Diplomarbeit

C++ Codeerzeugung aus OMT-Modellen

von

Frank Bohlmann

betreut durch

Prof. Dr. Jürgen Ebert
Dipl.-Inform. Roger Süttenbach

Universität Koblenz

April 1996

Inhaltsverzeichnis

I Einleitung

1 Allgemeines 4

1.1 Ziel dieser Diplomarbeit 4
1.2 Object Modeling Technique (OMT) 5
1.2.1 Objekt-Modell 5
1.2.2 Dynamisches Modell 5
1.2.3 Funktionales Modell 6
1.3 C++ 6
1.4 Graphen 7
1.5 Gliederung 7
1.6 Danksagungen 7

II Modellierung

2 Umsetzung der OMT-Komponenten in Schema-Diagramme 9

3 Schema-Diagramme für das Objekt-Modell 12

3.1 Schema-Diagramme für die einzelnen Komponenten des Objekt-Modells 12
3.2 Das Objekt-Modell in der Gesamtsicht 21

4 Schema-Diagramme für das dynamische Modell 24

4.1 Schema-Diagramme für die einzelnen Komponenten des dynamischen Modells 24
4.2 Das dynamische Modell in der Gesamtsicht 32

5 Schema-Diagramme für das funktionale Modell 35

5.1 Schema-Diagramme für die einzelnen Komponenten des funktionalen Modells 35
5.2 Das funktionale Modell in der Gesamtsicht 41

6 Schema-Diagramme für die Beziehungen zwischen den drei Modellen 43

6.1 Schema-Diagramme für die einzelnen Beziehungen 43
6.2 Die Beziehungen in der Gesamtsicht 46

III Umsetzung in C++ Code

7 Allgemeines zur Umsetzung der einzelnen Komponenten in C++ Code 48

8 Umsetzung des Objekt-Modells in C++ Code 49

8.1 Ideen 49
8.2 Beispiele 59

9 Die Umsetzung des funktionalen Modells in C++ Code 70

9.1 Ideen 70
9.2 Beispiele 72

10 Die Umsetzung des dynamischen Modells in C++ Code 79

10.1 Ideen 79
10.2 Beispiele 84

11 Pseudocode-Algorithmen 98

11.1 Allgemeines zum Pseudocode 98
11.1.1 Benutzung des EMS-Graphenlabors 98
11.1.2 Methoden- und Kontrollstrukturbeschreibungen 99
11.1.3 Beispiel 101
11.1.4 Sonstiges 103
11.2 Algorithmen 104
11.2.1 Der grundlegende Algorithmus (für alle Klassen) 104
11.2.2 Verfeinerung der abstrakten Pseudocode-Anweisungen 107

12 Ausblick 131

Literaturverzeichnis 135

Erklärung 136

I Einleitung

1 Allgemeines

1.1 Ziel dieser Diplomarbeit

In der Realität vorkommende Systeme können modelliert werden. Dies kann unter anderem folgende Gründe haben:

- Das vorhandene reale System soll durch die Modellierung verständlich gemacht werden.
- Manchmal kann man Experimente am realen System nicht durchführen. Verändert man Komponenten des Modells, so kann man die möglichen Auswirkungen auf das reale System erkennen, ohne in den realen Ablauf einzugreifen.
- Reale Systeme werden vor ihrer Realisierung modelliert, um so den späteren Systembenutzern das System vorzustellen. Fehler, die nach Abschluß der Realisierungsphase teuer zu beheben wären, können so vorzeitig erkannt und verhindert werden.

Oft ist es wünschenswert, aus Modellen Programmcode, der einer bestimmten Programmiersprache entspricht, zu entwickeln. Dies kann z.B. zur Automatisierung modellierter Abläufe dienen.

Diese Diplomarbeit entstand aus dem Wunsch, objektorientierte Modellierungen per Programmgenerator in eine objektorientierte Programmiersprache umzusetzen.

Die „Object Modeling Technique“ (im folgenden auch mit „OMT“ bezeichnet), die in [Rumbaugh 91] beschrieben wird, stellt eine Modellierungsart zur Verfügung, die dem objektorientierten Paradigma unterliegt. Modelle dieser Art sollten also automatisch in eine objektorientierte Programmiersprache umgesetzt werden.

Als Zielprogrammiersprachen standen neben anderen C++, Eiffel oder Smalltalk zur Verfügung. Im Rahmen dieser Arbeit wurde C++ als Zielsprache gewählt, da sie am Institut für Softwaretechnik der Universität Koblenz überwiegend Verwendung findet.

Somit war es Ziel dieser Arbeit, einen Programmgenerator zu entwerfen, der OMT-Modelle möglichst ohne Informationsverlust in C++ Code umsetzt. Der Programmgenerator selbst sollte aus den oben genannten Gründen auch in C++ implementiert werden.

Voraussetzung zur Lektüre dieser Arbeit ist Vertrautheit mit dem objektorientierten Paradigma. Die Programmiersprache C++ ist vollständig in [Stroustrup 92] beschrieben. Um diese Arbeit in allen Einzelheiten verstehen zu können, sollte der Leser das Buch „Object-oriented Modeling and Design“ ([Rumbaugh 91]) oder die deutsche Übersetzung ([Rumbaugh 92]) zur Hand haben, denn gelegentlich nehme ich direkten Bezug auf Stellen in den genannten Büchern. Da OMT, die Programmiersprache C++ und Graphen elementare Bestandteile dieser Arbeit sind, werden ihre Grundzüge in den folgenden Kapiteln kurz vorgestellt.

1.2 Object Modeling Technique (OMT)

Die Object Modeling Technique, beschrieben in [Rumbaugh 91], ist die objektorientierte Modellierungsart, die in dieser Arbeit betrachtet wird. Jedes Modell, welches dieser Modellierungsart entspricht, soll in C++ Coderahmen umgesetzt werden können.

Im folgenden beschreibe ich die Object Modeling Technique in ihren Grundzügen.

Es gibt drei Sichtweisen auf ein System:

1. die datenorientierte Sicht
2. die zeitorientierte Sicht
3. die funktionale Sicht

Bisherige Modellierungsarten stellten jeweils einen dieser drei Ansätze in den Vordergrund. Dabei wurden die anderen Sichten jeweils vernachlässigt. Die Object Modeling Technique betrachtet diese drei Sichtweisen gleichwertig und integriert sie zu einem Modell. Dies geschieht, indem in jedem Teil-Modell der Object Modeling Technique Komponenten vorkommen, die sich in anderer Form in den beiden anderen Modellen wiederfinden.

Die drei grundlegenden Modelle der Object Modeling Technique sind das Objekt-Modell, das dynamische Modell und das funktionale Modell. Das Objekt-Modell entspricht der datenorientierten Sicht, das dynamische Modell der zeitorientierten Sicht und das funktionale Modell der funktionalen Sicht. Die Aufgaben und Eigenschaften werden in den folgenden Unterkapiteln kurz beschrieben.

1.2.1 Objekt-Modell

Das Objekt-Modell beschreibt die Struktur der auftretenden Objektklassen und Objekte. Es zeigt, welche Attribute ein Objekt hat und welche Operationen auf diesem Objekt ausgeführt werden können. Weiterhin stellt es heraus, welche Beziehungen zwischen welchen Objekten bestehen können. Das Objekt-Modell beantwortet die Frage nach dem „Was“ in einem System. Es entspricht der oben erwähnten datenorientierten Sicht.

1.2.2 Dynamisches Modell

Das dynamische Modell beschreibt das zeitliche Verhalten eines Systems. Es werden Zustandsübergangsdiagramme benutzt. Jedes Zustandsübergangsdiagramm ist genau einer Objektklasse des Objekt-Modells zugeordnet. Ereignisse, die Zustandsübergänge auslösen, finden sich als Operationen der entsprechenden Klasse im Objekt-Modell wieder. Gleiches gilt für Aktionen und Aktivitäten, die auch im dynamischen Modell vorkommen. Sie tauchen auch als Operationen im Objekt-Modell auf. Durch das Wissen, in welcher Reihenfolge die Operationen des Objekt-Modells vorkommen, wird die Frage nach dem „Wann“ beantwortet. Das dynamische Modell entspricht der oben erwähnten zeitorientierten Sicht.

1.2.3 Funktionales Modell

Das funktionale Modell beschreibt die interne Struktur der Operationen. Es verdeutlicht, wie die Eingabe-Werte in die Ausgabe-Werte transformiert werden. Weiterhin werden funktionale Abhängigkeiten der Operationen untereinander beschrieben. Beschreibungsmittel sind Datenflußdiagramme, die Prozesse enthalten. Diese Prozesse finden sich als Operationen im Objekt-Modell wieder. Das funktionale Modell beantwortet die Frage nach dem „Wie“. Es entspricht der oben beschriebenen funktionalen Sichtweise.

1.3 C++

Die Programmiersprache C++ ist die im Rahmen dieser Arbeit gewählte Zielsprache, also die Sprache, in die die eben beschriebenen OMT-Modelle umgesetzt werden sollen.

C++ ist eine Erweiterung der Programmiersprache C und entspricht zusätzlich dem objektorientierten Paradigma. Im folgenden finden die grundlegenden Elemente von C++, die für diese Arbeit wichtig sind, Erwähnung.

C++ unterstützt folgende Prinzipien, die dem objektorientierten Paradigma entsprechen:

- das Abstraktionsprinzip (Objektklassen stellen abstrakte Datentypen bereit)
- das Kapselungsprinzip (interne Informationen eines Objekts sind anderen Objekten nicht zugänglich)
- das Identitätsprinzip (genaue Identifizierung jedes Objekts)
- das Vererbungsprinzip (Vererbung von Struktur und Verhalten)

Sonstige objektorientierte Prinzipien, wie Polymorphismus werden in dieser Arbeit nicht benötigt und fehlen somit in dieser Liste.

Das Hauptelement in C++ ist die Objektklasse. Eine Objektklasse stellt einen abstrakten Datentyp zur Verfügung. Sie beschreibt die Struktur und das Verhalten aller Elemente, die dieser Objektklasse entstammen. Die Struktur wird durch die Variablen, die ausschließlich zu dieser Klasse gehören, beschrieben. Das Verhalten wird durch Methoden beschrieben. Hiermit werden das Abstraktionsprinzip und das Kapselungsprinzip eingehalten.

Objekte sind Instanzen dieser Objektklassen. Jede vorkommende Instanz in einem C++ Programm gehört zu genau einer Objektklasse. Die Instanziierung geschieht durch Zuweisung einer Variablen zu einer Objektklasse. Durch diese eindeutige Zuordnung wird das Identitätsprinzip eingehalten.

Durch das Vererbungsprinzip können Objekten einer Klasse Eigenschaften und Verhaltensweisen einer anderen Klasse vererbt werden. Dieses Konzept wird von C++ unterstützt. Als Sonderfall der Vererbung unterstützt C++ die Idee der virtuellen Klassen und der virtuellen Methoden, die in der objektorientierten Modellierung auch benutzt werden. Weiterhin ist sogar die Mehrfachvererbung möglich.

1.4 Graphen

Um die OMT-Modelle, die den Input des Codegenerators darstellen, überhaupt verarbeiten zu können, müssen sie in einer vom Rechner verarbeitbaren Form vorliegen. Als interne Repräsentation bieten sich Graphen an. Das Institut für Softwaretechnik der Universität Koblenz stellt mit dem EMS-Graphenlabor (siehe [Dahm 95]) dem Anwender eine äußerst leistungsfähige Schnittstelle zur Verwaltung von Graphen zur Verfügung.

Die Graphen können in Form von TGraphen (siehe [Ebert 94]) vorliegen.

TGraphen sind

- gerichtet
  das heißt, die Richtung einer Kante zwischen zwei Knoten ist eindeutig
- angeordnet
  das heißt, die Kanten eines Knotens sind geordnet
- typisiert
  das heißt, jeder Knoten und jede Kante ist einem bestimmten Typ zugeordnet
- attributiert
  das heißt, den Knoten und Kanten können Attribute zugeordnet werden.

Eine Menge von TGraphen kann zu einer Graphklasse zusammengefaßt werden. Graphklassen können durch Schemadiagramme definiert werden. Dieser Vorgang ist genauer in [Carstensen 94] beschrieben.

1.5 Gliederung

In diesem ersten Teil der Diplomarbeit habe ich allgemeine Voraussetzungen zum Verständnis dieser Arbeit benannt.

Eine Überführung jeder möglichen OMT-Modellierung in TGraphen wird von mir im zweiten Teil dieser Diplomarbeit beschrieben, so daß diese später als Input des Programmgenerators benutzt werden können. Ich entwickle also für jede Komponente der drei OMT-Modelle die zugehörigen Graphklassen und beschreibe sie durch Schemadiagramme.

Im dritten Teil der Diplomarbeit setze ich dann unter Verwendung der in den vorherigen Kapiteln geschaffenen Voraussetzungen die einzelnen OMT-Komponenten in C++ Code um. Der gesamte Erzeugungsalgorithmus wird im letzten Kapitel in Pseudocode beschrieben.

1.6 Danksagungen

Ich möchte mich bei Herrn Prof. Dr. Jürgen Ebert und Herrn Dipl.-Inform. Roger Süttenbach für die intensive Betreuung meiner Diplomarbeit bedanken. Die häufig sehr langen Gespräche haben mich und meine Arbeit gefördert und haben darüber hinaus auch Spaß gemacht. Herrn Süttenbach danke ich besonders für die Zeit, die er mir auch noch nach der offiziellen Dienstzeit gewidmet hat. Für das Fehlerlesen bedanke ich mich bei meinen Eltern und Dr. Michael Sennewald. Bei Oliver Breuckmann möchte ich mich für die Unterstützung bei der Textformatierung bedanken. Schließlich geht ein Dank auch an meine Freundin Cornelia, die sich während dieser Arbeit das ein oder andere Wochenende an die zweite Stelle setzen ließ.

Frank Bohlmann
Koblenz, April 1996

II Modellierung

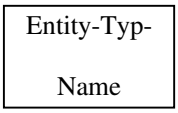
2 Umsetzung der OMT-Komponenten in Schema-Diagramme

In diesem Teil der Diplomarbeit werden die einzelnen Komponenten aus OMT in ihre abstrakte Syntax in Form von Schema-Diagrammen überführt. Diese Schema-Diagramme beschreiben jeweils eine Graphklasse. TGraphen, die diesen Graphklassen entsprechen, dienen im dritten Teil dieser Arbeit als Eingabe für den Codegenerator.

Schema-Diagramme:

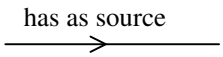
Folgende Konventionen und Schreibweisen bezüglich der Schema-Diagramme gelten in den nächsten Kapiteln:

Entity-Typen werden als Rechtecke dargestellt und mit einem Namen versehen, der mit einem Großbuchstaben beginnt:



Beziehungstypen zwischen Entity-Typen werden durch eine Kante zwischen diesen Entity-Typen ausgedrückt. Diese Kanten sind jeweils mit einem Pfeil versehen, der die Leserichtung zwischen den Entity-Typen festlegt. Dieser ist in den meisten Fällen in der Mitte der Kante.

Als Beschreibung der Kante wird eine Verbalphrase benutzt, die klein geschrieben wird:



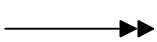
Weiterhin besitzt die Kante Kardinalitätspfeile an ihren beiden Enden. Diese sind ausgefüllt oder hohl, treten einzeln oder zu zweit auf und beschreiben, wieviele Entities der einen Seite mit einem Entity der anderen Seite in Beziehung stehen.

Hierbei gelten folgende Häufigkeiten :

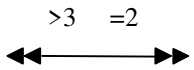
Null oder Eins:

Null oder Viele :

Genau Eins:

Eins oder Viele : 

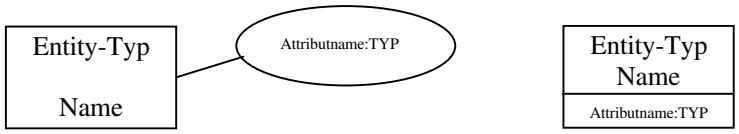
Die Häufigkeit kann durch die Angabe einer Zahl hinter einem Gleichheitszeichen, einer Größer als- oder einer Kleiner als-Relation an dem jeweiligen Entity-Typ genau spezifiziert werden:



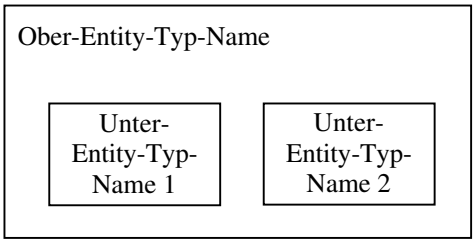
Attribute zu Entity-Typen und Beziehungstypen werden in Ellipsen beschrieben. Sie können auch in das Rechteck des Entity-Typs unterhalb des Namens geschrieben werden. Ihr Name beginnt mit einem Großbuchstaben, der Datentyp steht nach einem Doppelpunkt. Ist dieses Attribut schon einmal vorgekommen, so kann der Datentyp weggelassen werden. In Komponenten, die Erweiterungen schon aufgeführter Komponenten beschreiben, können die Attribute ganz weggelassen werden, außer sie sind direkt an der Erweiterung beteiligt.

Datentypen, die hier verwendet werden, sind die in prozeduralen Programmiersprachen üblichen wie Integer, String, Bool u.s.w.. Sie werden groß geschrieben (alle Buchstaben).

Aufbau eines Attributs (hier zu einem Entity-Typ):

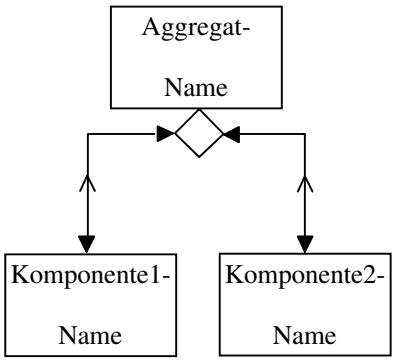


Als weiteres Konzept wird die Generalisierung von Entity-Typen und deren Beziehungstypen benutzt. Dabei werden mehrere Entity-Typen zu einem übergeordneten Entity-Typ zusammengefügt, um diesen später weiter zu benutzen. Die einzelnen Entity-Typen werden von einem Kasten umrandet. Dieser Kasten stellt den Ober-Entity-Typ dar und erhält einen Namen links oben in der Ecke:



Die Aggregation mehrerer Entity-Typen zu einem Entity-Typ, der die aggregierten Entity-Typen als Teile enthält, wird folgendermaßen ausgedrückt:

Der übergeordnete Entity-Typ (Aggregat) bekommt an sein Rechteck ein kleines Quadrat, das es nur mit einer Ecke berührt. Von diesem kleinen Quadrat verlaufen Kanten zu den Komponenten. Die Kardinalitätspfeile werden wie bei normalen Beziehungstypen benutzt:



Modellierung der OMT-Komponenten:

In [Rumbaugh 91] bzw. [Rumbaugh 92] werden bei der Vorstellung der einzelnen Komponenten der Object Modeling Technique die Grundzüge jeweils erläutert und dann beispielhaft beschrieben. An einer Stelle in den Büchern werden dann alle Komponenten noch mal tabellarisch aufgeführt. Dies geschieht in [Rumbaugh 91] auf den Faltseiten zu Beginn (Objekt-Modell) und Ende (dynamisches und funktionales Modell) des Buches, in [Rumbaugh 92] in Anhang A. Ich halte mich bei der Umsetzung der einzelnen Komponenten in Schema-Diagramme an die Reihenfolge, die Rumbaugh in diesen Tabellen einhält, und zwar von links nach rechts und dann von oben nach unten.

Einige Komponenten, die im Anhang A bzw. auf den Faltseiten nicht aufgeführt sind, im Buch jedoch benutzt werden und für diese Diplomarbeit auch relevant sind, wurden hinzugefügt.

Der Name der einzelnen Modellierungs-Komponenten steht in Fettschrift als Substantiv über der Grafik. Nach einer kurzen Beschreibung der Modellierungs-Komponente wird sie gegebenenfalls durch numerierte, kursiv geschriebene Constraints eingeschränkt. Constraints sind in dieser Arbeit natürlichsprachliche Bedingungen, die das jeweilige Schema-Diagramm einschränken. Die Constraints beginnen mit einem „O“ im Objekt-Modell, mit einem „D“ im dynamischen Modell und mit einem „F“ im funktionalen Modell. Danach folgt eine natürliche Zahl.

Beispiel: *Constraint* O2: Eine Klasse sollte nicht Superklasse von sich selbst sein.

Kapitelübersicht:

Auf die in diesem Kapitel beschriebene Art werden in den nächsten Kapiteln die Komponenten der Object Modeling Technique in Schema-Diagramme umgesetzt. In Kapitel 3 wird das Objekt-Modell, in Kapitel 4 das dynamische Modell und in Kapitel 5 das funktionale Modell umgesetzt. In Kapitel 6 werden die für diese Arbeit wichtigen Beziehungen zwischen den Modellen durch Schema-Diagramme beschrieben.

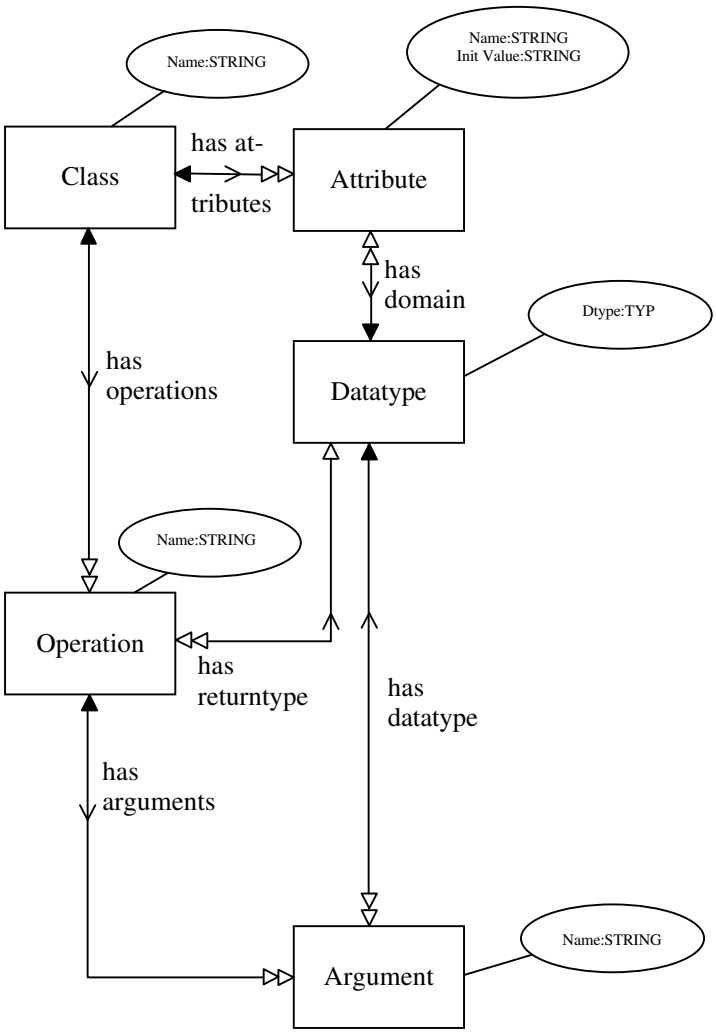
3 Schema-Diagramme für das Objekt-Modell

In diesem Kapitel wird auf in Kapitel 2 beschriebene Weise das Objekt-Modell in Schema-Diagramme umgesetzt. Es werden zunächst die einzelnen Modellierungs-Komponenten des Objekt-Modells überführt. Später werden sie verschmolzen und in einer Gesamtsicht dargestellt.

3.1 Schema-Diagramme für die einzelnen Komponenten des Objekt-Modells

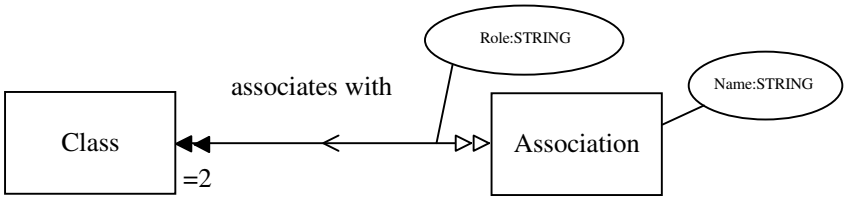
In diesem Unterkapitel werden die einzelnen Komponenten des Objekt-Modells in der Reihenfolge der Faltseiten in [Rumbaugh 91] in Schema-Diagramme umgesetzt.

Class:



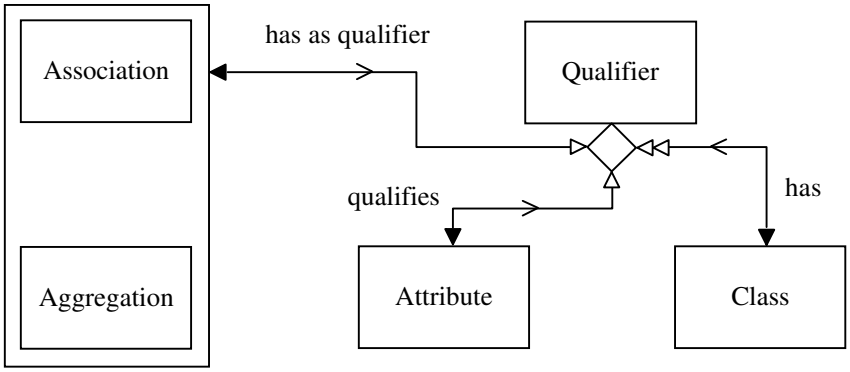
Der Entity-Typ „Class“ ist mit seinen Attributen und Operationen der grundlegende Entity-Typ des Objektmodells. Sollte ein Attribut keinen Datentyp haben (bei Rumbaugh möglich), so wird ihm der Typ VOID zugeordnet. So hat also jedes Attribut einen „Datatype“.

Association:



Die Assoziation stellt Beziehungen zwischen Klassen dar. Im vorliegenden Fall sind genau zwei Klassen an einer Assoziation beteiligt. Die Assoziation hat einen Namen, und jeder an der Assoziation beteiligten Klasse muß ein Rollenname zugeordnet werden.

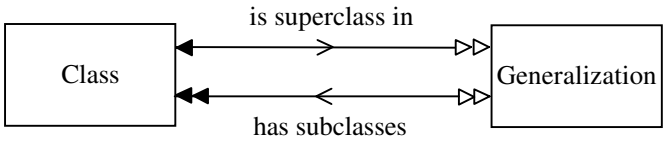
Qualified Association:



Der „Qualifier“ ist ein spezielles Attribut einer an der Assoziation beteiligten Klasse, das die in der übernächsten Grafik eingeführte Kardinalität herabsetzt. Bei Rumbaugh kann auch eine (drei Grafiken tiefer eingeführte) Aggregation einen Qualifier haben, was hier bereits eingearbeitet ist.

Constraint O1: Die am „Qualifier“ beteiligte Assoziation oder Aggregation muß zwischen der am „Qualifier“ beteiligten Klasse, und der Klasse, der das qualifizierte Attribut angehört, bestehen.

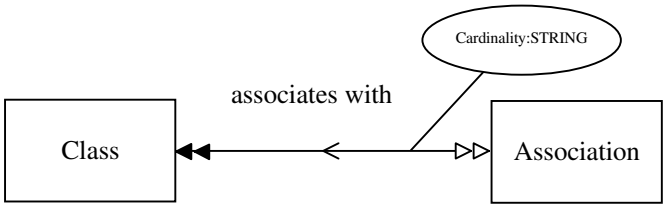
Generalization:



Hiermit werden Sub- und Superklassen beschrieben. Die Kardinalitätspfeile erlauben bereits Mehrfachvererbung.

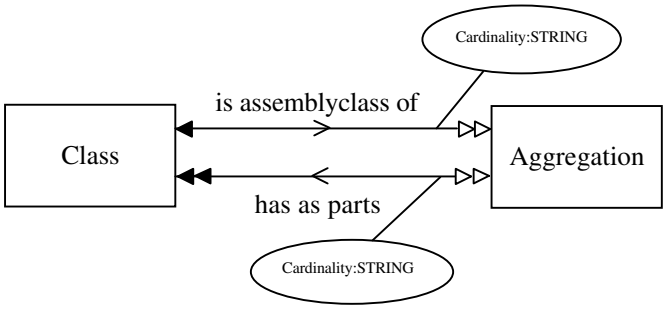
Constraint O2: Eine Klasse sollte nicht Superklasse von sich selbst sein.

Multiplicity of Association:



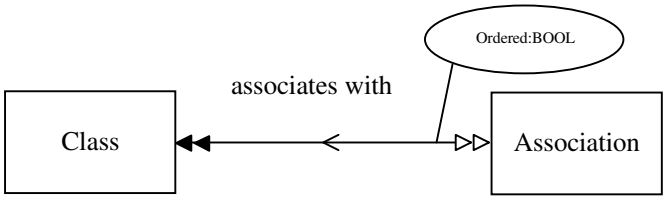
Die Kardinalität beschreibt die Häufigkeit, mit der die jeweiligen Klasseninstanzen an der Assoziation beteiligt sind.

Aggregation:



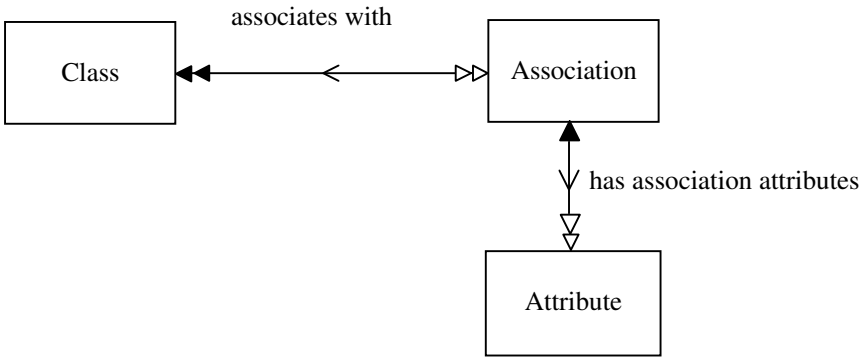
Die Aggregation beschreibt die Teil-von-Beziehung, bei der eine Klasse aus mehreren Teilklassen besteht. Die Teilklassen treten wiederum in der Häufigkeit der eben eingeführten Kardinalität auf.

Ordering:



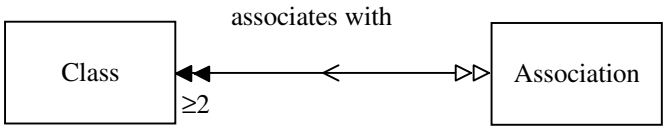
Geordnete Assoziationen werden verwendet, falls die Anordnung der vorhandenen Links relevant ist.

Link Attributes:



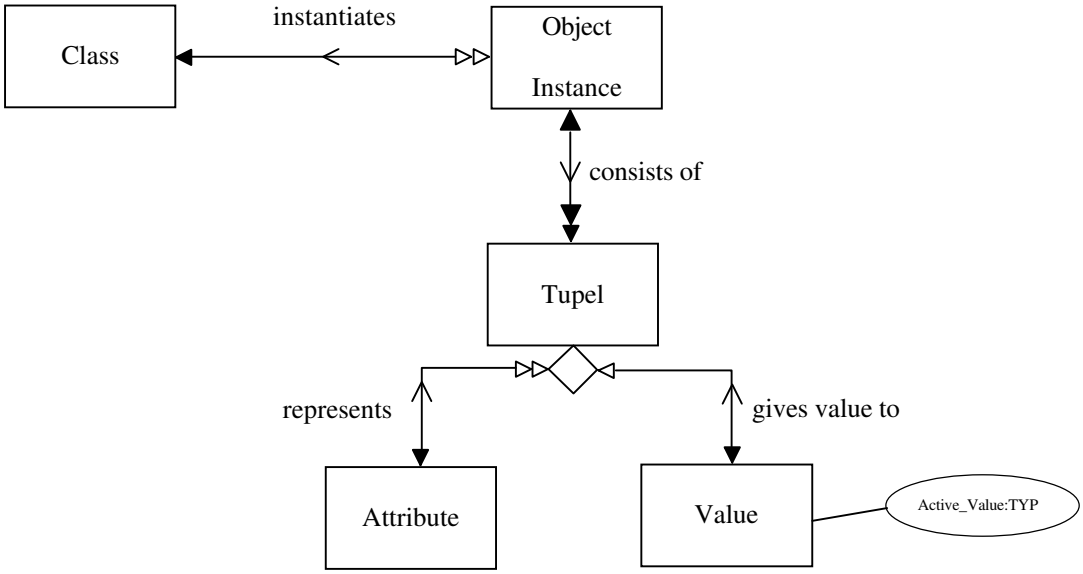
Assoziationen können eigene Attribute besitzen.

Ternary Association:



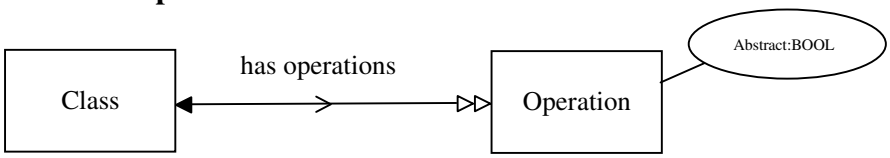
Durch die Bedingung, daß zwei oder mehrere Klassen in Beziehung mit einem Assoziationsknoten stehen (insbesondere 3), sind ternäre bzw. n-äre Assoziationen möglich.

Object Instances / Instantiation Relationship:



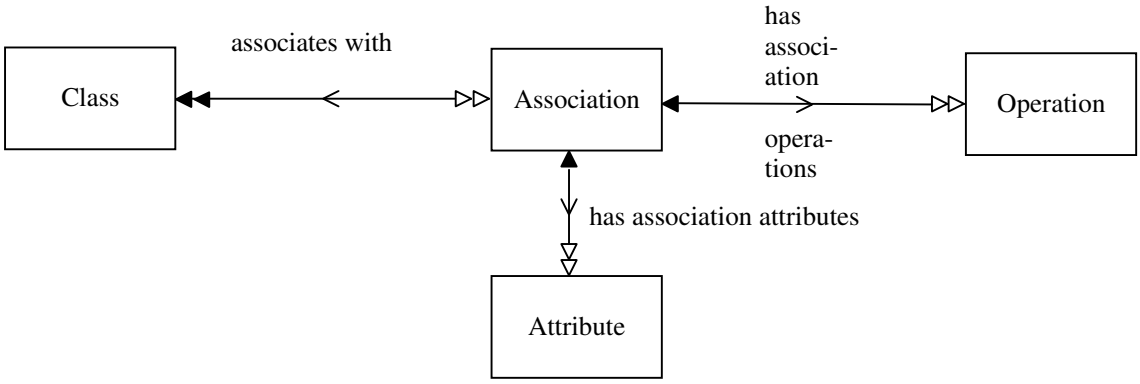
Constraint O3: Alle beteiligten Attribute müssen über eine „has as attributes“-Kante mit der entsprechenden Klasse verbunden sein.

Abstract Operations:



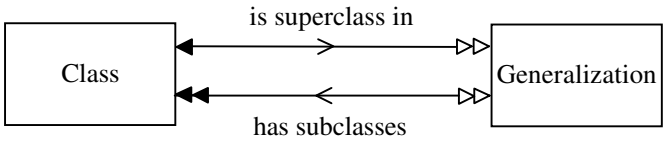
Solch eine Operation ist in dieser Klasse abstrakt und wird erst bei Unterklassen konkret.

Association as Class:



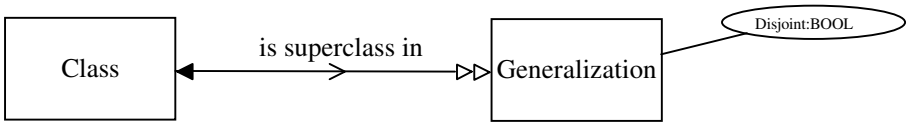
Assoziationen können neben Attributen auch Operationen enthalten.

Multiple Inheritance:



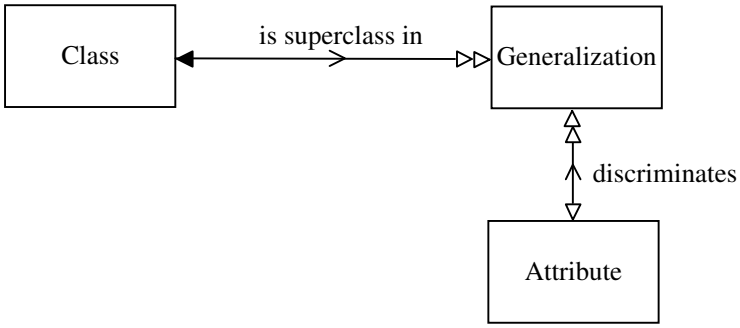
Die Mehrfachvererbung ist durch die Null-oder-Viele-Kardinalität der „has subclasses“-Kante auf der Generalisierungsseite sichergestellt.

Disjoint Subclasses:



Hierbei wird die Information verarbeitet, ob die Unterklassen der beteiligten Superklasse disjunkt sind.

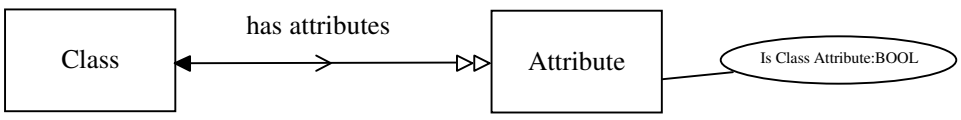
Discriminator:



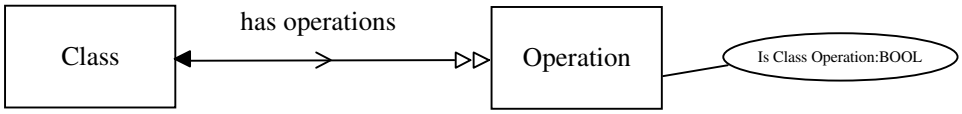
Der Diskriminator ist ein Attribut, welches für die verschiedenen Unterklassen einer Superklasse verschiedene Wertebereiche besitzt.

Constraint O4: Das Attribut muß über eine „has as attributes“-Kante mit der entsprechenden Klasse verbunden sein.

Class Attributes:

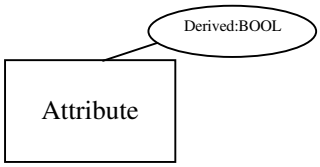


Class Operations:

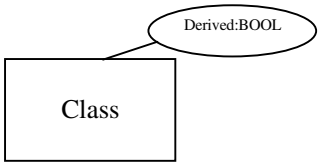


Klassen-Attribute und Klassen-Operationen sind solche, die nur für die Klasse und nicht für Instanzen gelten.

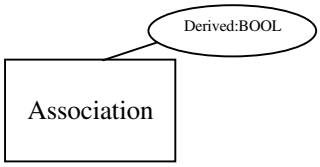
Derived Attributes:



Derived Class:

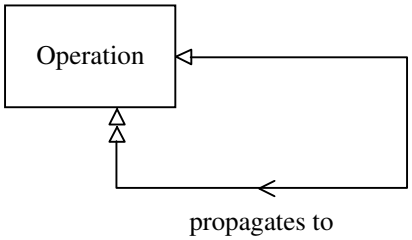


Derived Association:



Bei den letzten drei Komponenten können die jeweiligen Entity-Typen von Entity-Typen derselben Art abgeleitet sein.

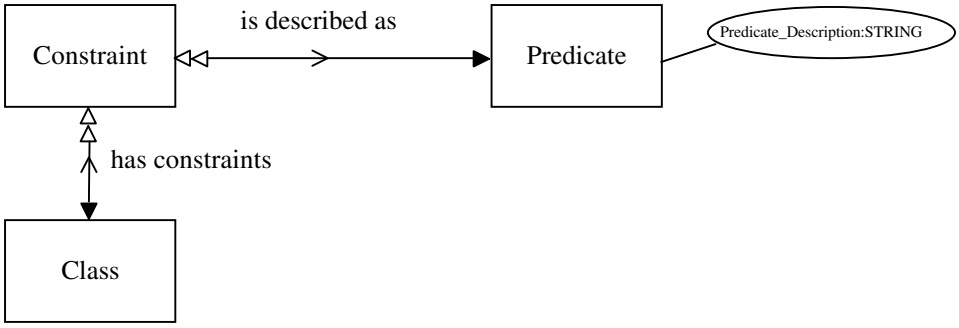
Propagation of Operations:



Hierbei liegt eine Fortpflanzung einer Operation, die einer bestimmten Klasse angehört, auf andere Klassen vor.

Constraint O5: Es muß eine Beziehung zwischen den Klassen, die über die „has as operations“-Kante mit den beteiligten Operationen verbunden sind, vorliegen (Aggregation oder Assoziation).

Constraints on Objects:

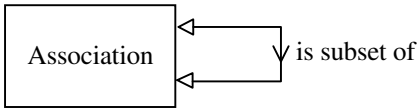


Hier sind Attributeinschränkungen aufzuführen.

Die „Predicate_Description“ wird hier als String beschrieben. Der Ausdruck sollte aber durch Attribute der beteiligten Klasse darstellbar sein.

Constraint O6: Die am „Predicate“ beteiligten Attribute müssen über eine „has as attributes“-Kante mit der entsprechenden Klasse verbunden sein.

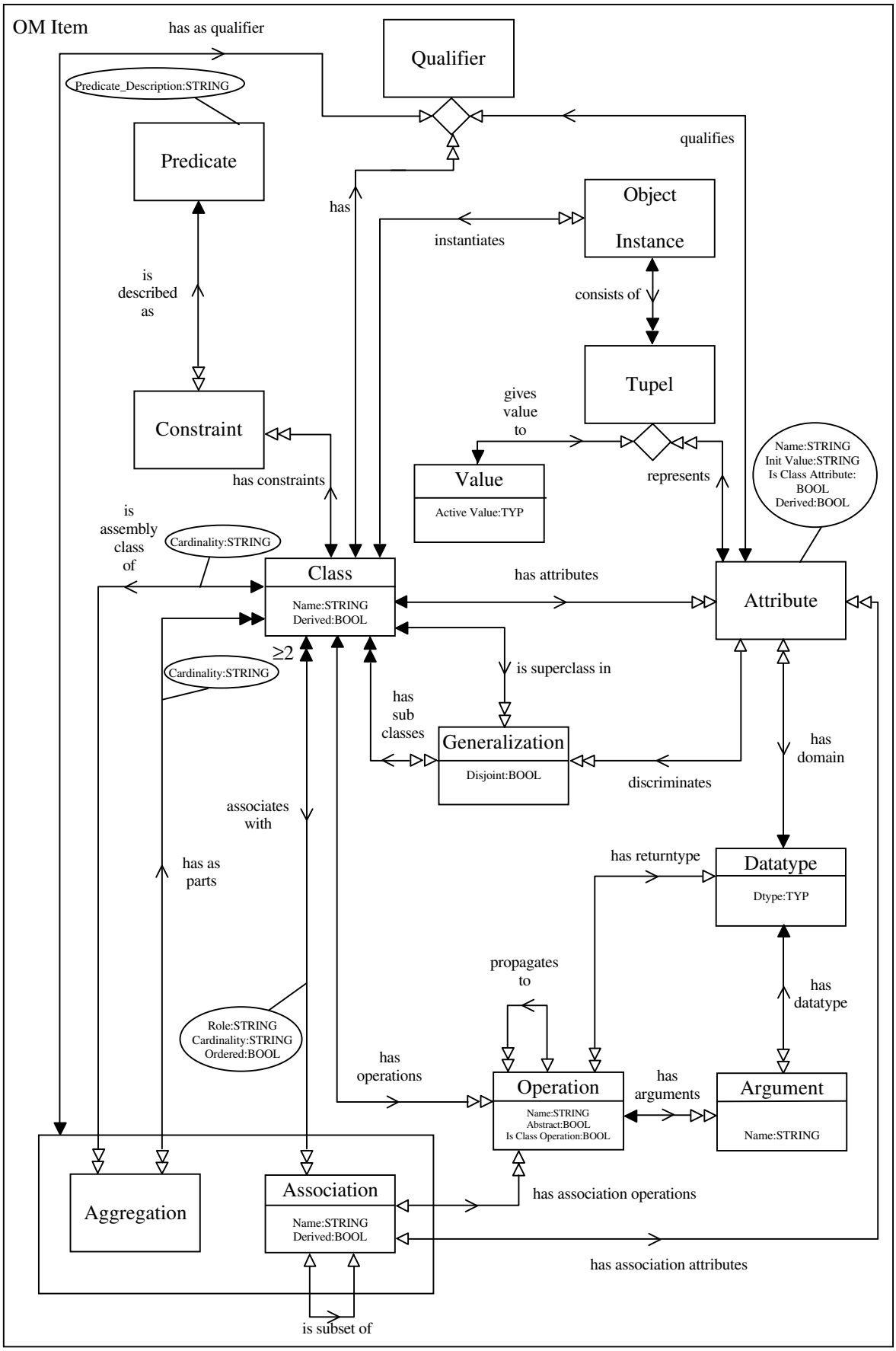
Constraints between Associations:



Als Einschränkungen zwischen Assoziationen ist hier nur die Teilmengenbeziehung gemeint.

3.2 Das Objekt-Modell in der Gesamtsicht

Bisher habe ich in Kapitel 3.1 die verwendeten Modellierungs-Komponenten des Objekt-Modells in einer Einzeldarstellung gezeigt. Im folgenden wird nun die Gesamtdarstellung präsentiert, wobei identische Entity- und Relations-Typen naturgemäß nur einmal vorkommen. Ferner werden die dazugehörigen Constraints des Objekt-Modells in ihrer Gesamtheit aufgeführt.


OM Item
has as qualifier
Qualifier
Predicate_Description:STRING
Predicate
qualifies
has
Object
Instance
instantiates
is
described
as
consists of
Tupel
gives
value
to
Constraint
Value
Active Value:TYP
represents
Name:STRING
Init Value:STRING
Is Class Attribute:
BOOL
Derived:BOOL
has constraints
is
assembly
class
of
Cardinality:STRING
Class
Name:STRING
Derived:BOOL
has attributes
Attribute
≥2
Cardinality:STRING
is superclass in
has
sub
classes
Generalization
Disjoint:BOOL
discriminates
has
domain
associates
with
has as
parts
has returntype
Datatype
Dtype:TYP
propagates
to
has
datatype
Role:STRING
Cardinality:STRING
Ordered:BOOL
has
operations
Operation
Name:STRING
Abstract:BOOL
Is Class Operation:BOOL
has
arguments
Argument
Name:STRING
Aggregation
Association
Name:STRING
Derived:BOOL
has association operations
has association attributes
is subset of

***Constraints* zum Objektmodell:**

Constraint O1: Die am „Qualifier“ beteiligte Assoziation oder Aggregation muß zwischen der am „Qualifier“ beteiligten Klasse und der Klasse, der das qualifizierte Attribut angehört, bestehen.

Constraint O2: Eine Klasse sollte nicht Superklasse von sich selbst sein.

Constraint O3: Alle Attribute, die mit der „Object Instance“ über „Tupel“ verbunden sind, müssen über eine „has as attributes“-Kante mit der an dieser „Object Instance“ beteiligten Klasse verbunden sein.

Constraint O4: Das Attribut, das über eine „discriminates“-Kante mit einem „Generalization“-Knoten verbunden ist, muß über eine „has as attributes“-Kante mit der entsprechenden „Class“ verbunden sein.

Constraint O5: Es muß eine Beziehung zwischen den Klassen, die über die „has as operations“-Kante mit den durch „propagates to“-Kanten verbundenen Operationen verbunden sind, vorliegen (Aggregation oder Generalisierung).

Unter Vorbehalt (da „Predicate“ noch näher beschrieben werden muß):
Constraint O6: Die am „Predicate“ beteiligten Attribute müssen über eine „has as attributes“-Kante mit der entsprechenden Klasse verbunden sein.

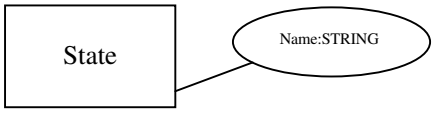
4 Schema-Diagramme für das dynamische Modell

In diesem Kapitel wird auf in Kapitel 2 beschriebene Weise das dynamische Modell in Schema-Diagramme umgesetzt. Es werden zunächst die einzelnen Modellierungs-Komponenten des dynamischen Modells überführt. Später werden sie verschmolzen und in einer Gesamtsicht dargestellt.

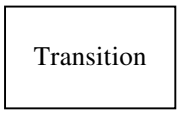
4.1 Schema-Diagramme für die einzelnen Komponenten des dynamischen Modells

Zuerst stelle ich das Hauptelement „State“ und die übrigen atomaren Elemente des dynamischen Modells vor, die bei Rumbaugh so nicht aufgeführt sind.

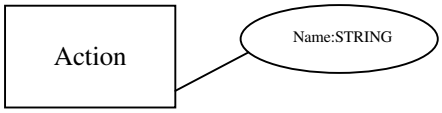
State:



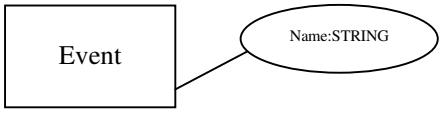
Transition:



Action:

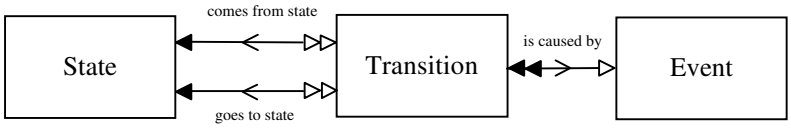


Event:



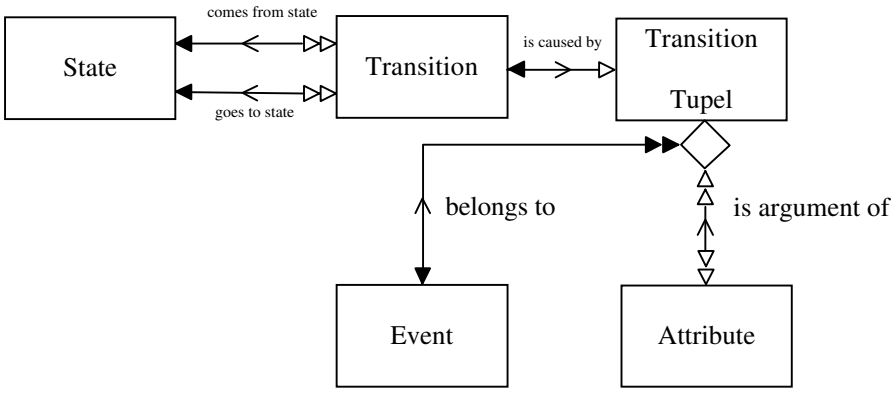
Im folgenden werden die einzelnen Komponenten des dynamischen Modells in der Reihenfolge der Faltseiten in [Rumbaugh 91] in Schema-Diagramme umgesetzt.

Event causes Transition between States:



Hiermit wird ein Zustandsübergang beschrieben, der durch ein Ereignis ausgelöst wird. Betont sei, daß es auch Transitionen geben kann, die nicht durch Ereignisse ausgelöst werden.

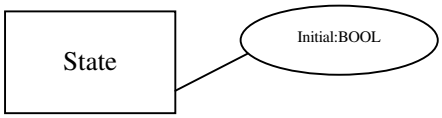
Event with Attributes:



Hierbei können zusätzlich Attribute bei den Transitionen verwaltet werden. Da an einer „Transition“ mehr als nur ein „Event“ beteiligt sein kann, wird der Entity-Typ „Transition Tupel“ eingeführt.

Die beiden folgenden Komponenten sagen aus, ob der beteiligte Zustand Start- oder Endzustand ist.

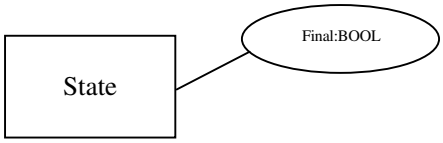
Initial State:



Mit „Initial State“ ist nicht der bei Rumbaugh aufgeführte „Anfangszustand“, sondern der kleine ausgefüllte Kreis gemeint. Rumbaugh führt hier eine Inkonsistenz ein, da (siehe [Rumbaugh 92], Seite 130, Abb. 5.23) dieser Kreis bereits über Transitionen mit anderen Zuständen verbunden sein kann.

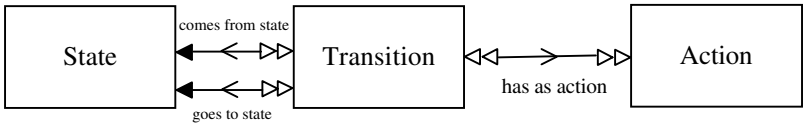
Constraint D1: Ein „Initial State“ darf nur mit *einer* Transition über eine „comes from state“-Kante verbunden sein, da das System sonst nicht mehr deterministisch ist.

Final State:



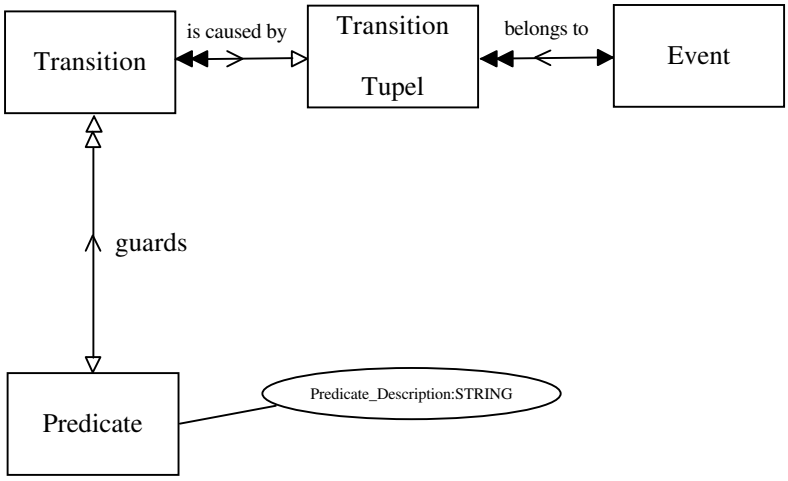
Mit „Final State“ ist der bei Rumbaugh aufgeführte umrandete ausgefüllte Kreis gemeint (analog zu Initial State).

Action on a Transition:



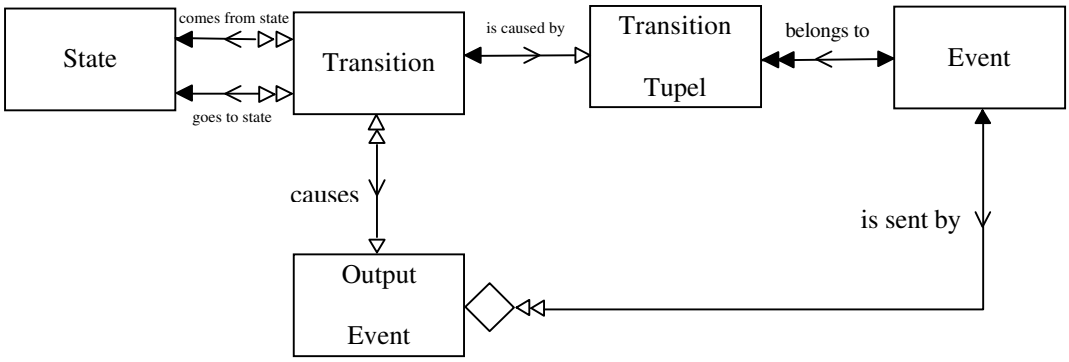
Transitionen können Aktionen auslösen. Dieser Sachverhalt wird hierdurch ausgedrückt.

Guarded Transition:



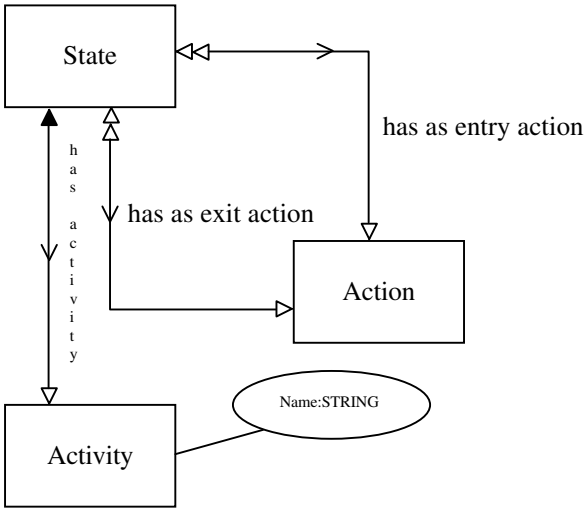
Eine „guarded Transition“ kann nur dann schalten, wenn das relevante Ereignis eintritt und die Guard den Wert True hat. Die Guard wird durch ein „Predicate“ beschrieben.

Output Event on a Transition:



Eine Transition kann ein Ereignis auch als Output haben. Hierzu wird der neue Entity-Typ „Output Event“ eingeführt, der später noch erweitert wird.

Actions and Activity while in a State:



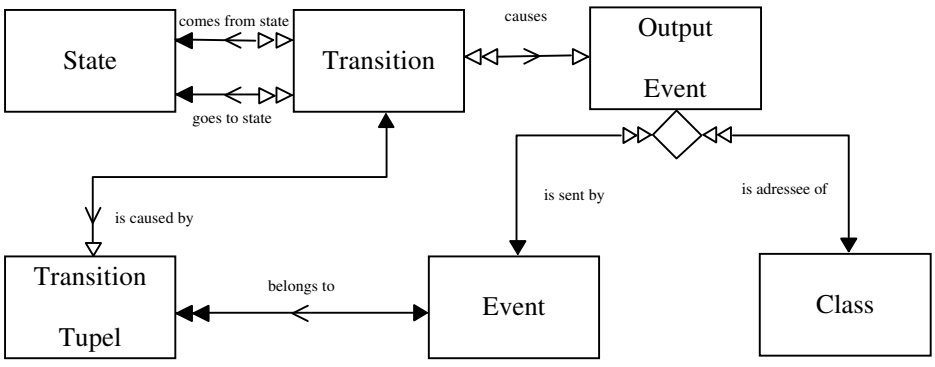
Die entry- und exit-Actions werden durch neue Kanten namens „has as entry action“ bzw. „has as exit action“ von „State“ zu „Action“ einem Zustand zugewiesen.

Die Transitionen innerhalb eines Zustandes werden über das normale Konzept als reflexive Transitionen modelliert. Dann muß also folgendes *Constraint* gelten:

Constraint D2: Bei einer Transition innerhalb eines Zustandes, also die bei [Rumbaugh 91] aufgeführten event1/action1 ... event n/action n, muß die „comes from state“- und die „goes to state“-Kante zum selben „State“ führen.

Hinzu kommt der neue Entitytyp Activity.

Sending an Event to another Object:



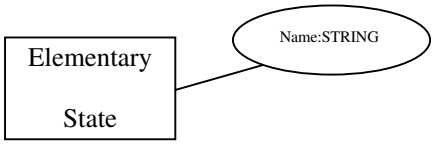
In diesem Fall stellt der Entitytyp „Output Event“ eine Verbindung zum Objektmodell dar.

Anmerkung zu den folgenden drei Komponenten:

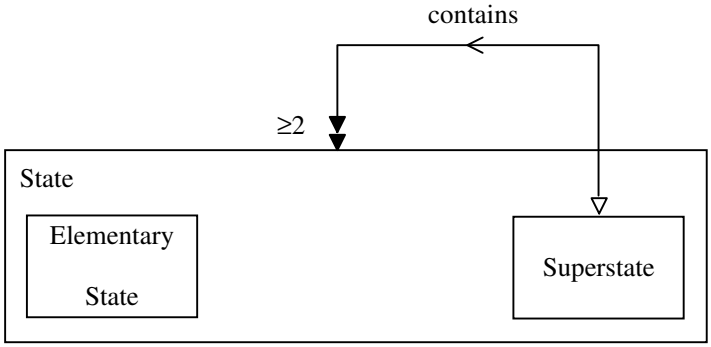
Bisher trat nur der Entity-Typ „State“ auf. Mit „State“ sind allerdings drei verschiedene Arten von Zuständen gemeint, auf die ich auf Grund der Reihenfolge im Anhang A bei [Rumbaugh 92] erst jetzt eingehe.

Dazu führe ich jetzt einen elementaren Zustand ein, der im Gegensatz zum „Superstate“ und dem „Concurrent Superstate“ keine Unterzustände hat.

Elementary State:



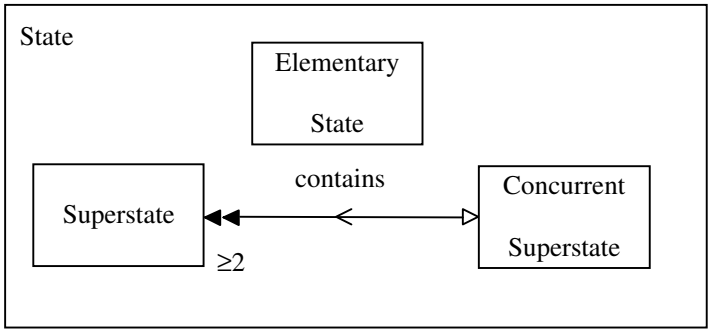
State Generalization :



Die vorliegende Grafik zeigt die Generalisierung mehrerer States zu einem Superstate. Dabei werden nur die beteiligten Substates aufgeführt. Die Transitionen zwischen den Substates sind dem Superstate bekannt, werden also nicht extra aufgeführt.

Constraint D3: Ein State sollte nicht Superstate von sich selbst sein.

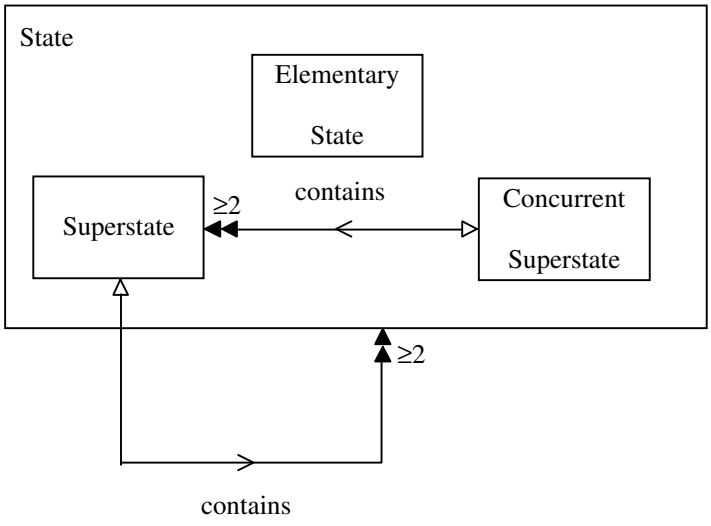
Concurrent Subdiagrams:



Hierbei werden einem State mehrere parallel abzuarbeitende Substates zugewiesen. Die Substates sind Superstates.

Constraint D4: Auch hier sollte ein State nicht gleichzeitig „Concurrent Superstate“ von einer Komponente sein, in der er selbst vorkommt.

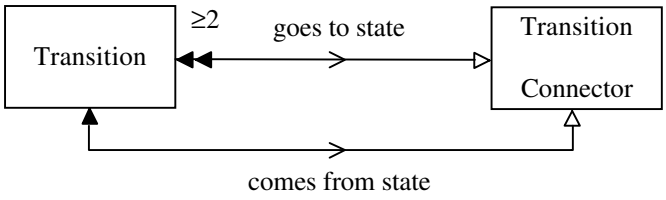
Zusammengenommen ergibt sich folgendes ER-Diagramm:



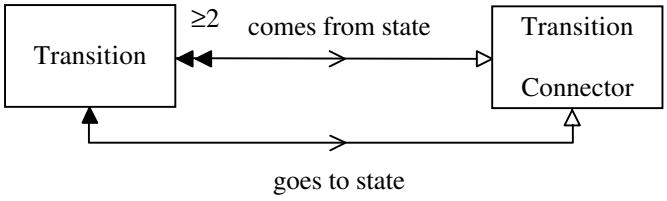
Splitting and Synchronization of Control:

Bei der Splittung wird eine Transition in mehrere Transitionen aufgespalten. Schaltet die Transition, schalten automatisch alle Transitionen, in die diese Transition aufgespalten wurde. Bei der Synchronisierung werden mehrere Transitionen zu einer verschmolzen, die erst dann schaltet, wenn alle einzelnen Transitionen geschaltet haben. Realisiert wird dies durch den Entity-Typ „Transition Connector“.

Splitting of Control:

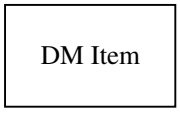


Synchronization of Control:



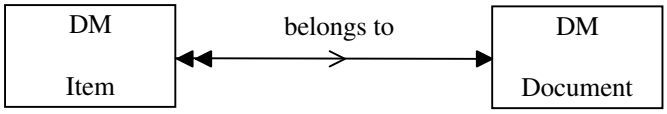
DM Item:

Alle bisher in diesem Kapitel aufgeführten Entity-Typen des dynamischen Modells werden zu einem Entity-Typ namens „DM Item“ generalisiert.



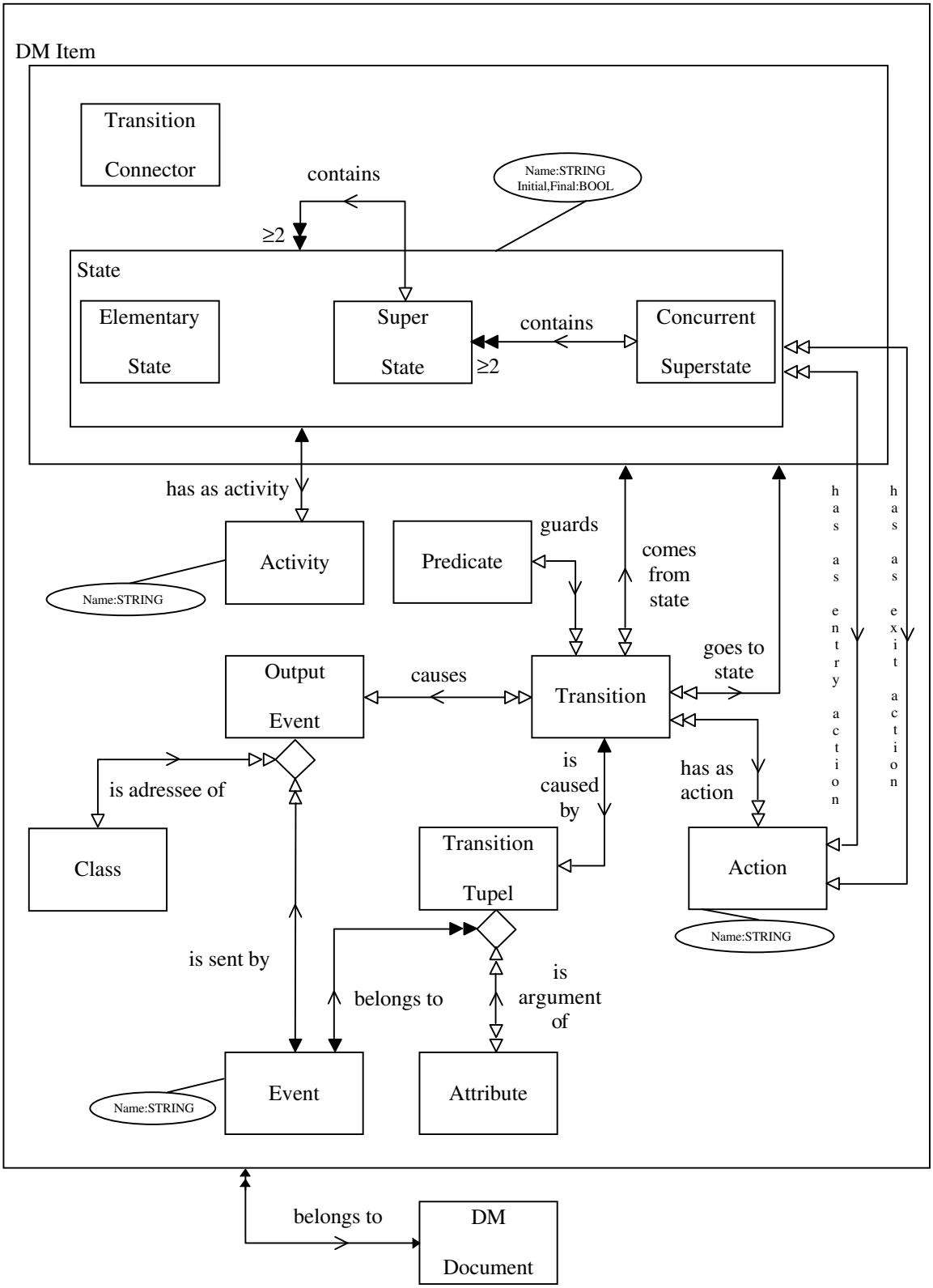
DM Document:

Ein „DM Document“ besteht wiederum aus mehreren „DM Items“:



4.2 Das dynamische Modell in der Gesamtsicht

Bisher habe ich in Kapitel 4.1 die verwendeten Modellierungs-Komponenten des dynamischen Modells in einer Einzeldarstellung gezeigt. Im folgenden wird nun die Gesamtdarstellung präsentiert, wobei identische Entity- und Relations-Typen naturgemäß nur einmal vorkommen. Ferner werden die dazugehörigen Constraints des dynamischen Modells in ihrer Gesamtheit aufgeführt.


DM Item
Transition Connector
contains
≥2
Name:STRING
Initial,Final:BOOL
State
Elementary State
Super State
contains
≥2
Concurrent Superstate
has as activity
Activity
Name:STRING
Predicate
guards
comes from state
has as entry action
has as exit action
goes to state
Output Event
causes
Transition
is adressee of
Class
is caused by
has as action
Transition Tupel
Action
Name:STRING
is sent by
belongs to
is argument of
Event
Name:STRING
Attribute
belongs to
DM Document

***Constraints* zum dynamischen Modell:**

Constraint D1: Ein „Initial State“ darf nur mit *einer* Transition über eine „comes from state“-Kante verbunden sein, da das System sonst nicht mehr deterministisch ist.

Constraint D2: Bei einer Transition innerhalb eines Zustandes muß die „comes from state“- und die „goes to state“-Kante zum selben „State“ führen.

Constraint D3: Ein „State“ sollte nicht „Superstate“ von sich selbst sein.

Constraint D4: Ein „State“ sollte nicht gleichzeitig „Concurrent Superstate“ von einer Komponente sein, in der er selbst vorkommt.

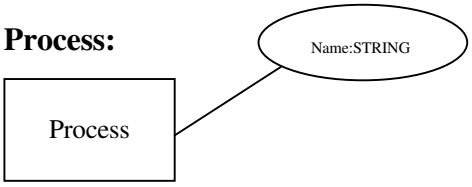
5 Schema-Diagramme für das funktionale Modell

In diesem Kapitel wird auf in Kapitel 2 beschriebene Weise das funktionale Modell in Schema-Diagramme umgesetzt. Es werden zunächst die einzelnen Modellierungs-Komponenten des funktionalen Modells überführt. Später werden sie verschmolzen und in einer Gesamtsicht dargestellt.

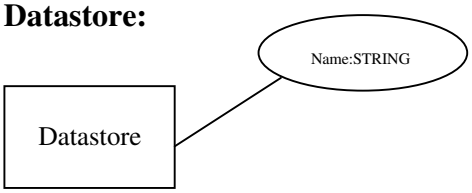
5.1 Schema-Diagramme für die einzelnen Komponenten des funktionalen Modells

Als erstes stelle ich die Hauptkomponente „Process“ und die anderen atomaren Komponenten des funktionalen Modells dar.

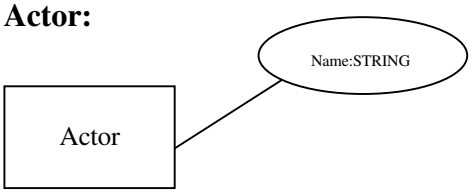
Process:



Datastore:

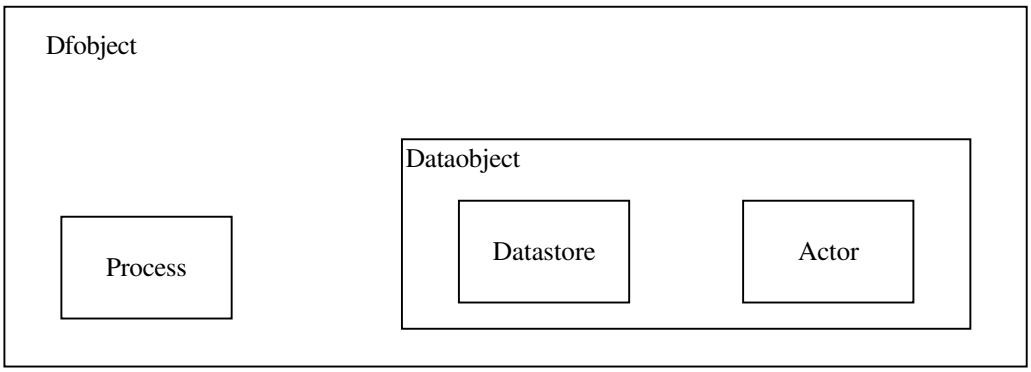


Actor:



Der Entity-Typ „Actor“ ist bei Rumbaugh nicht aufgeführt, da er aber ein grundlegender Entity-Typ ist, erwähne ich ihn hier.

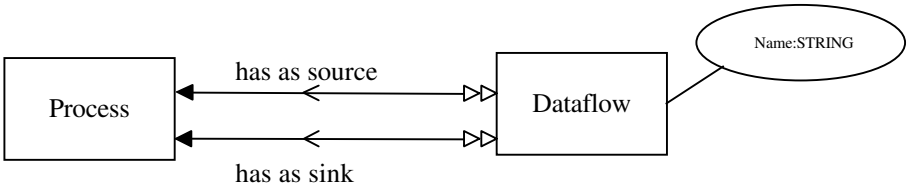
Dfobject:



Dies sind Generalisierungen, die später einige Vereinfachungen bringen. Sie sind in dieser Form bei Rumbaugh nicht aufgeführt.

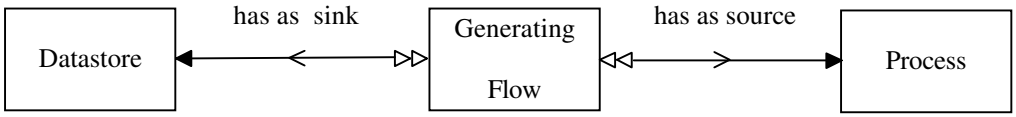
Im folgenden werden die einzelnen Komponenten des funktionalen Modells in der Reihenfolge der Faltseiten in [Rumbaugh 91] in Schema-Diagramme umgesetzt.

Data Flow between Processes:



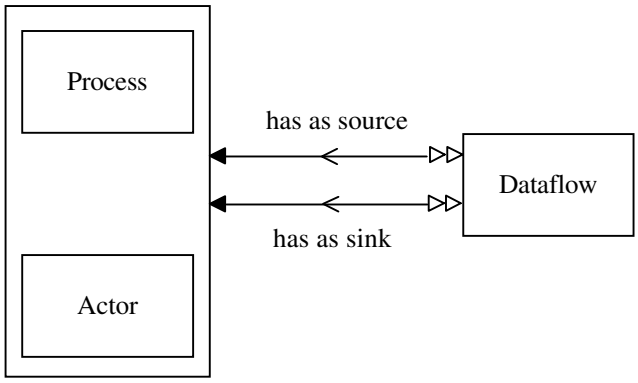
Dieses Konstrukt stellt den einfachsten Fall dar, nämlich den „normalen“ Datenfluß zwischen zwei Prozessen.

Data Flow that results in a Data Store:



Der „Generating Flow“ erzeugt neue Objekte in einem „Datastore“.

Actor as source or sink of flow:



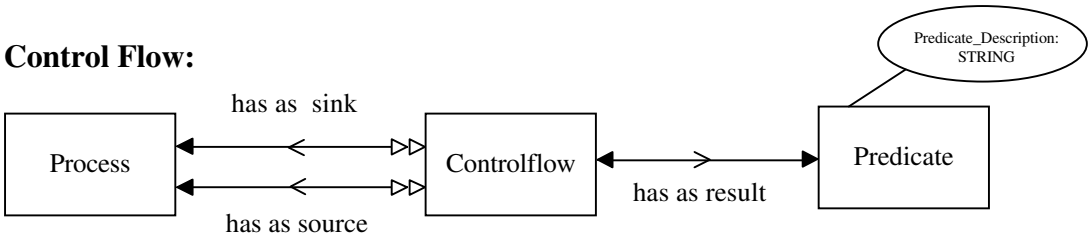
Hier ist der Fall erwähnt, daß Aktoren Quelle oder Ziel von Prozessen sind.

Constraint F1:

Wenn die „has as source“-Kante den „Dataflow“ mit einem „Actor“ verbindet, so muß an der „has as sink“-Kante ein „Process“ stehen.

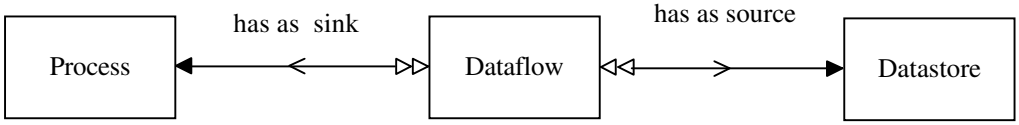
Wenn die „has as sink“-Kante den „Dataflow“ mit einem „Actor“ verbindet, so muß an der „has as source“-Kante ein „Process“ stehen.

Control Flow:



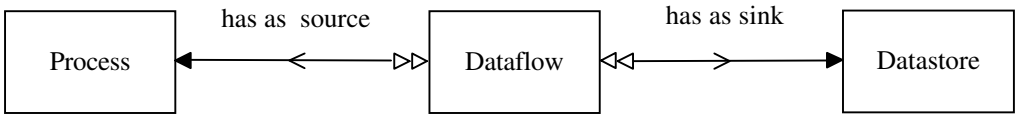
Rumbaughs Definition, daß ein Prozeß auch immer Quelle des Kontrollflusses ist, kann offensichtlich noch dahingehend erweitert werden, daß sogar ein „Dfobject“ an der „has as source“-Kante stehen kann. Für diese denkbare Situation habe ich allerdings noch kein Beispiel gefunden. Aus diesem Grund belasse ich es im folgenden bei der von Rumbaugh eingeführten Konvention.

Access of Data Store Value:



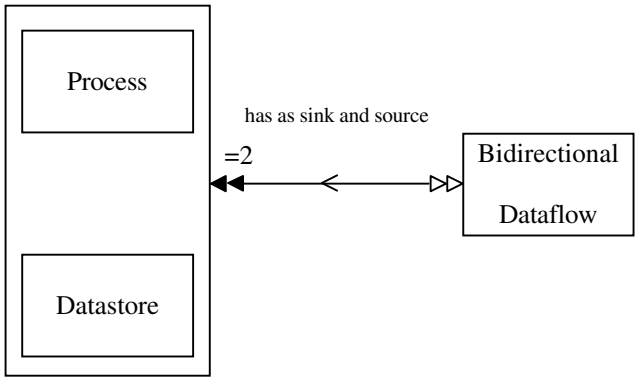
Dies stellt den lesenden Zugriff auf einen Datenspeicher dar.

Update of Data Store Value:



Hiermit ist der schreibende Zugriff auf einen Datenspeicher gemeint.

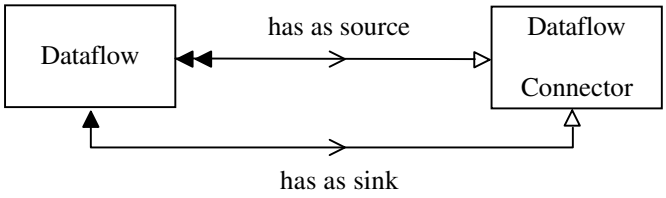
Access and Update of Data Store Value:



Hiermit ist die Kopplung der beiden obigen Zugriffe beschrieben.

Constraint F2: Jeweils eines der beteiligten Entities vom Typ „Dfobjekt“ muß ein „Process“ und das andere ein „Datastore“ sein.

Composition of Data Value:



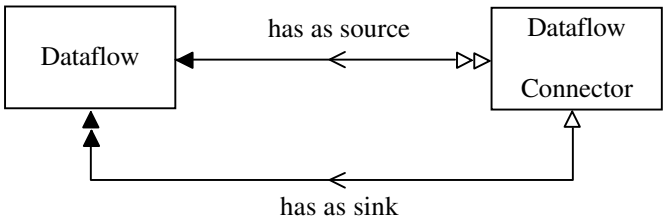
Bei der Komposition werden mehrere Datenflüsse über den Dataflow Connector zu einem Datenfluß zusammengefaßt.

Constraint F7: Bei der Komposition von Datenflüssen muß der „Dataflow Connector“ zwei oder mehr Quelldatenflüsse und genau einen Zieldatenfluß haben.

Duplication of Data Value:

Die Duplikation von Datenflüssen kann genau wie die in der nächsten Grafik eingeführte Dekomposition modelliert werden. Hierbei haben alle Datenflüsse den gleichen Bezeichner.

Decomposition of Data Value:



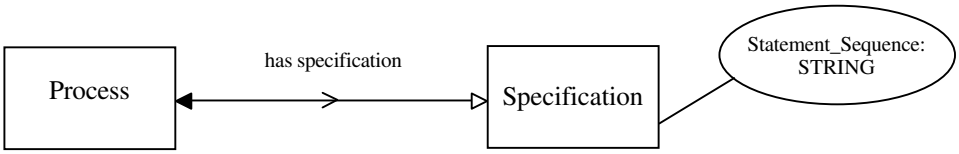
Bei der Dekomposition wird ein Datenfluß über den Dataflow Connector in Teildatenflüsse zerlegt.

Constraint F8: Bei der Dekomposition von Datenflüssen muß der „Dataflow Connector“ zwei oder mehr Zieldatenflüsse und genau einen Quelldatenfluß haben.

Process Specification:

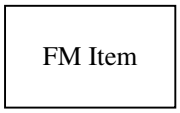
Die Voraussetzung für eine sinnvolle automatisierte Implementierung besteht darin, daß den Blattprozessen eine Beschreibung ihres C++ Codes bzw. eine primitive Prozeßspezifikation zuvor zugeordnet worden ist. Dies geschieht bei OMT in der Objektentwurfsphase, die eigentlich der „Implementierer“ und nicht der „Modellierer“ vornehmen sollte (siehe [Rumbaugh 92], S.320, Punkt 2).

Um die Zuordnung zu gewährleisten, muß das Meta-Modell für das funktionale Modell um den Entity-Typ „Specification“ erweitert werden.

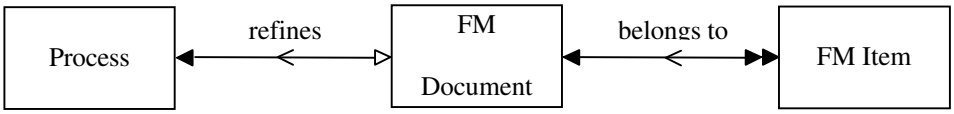


FM Item:

Alle bisher in diesem Kapitel aufgeführten Entity-Typen des funktionalen Modells werden zu einem Entity-Typ namens „FM Item“ generalisiert.



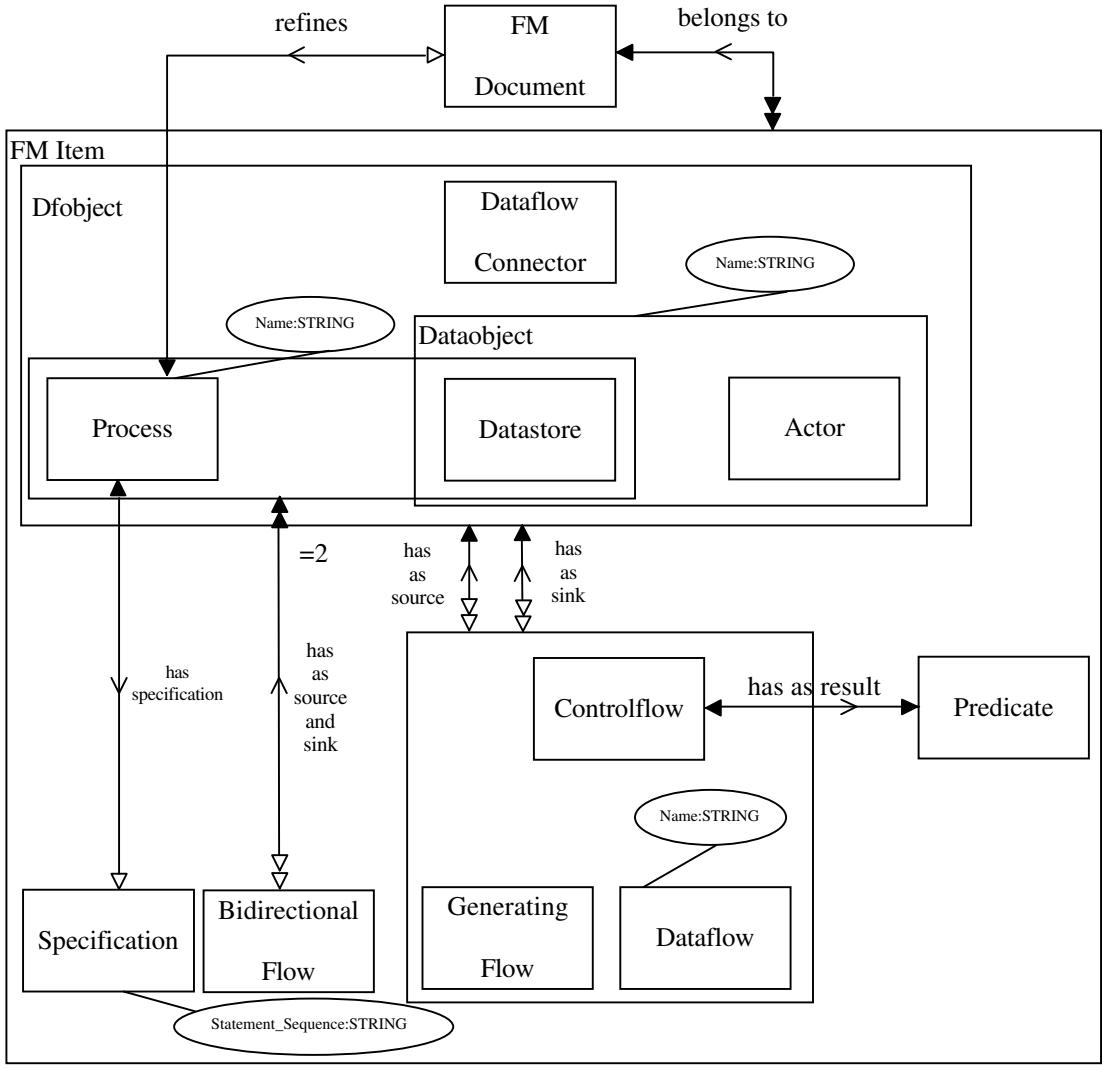
Process Refinement:



Diese Prozeßverfeinerung wird zwar in [Rumbaugh 92], Abbildung 6.10. verwendet, nicht aber in Anhang A aufgeführt. Sie stellt mit ihrem ebenenübergreifenden Charakter allerdings erst die Mächtigkeit des funktionalen Modells dar.

5.2 Das funktionale Modell in der Gesamtsicht

Bisher habe ich in Kapitel 5.1 die verwendeten Modellierungs-Komponenten des funktionalen Modells in einer Einzeldarstellung gezeigt. Im folgenden wird nun die Gesamtdarstellung präsentiert, wobei identische Entity- und Relations-Typen naturgemäß nur einmal vorkommen. Ferner werden die dazugehörigen Constraints des funktionalen Modells in ihrer Gesamtheit aufgeführt.



***Constraints* zum funktionalen Modell:**

Constraint F1:

Wenn die „has as source“-Kante den „Dataflow“ mit einem „Actor“ verbindet, so muß an der „has as sink“-Kante ein „Process“ stehen.

Wenn die „has as sink“-Kante den „Dataflow“ mit einem „Actor“ verbindet, so muß an der „has as source“-Kante ein „Process“ stehen.

Constraint F2: Jeweils eines der beteiligten Entities vom Typ „Dfobjekt“ beim „Bidirectional Flow“ muß ein „Process“ und das andere ein „Datastore“ sein.

Constraint F3: Jedes Dfobject hat Kontakt zu mindestens einem anderen Dfobject. Ausnahme ist der Prozeß auf der Kontextdiagrammebene.

Constraint F4: „Control Flow“s stehen nur in Beziehung mit Prozessen.

Constraint F5: „Generating Flow“s haben als Senke immer einen „Datastore“.

Constraint F6: Der normale „Dataflow“ sollte nicht gleichzeitig als Quelle und als Ziel einen „Datastore“ haben.

Constraint F7: Bei der Komposition von Datenflüssen muß der „Dataflow Connector“ zwei oder mehr Quelldatenflüsse und genau einen Zieldatenfluß haben.

Constraint F8: Bei der Dekomposition von Datenflüssen muß der „Dataflow Connector“ zwei oder mehr Zieldatenflüsse und genau einen Quelldatenfluß haben.

6 Schema-Diagramme für die Beziehungen zwischen den drei Modellen

In diesem Kapitel werden relevante Beziehungen zwischen den drei Modellen in Schema-Diagramme umgesetzt.

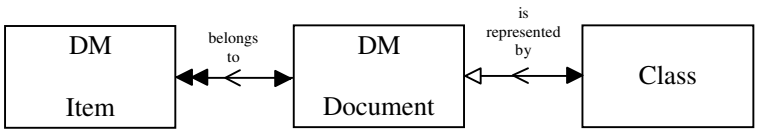
6.1 Schema-Diagramme für die einzelnen Beziehungen

In diesem Unterkapitel werden aus [Rumbaugh 92] Textstellen zitiert, die sich mit der Verschmelzung der drei Modelle beschäftigen. Diese werden dann jeweils in ein Schema-Diagramm umgesetzt, wobei die in den bisherigen Kapiteln eingeführten Entity-Typen verwendet werden. In Kapitel 6.2 werden die einzelnen Schema-Diagramme zu einem zusammengefügt.

Über jeder Textstelle steht das Kapitel, aus dem die Textstelle zitiert ist und die Seitenzahl.

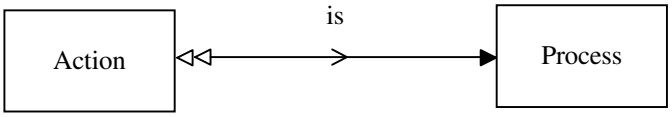
Kapitel 2.2.2, S.22

> *„Jedes Zustandsdiagramm zeigt den Zustand und die Ereignisfolgen, die in einem System für eine Klasse von Objekten zulässig ist.“*

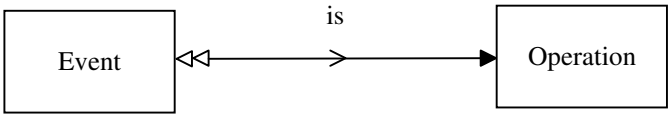


> *„Aktionen in Zustandsübergangsdiagrammen entsprechen Funktionen des funktionalen Modells.“*

Anmerkung : Wenn Rumbaugh von Funktionen spricht, meint er Prozesse.

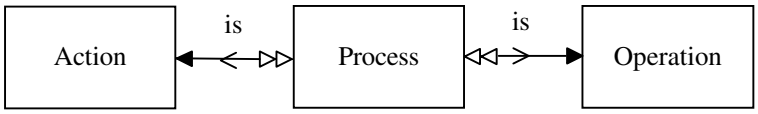


„Ereignisse in einem Zustandsdiagramm werden im Objektmodell zu Operationen auf Objekten.“



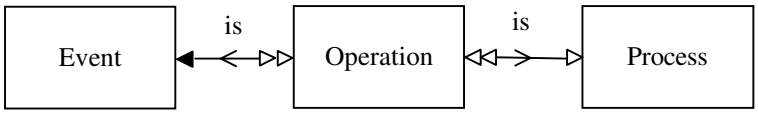
Kapitel 2.2.3, S.22

„Funktionen werden als Aktionen im dynamischen Modell aufgerufen und als Operationen auf Objekten im Objektmodell dargestellt.“

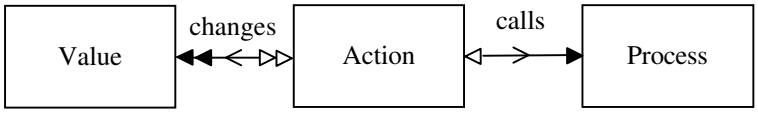


Kapitel 2.2.4, S.22f

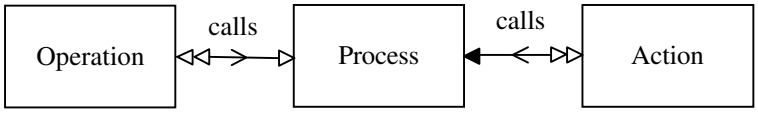
„Die Operationen im Objektmodell entsprechen den Ereignissen im dynamischen Modell und den Funktionen im funktionalen Modell.“



„Das dynamische Modell... kann Aktionen auslösen, die Objektwerte ändern und Funktionen aufrufen.“

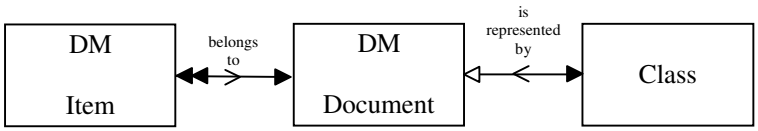


„Das funktionale Modell beschreibt Funktionen, die von Operationen im Objektmodell und Aktionen im dynamischen Modell aufgerufen werden.“



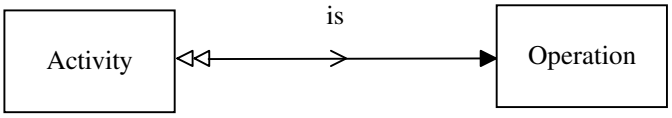
Kapitel 5.1, S.104

„Das Geflecht der Ereignisse, Zustände und Zustandsübergänge für eine gegebene Klasse kann abstrahiert als Zustandsdiagramm repräsentiert werden.“

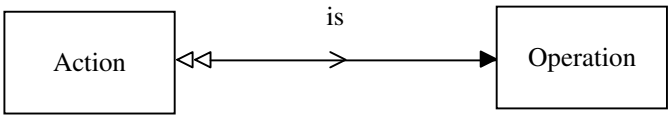


Kapitel 5.2.1, S.112f

„Eine Aktivität ist eine Operation, ...“

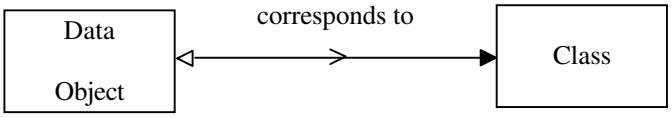


„Eine Aktion ist eine auf bestimmte Zeit beschränkte Operation.“



Kapitel 6.2.4, S.153f

„Sowohl Handlungsobjekte als auch Datenspeicher sind Objekte.“

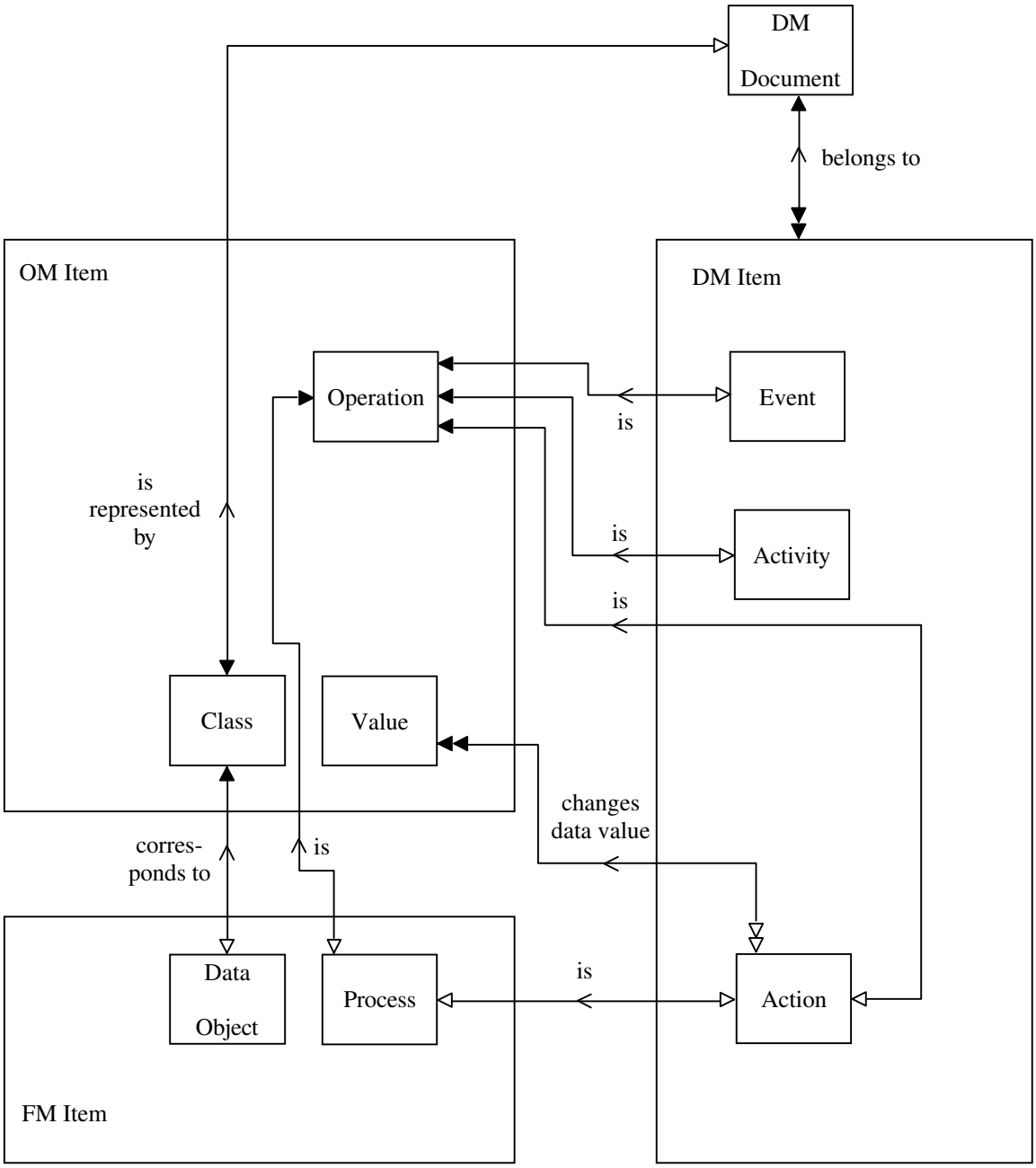


6.2 Die Beziehungen in der Gesamtsicht

In diesem Unterkapitel werden alle in Kapitel 6.1 aufgeführten Schema-Diagramme zu einem zusammengefügt.

Anmerkungen:

Kanten, die auf Grund der Textstellen bei Rumbaugh zwischen zwei Entity-Typen in beiden Richtungen vorkommen, habe ich zu einer verschmolzen. Ein Beispiel hierfür ist die „is“-Kante von „Process“ zu „Operation“ und die „is“-Kante von „Operation“ zu „Process“. Bei verschiedenen Namen solcher Kanten habe ich den jeweils günstigeren gewählt.



III Umsetzung in C++ Code

7 Allgemeines zur Umsetzung der einzelnen Komponenten in C++ Code

Im folgenden werden die in Teil II dieser Arbeit aufgeführten Komponenten des Meta-Modells in C++ Pseudocode umgesetzt. Dabei gehe ich folgendermaßen vor:

Für die einzelnen Modelle der Object Modeling Technique (Objekt-Modell, dynamisches Modell, funktionales Modell) stelle ich nacheinander jeweils zuerst die Ideen zur Umsetzung in Code und dann Beispielcodeerzeugungsvorgänge dar.

Auf Grund der Abhängigkeiten der drei Modelle untereinander (siehe Kapitel 6.2) benutze ich hier eine andere Reihenfolge: Das Objekt-Modell steht auch bei der Codeerzeugung im Mittelpunkt, wird also wegen seiner Bedeutung als erstes behandelt. Das funktionale Modell und das dynamische Modell hängen vom Objekt-Modell und auch voneinander ab. Hierbei ist allerdings das dynamische Modell mehr vom funktionalen Modell abhängig als umgekehrt. Wenn man dem Objekt-Modell das Wort „Was“, dem funktionalen Modell das Wort „Wie“ und dem dynamischen Modell das Wort „Wann“ zuordnet (siehe [Rumbaugh 92], Kapitel 6 Einleitung), wird deutlich: Bevor man das „Wann“ implementiert, sollte das „Wie“ bereits feststehen (anders bei der Modellierung, bei der das „Wie“ eher im Hintergrund steht). In den nächsten Kapiteln behandle ich also nach dem Objekt-Modell zunächst das funktionale Modell und dann erst das dynamische Modell.

Nun führe ich die grundlegenden Inhalte der Unterkapitel auf:

Ideen:

Für die einzelnen Komponenten der drei Modelle beschreibe ich zunächst die Ideen zur Umsetzung in C++ Code natürlichsprachlich. Dabei erwähne ich die Datenstruktur, eventuelle Voraussetzungen (erzeugter Code, der bereits vorhanden sein muß) und - falls relevant - die Stelle im zu erzeugenden Zielprogramm.

Beispiele:

Im Unter-Kapitel „Beispiele“ stelle ich Codeerzeugungsvorgänge exemplarisch dar. Als Eingabe dienen TGraphen, die den Schema-Diagrammen aus Teil II dieser Arbeit entsprechen. Ausgabe ist jeweils eine C++ Code-Sequenz, die auf Grund des jeweiligen Algorithmus in Kapitel 11 erzeugt werden sollte. Erzeugte Code-Zeilen werden grau schattiert dargestellt.

Algorithmen:

Im Kapitel 11 („Pseudocode-Algorithmen“) setze ich dann die Ideen in konkreten Pseudocode um. Die genaue Beschreibung des Pseudocodes befindet sich zu Beginn des Kapitels 11.

8 Umsetzung des Objekt-Modells in C++ Code

In diesem Kapitel beschreibe ich die Ideen zur Umsetzung der Komponenten des Objekt-Modells aus Kapitel 3 in C++ Code (Unterkapitel „Ideen“) und stelle dann Codeerzeugungsvorgänge exemplarisch dar (Unterkapitel „Beispiele“).

8.1 Ideen

In diesem Unterkapitel werden die Ideen zur Umsetzung der Komponenten des Objekt-Modells in C++ Code natürlichsprachlich beschrieben.

Class:

Jede im Objekt-Modell vorkommende Klasse wird im C++ Programm als Klasse mit ihren dazugehörigen Attributen und Operationen verwaltet. Gleichzeitig werden die Zusatzinformationen aus dem Objekt-Modell wie z.B. Datentyp, Argumente der Operationen und Ergebnistypen in die Klassendeklaration mit aufgenommen. Zusätzlich wird jeder Klasse ihre Konstruktor- und Destruktor-Methode hinzugefügt (sowohl im Definitions- als auch im Implementationsteil der Deklaration).

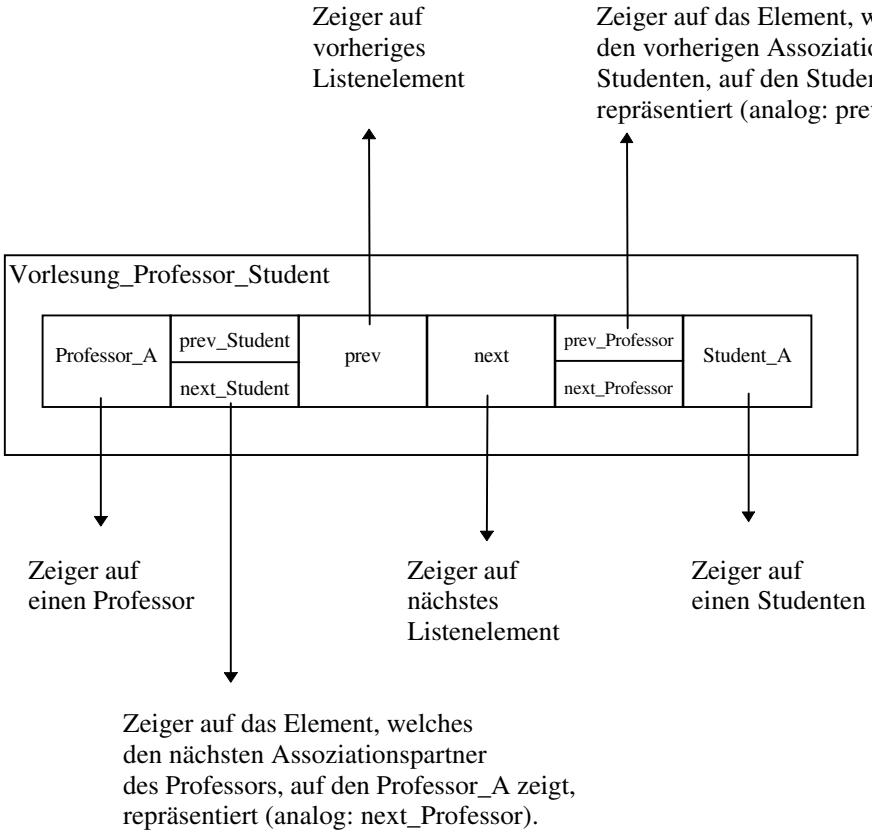
Association + Multiplicity of Association + Link Attributes + Association as Class / Aggregation

Assoziationen und Aggregationen werden als eigenständige Klassen verwaltet. Dabei werden die Attribute und Operationen analog zur Klassendeklaration (s.o.) in die Assoziations-Klassen-Deklaration aufgenommen.

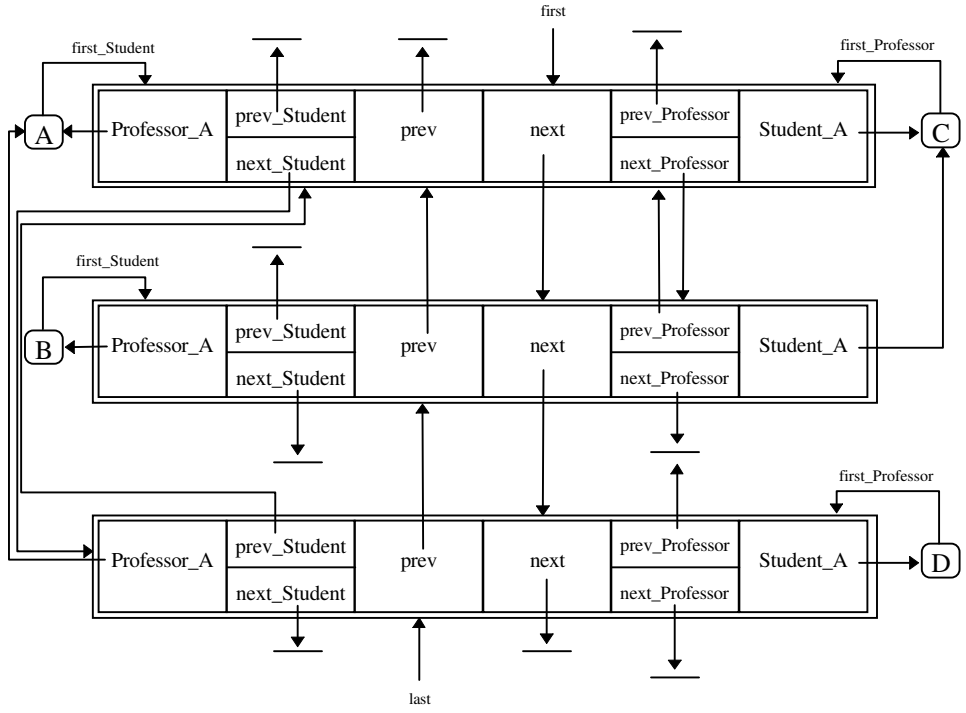
Voraussetzung im Rahmen dieser Arbeit ist, daß n-äre Assoziationen und Aggregationen binär modelliert vorliegen. Eine ternäre Assoziation wird vom Modellierer also in drei binäre Assoziationen unterteilt.

Im folgenden stelle ich das Konzept am Beispiel einer Assoziation namens „Vorlesung“, an der Instanzen der Klassen „Professor“ und „Student“ beteiligt sind, vor:

Grundlegende Datenstruktur eines einen Link repräsentierenden Listenelements: (Assoziationspartner sind Instanzen der Klassen Professor und Student (m:n-Assoziation))



Typische Listenstruktur:



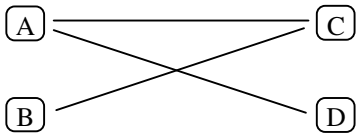
Diese Liste beinhaltet folgende Assoziationen:

Professor A liest für Student C und D

Professor B liest für Student C

Student C hört bei Professor A und B

Student D hört bei Professor A



Hierbei liegt in dem Satz „Student C hört bei Professor A und B“ gleichzeitig die Information, daß die Professoren A und B in genau dieser Reihenfolge in der Liste des Studenten C gespeichert sind. Dies gilt auch für das Beispiel auf den nächsten Seiten.

Erstmalige Beteiligung einer Objekt-Instanz an einer Assoziation:

Bei erstmaliger Beteiligung einer Objekt-Instanz an einer für sie erlaubten Assoziation wird von dieser Objekt-Instanz ein first-Zeiger auf das entsprechende Assoziations-Listenelement gesetzt.

Ermittlung aller Assoziationspartner einer Objekt-Instanz:

Wenn man alle Assoziationspartner einer Objekt-Instanz ermitteln will, steigt man über den first-Zeiger ein und kann alle beteiligten Assoziationspartner erreichen.

- Sei o eine Objekt-Instanz der Klasse Student. (s.o.).
- Man kann dann über den o–>first_Professor-Zeiger den ersten Professor ermitteln, bei dem o hört.
- Danach kann man über den next_Professor-Zeiger alle weiteren Professoren ermitteln.
- Sollte der letzte Professor erreicht sein, so zeigt next_Professor auf NULL.
- Über den prev_Professor-Zeiger kann man auch in die andere Richtung die beteiligten Professoren ermitteln.

Gebrauch der allgemeinen Zeiger:

Die allgemeinen next- und prev-Zeiger der Listenelemente, gekoppelt mit dem allgemeinen first- und last-Zeiger, erlauben eine Traversierung zur einfacheren Ermittlung aller aktuellen Assoziationen. Dies ist für Anfragen der folgenden Art sinnvoll:

Annahme:

> Es ist nicht bekannt, welche Professoren zur Zeit überhaupt an Assoziationen beteiligt sind. Das Problem ist also, daß man keine Einstiege über die first-Zeiger zur Verfügung hat.

Die folgende Anfrage wäre ohne allgemeinen prev- und next-Zeiger nicht möglich (die Gehaltsstufe ist ein Attribut der Klasse Professor; die Semesterzahl ist ein Attribut der Klasse Student):

> „Zeige mir alle Professoren, die eine C4-Professur innehaben und die zugehörigen Studenten des siebten Semesters, die bei diesem Professor Vorlesungen hören“

Durch die allgemeinen next- und prev-Zeiger ist die Ermittlung aller dieser Assoziationen möglich.

Einfügen und Löschen von Listenelementen:

Das Einfügen von Listenelementen geschieht im Konstruktor der Assoziationsklasse, das Löschen von Listenelementen im Destruktor.

Folgende Regeln gelten bezüglich der oben aufgeführten Listenstruktur der Assoziation „Vorlesung_Professor_Student“ beim Einfügen und Löschen von Assoziationsinstanzen:

- Das zu löschende Element hat drei Vorgänger und drei Nachfolger.
- Beim Löschen werden der prev- und der next-Zeiger des Vorgängers und des Nachfolgers so verbogen, daß auf das zu löschende Element kein Zeiger mehr zeigt.
- Dann gibt es jeweils noch einen Vorgänger und Nachfolger für die beiden beteiligten Klassen. Die Nachfolger sind über den next_Student- und den next_Professor-Zeiger, die Vorgänger über den prev_Student- und den prev_Professor-Zeiger zu ermitteln. Die Zeigerverbiegungen laufen dann nach dem üblichen Schema ab (wie auch bei prev und next).
- Wurden alle diese Zeiger verbogen, so ist das zu löschende Element verloren.
- Sonderfall ist, wenn auf das zu löschende Element ein first-Zeiger einer an der Assoziation beteiligten Objekt-Instanz zeigt, da es dann keinen Vorgänger gibt. Der first-Zeiger muß in diesem Fall verbogen werden.
- Das Einfügen beinhaltet auch die drei verschiedenen Einfüge-Operationen, die wie normale Einfüge-Operationen in verkettete Listen gehandhabt werden. Das Einfügen eines neuen Elementes geschieht in der allgemeinen Liste an letzter Stelle. Das Einfügen in die internen Listen geschieht jeweils an erster Stelle. Nach dem Einfügen zeigen also der first_Professor- und der first_Student-Zeiger auf das neue Element.

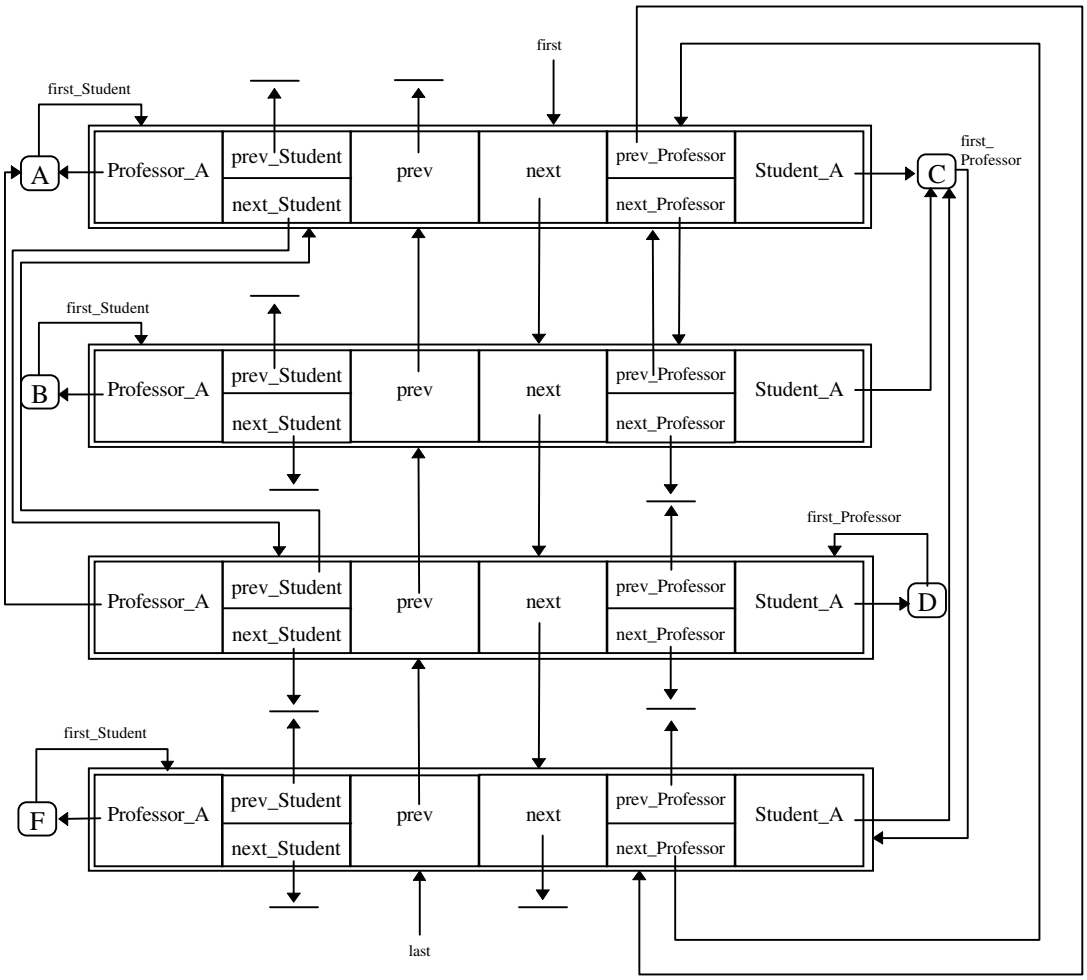
Beispielhaft wird im folgenden die bestehende Struktur erst erweitert und danach um ein Element verringert.

Die Folge der Operationen im Beispiel:

Vorlesung_Professor_Student::Vorlesung_Professor_Student(*F , *C) ;
Vorlesung_Professor_Student::Vorlesung_Professor_Student(*A , *E) ;
Vorlesung_Professor_Student::~Vorlesung_Professor_Student // this = (*B, *C);

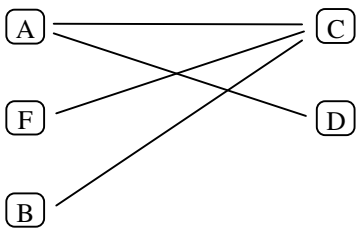
Vorlesung_Professor_Student::Vorlesung_Professor_Student(*F , *C) ;

Professor F ist neu, für Student C ist F der dritte Professor, bei dem er Vorlesungen hört.
Die Liste nach Durchführung dieser Methode:



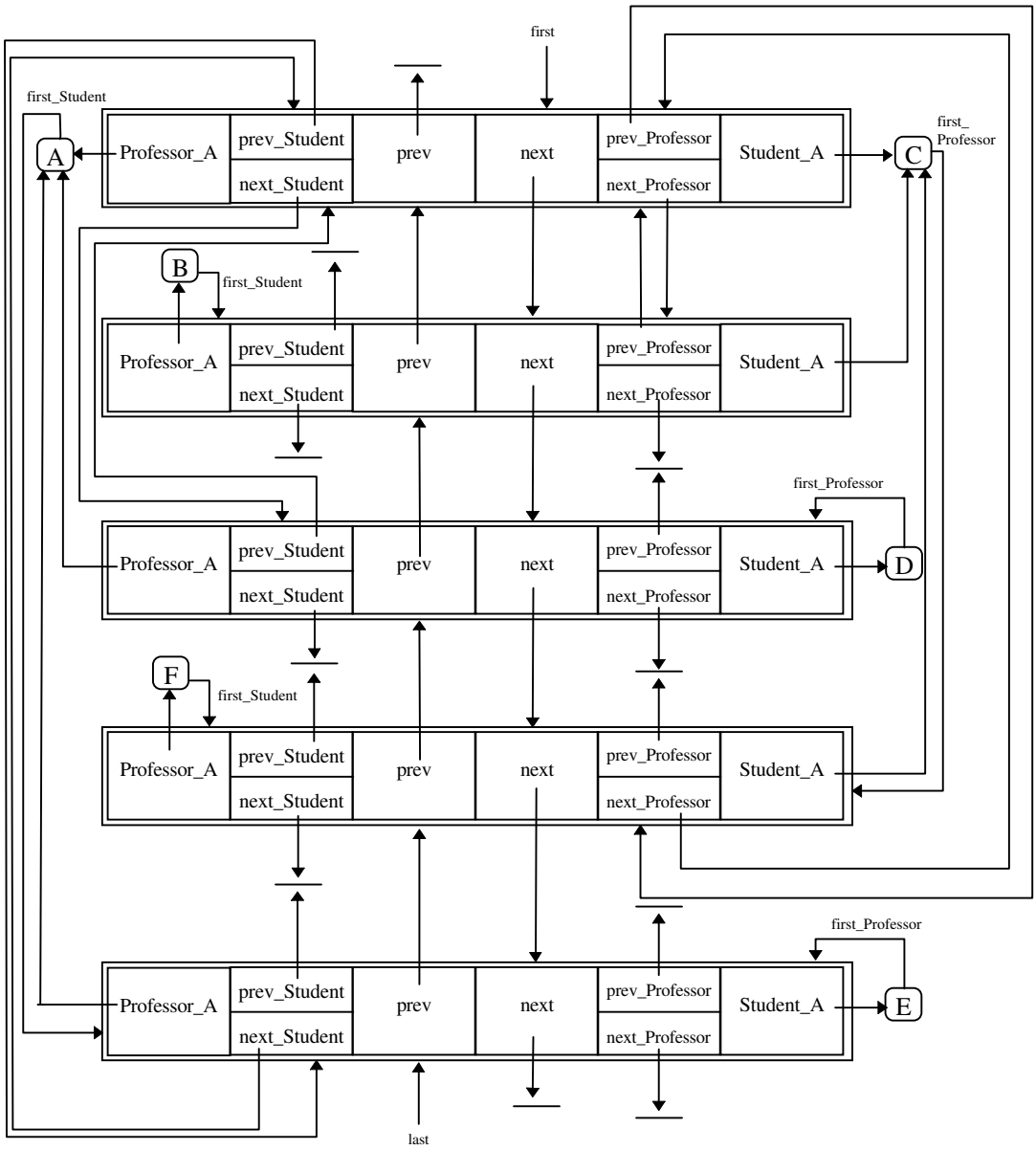
Die Liste beinhaltet nun folgende Assoziationen:

Professor A liest für Student C und D
Professor B liest für Student C
Professor F liest für Student C
Student C hört bei Professor F, A, und B
Student D hört bei Professor A



Vorlesung_Professor_Student::Vorlesung_Professor_Student(*A , *E) ;

Student E ist neu, er hört bei Professor A, für den er der dritte Besucher seiner Vorlesung ist.



Die Liste beinhaltet nun folgende Assoziationen:

Professor A liest für Student E, C und D

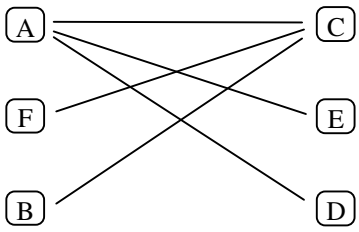
Professor B liest für Student C

Professor F liest für Student C

Student C hört bei Professor F, A und B

Student E hört bei Professor A

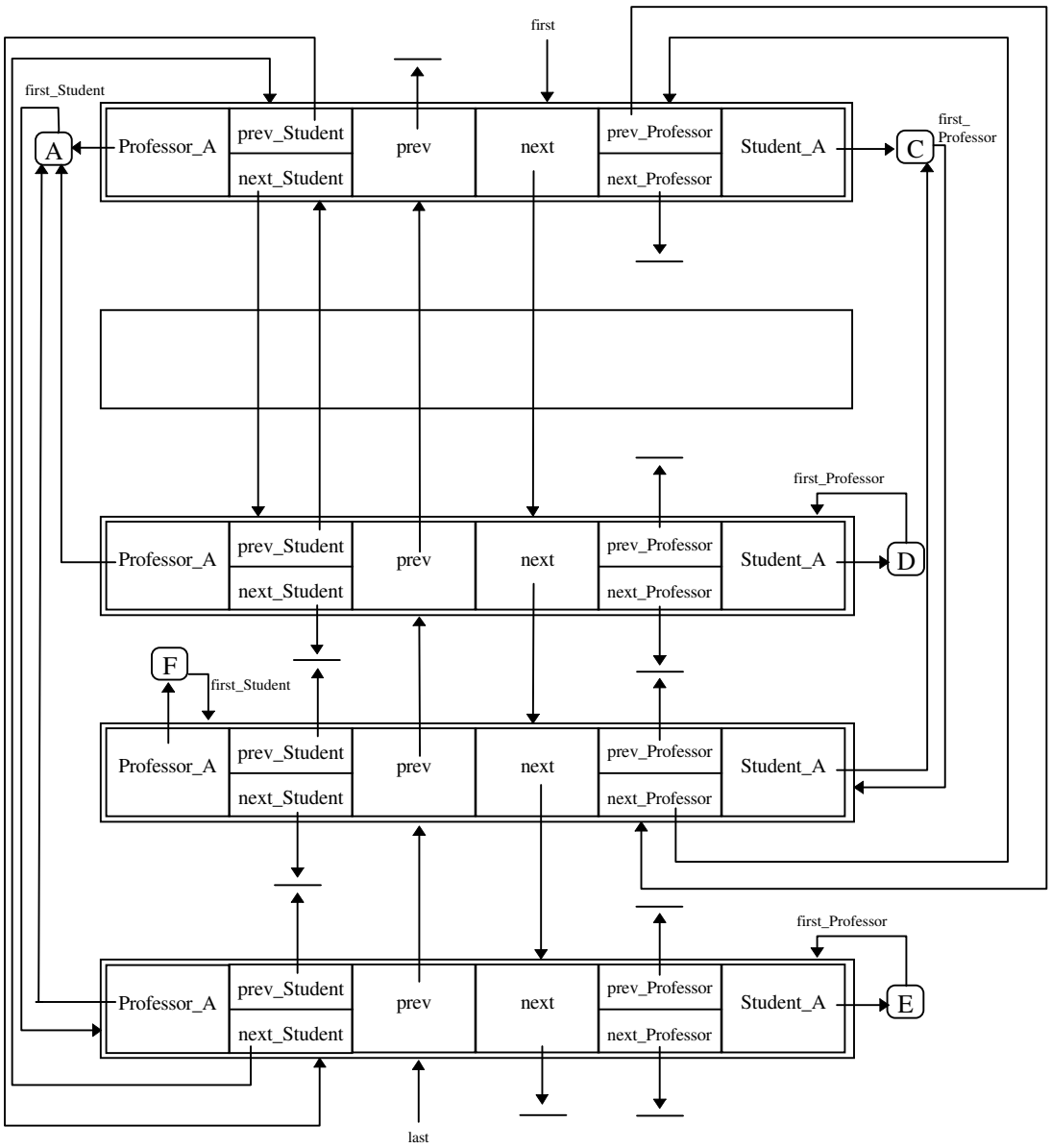
Student D hört bei Professor A



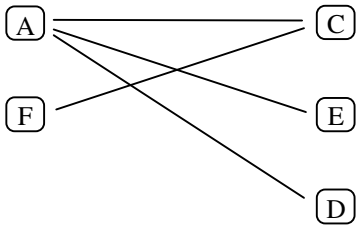
Vorlesung_Professor_Student::~Vorlesung_Professor_Student // this = (*B, *C);

Professor Bs Vorlesung wird wegen geringer Beteiligung gestrichen.

Student C besucht somit nur noch zwei Vorlesungen.



Diese Listenstruktur beinhaltet nun folgende Assoziationen:



Professor A liest für Student E, C und D

Professor F liest für Student C

Student C hört bei Professor F und A

Student E hört bei Professor A

Student D hört bei Professor A

Feststellung: Das Assoziationskonzept eignet sich für m:n-Assoziationen. Da 1:n und 1:1-Assoziationen Spezialfälle sind, können sie auch mit diesem Konzept bearbeitet werden.

Qualified Association

Der „Qualifier“ ist eine gute Hilfestellung für das Verstehen des Modells, besitzt aber keine Implementierungseigenschaft.

Generalization + Multiple Inheritance

Dies wird in C++ über das Konstrukt

class Unterklasse : public Oberklasse [(, public Oberklasse)*]

realisiert.

Voraussetzung ist, daß alle Klassendeklarationen schon vorhanden sind.

Die Implementierung der „Subclass“-Eigenschaft einer bereits deklarierten Klasse wird durch Anfügen der Superklassen an den bereits bestehenden Header der Klassendeklaration erreicht.

Aggregation:

Die „Aggregation“ ist eine spezielle „Association“, wird also auch wie sie behandelt (siehe oben). Eventuell kann in der Deklaration ein Extra-Attribut eingefügt werden, das beschreibt, daß eine Aggregation vorliegt.

Ordering:

Für geordnete Assoziationen gibt es kein äquivalentes Programmiersprachen-Konstrukt. Trotzdem kann das Attribut bei der entsprechenden Assoziation als True-Konstante verwaltet werden.

Object Instances / Instantiation Relationship:

Das System, welches modelliert wird, befindet sich zum Zeitpunkt der Modellierung in einem konkreten Zustand. Es liegen also aktuelle Attribut-Belegungen der Objekt-Instanzen vor. Die entsprechenden Attribute müssen also auf diese (Start-)Werte gesetzt werden. Die Informationen über den (Start-)Wert eines Attributs bekommt man aus dem Objekt-Modell über die Entitäten vom Typ „Value“, die über die „gives value to“-Kante mit der „Object Instance“ verbunden sind. Pro „Object Instance“ wird im Hauptprogramm eine Variable (die von einem Namensgenerator erzeugt wird) instantiiert.

Abstract Operations:

Dieses Konstrukt wird in C++ durch virtuelle Methoden umgesetzt.

Disjoint Subclasses:

Die Information, daß Unterklassen disjunkt sind, wird als Kommentar bei der entsprechenden Superklasse eingefügt.

Discriminator:

Der Diskriminator wird als Kommentar bei der Superklasse in der folgenden Form abgelegt:
„Das Attribut ... ist Diskriminator für die Unterklassen ...“

Class Operations + Class Attributes:

Dies wird in C++ über das Schlüsselwort „static“ vor der entsprechenden Attribut- oder Operationsdeklaration umgesetzt.

Derived Class + Derived Association:

Die Information wird bei der entsprechenden Klasse bzw. Assoziation als Kommentar abgelegt.

Derived Attributes:

Die Information, ob ein Attribut abgeleitet ist, wird als Kommentar unterhalb der Attribut-Deklarationen aufgeführt.

Propagation of Operations:

Diese Information wird bei der entsprechenden Operation als Kommentar der Form „pflanzt sich fort zu Operation ...“ verwaltet.

Constraints on Objects:

Die Constraints werden bei den jeweiligen Klassendeklarationen als Kommentar eingefügt.

Constraints between Associations:

Die Subset-Beziehung wird bei der jeweiligen Assoziations-Deklaration als Kommentar der Form „ist Teilmenge von ...“ eingefügt.

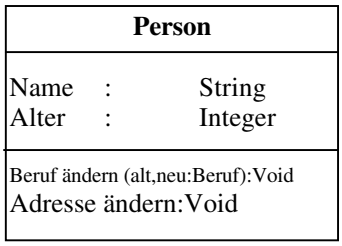
8.2 Beispiele

In diesem Unterkapitel wird auf Grund der Ideen des letzten Unterkapitels beispielhaft Code aus dem Objekt-Modell erzeugt. Dabei werden für die einzelnen Komponenten jeweils drei Entwicklungsstufen dargestellt:

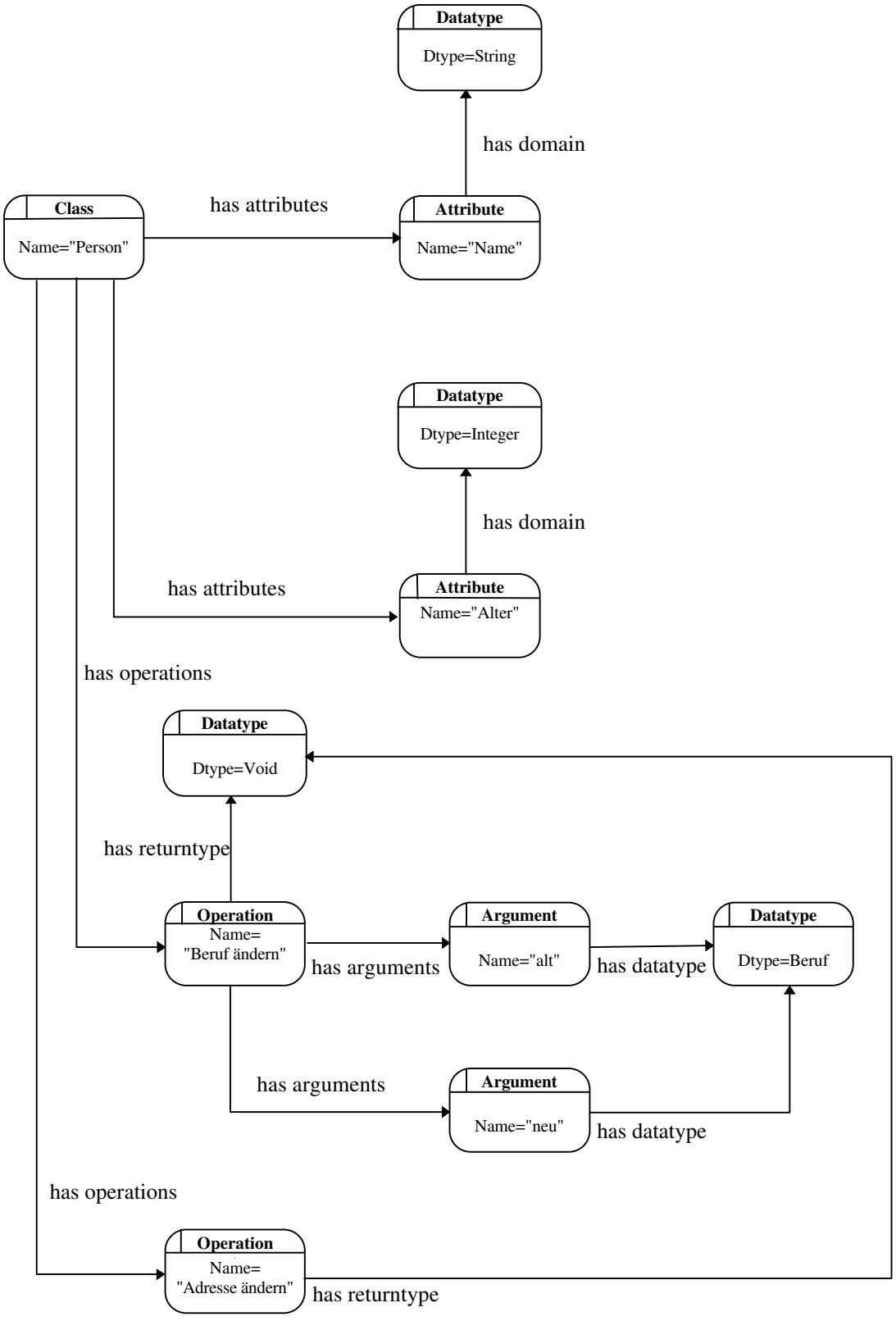
1. Die Modellierung nach Rumbaugh
2. Der zugehörige TGraph
3. Der daraus erzeugte C++ Code

Class:

Dieses Beispiel befindet sich in [Rumbaugh 92] auf S. 33, Abbildung 3.4, und ist um die Argumente „alt“ und „neu“ bei der Operation „Beruf ändern“ erweitert:



Daraus resultierender TGraph:



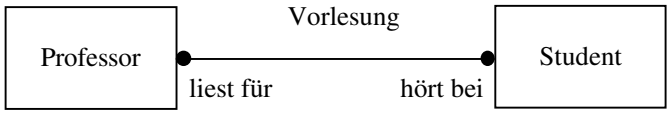
Folgender Code sollte erzeugt werden:

```
class Person {
        string Name ;
        int Alter ;
        Person() ;
        ~Person() ;

        void Beruf_ändern(beruf alt, beruf neu) ;
        void Adresse_ändern() ;
};
void Person::Person() {    }
void Person::~Person() {   }
void Person::Beruf_ändern(beruf alt, beruf neu) {   }
void Person::Adresse_ändern() {   }
```

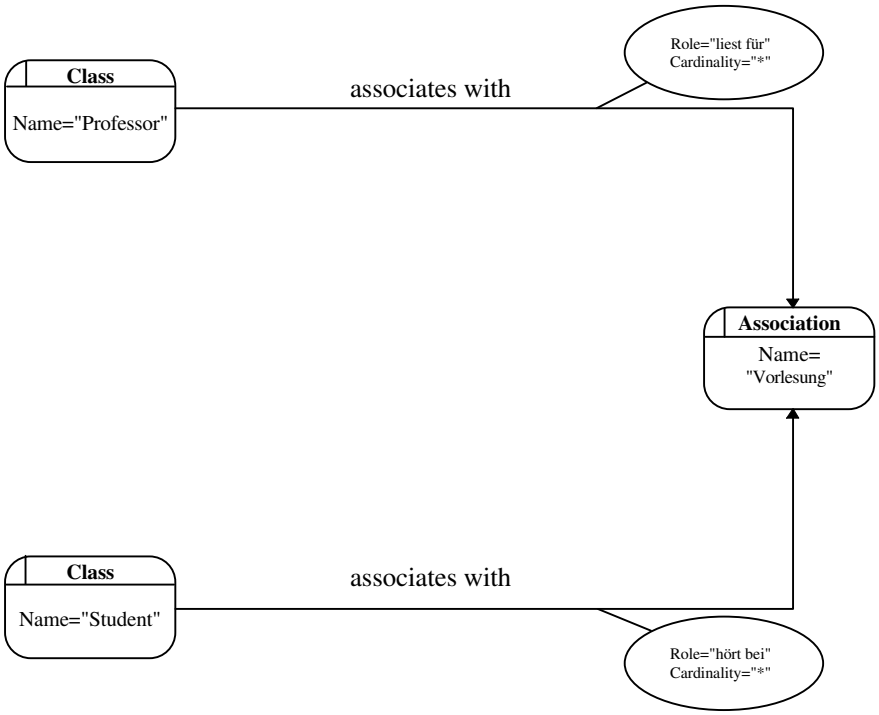
Association + Multiplicity of Association + Link Attributes + Association as Class

Bei diesem Beispiel habe ich Assoziations-Attribute und Assoziations-Operationen weggelassen, da bei der Codeerzeugung genauso vorgegangen wird, wie bei Klassen (siehe oben).

Modellierung nach Rumbaugh:



Daraus resultierender TGraph:



Daraus erzeugter Code:

Die beteiligten Klassen werden um folgende Zeilen erweitert:

```
class Professor
{ ...
Vorlesung_Professor_Student* first_Student ;
```

```
class Student
{ ...
Vorlesung_Professor_Student* first_Professor ;
```

Die Assoziation wird als eigenständige Klasse verwaltet:

```
class Vorlesung_Professor_Student ;
{
        Grundlegende Datenstruktur ist die folgende:
        class Professor* Professor_A ;
        class Student* Student_A ;

        Vorlesung_Professor_Student* next_Professor ;
        Vorlesung_Professor_Student* next_Student ;
        Vorlesung_Professor_Student* prev_Professor ;
        Vorlesung_Professor_Student* prev_Student ;

        Vorlesung_Professor_Student* next ;
        Vorlesung_Professor_Student* prev ;

        static Vorlesung_Professor_Student* first ;
        static Vorlesung_Professor_Student* last ;

        Vorlesung_Professor_Student( Professor* oid_Professor , Student* oid_Student) ;
        ~Vorlesung_Professor_Student() ;

}
```

Anmerkung: Der Präfix „oid“ steht für „object identifier“

Die Implementierung des Konstruktors:

```
Vorlesung_Professor_Student::
Vorlesung_Professor_Student(Professor* oid_Professor , Student* oid_Student)

{
        derived = FALSE ;
        // Element in die allgemeine Liste ans Ende einfügen
        if (last==NULL)
        {
                first = new Vorlesung_Professor_Student ;
                last = first ;
        }
        else
        {
                last–>next = new Vorlesung_Professor_Student ;
                last–>next–>prev = last ;
                last = last–>next ;
        }

        // Zeiger auf übergebene Objekte setzen
        last–>Professor_A = oid_Professor ;
        last–>Student_A = oid_Student ;

        // Professor in Liste des Studenten an 1. Stelle einfügen
        last–>next_Professor = oid_Student–>first_Professor ;
        last–>next_Professor–>prev_Professor = last ;
        last–>prev_Professor = NULL
        oid_Student–>first_Professor = last ;

        // Student in Liste des Professors an 1.Stelle einfügen
        last–>next_Student = oid_Professor–>first_Student ;
        last–>next_Student–>prev_Student = last ;
        last–>prev_Student = NULL
        oid_Professor–>first_Student = last ;
}
```

Die Implementierung des Destruktors:

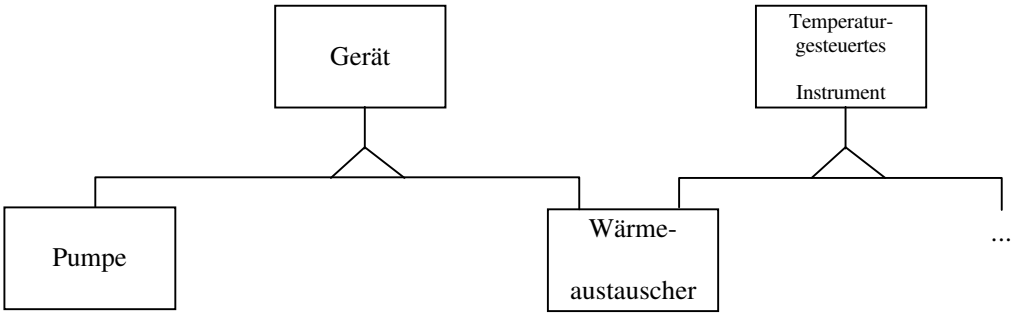
```
Vorlesung_Professor_Student::~Vorlesung_Professor_Student()
{
        // das zu löschende Element ist über den this-Zeiger zu erreichen


        // first-Zeiger sichern, falls this das jeweils erste Element ist
        if this–>Student_A–>first_Professor == this
        {
                this–>Student_A–>first_Professor = this–>next_Professor
        }
```

```
        if this–>Professor_A–>first_Student == this
        {
                this–>Professor_A–>first_Student = this–>next_Student
        }

        // Aus der Professor-Liste Student streichen
        if this–>next_Student != NULL
        {
                this–>next_Student = this_next_Student–>next_Student ;
        }

        if this–>prev_Student != NULL
        {
                this–>prev_Student = this–>prev_Student–>prev_Student ;
        }

        // Aus der Student-Liste Professor streichen
        if this–>next_Professor != NULL
        {
                this–>next_Professor = this–>next–>Professor–>next_Professor ;
        }
        if this–>prev_Professor != NULL
        {
                this–>prev_Professor = this–>prev_Professor–>prev_Professor ;
        }

        // Element aus der allgemeinen Liste löschen
        if this–>next != NULL
        {
                this–>next–>prev = this–>prev ;
        }
        if this–>prev != NULL
        {
                this–>prev–>next = this–>next ;
        }
}
```

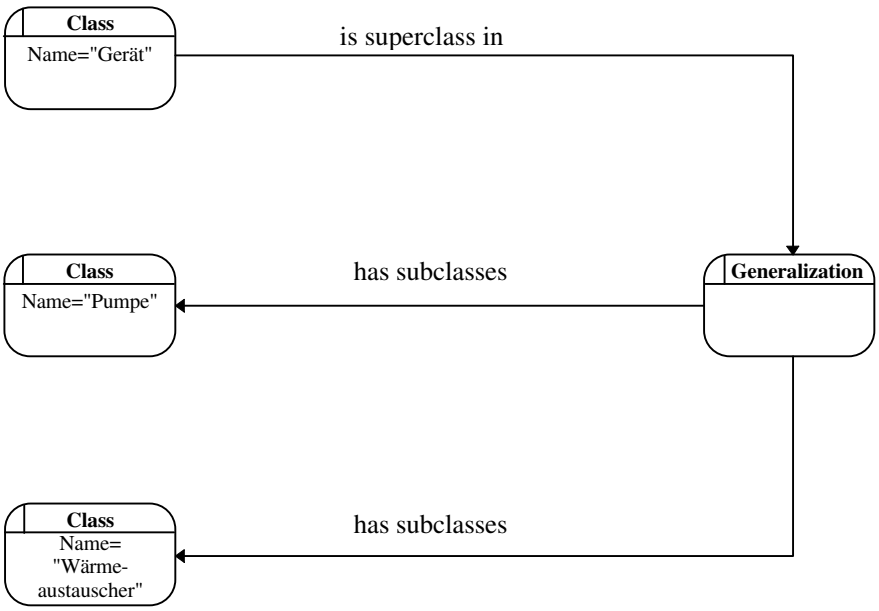
Das hier vorgestellte Idee der Codeerzeugung für Assoziationen läßt sich fast analog für Aggregationen durchführen. Da Aggregationen keinen Namen besitzen, wird dieser per Namensgenerator erzeugt. Weiterhin fällt die Initialisierung des Assoziationsattributs „derived“ weg.

Generalization + Multiple Inheritance

Dieses Beispiel befindet sich in [Rumbaugh 92] S.49, Abb.3.23 und ist um die Klasse „Temperaturgesteuertes Instrument“ erweitert



Daraus resultierende Instanz des Meta-Modells:



bereits erzeugter Code (Annahme):

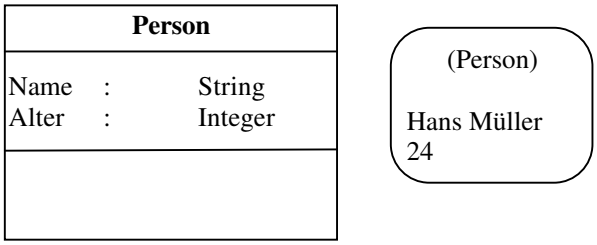
```
class Pumpe

class Wärmeaustauscher : public Temperaturgesteuertes_Instrument
```

nach Erzeugungsalgorithmus:

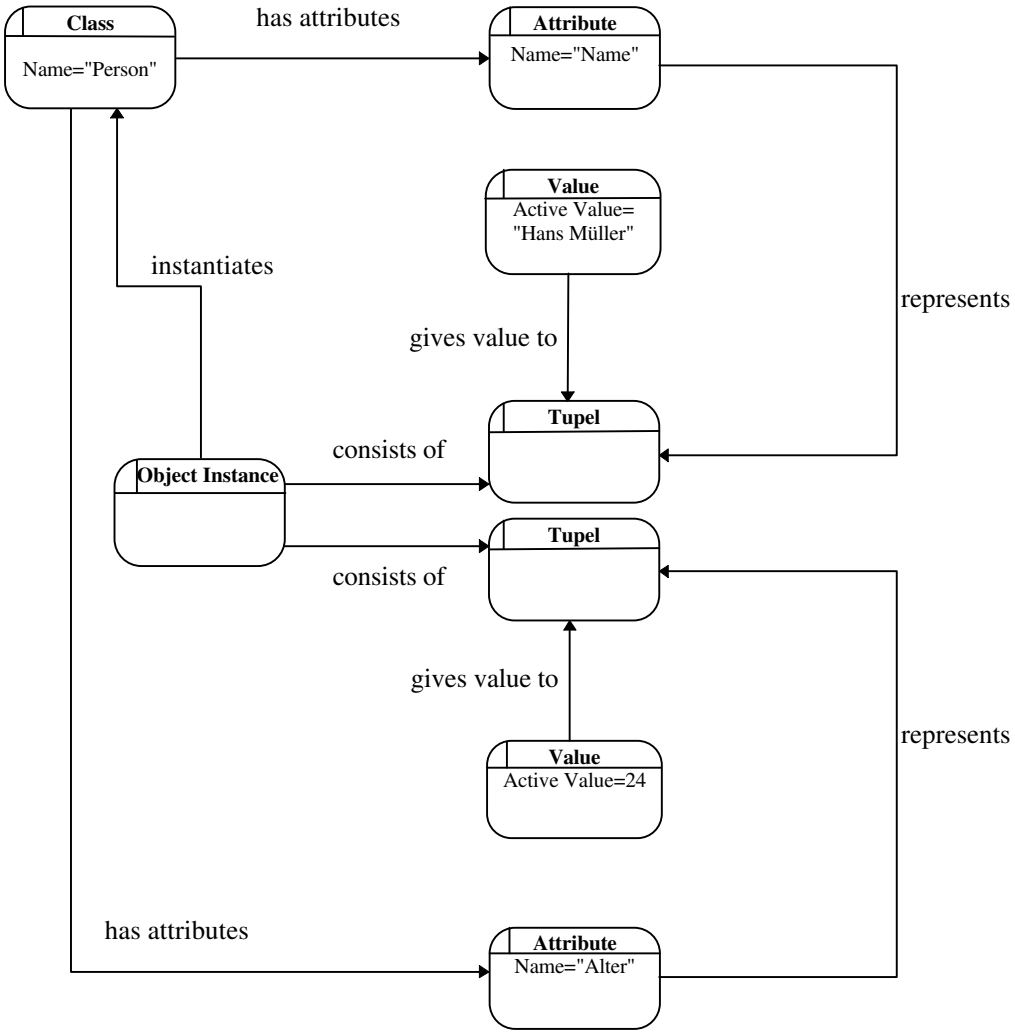
```
class Pumpe : public Gerät

class Wärmeaustauscher : public Temperaturgesteuertes_Instrument, public Gerät
```

Object Instances / Instantiation Relationship:

Dieses Beispiel befindet sich in [Rumbaugh 92], S.30, Abb.2



TGraph:



Erzeugter Code:

```
Person Objekt_77 ;
```

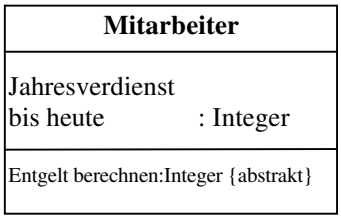
Der Name Objekt_77 ist Ergebnis eines Namensgenerators (sollte auch heißen: dieses Objekt ist das 77. im System).

```
Objekt_77–>Name = "Hans Müller" ;
```

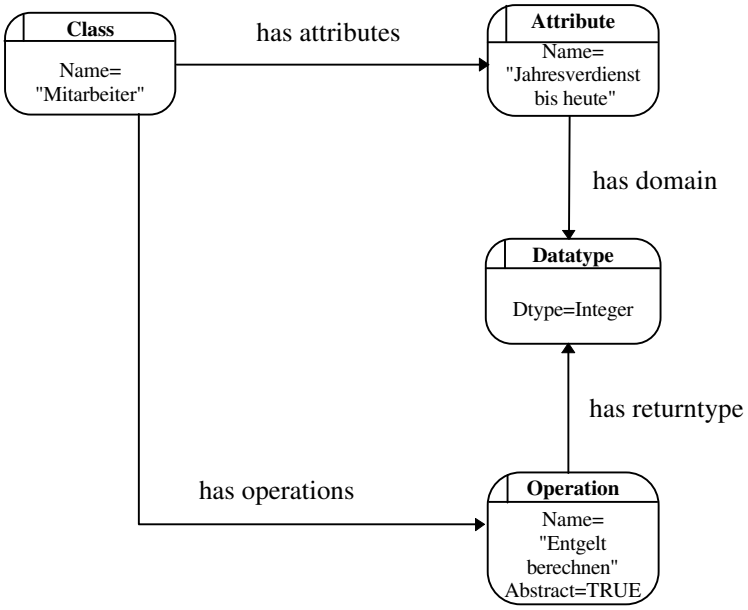
```
Objekt_77–>Alter = 24 ;
```

Abstract Operations:

Dieses Beispiel ist aus [Rumbaugh 92] S.76, Abb. 4.7



Dazugehöriger TGraph:



Später sollte dieser Code vorliegen:

```
class Mitarbeiter {
...
virtual int Entgelt_berechnen()
...
}
```

Vor der Erzeugung der Code-Sequenz

int Entgelt_berechnen()

wird das Schlüsselwort

virtual

eingefügt.

Die Beispiele für „Class Operations“, „Class Attributes“, „Derived Attributes“, Disjoint Subclasses“, „Discriminator“, „Derived Class“, „Propagation of Operation“, „Constraints on Objects“ und „Constraints between Associations“ laufen analog zu „Abstract Operations“ ab. Hier werden entsprechende Schlüsselwörter vor bzw. eine Kommentarzeile hinter die entsprechende Deklaration gesetzt.

9 Die Umsetzung des funktionalen Modells in C++ Code

In diesem Kapitel beschreibe ich die Ideen zur Umsetzung der Komponenten des funktionalen Modells aus Kapitel 5 in C++ Code (Unterkapitel „Ideen“) und stelle dann Codeerzeugungsvorgänge exemplarisch dar (Unterkapitel „Beispiele“).

9.1 Ideen

In diesem Unterkapitel werden die Ideen zur Umsetzung der Komponenten des funktionalen Modells in C++ Code natürlichsprachlich beschrieben.

Process / Process Refinement:

Für jeden Prozeß auf der untersten Ebene (in den keine refines-Kante mehr eingeht) wird die dazugehörige Operation gesucht (über die Process.is.Operation-Kante zwischen funktionalem und Objekt-Modell) und die entsprechende Spezifikation als Kommentar eingefügt.

Danach werden die Prozesse, die eine Stufe höher stehen, betrachtet. Im entsprechenden Operations-Body werden die Aufrufe der untergeordneten Prozesse (bzw. deren entsprechenden Operationen) in Kommentarzeilen plaziert. Der Implementierer weiß so, welche Methoden von der jeweiligen Methode aus aufgerufen werden.

Die Parameter und Ergebnistypen der einzelnen Operationen, die man beim Aufruf benötigt, bekommt man aus dem Objekt-Modell. Dabei müssen z.B. das „Argument“ aus dem Objekt-Modell und der „Dataflow“ aus dem funktionalen Modell zueinander passen.

So durchläuft man analog zum beschriebenen Verfahren den Prozeß-Baum bis zur Wurzel. Der Wurzel-Prozeß hat die Eigenschaft, daß er eingehende, aber keine ausgehende refines-Kanten besitzt.

Data Store / Actor:

Diese sind im funktionalen Modell Ein- bzw. Ausgaben von Prozessen. Sie sind also Parameter, die entsprechend der Idee von oben in die Berechnung eingehen, also über die Argumentenliste der Operation bzw. über den „Return-Type“ aus dem Objekt-Modell.

Data Flow between Processes:

Dies ist bei Process / Process Refinement bereits enthalten. Hierbei liegt der Fall vor, daß das Ergebnis eines Prozesses als Parameter in einen anderen Prozeß eingeht. Der Wert, den der Dataflow repräsentiert, ist also Argument der Operation, die den Ziel-Prozeß repräsentiert.

Data Flow that results in a Data Store:

Hierbei wird das Ergebnis in einem Datenspeicher abgelegt. Die Operation wird also einer Variablen zugeordnet.

Actor as source or sink of flow:

Man verfährt wie oben (Actor ist spezieller Datenspeicher)

Control Flow:

Hier geht der Control Flow als Parameter in den Prozeß ein (Boolescher Wert). Dies gilt natürlich nur für Prozesse, die über die „has as source"-Kante mit einem Control Flow verbunden sind.

Beim Programmlauf könnte vor Aufruf des Prozesses (also der entsprechenden Operation) geprüft, ob das Prädikat den Wert TRUE hat. Falls ja, würde der Prozeß gestartet, falls nicht, würde er verzögert.

Die Umsetzung des „Control Flow" wird in dieser Arbeit nicht durchgeführt, da sie sich auf den Programmlauf bezieht.

Access of Data Store Value / Update of Data Store Value /Access and Update of Data Store Value:

Dies ist ebenfalls beim Prozeß-Konzept eingearbeitet. Hierbei werden die Parameter aus einem Datenspeicher geholt (Access) bzw. das Ergebnis in einem Datenspeicher abgelegt (Update).

Composition / Decomposition / Duplication of Data Value:

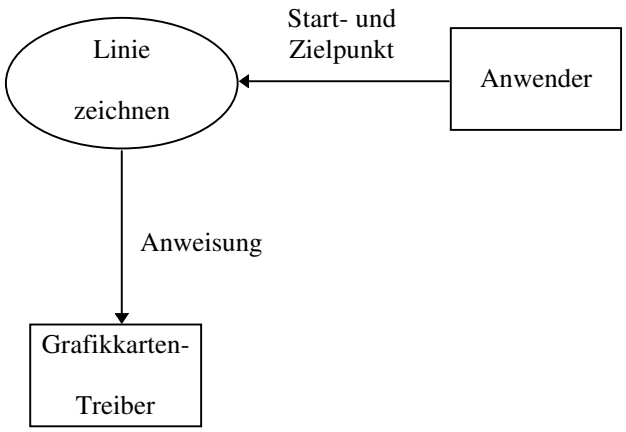
Diese Information bezüglich des Zusammenflusses bzw. des Zerlegens von Datenflüssen ist zum Verstehen des Modells notwendig. Aggregierte Datenwerte werden allerdings nicht besonders behandelt, sondern genau wie die oben erwähnten normalen Datenwerte.

9.2 Beispiele

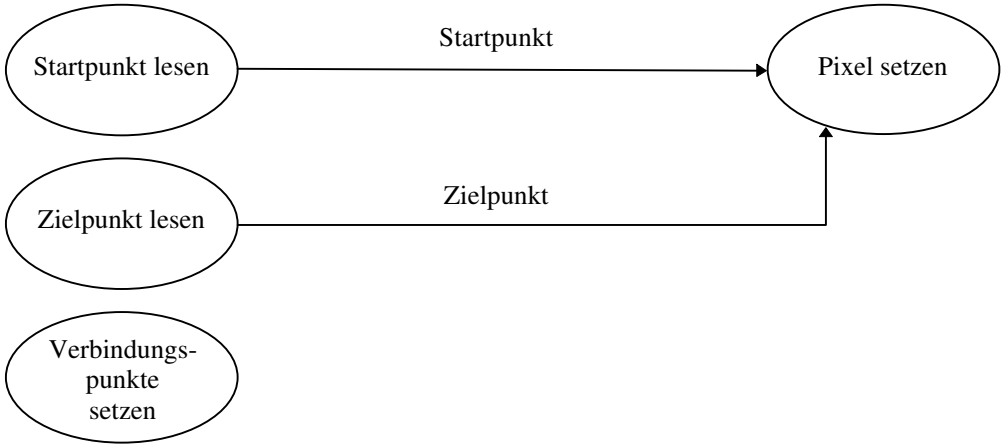
In diesem Unterkapitel wird auf Grund der Ideen des letzten Unterkapitels beispielhaft Code aus dem funktionalen Modell erzeugt. Dabei werden für die einzelnen Komponenten jeweils drei Entwicklungsstufen dargestellt:

1. Die Modellierung nach Rumbaugh
2. Der zugehörige TGraph
3. Der daraus erzeugte C++ Code

Beispiel nach Rumbaugh:



Verfeinerung von Prozeß „Linie zeichnen“

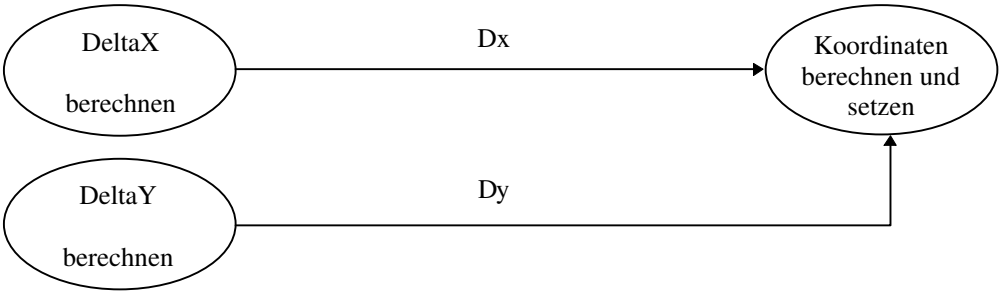


Spezifikation für „Startpunkt lesen“: "Liest den Startpunkt "

Spezifikation für „Zielpunkt lesen“: "Liest den Zielpunkt "

Spezifikation für „Pixel setzen“: "Setzt Pixel mit den übergebenen Koordinaten "

Verfeinerung von Prozeß „Verbindungspunkte setzen“



Spezifikation für „DeltaX berechnen“:

"Dx = (Zx - Sx)/Normalenvektor"

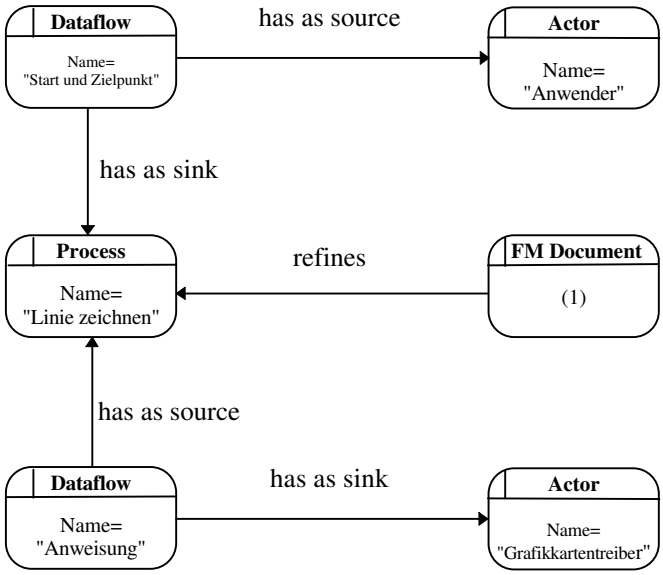
Spezifikation für „DeltaY berechnen“:

"Dy = (Zy - Sy)/Normalenvektor"

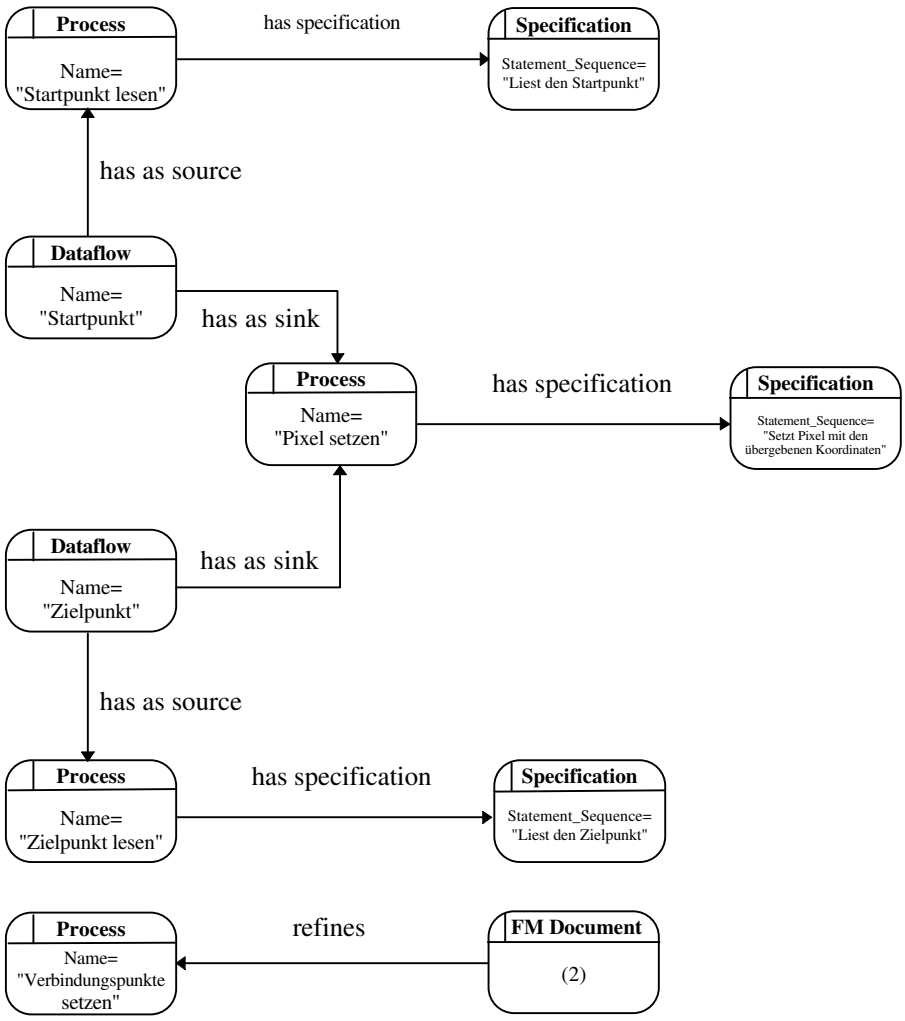
Spezifikation für „Koordinaten berechnen und setzen“:

"vector i
for (i=(Sx,Sy) ; i<(Zx,Zy) ; i=i+(Dx,Dy))
{ draw i }"

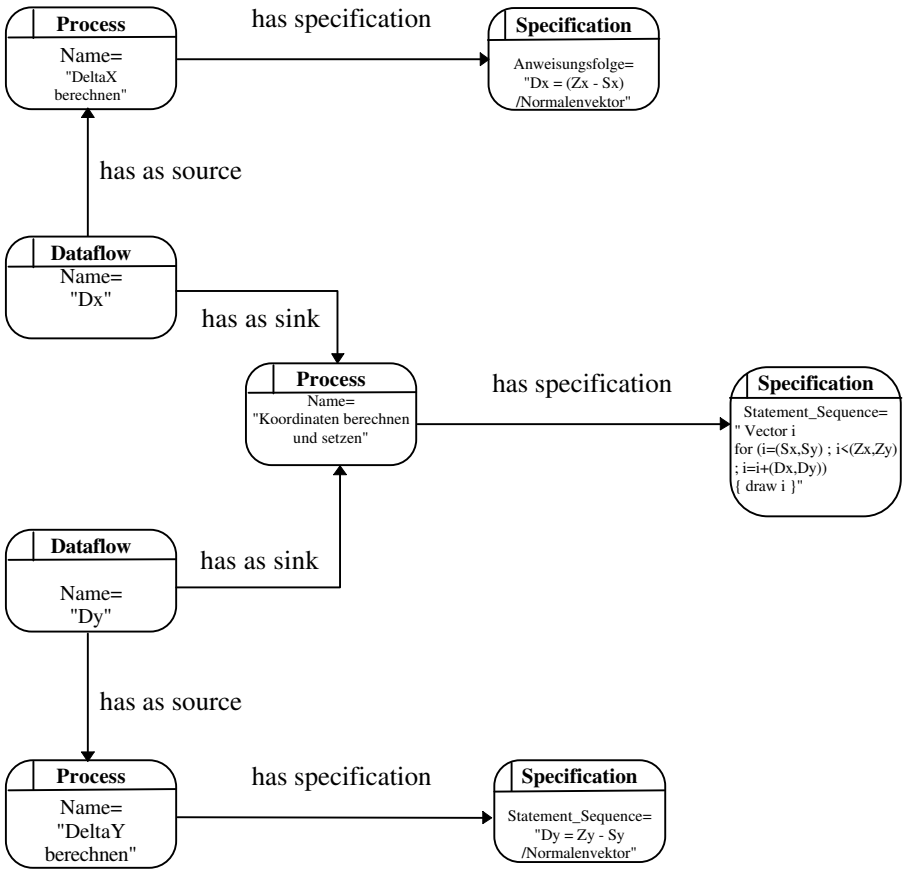
Daraus resultierender TGraph:



Alle auf dieser Seite aufgeführten Entities (bis auf „FM Document(2)“) sind über eine „belongs to“-Kante mit „FM Document(1)“ verbunden.



Alle auf dieser Seite aufgeführten Entities sind über eine „belongs to“-Kante mit „FM Document(2)“ verbunden.



Erzeugter Code:

Voraussetzung (zur Vereinfachung des Beispiels): Alle Operationen op, die über die is-Kante mit einem Prozeß p verbunden sind, haben den gleichen Namen (p.Name == op.Name).

Wenn ein Blattprozeß bearbeitet wird, wird der folgende Code in den entsprechenden Operations-Body eingefügt.

Bereits erzeugter Code, der jeweils erweitert wird, nachdem die Informationen über die Signatur dem Objekt-Modell entnommen worden ist und das "{" der entsprechenden Blattprozeß-Operation erzeugt worden ist):

```
void Startpunkt_lesen (vector (Sx,Sy), (Zx,Zy))
{

void Zielpunkt_lesen (vector (Sx,Sy), (Zx,Zy))
{

void Pixel_setzen (vector (x,y))
{

int DeltaX_berechnen (vector (Sx,Sy), (Zx,Zy))
{

int DeltaY_berechnen (vector (Sx,Sy), (Zx,Zy))
{

void Koordinaten_berechnen_und_setzen (vector (Sx,Sy), (Zx,Zy), (Dx,Dy))
{
```

nach dem Durchlauf der Schleifen, die die jeweiligen Operationen behandeln, liegt folgender Code vor:

```
void Startpunkt_lesen (vector (Sx,Sy), (Zx,Zy))
{ // Liest den Startpunkt

void Zielpunkt_lesen (vector (Sx,Sy), (Zx,Zy))
{ // Liest den Zielpunkt

void Pixel_setzen (vector (x,y))
{ // Setzt Pixel mit den übergebenen Koordinaten

int DeltaX_berechnen (vector (Sx,Sy), (Zx,Zy))
{ // Dx = (Zx - Sx)/Normalenvektor

int DeltaY_berechnen (vector (Sx,Sy), (Zx,Zy))
{ // Dy = (Zy - Sy)/Normalenvektor

void Koordinaten_berechnen und setzen (vector (Sx,Sy), (Zx,Zy), (Dx,Dy))
{
// vector i; for (i=(Sx,Sy) ; i<(Zx,Zy) ; i = i+(Dx,Dy)) { draw (i) }
```

Bei den Prozessen, die keine Blattprozesse sind, geschieht folgendes:

Vorher erzeugter Code:

```
void Verbindungspunkte_setzen (vector (Sx,Sy), (Zx,Zy))
{

void Linie_zeichnen (vector (Sx,Sy), (Zx,Zy))
{
```

hinterher vorliegender Code:

```
void Verbindungspunkte_setzen (vector (Sx,Sy), (Zx,Zy))
{
// DeltaX_berechnen( (Sx,Sy), (Zx,Zy) );
// DeltaY_berechnen( (Sx,Sy), (Zx,Zy) );
// Koordinaten_berechnen und setzen ((Sx,Sy), (Zx,Zy), (Dx,Dy));
```

Anmerkung:

Der hier erzeugte Code kann sich an mehreren Stellen in der entsprechenden Methode befinden. Dies wird später beim dynamischen Modell deutlich.

10 Die Umsetzung des dynamischen Modells in C++ Code

In diesem Kapitel beschreibe ich die Ideen zur Umsetzung der Komponenten des dynamischen Modells aus Kapitel 4 in C++ Code (Unterkapitel „Ideen“) und stelle dann Codeerzeugungsvorgänge exemplarisch dar (Unterkapitel „Beispiele“).

10.1 Ideen

In diesem Unterkapitel werden die Ideen zur Umsetzung der Komponenten des dynamischen Modells in C++ Code natürlichsprachlich beschrieben.

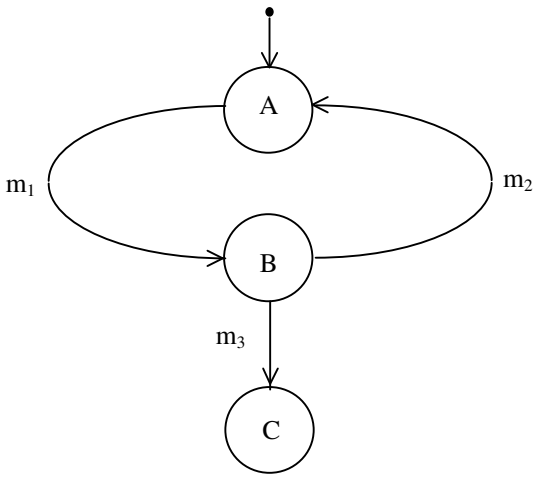
Das dynamische Modell besteht aus mehreren Zustandsübergangsdiagrammen. Jedes dieser Zustandsübergangsdiagramme ist genau einer Klasse zugeordnet.

Szenario:

Klasse:

Klasse 1
a_1 ... a_n
m_1 ... m_n

zugehöriges Zustandsübergangsdiagramm:



Die Methoden m_1 bis m_3 kommen also als Ereignisse im Zustandsübergangsdiagramm vor.

Bei der Erzeugung der besagten Methoden wird folgendermaßen vorgegangen (es ist bereits alles deklariert, wir befinden uns im Body der Methode direkt hinter dem erzeugten "{"):

Es wird eine switch-Abfrage eingefügt, die, abhängig vom aktuellen Zustand der entsprechenden Klasseninstanz, bestimmte Anweisungen ausführt.

Es wird geprüft,

- ob überhaupt ein Aufruf dieser (Event-)Methode möglich ist (sonst Fehlermeldung)
- welcher Zustandsübergang ausgeführt werden muß.

Falls ein Zustandsübergang möglich ist, werden - falls vorhanden - nacheinander folgende Anweisungen ausgeführt:

1. Exit-Action des noch aktuellen States aufrufen
2. eigentliche Methode ausführen (liegt als Spezifikation beim entsprechenden Prozeß des funktionalen Modells vor)
3. Transitions-Aktion aufrufen
4. Output-Events aufrufen
5. Event, das an eine andere Klasse gesendet wird, aufrufen
6. Den Zustand auf den Nachfolgezustand setzen
7. Nachfolge-Zustand-Entry-Aktion aufrufen
8. Nachfolge-Zustand-Inside-Aktivität aufrufen

Um dieses so realisieren zu können, müssen folgende Voraussetzungen geschaffen werden:

1. Jeder Klasse müssen ihre möglichen Zustände als Aufzählungstyp zugeordnet werden. Dies geschieht in der privaten Variablen „state“.
2. Diese Variable muß für jede Klasseninstanz auf den aus dem Zustandsübergangsdiagramm ermittelbaren Startzustand gesetzt werden (geschieht nach der Instanziierung der Variablen im Konstruktor).

Eine (Event-)Methode könnte also folgendermaßen aussehen:

```
void m1();
{
        switch (state)
        {
                case (state1) :
                {
                        Exit-Aktion von state1 aufrufen ;
                        // eigentliche Funktion von m1 liegt hier als Spezifikation vor
                        // (Information aus dem funktionalen Modell)
                        Transitions-Aktion aufrufen;
                        Output-Event aufrufen ;
                        Event, das an eine andere Klasse gesendet wird, aufrufen;
                        state = Nachfolge-Zustand ;
                        Nachfolge-Zustands-Entry-Aktion aufrufen ;
                        Nachfolge-Zustands-Inside-Activity aufrufen ;
                }
                break ;
                case (state2) :
                {
                        ....
                }
                break ;
                ...
                default         :  error();
        }
}
```

Anmerkung: Die Fehlerbehandlung sollte durch eine globale Methode error() realisiert werden und eventuell den aktuellen Zustand und den Event-Namen übergeben bekommen.

Weiterhin gilt: Es muß eine 1:1-Zuordnung vorliegen zwischen

1. Events im dynamischen Modell und Operationen im Objekt-Modell (somit auch zu einer bestimmten Methode im bereits erzeugten Code)
2. Transitions-Aktionen im dynamischen Modell und Operationen im Objekt-Modell
3. Zustands-Entry-Aktionen im dynamischen Modell und Operationen im Objekt-Modell
4. Zustands-Inside-Activities im dynamischen Modell und Operationen im Objekt-Modell
5. Zustands-Exit-Aktionen im dynamischen Modell und Operationen im Objekt-Modell (siehe Kapitel 6.2 - Die Beziehungen zwischen den drei Modellen in der Gesamtsicht).

Auf folgende Problematik muß noch hingewiesen werden:

Events, die in einem Zustandsübergangsdiagramm, welches ja einer bestimmten Klasse zugeordnet ist, vorkommen, müssen nicht unbedingt zu dieser Klasse gehören.

In dieser Arbeit wird allerdings angenommen, daß Events, die Zustandsübergänge einer bestimmten Klasse auslösen, auch zu dieser Klasse gehören.

Hieraus ergibt sich nun folgende Vorgehensweise des Code-Erzeugungsalgorithmus bezüglich der Umsetzung des dynamischen Modells in C++ Code:

1. Es werden alle im dynamischen Modell vorkommenden Events durchlaufen.
2. Die zu dem Event gehörenden Transitionen werden nacheinander ermittelt.
3. Die zur jeweiligen Transition gehörende Klasse wird ermittelt
4. Die switch-Abfrage wird abhängig von den möglichen Zuständen der Klassen aufgebaut.

Im folgenden nun die Ideen für die Umsetzung der einzelnen Komponenten des dynamischen Modells:

State:

Zu jeder Klasse des Objekt-Modells wird das entsprechende Zustandsübergangsdiagramm ermittelt. Alle dort auftretenden States werden bei der Klassendeklaration in einem Aufzählungstyp in der Variablen „state“ abgelegt:

```
enum { A,B,C } state ;
```

Event / Transition / Action on a Transition:

Die oben beschriebene switch-Anweisung wird in der Methode, die dem Event entspricht, erzeugt (falls noch nicht vorhanden). Für jeden möglichen Zustandsübergang, der durch dieses Event ausgelöst werden kann, wird eine case-Abfrage eingefügt. In dieser werden dann, wie oben beschrieben, nacheinander alle beteiligten Methoden aufgerufen.

Initial State / Final State:

Bei Instanziierung der einzelnen Klasseninstanzen wird danach die Variable „state“ auf den aus dem dynamischen Modell ermittelbaren „Initial State“ gesetzt. Dies geschieht in der Konstruktor-Methode der entsprechenden Klasse.

Zusätzlich werden unterhalb der enum-Deklaration als Kommentare die Informationen über Start- und Endzustände abgelegt.

Event with Attributes:

Im entsprechenden case-Fall der (Event-)Methode wird der folgende Kommentar eingefügt:

```
// Dieses Ereignis könnte als Argumente
// a,b,c
// übergeben bekommen
```

Anmerkung: Jeder case-Fall entspricht einer möglichen Transition.

Guarded Transition:

Analog zu „Event with Attributes“ wird im entsprechenden case-Fall eine Kommentarzeile erzeugt:

```
// Bei Eintritt dieses Ereignisses tritt nur der Nachfolgezustand ein,
// falls folgendes Prädikat gilt:
// Prädikatsbeschreibung
```

Output Event / Sending an event to another Object:

Die entsprechende (Event-)Methode wird ermittelt und aufgerufen.

State Generalization:

Die Information über Unterzustände wird unterhalb der enum-Deklaration als Kommentar abgelegt.

Falls so modelliert würde, daß stets nur einer der Unterzustände Startzustand wäre, könnte der Nachfolgezustand immer auf diesen „Initial-Sub-State“ gesetzt werden.

Concurrent Subdiagrams / Splitting and synchronizing of Control:

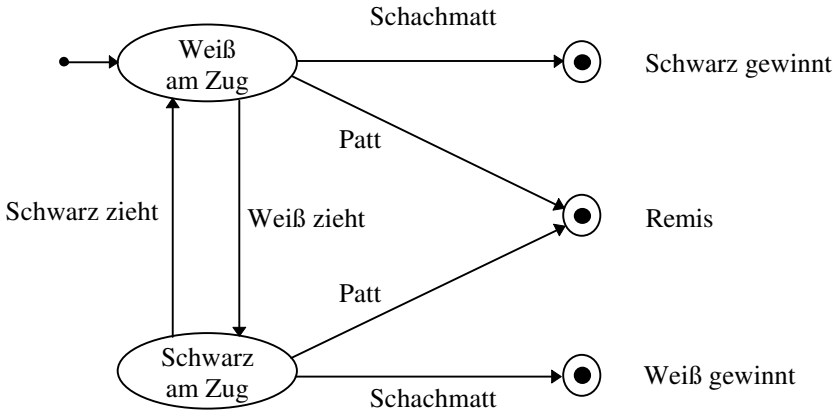
Diese Konstrukte werden im Rahmen dieser Arbeit bei der Umsetzung in Code nicht berücksichtigt, da ihre Semantik zu komplex ist.

10.2 Beispiele

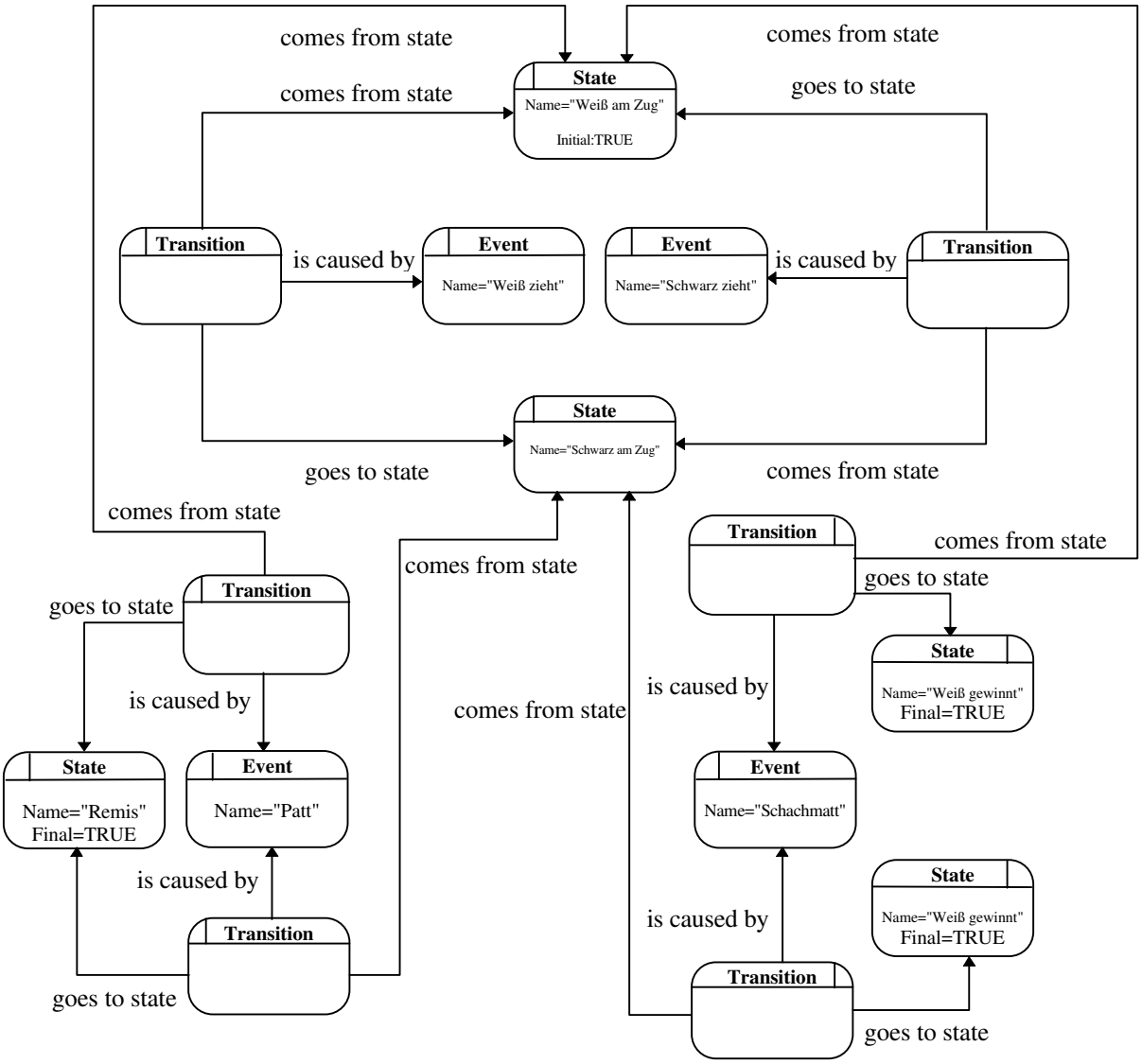
In diesem Unterkapitel wird auf Grund der Ideen des letzten Unterkapitels beispielhaft Code aus dem dynamischen Modell erzeugt. Dabei werden für die einzelnen Komponenten jeweils drei Entwicklungsstufen dargestellt:

1. Die Modellierung nach Rumbaugh
2. Der zugehörige TGraph
3. Der daraus erzeugte C++ Code

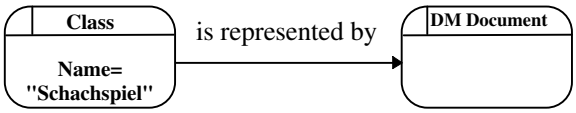
Zuerst ein Gesamtbeispiel: [Rumbaugh 92], Seite 111, Abbildung 5.6



Daraus resultierender TGraph:



Alle auf dieser Seite aufgetretenen Entities sind mit folgendem DM Document verbunden:



Bereits erzeugter Code:

```
class Schachspiel
{
        ...

void Schwarz_zieht() ;
{
        ...

void Weiss_zieht() ;
{
        ...

void Schachmatt() ;
{
        ...

void Patt() ;
{
        ...
```

wird jeweils (bei Abarbeitung der entsprechenden Operation des Objekt-Modells) erweitert zu:

```
class Schachspiel
{
        ...
        enum
        {Schwarz_am_Zug, Weiss_am_Zug,
        Schwarz_gewinnt, Weiss_gewinnt, Remis } state ;

void Schwarz_zieht() ;
{
        ...
        switch (state)
        {
        case    Schwarz_am_Zug :
                {
                        state = Weiss_am_Zug ;
                }
                break ;
                default : error() ;
        }
```

```
void Weiss_zieht() ;
{
        ...
        switch (state)
        {
        case    Weiss_am_Zug :
                {
                        state = Schwarz_am_Zug ;
                }
                break ;
                default : error() ;
        }
```

```
void Schachmatt() ;
{
        ...
        switch (state)
        {
        case    Weiss_am_Zug :
                {
                        state = Schwarz_gewinnt ;
                }
                break ;

        case    Schwarz_am_Zug :
                {
                        state = Weiss_gewinnt ;
                }
                break ;

                default : error() ;
        }
```

```
void Patt() ;
{
        ...
        switch (state)
        {
        case    Weiss_am_Zug :
                {
                        state = Remis ;
                }
                break ;

        case    Schwarz_am_Zug :
                {
                        state = Remis ;
                }
                break ;

                default : error() ;
        }
```

In der Konstruktor-Methode wird die Variable state auf den Startzustand gesetzt:

```
Schachspiel sp1 ;       // Instanziierung eines Objekts der Klasse Schachspiel
                        //–> ruft Konstruktor auf
```

```
void Schachspiel::Schachspiel();
{
        ...
        state = Weiss_am_Zug ;
```

Feststellung:

Zur Vereinfachung nehme ich hier an, daß die Event-Namen aus dem dynamischen Modell und die Operations-Namen aus dem Objekt-Modell (die ja auch den Namen der erzeugten Methode festlegen) gleich sind. Dies muß laut bisheriger Modellierung nicht unbedingt der Fall sein.

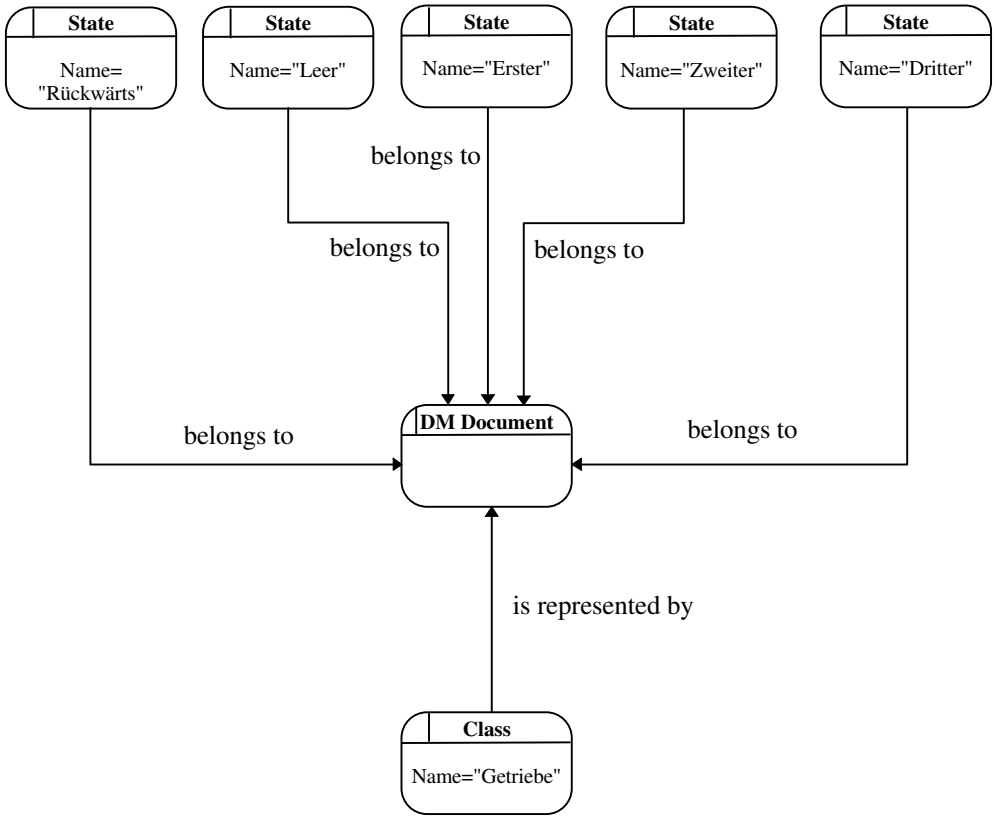
Es folgen Beispiele für die Umsetzung der einzelnen Komponenten des dynamischen Modells:

State:

[Rumbaugh 92], S.108, Text:

> *„so könnte sich das Getriebe eines Autos in den Zuständen Rückwärts, Leer, Erster, Zweiter, Dritter befinden“*

Hieraus resultiert folgender TGraph:



Bereits erzeugter Code:

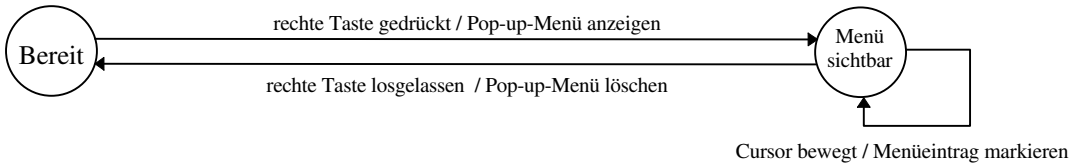
```
class Getriebe
{
        ...
```

wird erweitert zu:

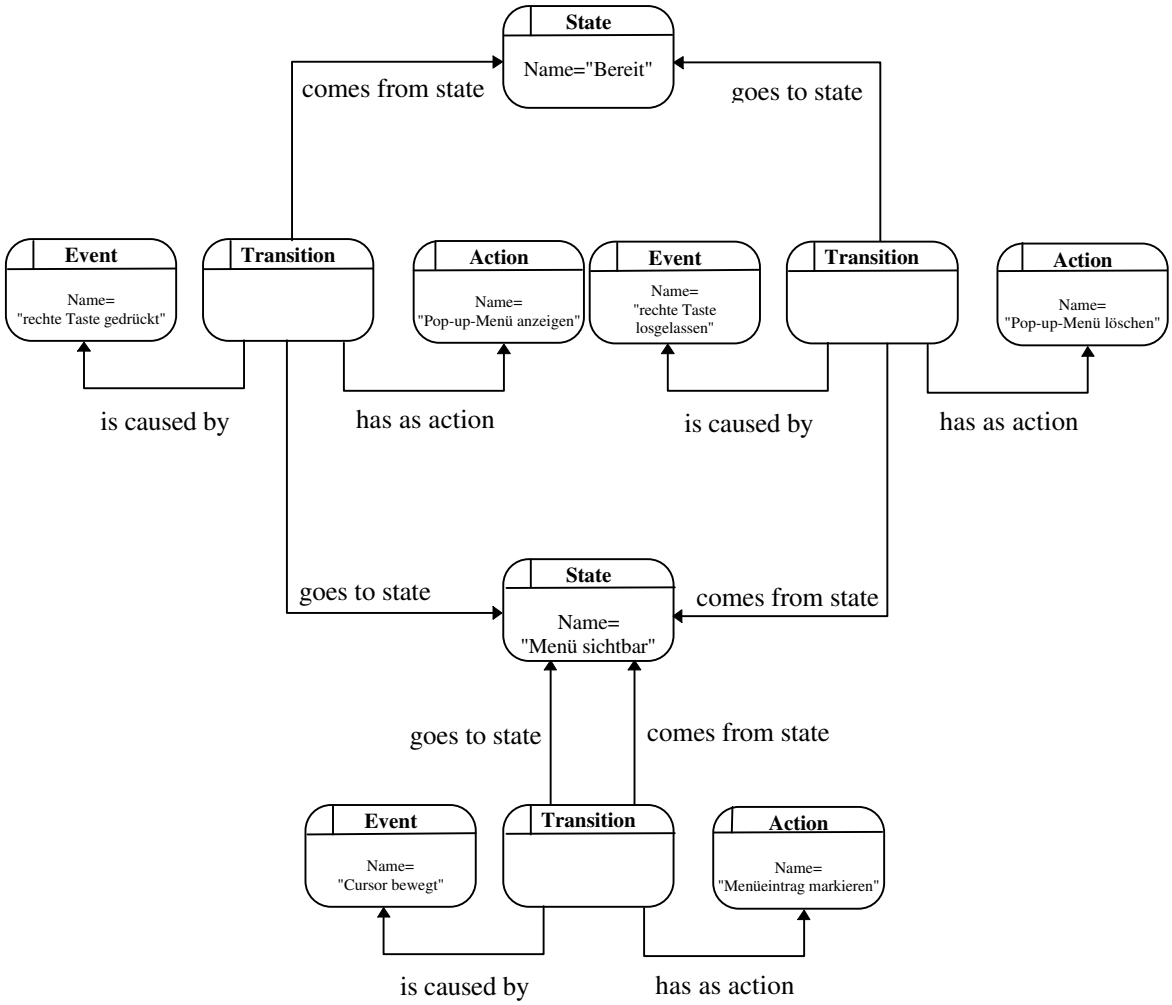
```
class Getriebe
{
        ...
        enum { Rueckwaerts, Leer, Erster, Zweiter, Dritter } state ;
```

Event causes Transition between states / Transition / Action on a Transition:

[Rumbaugh 92], S.113, Abb. 5.8



Daraus resultierender TGraph:



Bereits erzeugter Code:

```
class Pop_up_menue
{
        ...

void show_pop_up_menue() ;
{
        // zeigt Menü am Bildschirm an
        ...
}

void clear_pop_up_menue() ;
{
        // löscht Menü vom Bildschirm
        ...
}

void mark() ;
{
        // markiert Menüeintrag, auf dem sich Cursor befindet
        ...
}

void got_mouse_click() ;
{
        ...

void lost_mouse_click() ;
{
        ...

void moved_cursor() ;
{
        ...
```

Daraus erzeugter Code:

```
class Pop_up_menue
{
        ...
        enum { Bereit, Menue_sichtbar } state ;
```

```
void got_mouse_click() ;
{
        ...
        switch(state)
        {
                case Bereit :
                {
                        show_pop_up_menue() ;
                        state = Menue_sichtbar ;
                } ;
                break ;
                default  : error();
        }
```

```
void lost_mouse_click() ;
{
        ...
        switch(state)
        {
                case Menue_sichtbar :
                {
                        clear_pop_up_menue() ;
                        state = Bereit ;
                } ;
                break ;
                default : error();
        }
```

```
void moved_cursor() ;
{
        ...
        switch(state)
        {
                case Menue_sichtbar :
                {
                        mark() ;
                        state = Menue_sichtbar ;
                } ;
                break ;
                default : error();
        }
```

Initial State / Final State:

[Rumbaugh 92], S.111, Abb.5.6 (siehe oben)

In der Konstruktor-Methode wird folgende Zeile erzeugt:

```
state = Weiss_am_Zug ;
```

Unterhalb der enum-Deklaration

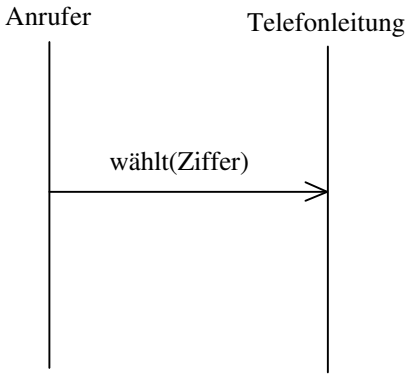
```
enum {Schwarz_am_Zug, Weiss_am_Zug, Schwarz_gewinnt, Weiss_gewinnt, Remis }
state;
```

werden folgende Kommentarzeilen eingefügt:

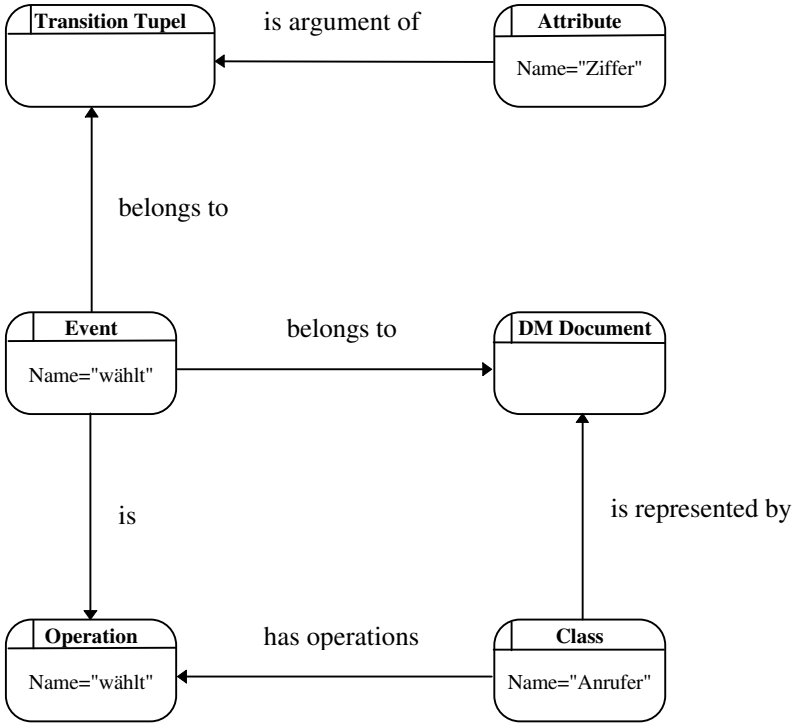
```
// Weiss_am_Zug ist Startzustand
// Schwarz_gewinnt, Weiss_gewinnt, Remis
// sind Endzustände
```

Event with Attributes:

[Rumbaugh 92], S.106, Abb. 5.3



TGraph:



Die Methode

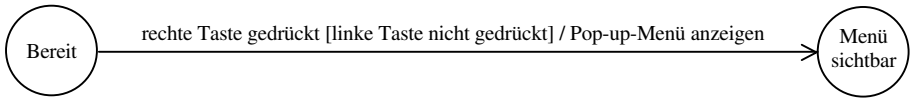
```
void waehlt() ;
{
        ...
```

bekommt als Kommentar hinzu:

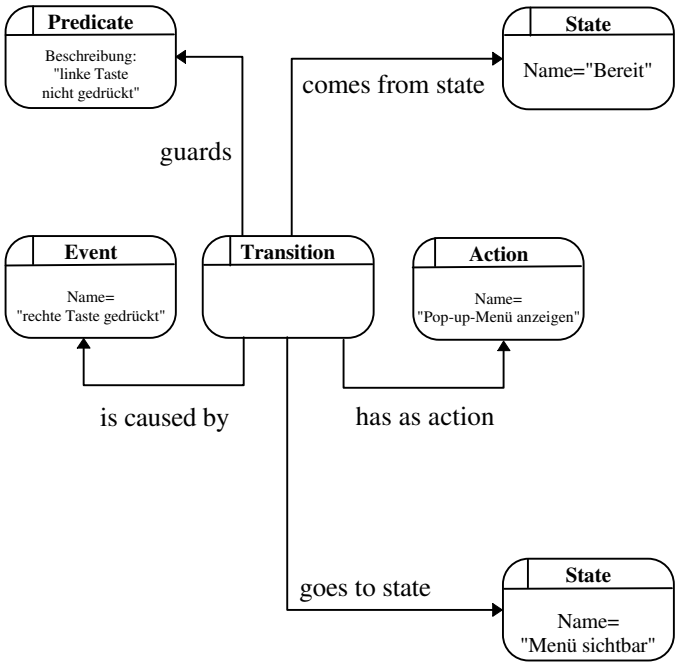
```
// Dieses Ereignis könnte als Argumente
// Ziffer
// übergeben bekommen
```

Guarded Transition:

[Rumbaugh 92], S.113, Abb. 5.8 erweitert durch den Guard „linke Taste nicht gedrückt“:



TGraph:



Die Methode

```
void got_mouse_click() ;
{
        switch(state)
        {
                case Bereit :
                {
```

wird endgültig erweitert zu:

```
void got_mouse_click() ;
{
        switch(state)
        {
                case Bereit :
                {
                        // Bei Eintritt dieses Ereignisses tritt der Nachfolgezustand
                        // nur dann ein,
                        // falls gilt:
                        // linke Taste nicht gedrückt
                        show_pop_up_menue();
                        state = Menue_sichtbar ;
                } ;
                break ;
                default : error();
        }
        ...
}
```

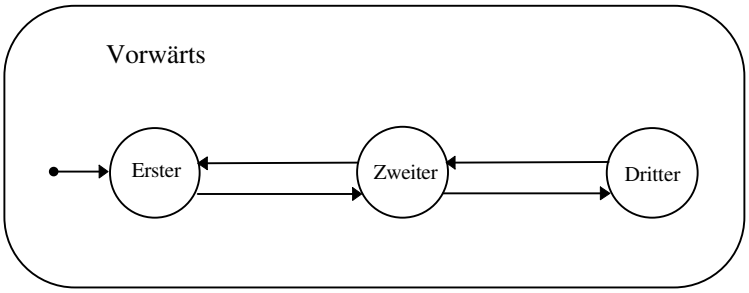
Output Event on a Transition / Sending an Event to another Object / Actions and Activity while in a state:

Diese Komponenten werden jeweils durch einen Aufruf innerhalb der entsprechenden (Event-)Methode, die den Zustandsübergang auslöst, realisiert.

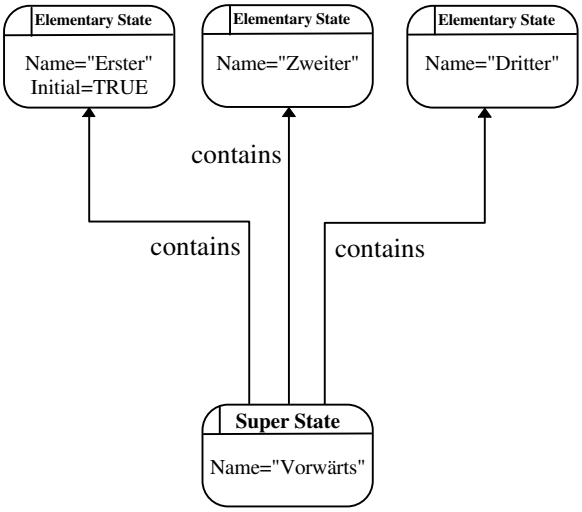
Die Reihenfolge wurde zu Beginn dieses Kapitels beschrieben (Beispielmethode m_1).

State Generalization:

[Rumbaugh 92], S.118, Abb.5.15 (relevanter Teilausschnitt)



Daraus resultierender TGraph:



Unterhalb der enum-Deklaration

```
enum {Leerlauf, Rueckwaerts, Vorwaerts, Erster, Zweiter, Dritter } state;
```

werden folgende Kommentarzeilen eingefügt:

```
// Der Zustand Vorwaerts hat als Unterzustände
// Erster, Zweiter, Dritter
```

11 Pseudocode-Algorithmen

In diesem Kapitel werden die Ideen zur Umsetzung in C++ Code aus den Kapiteln 8, 9 und 10 in konkreten Pseudocode umgesetzt. Die natürlichsprachlichen Ideen aus den oben genannten Kapiteln, die durch Beispiele erläutert wurden, werden nun in eine formale, implementationsnahe Form gebracht. Ich gehe folgendermaßen vor: Zuerst wird im Unterkapitel 11.1 der später verwendete Pseudocode beschrieben. Im Unterkapitel 11.2 ist dann der komplette Pseudocode-Algorithmus, der die Umsetzung aller Ideen in C++ Code zur Aufgabe hat, aufgeführt.

11.1 Allgemeines zum Pseudocode

In diesem Unterkapitel werden Grundlagen zum besseren Verständnis der Pseudocode-Algorithmen aufgeführt.

11.1.1 Benutzung des EMS-Graphenlabors

Voraussetzung für eine Codeerzeugung ist, daß alle 3 OMT-Modelle in Form eines TGraphen gemäß [Carstensen 94] vorliegen. Dabei ist das Typsystem und das Attributschema für jeden einzelnen Entity-Typ den in Teil II dieser Diplomarbeit beschriebenen Schema-Graphen zu entnehmen.

Die Informationen, die aus dem Graphen gewonnen werden sollen, bekommt man über die bereitgestellten Methoden und Kontrollstrukturen des Graphenlabors [Dahm 95].

Es gibt zwei Hauptdatentypen, auf denen diese arbeiten:

1. vertex, die Menge aller Knoten
2. edge, die Menge aller Kanten.

Die benutzten Methoden und Kontrollstrukturen, die im Pseudocode fett geschrieben sind, werden im folgenden kurz aufgeführt:

Anmerkung: Da nur ein gemeinsamer Graph verwendet wird, lasse ich die Identifizierung des Graphen (1. Parameter bei allen Methoden) weg. Gleichzeitig vernachlässige ich den Präfix „G_“ vor jeder Methode und jedem Datentyp, da er für meine Algorithmen irrelevant ist.

Beispiel:

Anstelle von

G_forAllOutEdges (G_Graph g, G_vertex v, G_edge e)

benutze ich

forAllOutEdges (vertex v, edge e).

11.1.2 Methoden- und Kontrollstrukturbeschreibungen

forAllVerticesWithType (const type t, vertex v) { StatementSequence }

Führt für jeden Knoten, der vom Typ type ist, nacheinander die StatementSequence aus.

forAllEdgesWithType (const type t, edge e) { StatementSequence }

Führt analog für jede Kante vom Typ type nacheinander die StatementSequence aus.

vertex **omega**(edge e)

Liefert den Endknoten der Kante e als Ergebnis.

vertex **alpha**(edge e)

Liefert den Startknoten der Kante e als Ergebnis.

edge **firstOut** (const type t, vertex v)

Liefert die erste Kante vom Typ type, die vom Knoten v ausgeht.

edge **firstIn** (const type t, vertex v)

Liefert die erste Kante vom Typ type, die in den Knoten v eingeht.

edge **nextOut** (const type t, edge e)

Liefert die auf e folgende Kante vom Typ type, die aus dem Knoten ausgeht, aus dem e ausgeht.

edge **nextIn** (const type t, edge e)

Liefert die auf e folgende Kante vom Typ type, die in den Knoten eingeht, in den e eingeht.

forAllOutEdgesWithType (const type t, vertex v, edge e) { StatementSequence }

Führt für jede ausgehende Kante (vom Typ type) des Knotens v nacheinander die StatementSequence aus.

forAllInEdgesWithType (const type t, vertex v, edge e) { StatementSequence }

Führt analog zu forAllOutEdges die StatementSequence für alle eingehenden Kanten aus.

type **getVType**(vertex v)

Liefert den Typ des Knotens v.

attribute **getVAttr** (vertex v)

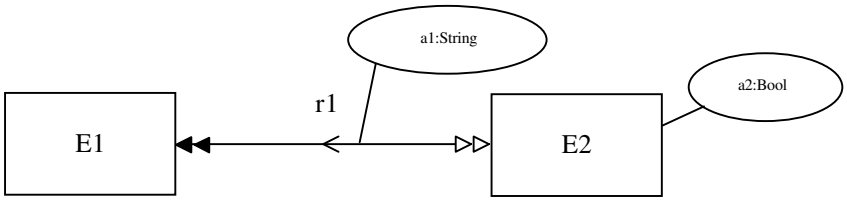
Liefert die Attribute des Knotens v.

attribute **getEAttr** (edge e)

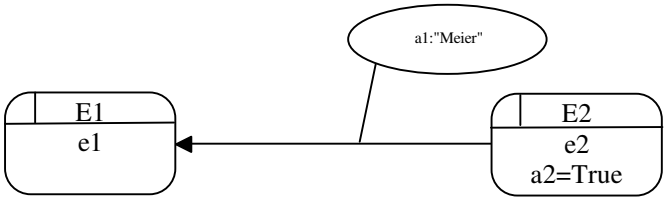
Liefert die Attribute der Kante e.

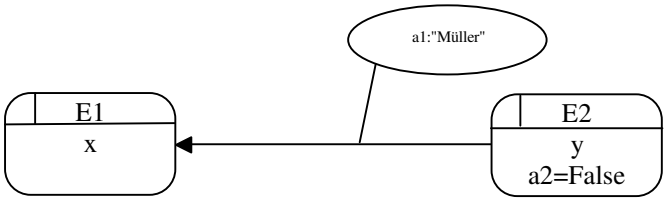
11.1.3 Beispiel

Folgendes Konzept-Schema sei gegeben:



Eine resultierende Instanz dieses Schemas, die wir mit den aufgeführten Methoden beispielhaft bearbeiten wollen:





forAllVerticesWithType (E1, v) { StatementSequence }

Führt für die beiden vorhandenen Knoten e1 und x vom Typ E1 die StatementSequence aus.

forAllEdgesWithType (r1, e) { StatementSequence }

Führt für die beiden Kante vom Typ r1 die StatementSequence aus.

Sei e die Kante der unteren Beziehung zwischen y und x:

v = **omega**(e) ;

Weist v den Knoten x vom Typ E1 als Endknoten der Kante e zu.

v = **alpha**(e) ; // e wie oben.

Weist v den Knoten y als Startknoten der Kante e zu.

e = **firstOut** (r1, e2) ;

Liefert die Kante mit Attribut a1="Meier" vom Typ r1, die vom Knoten e2 ausgeht.

e = **nextOut** (r1, e)

Liefert **EdgeNull**, da keine weitere Kante vom Typ r1 aus e2 ausgeht.

e = **firstIn** (r1, x)

Liefert die Kante mit Attribut a1="Müller" vom Typ r1, die in den Knoten x eingeht.

e = **nextIn** (r1, e)

Liefert **EdgeNull**, da keine weitere Kante vom Typ r1 in den Knoten x eingeht.

forAllOutEdgesWithType (r1, e2, e) { StatementSequence }

Führt für die ausgehende Kante (vom Typ r1) des Knotens e2 die StatementSequence aus.

forAllInEdgesWithType (r1, e1, e) { StatementSequence }

Führt die StatementSequence analog für die eingehende Kante aus.

getVType(y) ;

Liefert E2 als Typ des Knotens y.

```
attribute eatr ; // e wie oben die Kante zwischen y und x
eatr = getEAttr(e) ;
eattr.a1 = "Schmidt" ; // vorher Müller
```

Holt das Attribut der Kante e in die Variable eatr und manipuliert sie.

```
attribute classatr ;
vertex c ;      // mit Typsystem und Attributschema gemäß dem Entity-Typ Class
                // aus dem Meta-Modell
                // inklusive: Das Attribut Name vom Typ String.
classatr = getVAttr(c) ;
classattr.Name = "Schmidt" ;
```

Holt die Attribute in die Variable classatr und setzt den Namen auf "Schmidt".

11.1.4 Sonstiges

Teile des Pseudocodes werden nur natürlichsprachlich beschrieben. Diese Teile werden von spitzen Doppelklammern eingeschlosen.

<< natürlichsprachlicher Pseudocode >>

Der Code dieser natürlichsprachlichen Teile wird später näher beschrieben.

Beispiel:

```
forAllVerticesWithType(Class, c)
{
        << Gib den Namen von c aus >>
}
...
<< Gib den Namen von c aus >>
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class, c)
attribute cattr ;
cattr = getVAttr(c) ;
cout << cattr.Name << '\n' ;
```

Kommentare des Pseudocodes werden durch // eingeleitet.

Klassen-, Attribut- und Operationsnamen, die bei Rumbaugh aus mehreren Worten (getrennt durch Leerzeichen) bestehen, werden in C++ durch Unterstriche "_" getrennt, da Leerzeichen dort nicht erlaubt sind.

Wird ein Datentyp, der bei Rumbaugh groß geschrieben wird, in C++ umgesetzt, so wird er automatisch angepaßt („String“ wird zu „string“, „Integer“ zu „int“, „Bool“ zu „bool“ u.s.w.). Dies gilt auch für Typen, die keine Standard-Datentypen sind (Beruf wird zu beruf).

Die zwei letzten erwähnten Umsetzungen werden bei den einzelnen Pseudocode-Algorithmen nicht jedesmal explizit erwähnt, weil dieser Vorgang durch jeweilige Aufrufe einer Methode der Art „Wandle um()“ geschehen müßte.

Der zu erzeugende Code wird in Fettbuchstaben geschrieben. Gleichzeitig werden die Zeilen zu 10% grau schattiert dargestellt. Hierbei werden C++ Schlüsselwörter in Anführungszeichen dargestellt.

Temporäre Variablen, die Instanzen der Entity-Typen der Meta-Modelle sind, werden im Pseudocode unterstrichen dargestellt.

11.2 Algorithmen

In diesem Unterkapitel wird der komplette Algorithmus in der im letzten Unterkapitel beschriebenen Art in Pseudocode aufgeführt. Im Unterkapitel 11.2.1 befindet sich der grundlegende Algorithmus. Im Unterkapitel 11.2.2 sind die Verfeinerungen der natürlichsprachlichen Pseudocode-Teile aus Kapitel 11.2.1 zu finden.

11.2.1 Der grundlegende Algorithmus (für alle Klassen)

```
vertex c ;
edge e
attribute cattr ;
forAllVerticesWithType(Class,c)
{
        cattr = getVAttr(c) ;
        "class" cattr.Name
        << Generalization + Multiple Inheritance >>
        "{"
        << Constraints on Objects >>
        << Derived Class >>
        << Disjoint Subclasses >>
        vertex at, dt ;
        forAllEdgesWithType(has_attributes, c, e)
        {
                at = omega(e) ;
                dt = omega(firstOut(has_domain, at)) ;
                attribute dtattr, atattr ;
                dtattr = getVAttr(dt) ;
                atattr = getVAttr(at) ;
                << Class Attributes >>
                dtattr.Dtype atattr.Name
                if (atattr.Init_Value != "")
                {
                        "=" atattr.Init_Value
                }
                ";"
                << Derived Attributes >>
                << Discriminator >>
        }
        << Association Attributes >>
        << State >>
        << Initial State / Final State >>
        << State Generalization >>

        // Konstruktor und Destruktor-Definition
        cattr.Name "(" ")" ";"
        "~" cattr.Name "(" ")" ";"
```

```
// andere Operationen definieren
vertex op ;
forAllEdgesWithType(has_operations, c, e)
{
        op = omega(e) ;
        dt = omega(firstOut(has_returntype, op)) ;
        attribute opattr ;
        opattr = getVAttr(op) ;
        dtattr = getVAttr(dt) ;
        << Abstract Operations >>
        << Class Operations >>
        dtattr.Dtype opattr.Name "("
        vertex arg ;
        edge e1 ;
        forAllOutEdgesWithType(has_arguments, op, e1)
        {
                arg = omega(e1) ;
                dt = omega(firstOut(has_datatype, arg)) ;
                attribute argattr ;
                argattr = getVAttr(arg) ;
                dtattr = getVAttr(dt) ;
                dtattr.Dtype argattr.Name
                if (nextOut(has_arguments, e1) != EdgeNull)
                {
                        ","
                }
        }
"}"
}
```

```
        // Konstruktor und Destruktor-implementieren
        "void" cattr.Name "(" ")" "{"
        << Set Initial State >>
        "}" ";"

        "void" "~" cattr.Name "(" ")" "{"   "}" ";"

        // andere Operationen implementieren
        forAllEdgesWithType(has_operations, c, e)
        {
                op = omega(e) ;
                dt = omega(firstOut(has_returntype, op)) ;
                attribute opattr ;
                opattr = getVAttr(op) ;
                dtattr = getVAttr(dt) ;
                << Abstract Operations >>
                << Class Operations >>
                dtattr.Dtype cattr.Name "::" opattr.Name "("
                << Propagation of Operations >>
                e1 = firstOut(has_arguments, op) ;
                while (e1 != EdgeNull)
                {
                        arg = omega(e1) ;
                        dt = omega(firstOut(has_datatype, arg)) ;
                        attribute argattr ;
                        argattr = getVAttr(arg) ;
                        dtattr = getVAttr(dt) ;
                        dtattr.Dtype argattr.Name
                        if (nextOut(has_arguments, e1) != EdgeNull)
                        {
                                ","
                        }
                        e1 = nextOut((has_arguments, e1)) ;
                }
                "{"
                << Event / Transition >>
                "}"
        }
<< Object Instance + Instantiation Relationship >>
}
<< Association >>
```

11.2.2 Verfeinerung der abstrakten Pseudocode-Anweisungen

In diesem Unterkapitel werden die abstrakten Pseudocodeanweisungen verfeinert.

Zur Orientierung ist hier noch einmal die bei der Codeerzeugung eingehaltene Reihenfolge der zu verfeinernden Pseudocodeanweisungen aufgeführt. Sie entspricht gleichzeitig der Reihenfolge in diesem Kapitel:

```
<< Class >> // grundlegender Algorithmus aus dem letzten Kapitel
        //
        // Kopfzeile
        << Generalization + Multiple Inheritance >>
        << Constraints on Objects >>
        << Derived Class >>
        << Disjoint Subclasses >>
        //
        // Attribut-Deklarationen
        << Class Attributes >>
        << Derived Attributes >>
        << Discriminator >>
        << Association Attributes >>
        << State >>
        << Initial / Final State >>
        << State Generalization >>
        //
        // Operations-Definitionen
        << Abstract Operations >>
        << Class Operations >>
        //
        // Operations-Implementationen
        << Set Initial State >>
        << Abstract Operations >>
        << Class Operations >>
        << Propagation of Operation >>
        << Event / Transition >>
                << Event with Attributes >>
                << Guarded Transition >>
                << Exit Action >>
                << Process + Process Refinement >>
                << Action on a Transition >>
                << Output Event >>
                << Sending an Event to another Object >>
                << Entry Action >>
                << Inside Activity >>
```

//
// Objekt-Instanziierung
<< Object Instance / Instantiation Relationship >>
<< Association >>
<< Constraints between Associations >>

Im folgenden werden also die Verfeinerungen der einzelnen Komponenten beschrieben:

<< Generalization + Multiple Inheritance >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)

// Der erzeugte Code befindet sich am Ende der bereits erzeugten Zeile class c

```
vertex super, g ;       // g ist der Generalisationsknoten, super die Laufvariable für die
                        // Superklassen
edge e, e1 ;
attribute superattr ;
int flag = 0 ;

e = firstIn(has_subclasses, c) ;
while (e != EdgeNull)
{
        g = alpha(e) ;
        e1 = firstIn(is_superclass_in, g) ;
        super = alpha(e1) ;
        superattr = getVAttr(super) ;
        if (flag == 0) // erste Superklasse
        {
                ":"
                flag = 1 ;
        }
        "public" superattr.Name
        if (nextIn(has_subclasses, e) != EdgeNull)
        {
                ","
        }
        e = nextIn(has_subclasses, e) ;


}
```

<< Constraints on Objects >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)

// Der erzeugte Code befindet sich hinter der Deklarationskopfzeile der Klasse c

```
edge constr, descr ;
vertex con, pred ;       // con ist vom Typ „Constraint“, pred vom Typ „Predicate“

constr = firstOut(has_constraints, c) ;
if (constr != EdgeNull)
{
        "//" Instanzen dieser Klasse unterliegen folgenden Constraints :
}
while (constr != EdgeNull)
{
        con = omega(constr) ;
        pred = omega(firstOut(is_described_as, con) ;
        "//" pred.Predicate_Description
        constr = nextOut(has_constraints, constr) ;


}
```

<< Derived Class >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)

// Der erzeugte Code befindet sich hinter der Deklarationskopfzeile der Klasse c

```
if (cattr.Derived == TRUE)
{
        "//" Diese Klasse ist eine abgeleitete Klasse


}
```

<< Disjoint Subclasses >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)

// Der erzeugte Code befindet sich vor den Attributdeklarationen der Klasse c

```
vertex g, sub ; // g ist vom Typ „Generalization“, sub ist vom Typ „Class“
edge e ; e1 ;
attribute gattr , subattr;

e = firstOut(is_superclass_in, c) ;
if (e != EdgeNull) // c hat Unterklassen
{
        while (e != EdgeNull)
        {
                g = omega(e) ;
                gattr = getVAttr(g) ;
                if (gattr.Disjoint == TRUE // c hat disjunkte Unterklassen, die nun aufgelistet
                werden
                {
                        // Diese Klasse hat als disjunkte Unterklassen :
                        e1 = firstOut(has_subclasses, g) ;
                        while (e1 != EdgeNull) // auf jeden Fall ein Durchlauf
                        {
                                sub = omega(e1) ;
                                subattr = getVAttr(sub) ;
                                "//" subattr.Name
                                e1 = nextOut(has_subclasses, e1) ;
                        }
                e = nextOut(is_superclass_in, e) ;
        }
}
```

<< Class Attributes >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)
// und innerhalb der Schleife **forAllOutEdgesWithType**(has_attributes, c, e)

// Der erzeugte Code befindet sich vor der Deklaration des Attributes at

```
if (atattr.Is_Class_Attribute == TRUE)
{
        "static"
}
```

<< Derived Attributes >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_attributes, c, e)

// Der erzeugte Code befindet sich hinter der Deklaration des Attributes at

if (atattr.derived == TRUE)
{
        "//" atattr.Name ist ein abgeleitetes Attribut
}
```

<< Discriminator >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_attributes, c, e)

// Der erzeugte Code befindet sich hinter der Deklaration des Attributes at

vertex g, sub ; // g ist vom Typ „Generalization“, sub ist vom Typ „Class“
edge discr, e1 ;
attribute gattr, subattr ;

discr = firstOut(discriminates, at) ;
if (discr != EdgeNull)
{
        g = omega(firstOut(discr) ;
        "//" Das Attribut atattr.Name ist Diskriminator für die Unterklassen
        forAllOutEdgesWithType(has_subclasses, g, e1)
        {
                sub = omega(e1) ;
                subattr = getVAttr(sub) ;
                "//" subattr.Name
        }
}
```

<< Association Attributes >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)

// Der erzeugte Code befindet sich unterhalb der Attributdeklarationen der Klasse c .
vertex as, c2 ; // as entspricht dem Assoziationsknoten, c und c2 sind die
// beteiligten Klassen
edge e1 ;
attribute c2attr, asattr ;

forAllOutEdgesWithType(associates_with, c, e1)
{
as = **omega**(e1) ;
c2 = **alpha**(**firstIn**(associates_with, as)) ;
if (c2 == c) // dieselbe Klasse als erstes gefunden
{
c2 = **alpha**(**nextIn**(associates_with, e1)) ; // jetzt muß c2 die beteiligte Klasse sein
}
cattr = **getVAttr**(c) ;
c2attr = **getVAttr**(c2) ;
asattr = **getVAttr**(as) ;
if << c2 ist markiert >>
{
as.Name _ c2.Name _ c.Name "*" first _ c2.Name ";"
}
else
{
as.Name _c.Name _ c2.Name "*" first _ c2.Name ";"
<< Markiere c >>
}
}

<< State >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// Der erzeugte Code befindet sich unterhalb der Attributdeklarationen der Klasse c .

vertex dmd, elemstate, ;        // dmd ist vom Typ „DM Document“, elemstate vom Typ „State“
edge repr, belongs ;
attribute elemstateattr, ;
int flag = 0 ;
repr = firstOut(is_represented_by, c) ;
if (repr != EdgeNull)
{
        "enum" "{"
        dmd = omega(repr) ;
        forAllInEdgesWithType(belongs_to, dmd, belongs)
        {
                elemstate = alpha(belongs) ;
                if (getVType(elemstate) == Elementary State)
                {
                        if (flag == 1) // nur beim ersten Durchlauf nicht
                        {
                                ","
                        flag = 1 ;
                        }
                        elemstateattr = getVAttr(elemstate) ;
                        elemstateattr.Name
                }
        }
        "}" state ";"


}
```

<< Initial State / Final State >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// dmd ist das zu c gehörende „DM Document“

// Der erzeugte Code befindet sich unterhalb der enum-Deklaration der Klasse c

vertex istate ;  // istate ist vom Typ „State“
attribute istateattr ;

forAllInEdgesWithType(belongs_to, dmd, belongs)
{
        istate = alpha(belongs) ;
        istateattr = getVAttr(istate) ;
        if (getVType(istate) == State)
        {
                if (istateattr.Initial == TRUE)
                {
                        // istateattr.Name ist Startzustand
                }
                if (istateattr.Final == TRUE)
                {
                        // istateattr.Name ist Endzustand
                }
        }


}
```

<< State Generalization >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// dmd ist das zu c gehörende „DM Document“

// Der erzeugte Code befindet sich unterhalb der enum-Deklaration

vertex superstate, substate ;   // Knoten selbsterklärend
attribute superstateattr, substateattr ;
int flag ;

forAllInEdgesWithType(belongs_to, dmd, belongs)
{
        superstate = alpha(belongs) ;
        if (getVType(superstate) == Superstate))
        {
                superstateattr = getVAttr(superstate) ;
                // Der Zustand superstateattr.Name hat als Unterzustände:
                //
                substate = omega(firstOut(contains, superstate)) ;
                flag = 0 ;
                while (substate != VertexNull)
                {
                        substateattr = getVAttr(substate) ;
                        substateattr.Name
                        if ((flag == 1) &&
                        (omega(nextOut(contains, belongs)) != VertexNull ))
                        {
                                ","
                        }
                        flag = 1 ;
                        substate = omega(nextOut(contains, belongs)) ;
                }
        }
}
```

<< Abstract Operations >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)

// und innerhalb der Schleife **forAllOutEdgesWithType**(has_operations, c, e)

// Der erzeugte Code befindet sich vor der Definition und vor der Implementation der
// Operation op

```
if (opattr.Abstract == TRUE)
{
        "virtual"
}
```

<< Class Operations >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)
// und innerhalb der Schleife **forAllOutEdgesWithType**(has_operations, c, e)

// Der erzeugte Code befindet sich vor der Deklaration und vor der Implementation der
// Operation op

```
if (opattr.Is_Class_Operation == TRUE)
{
        "static"
}
```

<< Set Initial State >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// Der erzeugte Code befindet sich bei der Implementierung des Konstruktors von c

vertex state, dmd ; edge dmdh ;        // dmd ist vom Typ „DM Document“, state vom Typ
„State“
attribute stateattr ;

dmd = omega(firstOut(is_represented_by,c)) ;
dmdh = firstIn(belongs_to, dmd) ;
state = alpha(dmdh) ;
if (getVType(state) == State)
{
        while (stateattr.Initial != TRUE);
        {
                state = alpha(nextIn(belongs_to, dmdh)) ;
                stateattr = getVType(state) ;
        }
}
state = stateattr.Name ";"
```

<< Abstract Operations >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)

// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich vor der Definition und vor der Implementation der
// Operation op

if (opattr.Abstract == TRUE)
{
        "virtual"
}
```

<< Class Operations >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich vor der Deklaration und vor der Implementation der
// Operation op

if (opattr.Is_Class_Operation == TRUE)
{
        "static"
}
```

<< Propagation of Operation >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich hinter der Kopfzeile der Implementation der Operation op

edge prop ;
vertex op1 ;    // op1 ist die „Operation“, zu der sich op fortpflanzt
attribute op1attr ;

prop = firstOut(propagates_to, op) ;
if (prop != EdgeNull)
{
        op1 = omega(prop) ;
        op1attr = getVAttr(op1) ;
        "//" Diese Operation pflanzt sich zu op1.Name fort
}
```

<< Event / Transition >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich im Body der Operation op

edge is eventh ;
vertex event, transtup, trans, fromstate, tostate ;      // Knotenbezeichner selbsterklärend
attribute fromstateattr, tostateattr ;

eventh = firstIn(is, op) ;
event = alpha(eventh) ;
while (getVType(event) != Event)
{
        event = alpha(nextIn(is, eventh)) ;
}
if (getVType(event) == Event) // sonst ist diese Operation kein Event
{
        "switch" "(" state ")"
        "{"
        transtup = omega(firstOut(belongs_to, event)) ;
        while (transtup != VertexNull)
        {
                trans = alpha(firstIn(is_caused_by, transtup)) ;
                fromstate = omega(firstOut(comes_from_state, trans) ;
                fromstateattr = getVAttr(fromstate) ;
                "case" "(" fromstateattr.Name ")" ":" "{"
                << Event with Attributes >>
                << Guarded Transition >>
                << Exit Action >>
                << Process + Process Refinement >>
                << Action on a Transition >>
                << Output Event >>
                << Sending an Event to another Object >>
                tostate = omega(firstOut(goes_to_state, trans)) ;
                tostateattr = getVAttr(tostate) ;
                state "=" tostateattr.Name ";"
                << Entry Action >>
                << Inside Activity >>
                "}"
                "break" ";"
                transtup = omega(nextOut(belongs_to, transtup)) ;
        }
        "default" ":" error "(" ")" ";"
}
```

<< Event with Attributes >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt
// und dort zu Beginn eines case-Falls

vertex atr ;      // atr ist vom Typ „Attribute“
edge transtuph ;
attribute atrattr ;

transtuph = firstIn(is_argument_of, transtup) ;
atr = alpha(transtuph) ;
if (atr != VertexNull) // es können also Argumente übergeben werden
{
        // Dieses Ereignis könnte als Argumente
        while (atr != VertexNull)
        {
                atrattr = getVAttr(atr) ;
                // atrattr.Name
                atr = alpha(nextIn(is_argument_of, transtuph)) ;
        }
// übergeben bekommen
}
```

<< Guarded Transition >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt
// und dort zu Beginn eines case-Falls

vertex pred ;    // pred ist vom Typ „Predicate“
attribute predattr ;

pred = alpha(firstIn(guards, trans)) ;
if (pred != VertexNull) // es gibt also einen Guard
{
        // Bei Eintritt dieses Ereignisses tritt der Nachfolgezustand nur dann ein, falls
        // gilt:
        predattr = getVAttr(pred) ;
        // predattr.Prädikatsbeschreibung
        // Prädikatsbeschreibung noch unspezifiziert
}
```

<< Exit Action >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)
// sie bekommt fromstate übergeben

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt


vertex act, op1 ;        // act ist vom Typ „Action“, op1 ist die „Operation“, die act entspricht
attribute op1attr ;

act = omega(firstOut(has_as_exit_action, fromstate)) ;
if (act != VertexNull)
{
        op1 = omega(firstOut(is, act));
        op1attr = getVAttr(op1) ;
        op1attr.Name
}
```

<< Process + Process Refinement >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich im Body der Operation op in einer Switch-Abfrage in einem
// case-Fall

edge e, e1 oph ;
vertex spec, p ;         // p ist vom Typ „Process“, spec ist die „Specification“, die p spezifiziert
attribute specattr ;

oph = firstIn(is, op) ;
p = alpha(oph) ;
while (getVType(p) != Process)
{
        p = alpha(nextIn(is, oph)) ;
}
spec = omega(firstOut(has_specification, p) ;
if (spec != VertexNull) // Blattprozeß
{
        specattr = getVAttr(spec) ;
        "//" specattr.Statement_Sequence
}
```

```
else // untergeordnete Prozesse von p suchen
{
	edge e2, e3, e4 ; // Hilfskanten
	vertex fmd, up, uop, dt1, arg1 ;
	// fmd ist vom Typ „FM Document“
	// up entspricht dem untergeordneten „Process“ von p, uop entspricht der „Operation“,
	// die up darstellt
	// dt1 entspricht dem „Datatype“ von uop, arg1 ist die Laufvariable für die Argumente
	// von uop
	attribute upattr, dt1attr, arg1attr, uopattr ;

	fmd = alpha(firstIn(refines, p) ;
	forAllInEdgesWithType(belongs_to, fmd, e2)
	{
		if (getVType(alpha(e2)) == Process)
		{
			up = alpha(e2) ;
			uop = omega(firstOut(is, up) ;
			dt1 = omega(firstOut(has_returntype, uop) ;
			dt1attr = getVAttr(dt1) ;
			uopattr = getVAttr(uop) ;
			dt1attr.Dtype uopattr.Name
			"("
			e3 = firstOut(has_arguments, uop) ;
			while (e3 != EdgeNull)
			{
				arg1 = omega(e3) ;
				arg1attr = getVAttr(arg1) ;
				arg1attr.Name
				if (nextOut(has_arguments, e3) != EdgeNull)
				{
					","
				}
				e3 = (nextOut(has_arguments, e3)) ;
			}
			")"
		}
	}
}
```

<< Action on a Transition >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)
// sie bekommt als aktuell zu bearbeitende Transition trans übergeben

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt, in einem
// case-Fall

vertex act, op1 ;        // act ist vom Typ „Action“, op1 ist die „Operation“, die act entspricht
attribute op1attr ;

act = omega(firstOut(has_as_action, trans)) ;
if (act != VertexNull)
{
        op1 = omega(firstOut(is, act)) ;
        op1attr = getVAttr(op1) ;
        op1attr.Name
}
```

<< Output Event >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt, in einem
// case-Fall

vertex outev, ev1, op1 ;        // outev ist vom Typ „Output Event“, ev1 das dazugehörige
„Event“
                        // und op1 die „Operation“, die ev1 entspricht
attribute op1attr ;

outev = omega(firstOut(causes, trans)) ;
if ( outev != VertexNull) // Output Event existiert
{
        ev1 = alpha(firstIn(is_sent_by, outev)) ;
        op1 = omega(firstOut(is, ev1)) ;
        op1attr = getVAttr(op1) ;
        op1attr.Name
}
```

<< Sending an Event to another Object >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt
// in einem case-Fall

vertex outev, ev1, op1, c1 ;   // outev ist vom Typ „Output Event“,
                               // ev1 das dazugehörige „Event“
                               // op1 die „Operation“, die ev1 entspricht und c1 die „Class“, zu
                               // der op1 gehört (an die gesendet werden muß)
attribute op1attr, c1attr ;

outev = omega(firstOut(causes, trans)) ;

if (outev != VertexNull)) // Output Event existiert
{
        ev1 = alpha(firstIn(is_sent_by, outev)) ;
        c1 = alpha(firstIn(is_adressee_of, outev)) ;
        op1 = omega(firstOut(is, ev1)) ;
        op1attr = getVAttr(op1) ;
        c1attr = getVAttr(c1) ;
        c1attr.Name "::" op1attr.Name
}
```

<< Entry Action >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)
// sie bekommt tostate übergeben

//Der erzeugte Code befindet sich  im Body der Operation op, die ein Event darstellt


vertex act, op1 ;        // act ist vom Typ „Action“, op1 ist die „Operation“, die act entspricht
attribute op1attr ;

act = omega(firstOut(has_as_entry_action, tostate)) ;
if (act != VertexNull)
{
        op1 = omega(firstOut(is, act)) ;
        op1attr = getVAttr(op1) ;
        op1attr.Name
}
```

<< Inside Activity >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)
// und innerhalb der Schleife forAllOutEdgesWithType(has_operations, c, e)
// sie bekommt tostate übergeben

// Der erzeugte Code befindet sich im Body der Operation op, die ein Event darstellt


vertex activ, op1 ;      // activ ist vom Typ „Activity“, op1 die „Operation“, die activ
repräsentiert
attribute op1attr ;

activ = omega(firstOut(has_as_activity, tostate)) ;
if (activ != VertexNull)
{
        op1 = omega(firstOut(is, activ)) ;
        op1attr = getVAttr(op1) ;
        op1attr.Name
}
```

<< Object Instance + Instantiation Relationship >>

```
// Diese Code-Sequenz befindet sich innerhalb der Schleife forAllVerticesWithType(Class,c)

// Der erzeugte Code befindet sich unterhalb der kompletten Klassendeklaration der Klasse c

vertex oi, tup, val, at  // oi ist vom Typ „Object Instance“, tup vom Typ „Tupel“,
                         // val vom Typ „Value“ und at vom Typ „Attribute“
edge e, e1 ;
attribute oiattr, valattr, atattr ;
string oiname ;

forAllInEdgesWithType(instantiates, c, e)
{
        oi = alpha(e) ;
        << oiname bekommt einen String des Namensgenerators zugeordnet >>
        cattr.Name oiname ";"
        forAllOutEdgesWithType(consists_of, oi, e)
        {
                tup = omega(e) ;
                val = alpha(firstIn(gives_value_to, tup)) ;
                at = alpha(firstIn(represents, tup)) ;
                attribute atattr, valattr ;
                oiattr.Name "." atattr.Name "=" valattr.Active_Value ";"
        }
}
```

<< Association >>

```
vertex as c2 ;  // as entspricht dem Assoziationsknoten, c und c2 sind die beteiligten
                // Klassen
edge e1 eh ;
attribute c2attr, asattr ;
forAllVerticesWithType(Association, as)
{
        eh = firstIn(associates_with, as))
        c = alpha(eh) ;
        c2 = alpha(nextIn(associates_with, eh)) ;
        cattr = getVAttr(c) ;
        c2attr = getVAttr(c2) ;
        asattr = getVAttr(as) ;

        "class" as.Name _ c.Name _ c2.Name
        << Markiere c >>
        {
        << Constraints between Associations >>
        "bool" derived ";"
        "class" c.Name"*" c.Name _ as.Name _ A ";"
        "class" c2.Name"*" c2.Name _ as.Name _ A ";"

        as.Name _ c.Name _ c2.Name "*" next _ c.Name ";"
        as.Name _ c.Name _ c2.Name "*" next _ c2.Name ";"
        as.Name _ c.Name _ c2.Name "*" prev _ c.Name ";"
        as.Name _ c.Name _ c2.Name "*" prev _ c2.Name ";"

        as.Name _ c.Name _ c2.Name "*" next ";"
        as.Name _ c.Name _ c2.Name "*" prev ";"

        "static" as.Name _ c.Name _ c2.Name "*" first ";"
        "static" as.Name _ c.Name _ c2.Name "*" last ";"

        // Konstruktor- und Destruktor-Definitionen
        as.Name _ c.Name _ c2.Name "(" ")" ";"
        ~ as.Name _ c.Name _ c2.Name "(" ")" ";"

        "}"

        // Implementierung des Konstruktors
        as.Name _ c.Name _ c2.Name "::" as.Name _ c.Name _ c2.Name
        "(" c.Name"*" oid _ c.Name "," c2.Name"*" oid _ c2.Name ")"
        "{"
                derived "=" FALSE ";"
```

```
        "//" Element in die allgemeine Liste ans Ende einfügen
        "if" "("last"=="NULL")"
        "{"
                first "=" "new" as.Name _ c.Name _ c2.Name ";"
                last "=" first ";"
        "}"
        "else"
        "{"
                last"–>"next "=" "new" as.Name _ c.Name _ c2.Name ";"
                last"–>"next"–>"prev "=" last ";"
                last "=" last"–>"next ";"
        "}"

        "//" Zeiger auf übergebene Objekte setzen
        last"–>"c.Name _ A "=" oid _ c.Name ";"
        last"–>"c2.Name _ A "=" oid _ c2.Name ";"

        "//" c.Name in Liste des c2.Name an 1. Stelle einfügen
        last"–>"next _ c.Name "=" oid _ c2.Name"–>"first _ c.Name ";"
        last"–>"next_ c.Name"–>"prev_ c.Name "=" last ";"
        last"–>"prev_ c.Name "=" "NULL"
        oid _ c2.Name"–>"first _ c.Name "=" last ";"

        "//" c2.Name in Liste des c.Name an 1.Stelle einfügen
        last"–>"next _ c2.Name "=" oid _ c.Name"–>"first _ c2.Name ";"
        last"–>"next_ c2.Name "–>"prev_ c2.Name  "=" last ";"
        last"–>"prev_ c2.Name "=" "NULL"
        oid _ c.Name"–>"first _ c2.Name "=" last ";"

"}"

// Implementierung des Destruktors
as.Name _ c.Name _ c2.Name "::" "~" as.Name _ c.Name _ c2.Name "(" ")"
"{"
        "//" das zu löschende Element ist über den this-Zeiger zu erreichen


        "//" first-Zeiger sichern, falls this das jeweils erste Element ist
        "if" "this"–>c2.Name_A–>first_ c.Name "==" "this"
        "{"
                "this" "–>" c2.Name_A"–>"first_ c.Name "="
                "this" "–>"next_ c.Name ";"
        "}"

        "if" "this" "–>"c.Name_A"–>"first_ c2.Name "==" "this"
        "{"
                "this" "–>"c.Name_A"–>"first_ c2.Name "="
                "this" "–>"next_ c2.Name ";"
        "}"
```

```
                "//" Aus der c.Name-Liste c2.Name streichen
                "if" "this" "–>"next_ c2.Name "!=" "NULL"
                "{"
                        "this" "–>"next_ c2.Name "="
                        "this" "–>"next_ c2.Name"–>"next_ c2.Name ";"
                "}"
                "if" "this" "–>"prev_ c2.Name "!=" "NULL"
                "{"
                        "this" "–>"prev_ c2.Name "="
                        "this" "–>"prev_ c2.Name"–>"prev_ c2.Name ";"
                "}"

                "//" Aus der c2.Name-Liste c.Name streichen
                "if" "this" "–>"next_ c.Name "!=" "NULL"
                "{"
                        "this" "–>"next_ c.Name "="
                        "this"–>"next_ c.Name"–>"next_ c.Name ";"
                "}"
                "if" "this" "–>"prev_ c.Name "!=" "NULL"
                "{"
                        "this" "–>"prev_ c.Name "="
                        "this" "–>"prev_ c.Name"–>"prev_ c.Name ";"
                "}"

                "//" Element aus der allgemeinen Liste löschen
                "if" "this" "–>"next "!=" "NULL"
                "{"
                        "this" "–>"next"–>"prev "=" "this" "–>"prev ";"
                "}"
                "if" "this" "–>"prev "!=" "NULL"
                "{"
                        "this" "–>"prev"–>"next "=" this"–>"next ";"
                "}"
        "}"
}
```

<< Constraints between Associations >>

// Diese Code-Sequenz befindet sich innerhalb der Schleife **forAllVerticesWithType**(Class,c)
// und innerhalb der Schleife **forAllOutEdgesWithType**(associates_with, c, e)
// und bekommt as übergeben

// Der erzeugte Code befindet sich hinter dem Klassennamen der Assoziation as

```
edge subset ;
vertex as1 ;     // as1 entspricht der Assoziation, von der as eine Teilmenge ist
attribute as1attr ;

subset = firstOut(is_subset_of, as) ;
if (subset != EdgeNull)
{
        as1 =  omega(subset) ;
        as1attr = getVAttr(as1) ;
        "//" Diese Assoziation ist eine Teilmenge von as1.Name
}
```

12 Ausblick

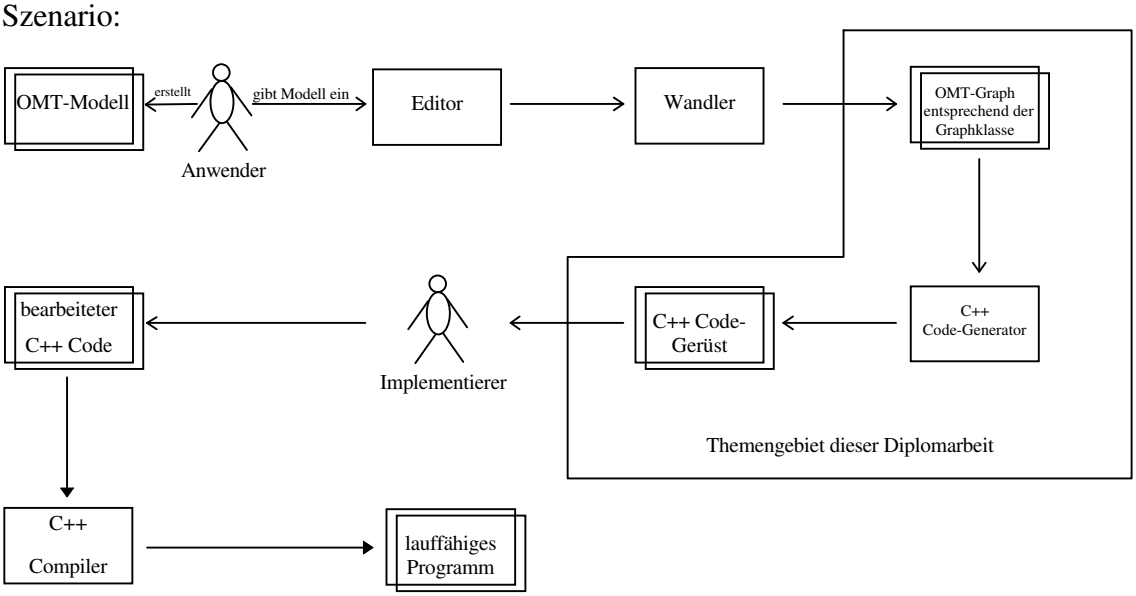
Mit dieser Arbeit wird der Pseudocode zur Implementierung eines C++ Codegenerators aus OMT-Modellen bereitgestellt. Aus Zeitgründen konnte dieser Algorithmus nicht mehr implementiert werden. Die bereitgestellte Graphklasse für OMT-Modelle ist bei weiteren Arbeiten am Institut jedoch ohne Einschränkungen zu verwenden.

In der Zukunft könnte der Ansatz, den diese Diplomarbeit geschaffen hat, erweitert werden.

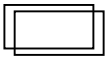
Wünschenswert wäre folgendes:

1. Der Anwender erstellt ein OMT-Modell.
2. Der Anwender gibt sein Modell in einen OMT-Editor ein. Der Editor enthält in seiner Grafikbibliothek alle Komponenten, die in OMT vorkommen können. Es liegt also kein Informationsverlust vor.
3. Ein Wandler überführt die Modelle aus dem Editor in einen Graphen, der der OMT-Graphklasse, die in dieser Arbeit beschrieben ist, entspricht.
4. Dieser Graph dient als Eingabe des C++ Code-Generators.
5. Der Code-Generator generiert auf in dieser Arbeit beschriebene Weise den C++ Code.
6. Der Implementierer bearbeitet den erzeugten Code dahingehend, daß die Semantik der Modelle korrekt im Code wiederzufinden ist.
7. Der bearbeitete C++ Code dient als Eingabe für einen C++ Compiler.
8. Endgültige Ausgabe ist das compilierte, lauffähige Programm.

Szenario:



Legende:



- entspricht einer Komponente, die eine Ein- oder Ausgabe darstellt.

- entspricht einer Komponente, die eine bestimmte Aufgabe ausführt.

Zusätzlich könnte man folgendes ins Auge fassen:

Der Editor wird erweitert. Er stellt dann für verschiedene objektorientierte Modellierungsarten Grafikbibliotheken zur Verfügung. So wird es dem Anwender ermöglicht, sich in verschiedenen Modellierungswelten zu bewegen, aber trotzdem nur einen Editor benutzen zu müssen.

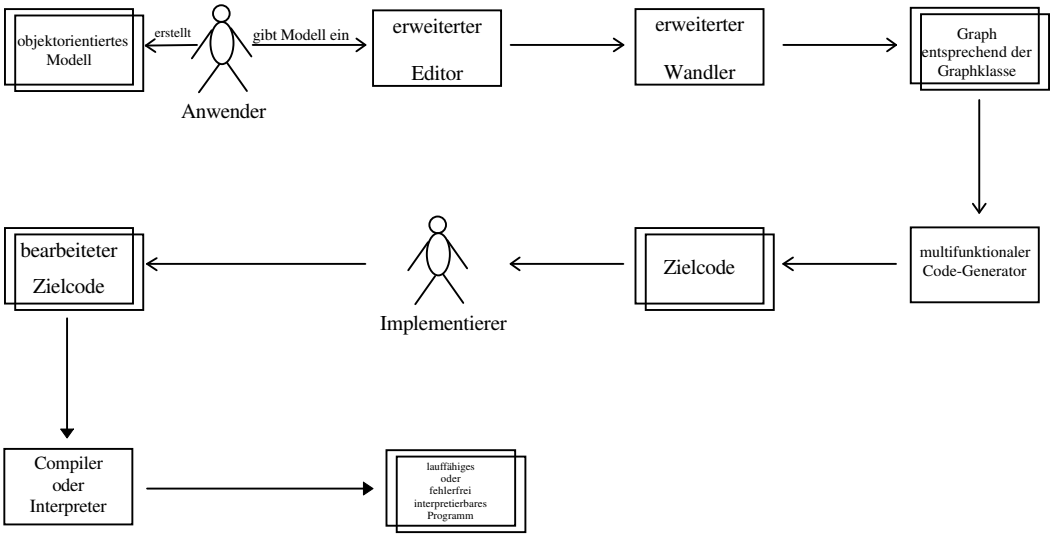
Der Wandler bekommt die Graphklassen der anderen Modellierungsarten zur Verfügung gestellt und ist so in der Lage, auch diese in den entsprechenden TGraphen umzuwandeln.

Der in dieser Arbeit beschriebene Code-Generator wird dahingehend erweitert, daß er auch Code anderer objektorientierter Sprachen erzeugen kann.

Der eben beschriebene Vorgang könnte dann so aussehen:

1. Der Anwender erstellt irgendein objektorientiertes Modell.
2. Der Anwender gibt sein Modell in den erweiterten Editor ein.
3. Der Wandler überführt die Modelle aus dem Editor in einen TGraphen, der der entsprechenden Graphklasse entspricht.
4. Dieser TGraph dient als Eingabe des multifunktionalen Code-Generators.
5. Der Code-Generator generiert den in einem Dialog mit dem Anwender gewählten Zielcode.
6. Der erzeugte und vom Implementierer bearbeitete Code dient - je nach gewählter Zielsprache - als Eingabe für einen Compiler oder einen Interpreter.
7. Ausgabe ist das compilierte, lauffähige oder das fehlerfrei interpretierbare Programm.

Szenario:



Legende:

wie oben.

Dieses Konzept könnte man dann immer an die neuesten Entwicklungen anpassen:

1. Es ist eine neue Modellierungsart entwickelt worden:
   - Man erstellt eine entsprechende Graphklasse für diese neue Modellierungsart.
   - Man erweitert den Editor um eine komplette Bibliothek für die neue Modellierungsart.
   - Man erweitert den Wandler, so daß er die neuen Modelle aus dem Editor in einen TGraphen entsprechend der neuen Graphklasse überführen kann.
2. Es ist eine neue objektorientierte Programmiersprache entwickelt worden:
   - Man erweitert den Codegenerator, so daß er Code in der neuen Programmiersprache erzeugen kann.

Die Zukunft wird zeigen, inwieweit die Gedanken dieser Arbeit erweitert werden.

Literaturverzeichnis:

[Carstensen 94] Carstensen, Martin; Ebert, Jürgen; Winter, Andreas.
Entity-Relationship-Diagramme und Graphklassen
Koblenz: Institut für Softwaretechnik, 1994

[Dahm 95] Dahm, Peter; Ebert, Jürgen; Litauer, Christoph.
Benutzerhandbuch EMS-Graphenlabor V3.0
Koblenz: Institut für Softwaretechnik 1995

[Ebert 94] Ebert, Jürgen; Franzke, Angelika.
A Declarative Approach to Graph Based Modeling
In: Mayr, E.; Schmidt, G.; Tinhofer, G.(Eds.).
Graph Theoretic Concepts in Computer Science
Berlin: Springer, LNCS 1994

[Rumbaugh 91] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William.
Object-Oriented Modeling and Design
Engelwood Cliffs: Prentice Hall 1991

[Rumbaugh 92] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William.
Objektorientiertes Modellieren und Entwerfen
München, Wien: Hanser Verlag 1992

[Stroustrup 92] Stroustrup, Bjarne.
Die C++–Programmiersprache
Bonn, München, Paris: Addison-Wesley 1992

Erklärung:

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Koblenz, April 1996

(Frank Bohlmann)