonderfälle benutzt werden. Dabei ist in der

Regel ein größerer Aufwand verbunden als mit der Definition einer SQL-Anfrage, und die Berechnung der Ergebnisse ist auch weniger effizient. Aber es können so alle Werte, die auf Basis der Daten in ODEM berechenbar sind, ermittelt werden.

Das Werkzeug sollte auf SUN SPARC Stations unter dem Betriebssystem Solaris benutzt werden können. Da die Implementierung mit Java erfolgte und auch *postgreSQL* für viele gängige Plattformen erhältlich ist, sollte es keine Probleme bereiten, MOOSE z.B. auch unter Windows oder Linux zu benutzen.

Das Werkzeug MOOSE wurde erprobt, indem es zur Erhebung der Metriken in QOOD verwendet wurde. Die Erprobung verlief erfolgreich. Alle für QOOD erforderlichen Informationen konnten ermittelt und in eine Tabellenkalkulation importiert werden.

Bei der Implementierung bestehen die folgenden Einschränkungen gegenüber den Anforderungen und der Konzeption:

- Templates werden nicht aussortiert (vgl. 3.1.1). Da für sie keine Sonderbehandlung implementiert wurde, tauchen sie als "normale" Modellelemente auf. Die Templateparameter fallen weg. Auch die Dummy-Elemente, die den Typ des Templateparameters angeben, tauchen als Modellelemente auf. Da keines der im Projekt benutzten Werkzeuge Templates unterstützt, war diese Einschränkung bisher nicht von Bedeutung.
- Es können keine Klassen- in Paketreports eingebunden werden (vgl. 4.3). Diese Option wurde für QOOD nicht benötigt. Eine nachträgliche Erweiterung ist jedoch einfach.

Einige Einschränkungen für die Brauchbarkeit von MOOSE ergeben sich schon aus der Aufgabenstellung:

### **Einschränkungen aufgrund von ODEM**

Von MOOSE können nur die Informationen aus einem UML-Modell ermittelt werden, die von ODEM abgedeckt sind. Die Abbildung auf ODEM-Elemente erfolgt nur für einige der nicht berücksichtigten Modellelemente. Dabei gehen Informationen über besondere Eigenschaften dieser Elemente verloren. Vor allem die Nichtberücksichtigung von Templates wiegt hier schwer, da diese durch die Wiederverwendung von Code die Qualität eines Entwurfs wesentlich beeinflussen können.

### **Einschränkungen aufgrund von XMI**

Obwohl viele UML-Werkzeuge in der Lage sind, Modelle als XMI-Dateien zu exportieren, so bietet doch die Mehrheit der Werkzeuge keine XMI-Unterstützung. Auch bei Werkzeugen mit XMI-Unterstützung können Fehler beim XMI-Export zur Verfälschung der Messergebnisse führen (vgl. 2.3.3). Fehler beim XMI-Export von UML-Werkzeugen können von MOOSE durch Filter korrigiert werden. Allerdings müssen diese Fehler zuerst festgestellt und die passenden Filter implementiert werden. Dies erfordert für jedes Werkzeug eine Analyse des generierten XMI. Wenn beim XMI-Export Informationen völlig verloren gehen, so können diese Fehler auch durch Filter nicht behoben werden.

### **Einschränkungen aufgrund unvollständiger Diagramme**

Bei der Bewertung von Entwürfen ist man darauf angewiesen, dass vom Entwerfer auch wirklich alle Elemente für die Messungen relevanten Elemente des Entwurfs eingetragen werden. Gerade Abhängigkeiten werden in UML-Modellen oft weggelassen oder sind nur implizit z. B. durch den Typ eines Parameters gegeben. Auch bei Assoziationen kann es vorkommen, dass statt dessen Attribute eingetragen werden. MOOSE kann aber nur die Daten erfassen, die vom Entwerfer explizit angegeben wurden.

### **Einschränkungen aufgrund von Tabellenkalkulationsprogrammen**

Werden die Messwerte als CSV in eine Tabellenkalkulation übernommen, so stehen keine Möglichkeiten zur Verfügung, eine Formatierung der Daten vorzugeben. HTML-Tabellen können jedoch nicht durch jedes Tabellenkalkulationsprogramm importiert werden.

Einige Einschränkungen müssen in Kauf genommen werden, wenn Entwürfe bewertet werden sollen, zu denen noch keine Implementierung besteht. Dies ist jedoch immer noch besser, als auf eine Qualitätssicherung in den frühen Entwurfsphasen zu verzichten.

## **6.3 Ausblick**

Außer der Implementierung der noch fehlenden Funktionen sind eine Reihe weiterer Änderungen an MOOSE in der Zukunft erforderlich oder wünschenswert:

- Die Entwicklung von MOOSE erfolgte basierend auf der UML-Spezifikation Version 1.3. Teil der UML-Spezifikation ist eine DTD für die Darstellung von UML-Modellen in XMI (vgl. 2.3.2). Diese basiert auf der XMI-Spezifikation Version 1.0. Der XMI-Export der im Projekt verwendeten Werkzeuge basierte ebenfalls auf der UML-Spezifikation 1.3. Seit September 2001 ist die Version 1.4 der UML-Spezifikation freigegeben (OMG, 2001). In dieser Version gibt es z.B. 'package' als vierten möglichen Sichtbarkeitsbereich. Außerdem nutzt die DTD für XMI aus dieser UML-Spezifikation die Möglichkeit, XML-Namensräume zu benutzen. Diese Möglichkeit besteht erst seit der XMI-Version 1.1 (OMG, 2000d). Eine XMI-Datei auf Basis dieser DTD kann von MOOSE nicht verarbeitet werden. MOOSE muss daher an die neuen Gegebenheiten angepasst werden.
- In einem UML-Entwurf müssen alle Abhängigkeiten zwischen Klassen/Interfaces explizit eingetragen werden, um von MOOSE erkannt zu werden. Einige Abhängigkeiten sind jedoch schon implizit im Modell vorhanden, z.B. durch die Verwendung einer Klasse als Typ eines Operationsparameters. Solche Abhängigkeiten könnten automatisch erkannt werden.
- Wünschenswert ist auch die Berücksichtigung der Sonderstellung von Templates. Hierfür müsste aber zunächst die Grundlage durch eine Erweiterung von ODEM geschaffen werden.
- Bisher wurde MOOSE nur benutzt, um Metriken für die Kriterien Knappheit, Strukturiertheit und Entkopplung von QOOD zu ermitteln. Im Zusammenhang mit

QOOD ergeben sich jedoch noch weitere Möglichkeiten, MOOSE zu nutzen. In QOOD werden Bewertungen von Kriterien, für die keine messbaren Attribute bestehen, mit Hilfe von Fragebögen ermittelt. Teilweise können die Fragen anhand der in ODEM enthaltenen Informationen beantwortet werden. Die Fragebögen können beispielsweise als HTML-Dokumente realisiert werden, bei denen die Antworten auf diese Fragen von MOOSE eingefügt werden.

## **6.4 Fazit**

Die Konzeption von MOOSE erscheint mir insgesamt gelungen. Durch die flexible Konfiguration der Metriken und der Ausgabe der Ergebnisse kann das Werkzeug an individuelle Erfordernisse angepasst werden. Wird MOOSE weiterentwickelt und gepflegt, so kann aus der prototypischen Implementierung ein Werkzeug entstehen, das auch in der Praxis mit großem Nutzen verwendet werden kann.

# Literaturverzeichnis

- Booch, G., Rumbough, J., Jacobson, I. (1999): **The Unified Modeling Language User Guide**, Addison-Wesley, Reading.
- Brodsky, S. (1999): **XMI Opens Application Interchange**, IBM,  
<http://www-3.ibm.com/software/ad/library/standards/xmi.html>
- Date, C.J. (2000): **An Introduction to Database Systems**, 7. Edition, Addison-Wesley, Reading.
- Dicken, H., Hipper G., Müßig-Trapp (2000): **Datenbanken unter Linux**, MITP-Verlag, Bonn.
- Duden Informatik**, 3. Auflage, Dudenverlag Mannheim, 2001.
- Gosling, J., Joy, B. , Steele, G., Steele G.L. (2000): **The Java(TM) Language Specification**, 2. Edition, Addison-Wesley, Boston.
- IEEE Computer Society (Hrsg.) (1987): **IEEE Standard for Projekt Management Plans**, IEEE Std 1058.1-1987,1987.
- Kemper A., Eickler A. (2001): **Datenbanksysteme**, 4. Auflage, Oldenbourg-Verlag, München.
- Kline, K., Kline, D. (2001): **SQL in a Nutshell**, O'Reilly, Beijing.
- MacLaughlin, B. (2001): **Java & XML**, 2. Edition, O'Reilly, Beijing.
- OMG (2000a): **Meta Object Facility (MOF) Specification**, Version 1.3, New Edition March 2000.
- OMG (2000b): **OMG Unified Modeling Language Specification**, Version 1.3.
- OMG (2000c): **OMG XML Metadata Interchange (XMI) Specification**, Version 1.0.
- OMG (2000d): **OMG XML Metadata Interchange (XMI) Specification**, Version 1.1.
- OMG (2001): **OMG Unified Modeling Language Specification**, Version 1.4.
- Reißing, R. (2001a): **Qualitätsbewertung beim objektorientierten Entwurf**.  
Erscheint in: Gesellschaft für Informatik (Hrsg.): Informatiktage 2001, 09. und 10. November 2001 im Neuen Kloster Bad Schussenried. Konradin Verlag.

- Reißing, R. (2001b): **Towards a Model for Object-Oriented Design Measurement**, In: Brito e Abreu, F. et al. (ed.): Proc. of the 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Budapest, 2001, 71-84.
- Rumbough, J., Jacobson, I., Booch, G. (1999): **The Unified Modeling Language Reference Manual**, Addison-Wesley, Reading.
- Schmider, C. (2002a): **Dokumentation MOOSE: Die Datenbank**, Internes Dokument, Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart.
- Schmider, C. (2002b): **Konfiguration von MOOSE für QOOD**, Internes Dokument, Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart.
- W3C (2000a): **Document Object Model (DOM) Level 2 Specification**, Version 1.0.
- W3C (2000b): **Extensible Markup Language (XML)**, Version 1.0 (Second Edition).

## **Erklärung**

Hiermit versichere ich, diese Arbeit selbständig verfasst  
und nur die angegebenen Quellen benutzt zu haben.

---

Christoph Schmider