

# **Semantisches Zoomen in Diagrammeditoren am Beispiel von UML**

Diplomarbeit im Fach Informatik

vorgelegt von

**Oliver Köth**

geboren am 28.02.1975 in Schweinfurt

angefertigt am

Institut für Informatik,  
Lehrstuhl für Programmiersprachen und Programmiermethodik  
Friedrich-Alexander-Universität Erlangen-Nürnberg  
(Prof. Dr. H. J. Schneider)

Betreuer: Mark Minas

Beginn der Arbeit: 2.10.2000

Abgabe der Arbeit: 22.2.2001



Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Programmiersprachen, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an in den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 22.2.2001



## Kurzfassung

Die Arbeit befasst sich mit der Darstellung und Bearbeitung großer Diagramme in grafischen Editoren, die mit Hilfe des DiaGen-Systems erzeugt und speziell auf eine bestimmte Diagrammsprache zugeschnitten sind. Wie gezeigt wird, erfordert das Arbeiten mit großen grafischen Darstellungen besondere Unterstützung durch den Editor, um Übersichtlichkeit und gute Handhabbarkeit zu garantieren.

Aus einer Betrachtung verschiedener Techniken, die dies erreichen sollen, wird ein Ansatz entwickelt, wie sich in DiaGen eine "semantische Fokus-und-Kontext-Darstellung" realisieren lässt. Diese ermöglicht unter guter Ausnutzung der vorhandenen Bildschirmfläche eine detaillierte Darstellung der gerade relevanten Diagrammbereiche, ohne dass dabei der Bezug zu ihrer Umgebung verloren geht. Das vorgeschlagene Konzept lässt sich mit den in DiaGen vorhandenen Mitteln gut realisieren und an die Gegebenheiten der jeweiligen Diagrammsprache anpassen; im Gegensatz zu klassischen "Fisheye"-Techniken wirkt die Darstellung des Diagramms natürlicher und verständlicher.

Anhand zweier beispielhafter Anwendungen – einem Editor für Baumdiagramme und einem Editor für UML-Klassendiagramme – wird die Anwendung der vorgeschlagenen Technik konkretisiert, und es werden die notwendigen Erweiterungen des DiaGen-Systems dargestellt. Diese betreffen vor allem die Graphtransformations-Fähigkeiten des Systems und die automatische Korrektur des Diagrammlayouts. Schließlich wird eine Benutzungsschnittstellen-Komponente vorgestellt, die eine einfache Anwendung der neuen Möglichkeiten erlaubt.

Die vorgeschlagene Technik zur Bearbeitung großer Diagramme konnte in den Beipielanwendungen erfolgreich eingesetzt werden und sollte sich einfach auf andere Diagrammsprachen übertragen lassen. Die Implementierung eines UML-Editors in DiaGen belegt, dass sich das System auch für umfangreiche praxisrelevante Diagrammsprachen einsetzen lässt. Für die Verbindung von Diagrammbearbeitung und Graphtransformation bieten sich damit vielfältige interessante Anwendungsmöglichkeiten.

Anmerkung zum Sprachgebrauch:

Die konsequente Verwendung geschlechtsneutraler Begriffe (im Stile von "der Nutzer/die Nutzerin") wirkt sich im Deutschen leider ausgesprochen negativ auf die Lesbarkeit von Texten aus. Wir haben uns daher entschieden in der vorliegenden Arbeit durchgängig die weiblichen Formen zu verwenden; Formulierungen wie "Programmiererin" werden daher geschlechtsneutral benutzt und schließen selbstverständlich auch männliche Programmierer mit ein.

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>i</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Das DiaGen-System . . . . .	1
1.2 Bearbeiten von Diagrammen in DiaGen-Editoren . . . . .	2
1.3 Ziel der Arbeit . . . . .	3
1.4 Aufbau und Überblick . . . . .	4
<b>2 Semantische Fokus-und-Kontext-Darstellung</b>	<b>7</b>
2.1 Probleme beim Arbeiten mit großen Diagrammen . . . . .	7
2.2 Techniken zur Darstellung großer Diagramme . . . . .	8
2.3 Weitere Darstellungsvarianten . . . . .	9
2.4 Semantisches Zoomen . . . . .	10
2.5 Probleme beim semantischen Zoomen . . . . .	12
2.6 Fokus und Kontext . . . . .	13
2.7 Optische Fisheye-Transformation . . . . .	15
2.8 Probleme der Fisheye-Darstellung . . . . .	16
2.9 Semantische Fokus-und-Kontext-Darstellung in DiaGen . . . . .	18
2.10 Vor- und Nachteile des realisierten Konzepts . . . . .	21
<b>3 Beispielhafte Diagrammsprachen</b>	<b>25</b>
3.1 Baumdiagramme und ihre interne Repräsentation . . . . .	26
3.2 Syntaktische Analyse . . . . .	26
3.3 Abstraktion in Baumdiagrammen . . . . .	29
3.4 UML-Diagramme . . . . .	31
3.5 Elemente von UML-Klassendiagrammen . . . . .	32
3.6 Abstraktion in UML-Klassendiagrammen . . . . .	34

<b>4</b>	<b>Analyse von UML-Diagrammen</b>	<b>37</b>
4.1	Die Metamodell-Hierarchie der UML . . . . .	37
4.2	Objektmodelle und Hypergraphen . . . . .	39
4.3	Generierung von abstrakten Syntaxgraphen . . . . .	41
4.4	Probleme mit abstrakten Syntaxgraphen . . . . .	43
<b>5</b>	<b>Graphtransformation</b>	<b>47</b>
5.1	Beschreibung von Diagrammtransformationen in DiaGen . . . . .	47
5.2	Aus- und Einblenden von Diagrammteilen . . . . .	49
5.3	Auswahl von Abstraktionsbereichen . . . . .	51
5.4	Austauschen von Diagrammkomponenten . . . . .	55
5.5	Generische Transformationsregeln . . . . .	57
5.6	Diagrammanalyse bei mehrstufigen Einbettungen . . . . .	59
5.7	Effizientes Parsen von Mengen . . . . .	60
5.8	Berücksichtigung der Bearbeitungsabfolge . . . . .	61
5.9	Zusätzliche wünschenswerte Erweiterungen . . . . .	63
<b>6</b>	<b>Diagrammlayout</b>	<b>67</b>
6.1	Techniken zur Layoutspezifikation . . . . .	68
6.2	Layout in Diagrammeditoren . . . . .	68
6.3	Layout für Baumdiagramme . . . . .	70
6.4	Layout für UML-Klassendiagramme . . . . .	71
6.5	Inkrementelles und vollständiges Layout . . . . .	72
6.6	Layout durch Constraint-Propagation . . . . .	74
6.7	Erzeugung der Constraints . . . . .	76
6.8	Layout durch Kräftesimulation . . . . .	77
6.9	Ein Kraftsystem zu Abstandskorrektur von Knoten . . . . .	79
6.10	Ein Kraftsystem zur Zoom-Korrektur . . . . .	81
<b>7</b>	<b>Vereinfachte Benutzung von Zoom-Transformationen</b>	<b>83</b>
7.1	Strukturbäume . . . . .	83
7.2	Operationen auf Baumknoten . . . . .	85
7.3	Kombination von Zoom-Transformationen . . . . .	87

Inhaltsverzeichnis	iii
<b>8 Schlussbemerkungen</b>	<b>91</b>
8.1 Ergebnisse . . . . .	91
8.2 Ansatzpunkte für Verbesserungen . . . . .	92
8.3 Ausblick . . . . .	94
<b>Literaturverzeichnis</b>	<b>97</b>
<b>Index</b>	<b>102</b>



# Kapitel 1

## Einführung

Wenn eine schlagwortartige Charakterisierung der Gegenwart gefragt ist, wird häufig der Begriff des "Informationszeitalters" herangezogen. Fast jeder sieht sich heute mit zunehmend komplexeren, digital erzeugten und verfügbaren Informationsumgebungen konfrontiert. Das Internet, besonders das World-Wide-Web, bietet ein Paradebeispiel für diese Entwicklung.

Die "Navigation" in großen Informationsmengen, d.h. das Erkennen und Verfolgen relevanter Informationen, wird dabei zunehmend zu einem Problem. Das Phänomen des "Lost in Hyperspace", bei dem die Anwenderin die digitale Orientierung verloren hat und nicht weiß, wie ihre aktuelle Sicht der virtuellen Welt in einen Gesamtkontext eingeordnet werden kann, wurde schon oft beschrieben. Nicht zuletzt aufgrund dieser Entwicklungen hat sich in den letzten 10–15 Jahren das Gebiet der Visualisierung großer Informationsmengen zu einem bedeutenden eigenständigen Zweig der Informatik entwickelt, auf dem die vorliegende Arbeit aufbaut.

### 1.1 Das DiaGen-System

Ähnliche Probleme treten nämlich, wie wir zeigen werden, im Kleinen auch beim Arbeiten mit grafischen Diagrammdarstellungen auf, mit denen wir uns im Rahmen des DiaGen-Projektes beschäftigt haben. Ziel dieses Projektes ist die Entwicklung eines Programmgenerators, der die einfache Erstellung von benutzungsfreundlichen Editoren für zweidimensionale grafische Diagramme eines bestimmten Typs – wie z. B. Flussdiagramme oder Petri-Netze – ermöglicht. Die Verwendung eines Programmgenerators erlaubt es dabei, die meisten Aspekte einer solchen "Diagrammsprache" in einer eigens dafür definierten kompakten Spezifikationssprache auszudrücken und trägt so wesentlich dazu bei, die Entwicklung von Diagrammeditoren zu vereinfachen.

Abbildung 1.1 zeigt, wie ein Diagrammeditor mit Hilfe des DiaGen-Systems erstellt wird: Ein Spezifikationsdokument legt die Struktur der Diagrammsprache

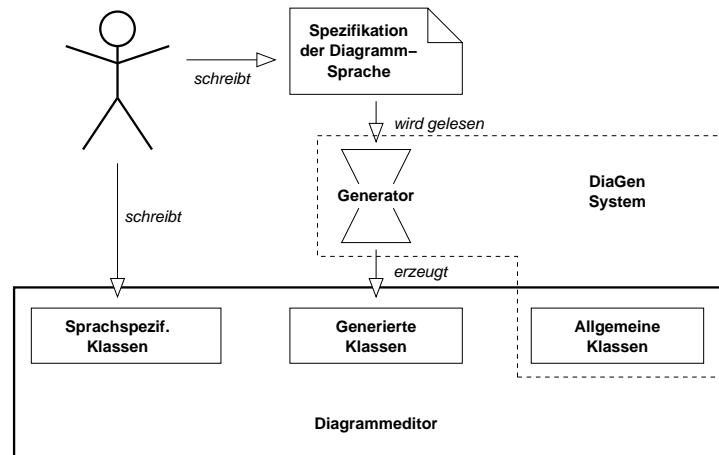


Abbildung 1.1: Erzeugung eines Diagrammeditors mit DiaGen

fest, d.h. es definiert die möglichen Bestandteile eines Diagramms (Diagrammkomponenten) und beschreibt, wie diese kombiniert werden können, um korrekte Diagramme zu erzeugen.

Aus dieser Spezifikation erzeugt das Generator-Modul von DiaGen dann den Quellcode einer Reihe von Klassen in der objektorientierten Programmiersprache Java. Einige Aspekte des zu beschreibenden Diagrammeditors eignen sich besser dazu, direkt durch Java-Code formuliert zu werden, anstatt sie in die Spezifikationssprache aufzunehmen,<sup>1</sup> deshalb müssen die generierten Klassen zum Teil nachbearbeitet oder durch weitere direkt programmierte Klassen ergänzt werden. Außerdem können sprachspezifische Klassen zusätzliche Funktionalität zur Verfügung stellen, die durch das allgemeine DiaGen-System nicht abgedeckt wird. Zusammen mit allgemeinen Klassen aus der DiaGen-Klassenbibliothek bilden generierte und handgeschriebene Klassen dann den fertigen Diagrammeditor. Die Klassenbibliothek liefert dabei den Teil der Editor-Funktionalität, der für alle Diagrammtypen gleich bleibt. Auf die interne Arbeitsweise des Systems wird im Verlauf des Textes noch genauer eingegangen, soweit dies erforderlich ist.

Sowohl das DiaGen-System selbst wie auch die generierten Diagrammeditoren sind vollständig in der Programmiersprache Java geschrieben und dadurch auf verschiedenen Hardware-Plattformen einsetzbar. Die erzeugten Editoren können auch als Module (Java-Beans) in größere Programmsysteme integriert werden.

## 1.2 Bearbeiten von Diagrammen in DiaGen-Editoren

Da die Spezifikation eines neuen Diagrammeditors möglichst wenig Aufwand erfordern soll (z. B. für Prototypen und experimentelle Diagrammsprachen), verfolgt DiaGen bei der Bearbeitung von Diagrammen einen grundsätzlich anderen

<sup>1</sup>Dabei handelt es sich vor allem um die grafische Darstellung und Manipulation der Diagrammkomponenten.

Ansatz als konventionelle Diagrammeditoren [29]: Diagramme werden vorzugsweise durch "direkte Manipulation" [22] modifiziert, d.h. die Benutzerin verändert die visuelle Darstellung des Diagramms durch das Bewegen von Griffen ("Handles") oder "Cut & Paste". Die Editoren werden also ähnlich bedient wie übliche Zeichenprogramme; im Unterschied zu diesen erkennen sie aber auch die abstrakte Struktur (bzw. "Bedeutung") der Diagramme, die durch ein Analyse-Modul aus der visuellen Darstellung abgeleitet wird. Deshalb ist es zwar möglich auch (vorübergehend) inkorrekte Diagramme zu erzeugen, bei denen die Anordnung der Diagrammkomponenten nicht den Regeln der Diagrammsprache entspricht; die Editoren können aber die Ergebnisse der Diagrammanalyse nutzen, um das Erstellen korrekter Diagramme zu erleichtern.

Im Gegensatz dazu bevorzugen übliche Diagrammeditoren strukturorientierte Modifikationsoperationen, welche speziell auf die jeweilige Diagrammsprache zugeschnitten sind und ihre abstrakte Struktur berücksichtigen. DiaGen erlaubt es auch, derartige Operationen zu benutzen [30], aber ihre Spezifikation und Verwendung ist optional, so dass Diagrammeditoren auch ohne die aufwendige Definition eines "vollständigen" Satzes von Modifikationsoperationen benutzbar sind; so können auch Manipulationen durchgeführt werden, die die Programmiererin beim Spezifizieren des Editors nicht vorhergesehen hat. Außerdem erweist sich das Bearbeiten durch direkte Manipulation oft als flexibler, da naheliegende Modifikationen eines Diagramms oft inkorrekte Zwischenzustände erfordern, welche sonst nicht erlaubt wären. Wir glauben, dass durch die Kombination beider Bearbeitungstechniken die Bedienbarkeit der Diagrammeditoren entscheidend verbessert wird (vgl. [30]).

### 1.3 Ziel der Arbeit

Zu Beginn der vorliegenden Arbeit war das DiaGen-System bereits in der skizzierten Form implementiert und benutzbar. Wir hatten daher die Möglichkeit, uns auf spezielle Anwendungen und Probleme zu konzentrieren. Ein Aspekt beim Bearbeiten von Diagrammen, der besondere Beachtung verdient, ist die Behandlung großer Diagramme.

Komplizierte Sachverhalte und Beziehungen können mit grafischen Notationen oft verständlicher dargestellt werden als durch sequentiellen Text; dies zeigt sich beispielsweise darin, dass unabhängig voneinander eine ganze Reihe von ähnlichen grafischen Notationen für Strukturen der objektorientierten Programmierung entwickelt wurden und wird auch durch die große Popularität der daraus hervorgegangenen UML-Diagramme belegt. Leider tendieren solche Darstellungen in der praktischen Anwendung aber dazu, sehr groß zu werden, und sie lassen sich durch die vielen Querverbindungen meist sehr viel schlechter in handhabbare Einzelstücke zerlegen als textuelle Beschreibungen. Schon in der "klassischen" Entity-Relationship-Modellierung von Unternehmensdaten kamen leicht Diagramme zustande, die ganze Wände bedecken.

Beim computergestützten Bearbeiten von Diagrammen mit Diagrammeditoren steht eine solche Bildschirmfläche natürlich nicht zur Verfügung. Aus diesem Grund erfordert der Umgang mit großen Diagrammen die Verwendung von speziell darauf zugeschnittenen Darstellungs- und Bearbeitungstechniken. In der vorliegenden Arbeit haben wir nun untersucht, wie sich derartige Konzepte in das DiaGen-System integrieren lassen.

## 1.4 Aufbau und Überblick

Die Arbeit ist folgendermaßen aufgebaut: Zunächst führt Kapitel 2 in die grundlegenden Probleme ein, welche beim Darstellen und Bearbeiten großer Diagramme auftreten und stellt verschiedene etablierte Techniken vor, um diesen Problemen zu begegnen. Daraus wird ein Ansatz entwickelt, wie sich derartige Abstraktionstechniken in Form einer "semantischen Fokus-und-Kontext"-Darstellung in die DiaGen-Architektur integrieren lassen und mit Hilfe von "Zoom-Transformationen" realisiert werden können. Die vorgeschlagene Lösung wird im Hinblick auf ihre Konsequenzen und ihre Unterschiede zu existierenden Ansätzen untersucht.

Kapitel 3 stellt zwei Diagrammsprachen vor, für die Beispieleditoren mit semantischer Fokus-und-Kontext-Darstellung implementiert wurden und welche in den folgenden Kapiteln als Anschauungsbeispiele verwendet werden. Als einfacher Prototyp dienen Baumdiagramme, während als Beispiel für eine komplexe und praxisnahe Anwendung ein Editor für UML-Klassendiagramme implementiert wurde. An diesen Beispielen wird erläutert, wie sich gewünschten Darstellungsmöglichkeiten in beiden Editoren auswirken. Um die zugrundeliegenden Techniken und Programmbestandteile später näher betrachten zu können, ist es notwendig, auch auf die Analyse der Diagramme und die damit erzeugten Semantikbeschreibungen einzugehen.

Kapitel 4 geht genauer auf die Analyse von UML-Diagrammen mit DiaGen ein: Es wird ausgeführt, wie sich die standard-konforme Beschreibung von UML-Diagrammen durch Objektmodelle auf "abstrakte Syntaxgraphen" abbilden lässt, die mit Hilfe des DiaGen-Systems erzeugt werden. Außerdem zeigen wir, welche Vorteile und zukünftige Entwicklungsmöglichkeiten damit verbunden sind.

Kapitel 5 befaßt sich mit der ersten wichtigen Voraussetzung für die semantische Fokus-und-Kontext-Darstellung: der Formulierung geeigneter Zoom-Transformationen. Zunächst betrachten wir diejenigen Erweiterungen der Diagrammtransformations-Komponente von DiaGen, die für die Beschreibung von Zoom-Transformationen essentiell notwendig sind. Anschließend werden zusätzliche Ergänzungen diskutiert, welche allgemein für die Beschreibung komplexer Diagrammtransformationen von Nutzen sind. Im Zusammenhang mit der Implementierung des UML-Editors haben sich daneben auch neue Konstrukte zur Diagrammanalyse als notwendig erwiesen, welche hier ebenfalls kurz vorgestellt

werden. Schließlich werfen wir einen kurzen Blick auf weitere Ideen zur internen Verarbeitung der Diagramme, die im Laufe der vorliegenden Arbeit entstanden sind, aber noch nicht realisiert wurden.

Das zweite der beiden Hauptprobleme bei der Implementierung des vorgestellten Ansatzes zur Diagrammabstraktion liegt in der Realisierung eines geeigneten automatischen Diagrammlayouts. Kapitel 6 diskutiert den grundlegenden Konflikt zwischen Allgemeinheit und Effizienz von Layout-Algorithmen unter besonderer Berücksichtigung der Anforderungen im Rahmen des DiaGen-Systems. Anschließend wird vorgestellt, wie sich basierend auf einem einfachen Constraint-Propagation-System und Techniken zur Kräftesimulation (Force-Directed Layout) ein geeigneter Layoutalgorithmus für komplexe graph-artige Diagrammtypen konstruieren lässt, welcher im Editor für UML-Klassendiagramme eingesetzt wird.

Nachdem Kapitel 5 und 6 sich mit der Analyse und Transformation von Diagrammen sowie dem Diagrammlayout befasst haben, wendet sich Kapitel 7 der Benutzungsschnittstelle zu. Mit den bis dahin diskutierten Mitteln können zwar komfortable Zoom-Transformationen definiert werden; ihre geeignete Anwendung ist allerdings noch völlig der Benutzerin überlassen. Deshalb wird noch eine allgemeine Schnittstellenkomponente vorgestellt, welche sich in Editoren für beliebige Diagrammsprachen mit hierarchischer Struktur einsetzen lässt. Die Anwendung vorgegebener Zoom-Transformationen kann mit dieser Komponente komfortabel gesteuert werden und es ist auf einfache Weise möglich, Abstraktionsansichten des gesamten Diagramms zu erzeugen. Kapitel 8 fasst noch einmal die wichtigsten Ergebnisse dieser Arbeit zusammen und bietet einen Ausblick auf mögliche zukünftige Entwicklungen.



## Kapitel 2

# Semantische Fokus-und-Kontext-Darstellung

Das folgende Kapitel befasst sich zunächst mit bekannten Forschungsergebnissen zur Arbeit mit umfangreichen zweidimensionalen Informationsdarstellungen: Wir erörtern typische Probleme und dafür vorgeschlagene Lösungsmöglichkeiten und entwickeln daraus ein Konzept, wie derartige Techniken für den Einsatz in Diagrammeditoren als eine erweiterte Anwendung von Diagrammtransformationen aufgefasst werden können. Es zeigt sich, dass sich diese Ideen gut in DiaGen integrieren lassen und dass sie gegenüber anderen Ansätzen einige wesentliche Vorzüge bieten.

### 2.1 Probleme beim Arbeiten mit großen Diagrammen

Die üblichen Navigationsmöglichkeiten für zweidimensionale grafische Darstellungen, bei denen ein rechteckiger Ausschnitt aus einem virtuellen Gesamtbild angezeigt wird, wobei die Anwenderin die Skalierung ("Zoom") und Position ("Pan") dieses "Fensters" bestimmen kann, und die schon von Beginn an in DiaGen implementiert waren, bieten beim Arbeiten mit großen Diagrammen nur eine begrenzte Hilfe; hier zeigen sich schnell zwei wichtige Defizite (vgl [7]):

1. Wenn eine hohe Vergrößerungsstufe gewählt wird, um einen bestimmten Ausschnitt im Detail betrachten zu können, ist keine Kontextinformation mehr sichtbar; d.h. die Benutzerin weiß nicht, welcher Teil des gesamten Diagramms gerade angezeigt wird und wie dieser Teil mit anderen Bereichen in Verbindung steht.
2. Wenn dagegen eine niedrige Vergrößerungsstufe gewählt wird, um derartige Überblicksinformationen zur Verfügung zu haben, wird die Darstellung durch die vielen angezeigten Details schnell unübersichtlich; die entsprechend winzigen Orientierungsmerkmale (Beschriftungen, Icons) sind

kaum zu erkennen, was das Erkennen der Zusammenhänge zusätzlich erschwert.

Diese Probleme sind natürlich nicht auf Diagrammeditoren beschränkt, sondern treten immer dann auf, wenn größere Mengen von zweidimensionaler geometrisch organisierter Information dargestellt und bearbeitet werden müssen. Deshalb sind allgemeine Techniken zur Informationspräsentation entwickelt und untersucht worden, die den Umgang mit solchen Darstellungen erleichtern.

## 2.2 Techniken zur Darstellung großer Diagramme

Zwei der bekanntesten und am meisten eingesetzten Techniken sind das *semantische Zoomen* und die *Fokus-und-Kontext-Darstellung*.

Die Fokus-und-Kontext-Darstellung<sup>1</sup> befasst sich mit dem ersten der beiden angesprochenen Probleme: Die Vergrößerungsstufe bleibt bei dieser Darstellungsform nicht über die gesamte sichtbare Informationsmenge gleich, sondern kann variiert werden. Informationen, die im Zentrum des Benutzerinteresses liegen werden stark vergrößert und nehmen verhältnismässig viel sichtbare Bildfläche ein. Die weniger bedeutsame Kontextinformation wird mit geringerem Vergrößerungsgrad dargestellt, so dass hier eine größere Informationsmenge sichtbar bleibt. Dadurch kann die Benutzerin die relevante Information in hohem Detailgrad betrachten und sieht trotzdem, wie sie in den Gesamtkontext einzuordnen ist. Empirische Untersuchungen [7] belegen, dass diese Technik tatsächlich das Arbeiten mit grafischen Darstellungen signifikant erleichtern kann.

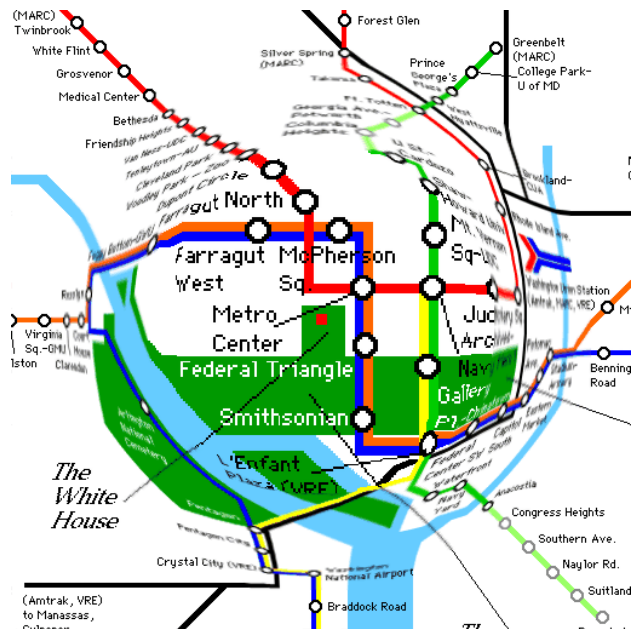


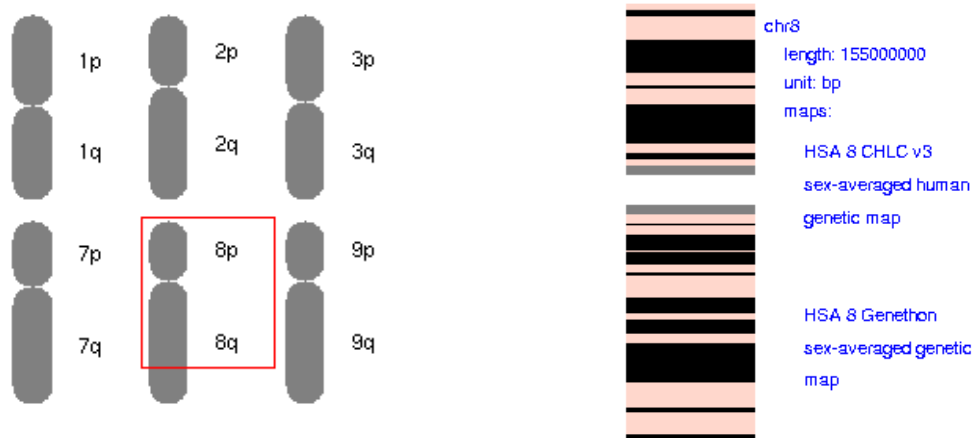
Abbildung 2.1: U-Bahn-Netz in Fisheye-Darstellung

Abbildung 2.1 zeigt als Beispiel den Plan eines U-Bahn-Netzes, bei dem ein Fokusbereich im Zentrum herausvergrößert wurde.<sup>2</sup> Der sich ergebende Effekt

<sup>1</sup>auch "Detail in Context" [4]

<sup>2</sup>Entnommen aus einer WWW-Demonstration des FAD-Toolkits,

<http://www.cs.indiana.edu/hyplan/tkeahay/research/fad/fad.html>



*Abbildung 2.2: Semantisches Zoomen in einer Darstellung des menschlichen Genoms*

ähnelt dem eines Weitwinkelobjektivs (engl. “Fisheye Lens”) in der Fotografie, da die unterschiedliche Vergrößerung durch eine kontinuierliche geometrische Transformation realisiert wird (mehr dazu in Abschnitt 2.7). Deshalb wird diese Darstellungstechnik oft auch als “Fisheye View” bezeichnet.

Das semantische Zoomen stellt einen Lösungsansatz für das zweite Problem dar. Hier wird nicht immer die gesamte Information dargestellt, die sich innerhalb des sichtbaren Bereichs befindet, sondern der Detailgrad der Darstellung wird an die Vergrößerungsstufe angepasst. Bei niedriger Vergrößerung und viel sichtbarer Information wird die Darstellung abstrahiert und Detailinformation unterdrückt, so dass die optische Komplexität über verschiedene Vergrößerungsstufen ungefähr gleich bleibt.

Zur Verdeutlichung zeigt Abbildung 2.2 zwei Ausschnitte aus einer Karte des menschlichen Genoms in unterschiedlichen Vergrößerungsstufen.<sup>3</sup> In der Übersichtsansicht links sind die einzelnen Chromosomen nur als Umriss dargestellt. Wenn man in dem markierten Ausschnitt hineinzoomt, wird die Detailstruktur des Chromosoms Nr. 8 zusammen mit zusätzlichen Informationen sichtbar.

Beide Darstellungstechniken lassen sich offensichtlich sinnvoll kombinieren, da sie voneinander unabhängig eingesetzt werden können und sich ergänzende Ziele verfolgen. In der vorliegenden Arbeit wurde eine solche “semantische Fokus- und-Kontext“-Darstellung (im Folgenden als “sFK-Darstellung” abgekürzt) für Diagrammeditoren realisiert.

## 2.3 Weitere Darstellungsvarianten

Natürlich existieren neben den genannten beiden Techniken auch noch andere Möglichkeiten, die den Umgang mit großen zweidimensionalen Informations-

<sup>3</sup>Entnommen aus einer WWW-Demonstration des zomit-Toolkits,  
<http://www.infobiogen.fr/services/zomit/zoommap.html>

darstellungen erleichtern sollen. Dazu seien drei in der Praxis angewandte Beispiele genannt:

Eine Möglichkeit bietet die gleichzeitige Darstellung von getrennten Ansichten in mehreren Vergrößerungsstufen durch mehrere oder geteilte Bildschirmfenster: Eine "Arbeitsansicht" zeigt den Bereich, der gerade von Interesse ist, in hoher Vergrößerung, während eine "Überblicksansicht" in niedriger Vergrößerung die Einordnung des Arbeitsbereichs in die Gesamtinformation ermöglicht. Die Überblicksansicht zeigt dabei typischerweise die gesamte Informationsmenge, wobei der Ausschnitt, der sich innerhalb der Arbeitsansicht befindet, durch eine Markierung hervorgehoben ist. Solche Lösungen finden sich zum Beispiel bei Diagrammeditoren wie "Together" oder auch bei Landkarten-Darstellungen.

Die Übersichtsansicht muss nicht einfach eine Kopie der Arbeitsansicht in geringerer Vergrößerung darstellen, sondern kann die Information anders organisieren, um eine alternative Navigationsstruktur zur Verfügung zu stellen: Hierbei bietet sich vor allem eine Baumdarstellung an, falls die Information eine hierarchische Struktur aufweist [3]. So verfügt beispielsweise die Mehrzahl der gängigen Umgebungen zur Softwareentwicklung (sowohl Design-Tools wie Rational Rose als auch Tools zur Code-Entwicklung wie JBuilder) über Navigationsfenster mit Baumdarstellungen.

Eine solche alternative Darstellung lässt sich natürlich auch gut mit der realisierten "semantischen Fokus-und-Kontext"-Lösung kombinieren; auf diesen Aspekt werden wir in Kapitel 7 zurückkommen. Voraussetzung ist dabei natürlich, dass die darzustellende Information zumindest teilweise als Baum organisiert werden kann.

Eine dritte Hilfe zur Orientierung in großen Informationsstrukturen besteht darin, die Abfolge von Navigationsschritten (Vergrößerung/Verkleinerung, Ausschnittverschiebung), welche bislang ausgeführt wurden, im Schnelldurchlauf zu wiederholen, um der Benutzerin in Erinnerung zu rufen, wie die aktuelle Ansicht entstanden ist [3].

Bei vielen Navigationsschritten wirkt dieses Vorgehen leicht verwirrend; stattdessen ist es wahrscheinlich verständlicher, die geraffte Darstellung eines direkten Übergangs von der Gesamtansicht zur aktuellen Ansicht anzubieten. Dieses Vorgehen wurde z. B. in manchen Computerspielen realisiert. Zur Untersuchung derartiger Navigationsvorgänge und zur Berechnung "optimaler" Übergänge haben sich sogenannte "Space-Scale-Diagramme" als geeignetes theoretisches Hilfsmittel bewährt [2].

## 2.4 Semantisches Zoomen

Nach diesem Exkurs soll im Folgenden ein genauerer Blick auf existierende Implementierungen der beiden Haupttechniken "semantisches Zoomen" und "Fokus und Kontext" geworfen werden. Dabei interessieren vor allem die typischen

damit verbundenen Einschränkungen und Auswirkungen im Hinblick auf eine Anwendung der Techniken in DiaGen bzw. damit entwickelten Diagrammeditoren:

Die Technik des semantischen Zoomens präsentiert den gesamten sichtbaren Bildbereich in einer einheitlichen Vergrößerungsstufe (einfache Skalierung), falls nicht explizit mehrere Ansichten angeboten werden (durch mehrere Fenster oder "Ports" [9]). Das bedeutet, dass sich die geometrische Struktur der Ansicht nicht zu ändern braucht; Objekte werden – abhängig von der Vergrößerungsstufe – in unterschiedlichem Detailgrad dargestellt, aber ihre Form, Fläche und Lagebeziehung im Koordinatensystem des dargestellten Modells bleibt unverändert.

Bei der Anwendung des semantischen Zoomens für praktische Probleme bestehen wenig Variationsmöglichkeiten. Deshalb bietet sich diese Technik für die Erstellung eines allgemeinen Toolkits an, auf dessen Basis dann ein breites Spektrum von Anwendungen realisiert werden kann.

Ein Beispiel für ein solches Toolkit ist die Pad++ Umgebung [9], die unter Führung von Benjamin Bederson entwickelt wurde. Pad++ stellt der Programmierin die Abstraktion einer unendlichen 2D-Fläche zur Verfügung, in der Objekte beliebiger Größe in beliebiger Verschachtelung platziert werden können. Unterstützung für semantisches Zoomen wird dadurch geboten, dass abhängig von der sichtbaren Größe der Objekte verschiedene Darstellungen (Detailstufen) definiert werden können. Während die Programmierin sich nur mit der Definition der Informationsobjekte beschäftigen muss, übernimmt das Toolkit das komplette Bildschirm-Management, d.h. die Bestimmung der sichtbaren Objekte, Auswahl der geeigneten Darstellung, falls mehrere Detailstufen zur Verfügung stehen, und die Visualisierung ("Rendering") der Objekte unter Berücksichtigung ihrer gegenseitigen Überlappung. Größtmögliche Effizienz wird dabei zum einen durch den Einsatz geeigneter geometrischer Datenstrukturen und zum anderen durch die Nutzung von moderner Hardware zur Grafik-Beschleunigung erreicht [10].

Auf der Basis von Pad++ wurden verschiedene Anwendungen (z. B. ein Hypertext-Browser) realisiert, bei denen das semantische Zoomen eine entscheidende Rolle spielt; diese zeichnen sich auch dadurch aus, dass versucht wurde, "alternative Benutzungsschnittstellen" [8] zu erforschen, die von den etablierten Fensterumgebungen im Stile von Xerox/Apple abweichen. Zu diesem Zweck bietet Pad++ z. B. auch eine vollständige alternative Darstellungs- und Bedienumgebung mit eigenen Konzepten wie "Ports" und "Linsen" statt der üblichen Fenster, eigenen Kontrollelementen usw.

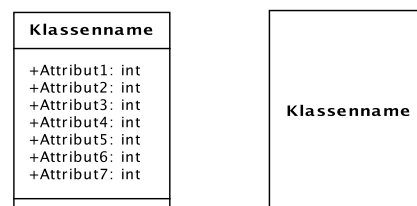
Infolgedessen zeigen diese Anwendungen zwar interessante alternative Ansätze, aber ihre Bedienbarkeit für die "Normalbenutzerin" wird dadurch nicht unbedingt vereinfacht. (Natürlich wurde Ähnliches vor zwanzig Jahren auch über die ersten Fensteroberflächen gesagt.) Für die Anwendung im Rahmen des DiaGen-Projektes wollten wir allerdings Benutzbarkeit über Experimentierfreude stellen.

Im Hinblick auf eine Integration des semantischen Zoomens in das DiaGen-Framework wäre offensichtlich eine Portierung des DiaGen-Systems auf die Basis des Jazz-Toolkits [11] – des in Java geschriebenen Nachfolgers von Pad++ – zu erwägen. Davon haben wir jedoch Abstand genommen, einerseits weil ein solches Vorgehen tiefgreifende Änderungen des bislang entwickelten Systemkerns von DiaGen erfordert hätte, vor allem aber, weil Pad++/Jazz, wie erwähnt, kein reines Darstellungstoolkit ist, sondern eine komplett eigenständige grafische Oberfläche. Daher ließe sich ein derartiger Editor nicht mehr mit einer Standard-Java-Anwendung kombinieren, welche das normale AWT/Swing-Toolkit nutzt. Die Philosophie von DiaGen besteht aber gerade darin, Diagrammeditoren als Bausteine (Java-Beans) zur Verfügung zu stellen, die einfach in größere Programmsysteme integriert werden können.

## 2.5 Probleme beim semantischen Zoomen

Wie bereits erwähnt, besteht ein Hauptcharakteristikum von üblichen Anwendungen des semantischen Zoomens darin, dass die geometrische Struktur der Information unabhängig von der gewählten Vergrößerungsstufe bleibt. Das vermeidet Verzerrungsprobleme und ermöglicht den Einsatz allgemeiner Toolkits, stellt aber für viele Anwendungen eine offensichtliche Einschränkung dar:

- Zum einen setzt dieses Vorgehen voraus, dass die geometrische Vergrößerung der Darstellung mit einer inhaltlichen Abstraktion korreliert, d.h. Objekte von höherer hierarchischer Ordnung umschließen geometrisch ihre weniger wichtigen Details, welche beim Herauszoomen verschwinden. Diese Eigenschaft muss aber durchaus nicht immer gegeben sein: So sind z. B. in Baumdarstellungen Elternknoten im Allgemeinen von größerer Wichtigkeit als ihre Kindknoten, obwohl sie diese nicht geometrisch enthalten; es würde keinen Sinn machen, Elternknoten auszublenden, aber Kindknoten weiterhin darzustellen.
- Selbst wenn die geometrische Struktur der Darstellung ihre Abstraktionsstruktur angemessen repräsentiert, kann es trotzdem ungünstig sein, dieselbe Objektform in verschiedenen Abstraktionsstufen beibehalten zu müssen. Als Beispiel sollen hier Klassendarstellungen in UML-Diagrammen dienen;<sup>4</sup> diese enthalten Informationen über Attribute und Operationen der Klassen, die bei niedrigerem Detailgrad sinnvollerweise ausgeblendet werden sollten.



**Abbildung 2.3:** Abstraktion von UML-Klassen bei unverändertem Seitenverhältnis

<sup>4</sup>Die Syntax und Bedeutung dieser Diagramme wird später genauer behandelt; sie ist für die in diesem Kapitel gezeigten Beispiele nicht von Bedeutung.

Die Darstellung einer Klasse mit vielen Attributen erfordert dabei die Form eines Rechtecks, das höher als breit ist; werden die Attribute jedoch ausgeblendet, erweist sich diese Form zur Darstellung des Klassennamens alleine als ungünstig.

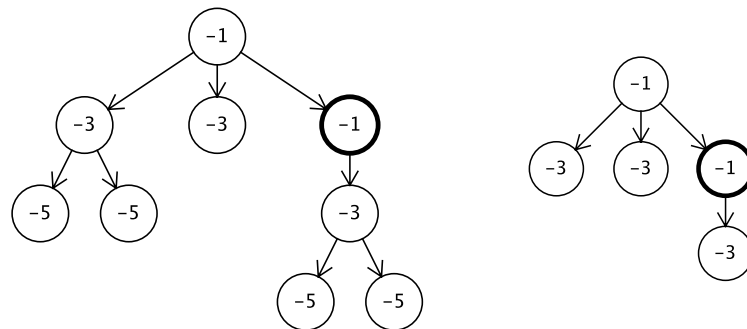
- Auch bei der Verwendung von speziellen Abstraktionsdarstellungen, die spezifisch für die jeweilige Informationsklasse (bzw. Diagrammsprache im Falle von Diagrammeditoren) definiert sind, kann sich die mangelnde geometrische Flexibilität der Darstellung als problematisch erweisen.

Als Konsequenz aus den angesprochenen Problemen erscheint es sinnvoll, zu überlegen, ob die starre Beibehaltung der geometrischen Informationsstruktur über verschiedene Vergrößerungsstufen, wie sie sich in Pad++ und ähnlichen Systemen findet, aufgegeben werden kann. Zur Grundidee des semantischen Zoomens steht es offensichtlich nicht im Widerspruch, wenn die Darstellung von Objekten in Form und Lage geringfügig an den gewählten Detailgrad angepasst wird; diese Veränderungen müssen nur gewährleisten, dass die Benutzerin die Identität von Objekten vor und nach einem Zoom-Vorgang noch problemlos erkennen kann. Außerdem dürfen solche Anpassungen natürlich nicht Objekte isoliert betrachten, sondern sie müssen immer den sichtbaren Informationsausschnitt als Ganzes berücksichtigen.

Das bedeutet, dass die Geometrie der Darstellung bei Veränderungen der Ansicht reorganisiert werden muss, um der veränderten Form und Lage einzelner Informationsobjekte (Diagrammkomponenten) Rechnung zu tragen und z. B. Überlappungen oder zu große Leerräume zu vermeiden. Eine geeignete automatische Anpassung der Darstellung ist offensichtlich keine einfache Aufgabe, und sie führt uns direkt zur Betrachtung der Fokus-und-Kontext-Techniken, welche mit demselben Problem zu kämpfen haben.

## 2.6 Fokus und Kontext

Wie bereits erwähnt, besteht die Grundidee des Fokus-und-Kontext-Prinzips darin, die Vergrößerungsstufe nicht im gesamten sichtbaren Bildausschnitt gleich zu lassen, sondern sie abhängig von der Relevanz der einzelnen Informationsobjekte zu variieren. Die erste generelle Formulierung dieses Prinzips findet sich bei Furnas [1]: Ob und wie groß ein Informationsobjekt dargestellt werden soll, ergibt sich aus einem (numerischen) Wert, der seine aktuelle Relevanz für die Benutzerin bemisst. Dieser "Degree of Interest" DOI setzt sich aus zwei Komponenten zusammen: dem "Level of Detail"-Wert  $LOD(x)$ , welcher die Wichtigkeit des Objekts *a priori* angibt und unabhängig von der aktuellen Wahl des Bildausschnitts ist, sowie dem "Distance from Focus"-Wert  $d(x, f)$ , welcher den Abstand des Objekts vom Zentrum des Benutzerinteresses  $f$  repräsentiert. Ein Objekt soll um so größer dargestellt werden, je größer es von Natur aus ist und je näher es



**Abbildung 2.4:** Baum mit Relevanzwerten, Abstraktion durch Entfernen von wenig relevanten Teilen

dem Zentrum der Aufmerksamkeit liegt. Die Begriffe “Größe” und “Abstand” kann man dabei für die praktische Umsetzung geometrisch interpretieren, aber sie stellen eigentlich kognitive Maße dar, und bieten daher Möglichkeiten, die Realisierung dieser Technik zu variieren.

Zur Verdeutlichung sei hier kurz ausgeführt, wie Furnas die eingeführten Begriffe auf Baumdarstellungen anwendet, ein Fall, der auch in der Implementierung der sFK-Darstellung in DiaGen noch auftauchen wird.

Der Fokus  $f$  ist dabei ein ausgewählter Knoten des Baums: den Abstand eines anderen Knotens zum Fokus kann man als die (geringstmögliche) Anzahl von Kanten definieren, über die sich der Fokus von diesem aus erreichen lässt. Die Wichtigkeit eines Knotens wird als die negative Tiefe des Knotens (Entfernung zur Wurzel  $w$ ) definiert. Der Relevanzwert eines Knotens berechnet sich dann als

$$\text{DOI}(x) = -d(x, w) - d(x, f)$$

Abbildung 2.4 zeigt einen Beispielbaum mit den Relevanzwerten, die sich bei der Wahl des markierten Knotens als Fokus ergeben. Für die Darstellung könnte man nun die Knoten abhängig von ihrer Relevanz verschieden groß anzeigen oder diejenigen Knoten ausblenden, deren Relevanzwert einen bestimmten Grenzwert unterschreitet, wie dies rechts in der Abbildung geschehen ist.

Natürlich sind die vorgestellten Definitionen willkürlich; abhängig vom gewünschten Darstellungsergebnis kann man z. B. auch noch Gewichtungsfaktoren oder nichtlineare Funktionen in die Berechnung einfließen lassen.

Man erkennt, dass das vorgestellte Prinzip viele Freiheiten und Wahlmöglichkeiten bei der praktischen Umsetzung erlaubt und für viele Informationsstrukturen anwendbar ist. Eine der ersten vorgeschlagenen Anwendungen bestand z. B. in der Auswahl von anzuzeigenden Textzeilen in Texteditoren; statt einfach eine Bildschirmseite zusammenhängenden Kontext anzuzeigen, sollten in einem Programmeditor besonders relevante Kontextzeilen ausgewählt werden (z. B. Schleifenanfänge). Die Anwendung, die jedoch typischerweise mit dieser Technik as-

soziiert wird, ist die nichtlineare Verzerrung von zweidimensionalen Ansichten (im Folgenden als "Fisheye"-Darstellung bezeichnet).

## 2.7 Optische Fisheye-Transformation

Abbildung 2.5 zeigt, wie sich die Darstellung eines regelmäßigen rechtwinkligen Gitters unter einer solchen Transformation verändert. Gitterfelder nahe dem Fokus im Zentrum werden vergrößert, während diejenigen, die näher am Rand liegen, verkleinert werden. Dies steht im Einklang mit der oben vorgestellten Definition von "Fokus-und-Kontext", wenn man als Modell zugrundelegt, dass alle Gitterfelder a priori von gleicher Wichtigkeit sind und der Abstand zum Fokus als geometrische Distanz interpretiert wird. Der optische Effekt ähnelt einer Darstellung auf einem Gummituch, welches am Fokuspunkt zum Betrachter hin ausgebeult wurde.

Realisiert wird eine solche Darstellung durch Anwendung einer nichtlinearen geometrischen Transformation auf die fertige zweidimensionale Informationsausgabe [5], welche hier durch das Gitternetz repräsentiert wird; in der Praxis könnten dies stattdessen z. B. Landkarten sein (vgl. Abb. 2.1), oder auch Diagramme. Da die veränderte Darstellung durch einen nachgeschalteten Transformati-onsschritt erreicht wird, ist diese Möglichkeit zur "Fokus-und-Kontext"-Darstellung auf beliebige Informationen anwendbar, und eignet sich daher auch dazu, in einem allgemeinen Toolkit implementiert zu werden. Das einzige uns bekannte derartige Toolkit ist allerdings kommerziell.<sup>5</sup>

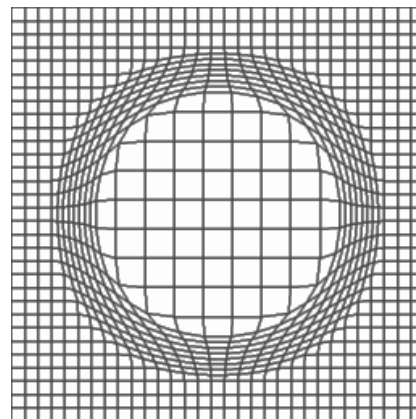


Abbildung 2.5: Gitter nach optischer Fisheye-Transformation

In der Literatur findet sich eine Reihe von Variationen dieser Fisheye-Technik: unter Beibehaltung des grundlegenden Prinzips einer geometrischen Ausgabetransformation, die vom dargestellten Inhalt unabhängig ist, kann man z. B. die Verzerrung auf begrenzte Bildbereiche beschränken und auch mehrere Fokuspunkte zulassen. Für manche Anwendungen ist es sinnvoll, die Verzerrung für  $x$ - und  $y$ -Koordinaten separat zu berechnen, was einer Verwendung von Manhattan-Distanz statt des euklidischen Abstands entspricht.

Es wäre möglich, einen derartige Transformationsschritt auch in die von DiaGen erzeugten Diagrammeditoren einzubauen. Neben der Navigation mit den üblichen "Zoom"- und "Pan"-Befehlen könnte die Benutzerin dann z. B. auch die Fokuspunkte und den Verzerrungsgrad bestimmen. Tatsächlich benutzen DiaGen-

<sup>5</sup>FAD-Toolkit, <http://www.visual-focus.com/>

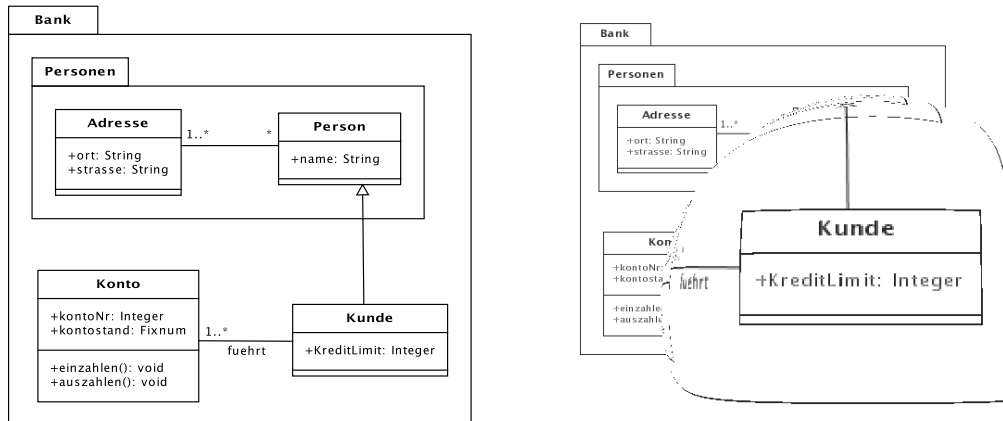


Abbildung 2.6: Optische Fisheye-Transformation eines UML-Diagramms

Editoren ohnehin einen nachgeschalteten Transformationsschritt, um die Diagrammdarstellung von "Weltkoordinaten" (mit denen die Zeichenbefehle arbeiten) auf Bildschirmkoordinaten umzurechnen. Für die dabei erforderliche Skalierung und Translation wird der integrierte Ausgabe-Transformationsmechanismus des benutzten Java-2D Toolkits eingesetzt. Dieser erlaubt allerdings nur affine Transformationen; für eine nichtlineare Fisheye-Transformation müsste das Java-2D API erweitert oder umgangen werden, was uns wenig wünschenswert erscheint. Die Anwendung einer nichtlinearen Transformation auf jeden einzelnen Bildpunkt, wie sie der optische Fisheye-Ansatz verlangt, wäre auch äußerst rechenaufwendig. Eine ausreichende Geschwindigkeit für interaktives Arbeiten könnte höchstens durch Integration von maschinenspezifischem Code erreicht werden – mit den sich daraus ergebenden Einschränkungen für die Portabilität von DiaGen.

## 2.8 Probleme der Fisheye-Darstellung

Die Effekte einer Fisheye-Transformation würden auch gerade bei Diagrammeditoren oft unnatürlich wirken: Diagramme sind typischerweise stark schematisch, sie bestehen aus einfachen geometrischen Bestandteilen, enthalten viele gerade Linien und rechte Winkel usw. Typische Fisheye-Transformationen können diese Eigenschaften nicht erhalten, weshalb die Darstellung unnatürlich erscheint. Auch enthalten Diagramme oft viel Text, welcher nach der Transformation nur schwer lesbar ist (vgl. Abb. 2.1). Nicht zuletzt lässt eine nachgeschaltete optische Fisheye-Transformation auch keine Berücksichtigung von spezifischen Gegebenheiten der Diagrammsprache zu. Zur Verdeutlichung zeigt Abbildung 2.6, wie die Anwendung einer Fisheye-Transformation auf ein UML-Diagramm aussehen könnte; Fokuspunkt ist dabei das Element "Kunde" rechts unten.

Die Transformation wurde mit Hilfe eines Bildbearbeitungsprogramms erzeugt und ist sicher nicht optimal auf die Erfordernisse abgestimmt, aber die Abbil-

ung macht deutlich, dass die Anwendung geometrischer Verzerrungen auf regelmäßige Diagrammstrukturen unnatürlich wirkt und bei der Bearbeitung stört.

Natürlich hat sich die Forschung mit dieser Problematik beschäftigt und nach allgemeinen Lösungen gesucht, welche die ungewohnte Geometrie von Fisheye-Darstellungen vertrauter machen oder kompensieren sollen [4]. Es wurde beispielsweise vorgeschlagen, Transformationen zu verwenden, welche (wie in Abb 2.5) möglichst große Teilbereiche des Kontextbereichs und auch die unmittelbare Umgebung des Fokuspunktes nur skalieren und Verzerrungen auf die Übergangsbereiche beschränken. Zusätzlich können verschiedene optische Hinweise die Art der Verzerrung verständlicher machen und die Wahrnehmung der Information erleichtern, so z. B. die Überlagerung der eigentlichen Information mit einem genauso transformierten Gitter oder die Verwendung von Licht- und Schatteneffekten, welche den Eindruck einer gewölbten dreidimensionalen Oberfläche hervorrufen sollen.

Allerdings können solche Ansätze nur eine eingeschränkte Hilfe sein, da es von der Art der jeweils dargestellten Information abhängt, welche geometrischen Eigenschaften (Orthogonalität, Lagebeziehungen, bestimmte Größenverhältnisse usw. [21]) bedeutungstragend sind, und welche problemlos verändert werden können. Gerade bei Diagrammsprachen mit einer definierten Semantik ist es aber notwendig, keine bedeutungsverändernden Eingriffe in die Darstellung vorzunehmen. Ein rein geometrisch orientierter "Fisheye"-Ansatz stößt hier zwangsläufig an seine Grenzen, da eine allgemeine optimale Lösung nicht möglich ist.

Statt den kompletten Bildausschnitt Bildpunkt für Bildpunkt zu transformieren, beschränken sich manche Implementierungen darauf, nur einzelne "Schlüssel-punkte" zu transformieren und die Ausgabe dann ausgehend von diesen Punkten aufzubauen. Dieses Vorgehen reduziert zum einen den Rechenaufwand so weit, dass keine Effizienzprobleme mehr zu befürchten sind; zum anderen werden manche Verzerrungen vermieden. Dafür kann es leicht zu Unregelmäßigkeiten wie Überlappungen eigentlich getrennter Bereiche kommen.

Als Beispiel sei hier eine "Fokus-und-Kontext"-Darstellung von Graphen genannt [6]: Es ist naheliegend, dabei nur die Knotenpositionen zu transformieren; die Kanten entstehen dann durch geradlinige Verbindung der verschobenen Knoten. Unregelmäßigkeiten zeigen sich hier z. B. darin, dass diese Kanten (die bei "vollständiger" Fisheye-Transformation gebogen dargestellt werden müssten) sich manchmal überkreuzen, obwohl sie sich in der unverzerrten Darstellung (auf welcher der Graphlayout-Algorithmus arbeitet) nicht berühren würden.

Als Konsequenz aus dieser Übersicht über existierende Fokus-und-Kontext-Lösungen können wir festhalten, dass bei einer Realisierung mit Hilfe einer nachgeschalteten Fisheye-Transformation im Allgemeinen wünschenswerte Layout-Eigenschaften zerstört werden. Eine optimale Lösung für Diagrammeditoren würde eine für die jeweilige Diagrammsprache spezifische Layoutanpassung nach der Transformation erfordern.

## 2.9 Semantische Fokus-und-Kontext-Darstellung in DiaGen

Die bisher betrachteten Ansätze waren entweder auf eine konkrete Anwendung spezialisiert oder setzen als Darstellungstransformationen erst auf der Ebene der Bilderzeugung an, ohne die Struktur der zugrundeliegenden Information näher zu berücksichtigen. Im Gegensatz dazu bietet das Konzept von DiaGen noch andere Ansatzpunkte zur verbesserten Darstellung großer Diagramme.

Für das im Folgenden vorgestellte Darstellungskonzept ist es entscheidend, dass DiaGen zum einen ein allgemeines System ist, welches zur Bearbeitung verschiedenster Diagrammtypen genutzt werden kann. Andererseits sind die generierten Editoren aber auf einen konkreten Diagrammtyp spezialisiert. Da diese Spezialisierung durch eine Beschreibung der Diagrammsprache in einer DiaGen-spezifischen Spezifikationsprache erfolgt (vgl. Abschnitt 1.1), besteht die Möglichkeit, in diese Spezifikation auch Informationen aufzunehmen, welche die zu entwickelnde sFK-Darstellung betreffen. Wie bereits ausgeführt, muss eine solche Darstellung ja nach Möglichkeit die zugrundeliegende Diagrammsprache berücksichtigen. Außerdem enthalten viele Diagrammsprachen ohnehin bereits Konstrukte zur Abstraktion, welche dann sinnvollerweise auch in einem Diagrammeditor genutzt werden sollten.

Um genauer erkennen zu können, wie solche Möglichkeiten in das System integriert werden können, betrachten wir die interne Struktur eines mit DiaGen entwickelten Editors noch etwas genauer.

Abbildung 2.7 zeigt die verschiedenen Module, aus denen sich ein DiaGen-Editor zusammensetzt. Bei dieser Darstellung handelt es sich um ein UML-Strukturmodell, das mit einem in dieser Arbeit beschriebenen Editor erstellt wurde. Pfeile zwischen den einzelnen Modulen repräsentieren Abhängigkeiten: Wenn sie mit der Beschriftung "modifiziert" versehen sind, greift das Quellmodul verändernd auf Bestandteile des Zielmoduls zu, sonst handelt es sich um Datenabhängigkeiten (d. h. der Datenfluss erfolgt entgegen der Pfeilrichtung vom Ziel zum Quellmodul). Die einzelnen Module haben folgende Aufgaben:

- Das Diagramm-Modul beschreibt Form und Position der Komponenten des Diagramms. Neben dieser Objekt-Repräsentation enthält es auch ein Graph-Modell des Diagramms, welches die Komponenten und ihre Lagebeziehungen durch einen Hypergraphen beschreibt.<sup>6</sup> Zusätzlich bietet dieses Modul die Möglichkeit zum Laden und Speichern von Diagrammen.

<sup>6</sup> Hypergraphen bilden eine Erweiterung gewöhnlicher Graphen, bei denen eine Kante über sogenannte "Tentakel" nicht nur zwei sondern eine beliebige Anzahl (1–n) von Knoten verbinden kann. DiaGen verwendet grundsätzlich typisierte Hypergraphen: jeder Hyperkante ist ein Typ (Markierung, "Label") zugeordnet, der definiert, wie viele Tentakel sie besitzt. Genauso, wie bei gerichteten Graphen die Orientierung einer Kante eine Rolle spielt, werden in diesem Modell die verschiedenen Tentakel einer Hyperkante unterschieden (anhand ihrer Ordnungsnummer); es spielt also eine Rolle, mit welchem ihrer Tentakel eine Hyperkante einen bestimmten Knoten berührt.

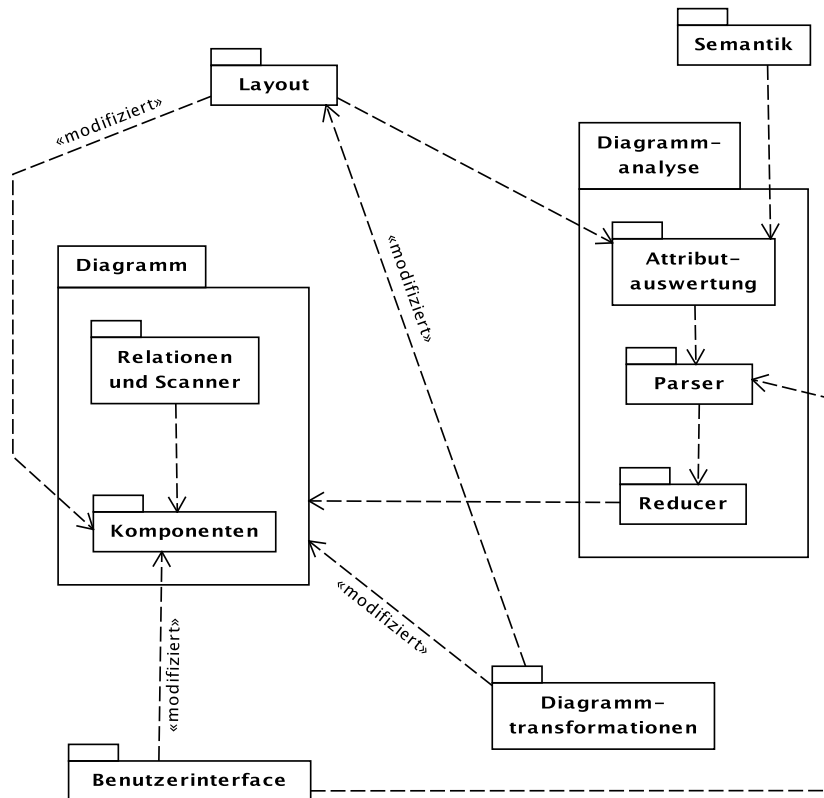


Abbildung 2.7: Interne Struktur eines DiaGen-Editors

- Das Benutzungsschnittstellen-Modul ermöglicht der Benutzerin die Interaktion mit dem Editor. Es stellt die Komponenten des Diagramms in einem oder mehreren Bildschirmfenstern dar und bietet Möglichkeiten, es durch direkte Manipulation zu modifizieren. Auch alle anderen Programmläufe, wie z. B. die Ausführung von Diagrammtransformationen, werden durch Steuerelemente der Benutzungsschnittstelle ausgelöst.
- Das Analyse-Modul erkennt in einem mehrstufigen Prozess Zusammenhänge zwischen den einzelnen Diagrammteilen und prüft, ob das Diagramm den Regeln der jeweiligen Diagrammsprache entspricht. Es baut dazu auf dem oben erwähnten Graph-Modell auf und verwendet Techniken aus dem Bereich der Graphgrammatiken. Als Ergebnis der Analyse wird eine geeignete interne Repräsentation des Diagramms ("Semantik") erzeugt, die dann außerhalb des Editors weiterverarbeitet werden kann.
- Das Layout-Modul unterstützt die Bearbeitung des Diagramms, indem es Form und Lage der Diagrammkomponenten automatisch so anpasst, dass die Korrektheit des Diagramms gewahrt bleibt. Dazu greift es auf die Informationen aus der Diagramm-Analyse zurück.
- Das Modul zur Diagrammtransformation ermöglicht umfangreichere pro-

grammgesteuerte Modifikationen des Diagramms, die in einer Art leistungsfähigem Makro-Mechanismus viele Bearbeitungsschritte zusammenfassen.

Statt nun lediglich das Benutzerschnittstellen-Modul um neue Darstellungsformen zu ergänzen, lassen sich die vom System gebotenen Möglichkeiten optimal nutzen, wenn die gewünschten Erweiterungen auch bei den anderen Modulen ansetzen. Das bedeutet, dass nicht nur die Darstellung des Diagramms verändert wird, sondern tatsächlich das Diagramm selbst abstrahiert wird, indem Teile entfernt und durch Abstraktionsdarstellungen ersetzt werden. Diese Abstraktionen sollten dabei denselben Größenmaßstab haben wie die detaillierten Objekte.<sup>7</sup> Zusammen mit der bereits vorhandenen Möglichkeit des "einfachen" optischen Zoomens kann dann dieselbe Wirkung erreicht werden wie durch semantisches Zoomen und Fokus-und-Kontext-Darstellung:

- Der Effekt des semantischen Zoomens ergibt sich, wenn die Vergrößerungsstufe verändert wird und durch gleichzeitige Abstraktion oder Verfeinerung des Diagramm der Detailgrad entsprechend angepasst wird.
- Eine Fokus-und-Kontext-Darstellung erhält man, indem man Diagrammteile mit zunehmender Entfernung vom Fokus immer weiter abstrahiert. Diese nehmen dadurch weniger Fläche ein, aber die Einordnung des Fokus in seine Umgebung bleibt trotzdem sichtbar.

Das Problem, wie die Geometrie des Diagramms bei Fokus-und-Kontext-Darstellung anzupassen ist, löst sich dann auch recht einfach: Das Layoutmodul, das ohnehin dafür zuständig ist, sprachspezifische Layoutkorrekturen vorzunehmen, kann auch dazu genutzt werden, das Diagrammlayout anzupassen, wenn Komponenten durch Abstraktion oder Verfeinerung ihre Größe geändert haben. Eine nichtlineare "Verzerrung" der Darstellung findet damit nicht statt.

Abbildung 2.8 zeigt als Beispiel eine sFK-Darstellung des UML-Beispieldiagramms aus Abbildung 2.6 mit dem Element "Kunde" als Fokus. Diese Element ist in allen Details sichtbar, während die umliegenden Kontextelemente abstrahiert worden sind. Wenn man den Effekt mit dem der optischen Fisheye-Transformation in Abbildung 2.6 vergleicht, sieht man, dass die Darstellung hier natürlicher wirkt und Verzerrungsprobleme vermeidet.

Zur Ausführung solcher Diagrammveränderungen bietet es sich natürlich an, das bereits vorhandene Transformationsmodul zu erweitern. Transformationsoperationen, die derartiges leisten, werden im Folgenden als "Zoom-Transformationen" bezeichnet. Das dabei notwendige Entfernen und Einfügen von Diagrammkomponenten durch Transformationen war als Grundbestandteil jeder

<sup>7</sup>Im Gegensatz dazu nehmen bei den anfangs vorgestellten Toolkits zum semantischen Zoomen die Abstraktionsdarstellungen dieselbe Fläche ein wie ihr kompletter Inhalt.

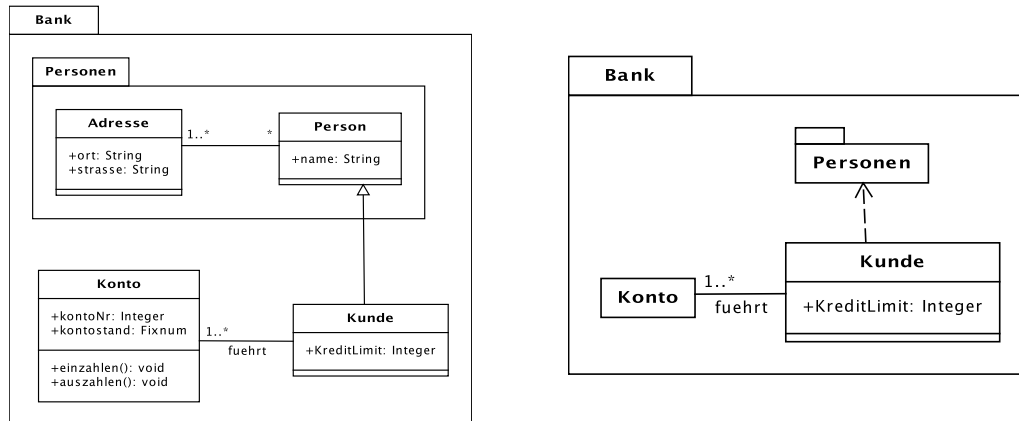


Abbildung 2.8: Semantische Fokus- und Kontext-Darstellung eines UML-Diagramms

Diagrammänderung bereits implementiert. Da das Diagramm selbst und nicht nur seine Darstellung verändert wird, ist eine abstrahierende Zoom-Transformation mit einem Informationsverlust verbunden. Sie muss natürlich trotzdem umkehrbar sein; es ist daher notwendig, zusätzliche Sprachmittel für die Definition von Transformationen einzuführen, um bei der Abstraktion "ausgeblendete" Information zu speichern und später wieder zu rekonstruieren. Durch die Nutzung sprachspezifischer Diagrammtransformationen zur Abstraktion, welche von der Programmiererin definiert werden, ist auch die Berücksichtigung spezifischer Gegebenheiten der Diagrammsprache auf einfache Weise möglich. Selbstverständlich sollte ein "Standardverhalten", soweit sinnvoll, mit möglichst wenig Aufwand spezifizierbar sein.

## 2.10 Vor- und Nachteile des realisierten Konzepts

Vergleicht man den vorgeschlagenen Ansatz mit einem rein geometrisch orientierten Vorgehen im Stile der zuvor beschriebenen Toolkits, so fallen eine Reihe von Vorteilen wie auch Nachteilen ins Auge.

Auf der positiven Seite sind dabei folgende Punkte zu berücksichtigen:

- Wie wir schon betont haben, besteht der fundamentale Vorzug darin, dass durch sprachspezifische Abstraktions- und Fokusoperatoren die Erfordernisse und Möglichkeiten der jeweiligen Diagrammsprache gezielt berücksichtigt werden können. Die Benutzerin hat es immer mit im Sinne der Diagrammsprache korrekten und unverzerrten Darstellungen zu tun, was der Verständlichkeit und Bedienbarkeit der generierten Editoren entgegenkommen sollte.
- Auch bei der Anpassung des Diagrammlayouts bieten sich viel mehr Möglichkeiten zu gezielten Korrekturen. Da Zoom-Transformationen beliebigen

Java-Code aufrufen können, kann über diesen Mechanismus auch das Layout abhängig von der gewählten Transformation explizit modifiziert werden, falls der allgemeine Layoutmechanismus nicht ausreicht.

Wenn solche automatischen Layoutkorrekturen in Einzelfällen nicht zu befriedigenden Ergebnissen führen, erlaubt DiaGen schließlich auch eine manuelle Nachkorrektur durch direkte Manipulation des Diagramms. Die flexible Struktur des Systems macht es dabei möglich, für solche manuellen Korrekturen noch zusätzliche Software-Unterstützung unter Rückgriff auf existierende Layoutalgorithmen zu bieten: Ein Diagrammeditor kann so z. B. Operationen anbieten wie "ziehe diese Teile näher zusammen" oder "suche eine Möglichkeit, diese Teile mit weniger überkreuzenden Verbindungen anzuordnen". Eine rein optische Verzerrung der Darstellung würde dagegen kaum Möglichkeiten zur manuellen Beeinflussung und Korrektur bieten.

- Einige bereits vorhandene Funktionen von DiaGen-Editoren können auch im Zusammenhang mit der sFK-Darstellung sinnvoll eingesetzt werden. So bietet DiaGen die Möglichkeit, bei Diagrammtransformationen nicht unmittelbar vom Ausgangs- zum Zieldiagramm zu wechseln, sondern automatisch eine Reihe von Zwischendiagrammen zu erzeugen und so einen animierten Übergang zu erreichen. Zusätzlich unterstützt DiaGen ein beliebig tiefes Rückgängig-Machen und Wiederholen aller Diagrammveränderungen. Diese Funktionalität lässt sich auch für Abstraktionstransformationen benutzen, da diese prinzipiell dieselben Mechanismen nutzen wie die bislang implementierten Diagrammtransformationen. Animation und Umkehrbarkeit sind zur Verbesserung der Bedienbarkeit und Verständlichkeit auch für "Fisheye"-Transformationen auf rein geometrischer Basis gefordert worden;<sup>8</sup> die Implementierung der sFK-Darstellung mittels Diagrammtransformationen liefert diese Möglichkeiten "umsonst".
- Schließlich kann man sich zunutze machen, dass der von DiaGen verfolgte Grammatik-Ansatz zur Diagrammanalyse bereits viel Hierarchieinformation zur Verfügung stellt. Diese Information kann nun durch Zoom-Transformationen sinnvoll zur Unterstützung der Benutzerin eingesetzt werden. Insgesamt fügen sich die vorgeschlagenen Erweiterungen nahtlos in das Gesamtkonzept von DiaGen ein.

Natürlich stehen den genannten Vorteilen des Zoom-Transformationen-Ansatzes auch einige Nachteile gegenüber:

- Die Zoom-Transformationen müssen für jede implementierte Diagrammsprache gesondert beschrieben werden und stehen so nicht von vornherein

---

<sup>8</sup>vgl. [4], dort werden diese Konzepte als "continuous and reversible transitions" bezeichnet

in jedem Editor zur Verfügung. Glücklicherweise ist jedoch die Formulierung von Zoom-Transformationen, die keine besonderen Zusatzbedingungen berücksichtigen müssen, mit relativ geringem Aufwand möglich, falls geeignete Sprachkonstrukte für die Spezifikation bereitgestellt werden. Unter Umständen können die Transformationen sogar fast vollautomatisch generiert werden. Diese Aspekte werden in den Kapiteln 5 und 7 genauer ausgeführt.

- Die Implementierung der von DiaGen vorausgesetzten diagrammtyp-spezifischen Layoutalgorithmen ist an sich schon aufwendig und wird durch die Berücksichtigung von Zoom-Transformationen noch weiter kompliziert. Um diese Problematik etwas zu entschärfen, ist es wünschenswert, eine Bibliothek von anpassbaren Basisalgorithmen für bestimmte Klassen von Diagrammsprachen zur Verfügung zu haben. Kapitel 6 beschreibt einen derartigen Algorithmus für graph-artige Diagramme.
- Da Zoom-Transformationen die interne Repräsentation des Diagramms verändern, muss das Diagramm vor und nach der Transformation nicht notwendigerweise semantisch äquivalent sein; dies liegt in der Verantwortung der Programmiererin, welche die Transformationen spezifiziert. Wenn dagegen nur die "Sicht" auf dasselbe Diagramm verändert würde, wäre die Äquivalenz automatisch sichergestellt. Als Konsequenz daraus ist es auch nicht möglich, gleichzeitig mit unterschiedlichen Ansichten desselben Datenmodells in verschiedenen Bildschirmfenstern zu arbeiten.

Alles in allem stellt der vorgeschlagene Ansatz sicher kein perfektes Universalmittel zur Darstellung großer Diagramme dar. Er bietet aber eine interessante neue Lösungsmöglichkeit, welche erst durch die spezifischen Eigenschaften des DiaGen-Konzeptes realisierbar wird und sich von vorangehenden Lösungen signifikant unterscheidet. Daher erscheint es gerechtfertigt, die praktische Umsetzbarkeit und die sich ergebenden Konsequenzen in einer Implementierung genauer zu untersuchen, wie dies im Hauptteil der vorliegenden Arbeit geschehen soll.



## Kapitel 3

# Beispielhafte Diagrammsprachen

Um die in Kapitel 2 vorgestellten Ideen zur semantischen Fokus-und-Kontext-Darstellung mittels Zoom-Transformationen in das DiaGen-System integrieren zu können, ist es natürlich notwendig, die Erweiterungen an Editorbeispielen zu testen. Im Rahmen der vorliegenden Arbeit wurden zwei Testeditoren implementiert, welche in den folgenden Kapiteln als Anschauungsbeispiele dienen sollen.

Zunächst haben wir einen Editor für Baumdiagramme, der bereits bei der Entwicklung vieler Komponenten des vorhandenen DiaGen-Basissystems als Testfall gedient hat, um Abstraktionsoperationen erweitert. Aufgrund seiner einfachen Spezifikation eignet sich dieser Editor besonders gut, um die essentiellen Voraussetzungen für Abstraktionsoperationen zu demonstrieren, welche die Graphtransformations- und Analyse-Module erfüllen müssen. Auf Layoutanpassungen kann günstigerweise völlig verzichtet werden, da der bereits existierende Baum-Layout-Algorithmus auch im Zusammenhang mit Zoom-Transformationen zufriedenstellend funktioniert

Um eine komplexere und besonders praxisrelevante Anwendung vorzuführen, wurde als zentrales Beispiel dieser Arbeit ein Editor für UML-Klassendiagramme implementiert. Dieser Editor erfordert zum einen relativ komplexe Zoom-Transformationen, wie sie mit einem allgemeinen Ansatz ohne sprachspezifische Anpassung nicht realisierbar wären. Daneben musste auch ein geeigneter Layoutalgorithmus entworfen und realisiert werden. Schließlich sollte der Editor als Beispiel dafür dienen, dass sich DiaGen auch zur Erstellung von Editoren für komplexe Diagrammsprachen eignet. Die Implementierung dieses aufwendigen Beispiels machte auch einige Erweiterungen der DiaGen-Komponenten zur Graphmanipulation notwendig, welche nicht direkt im Zusammenhang mit der Behandlung von Zoom-Transformationen stehen. Diese werden in Kapitel 5 ebenfalls besprochen.

### 3.1 Baumdiagramme und ihre interne Repräsentation

Baumdiagramme gehören zu den einfachsten wohlstrukturierten Diagrammtypen. Ihre grafischen Bestandteile (Diagrammkomponenten) sind Knoten, repräsentiert durch Kreise mit Textbeschreibungen, und gerichtete Kanten, repräsentiert durch Pfeile. Die einzige relevante räumliche Beziehung zwischen den Komponenten tritt dann auf, wenn das Ende einer Kante einen Knoten berührt oder sich innerhalb eines Knotens befindet. (Im zweiten Fall sorgt die Layoutkorrektur dafür, dass das Ende zum Rand des Knotens verschoben wird.) Die wesentlichen Aspekte der DiaGen-Spezifikation von Baumdiagrammen sollen hier kurz diskutiert werden; dabei müssen wir auch die Arbeitsweise des Analyse-Moduls von DiaGen etwas genauer betrachten.<sup>1</sup>

Wie bereits angesprochen, besteht ein Diagramm in DiaGen nicht nur aus seinen sichtbaren Komponenten, sondern auch aus einem internen Hypergraph-Modell (HGM), welches jede Komponente auf eine Hyperkante abbildet. Jede Komponente besitzt "Konnektoren", d. h. Teile, an denen die Komponente mit anderen in bedeutungstragenden räumlichen Beziehungen stehen kann. Im Hypergraph-Modell wird jeder dieser Konnektoren auf eine Tentakel der Komponenten-Hyperkante abgebildet; jede solche Tentakel besucht grundsätzlich genau einen (anonymen) Knoten. Die Komponenten eines Baumdiagramms sind Kreise ("circle"), welche die Knoten darstellen und über einen Konnektor verfügen und Pfeile ("arrow") zur Darstellung der Kanten, welche zwei Konnektoren für ihre beiden Enden besitzen.

Die räumlichen Beziehungen werden durch binäre Hyperkanten ("Relationen-Hyperkanten") dargestellt, welche die entsprechende Tentakel (bzw. die daran hängenden Knoten) verbinden. Im Fall von Baumdiagrammen sind dies die "inside"-Hyperkanten, welche die oben beschriebene Relation repräsentieren: Sie verbinden Pfeilenden mit den Knoten, innerhalb derer oder an deren Rand sie liegen. Das Entfernen einer Komponenten-Hyperkante löscht auch die zu ihr gehörigen Knoten und damit immer auch alle mit ihr verbundenen Relationskanten.

Abbildung 3.1 zeigt ein einfaches Baumdiagramm und das entsprechende HGM. Hier wie in allen folgenden Abbildungen von Hypergraphen sind Knoten als Punkte repräsentiert und Hyperkanten als Rechtecke, deren (ggf. nummerierte) Tentakel diese Knoten besuchen. Relationen-Hyperkanten, die immer genau zwei Tentakel besitzen, sind als Pfeile dargestellt.

### 3.2 Syntaktische Analyse

Aus dem so gebildeten Hypergraph-Modell des Diagramms wird nun vom Reduzierer-Modul ein "reduziertes Hypergraph-Modell" (rHGM) erzeugt. Dabei

<sup>1</sup>Eine ausführliche Darstellung dieser Konzepte und ihre formale Fundierung finden sich in [32].

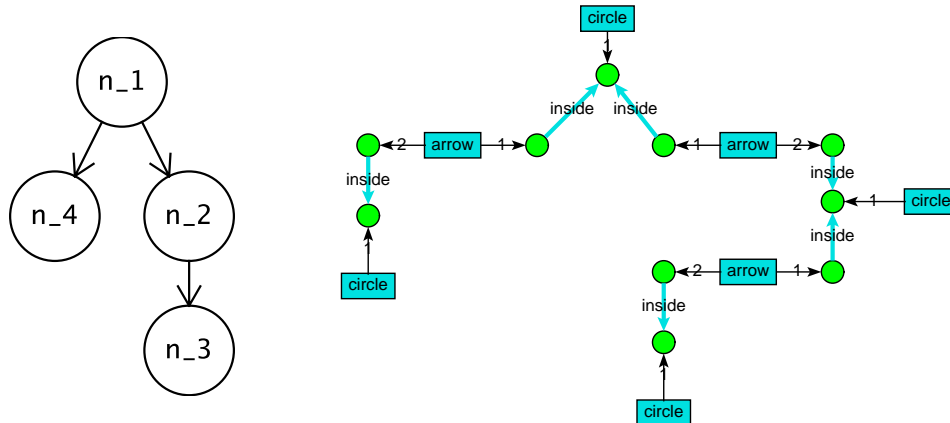


Abbildung 3.1: Ein Baumdiagramm und sein Hypergraph-Modell

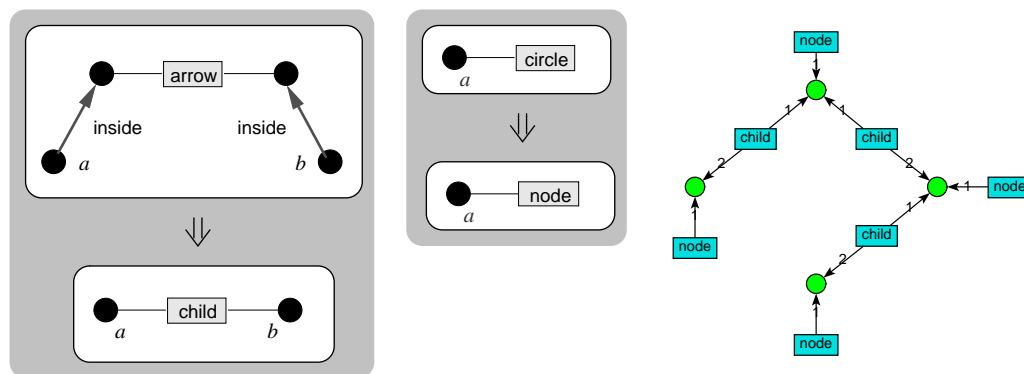
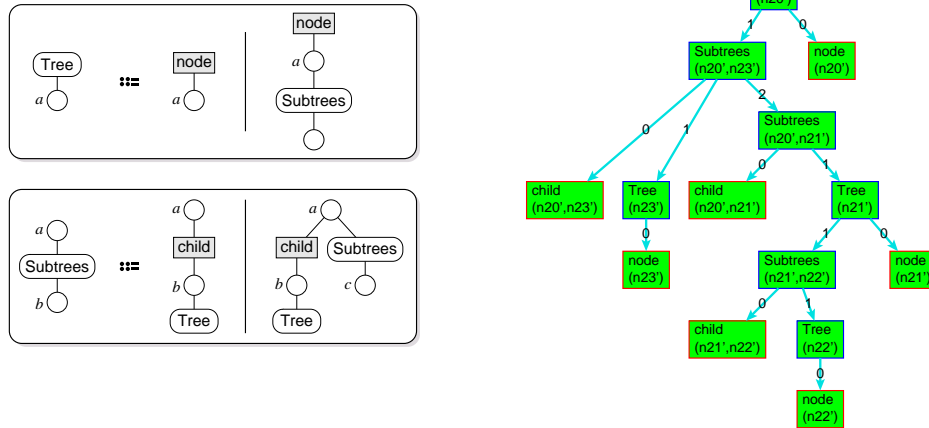


Abbildung 3.2: Reduzierer-Regeln für Baumdiagramme und ein reduziertes Hypergraph-Modell

wird eine Menge von Graphtransformationen („Reduzierer-Regeln“) auf jede mögliche Passung („Match“) des jeweiligen Regelmusters genau einmal angewandt. Reduzierer-Regeln können deshalb keine rekursiven Strukturen beschreiben; dafür dürfen ihre Muster aber beliebig komplex sein und verschiedene Arten von Pfadausdrücken, positiven und negativen Kontext und weitere Anwendungsbedingungen enthalten.

Bei Baumdiagrammen ist die Aufgabe des Reduzierers lediglich, einen Kantenpfeil zwischen zwei Knoten mit den entsprechenden Relationskanten zu einer Terminalkante der Hypergraph-Grammatik zu vereinigen. Ein Knoten (Kreis) wird direkt auf eine Terminalkante abgebildet. Abbildung 3.2 zeigt die Reduzierer-Regeln für die Spezifikation von Baumdiagrammen sowie das rHGM, das mit Hilfe dieser Regeln aus dem HGM von Abbildung 3.1 erzeugt wird. Die Regel besteht aus einem Muster im HGM und dem erzeugten Teilgraphen im rHGM; Knotenbezeichnungen *a*, *b* usw. markieren äquivalente Knoten im HGM und rHGM. Wir haben in dieser Arbeit immer eine grafische Darstellung für Graphtransformationen gewählt, obwohl DiaGen gegenwärtig nur eine textuelle



**Abbildung 3.3:** Grammatik für Baumdiagramme und ein Ableitungsbaum

Spezifikation unterstützt. Die grafische Form ist anschaulicher und soll, sobald keine größeren Änderungen an der verwendeten Sprache mehr vorgenommen werden müssen, auch für das DiaGen-System verwendet werden.

Die vom Reduzierer generierten Terminal-Hyperkanten (das rHGM) werden dann von einem Graphparser mit Hilfe der in der Spezifikation festgelegten Grammatik weiter analysiert. Die Produktionen dieser Grammatik beschreiben, wie sich die Terminal-Hyperkanten eines korrekten rHGM über nichtterminale Hyperkanten zum Startsymbol reduzieren lassen.<sup>2</sup> Im Gegensatz zu Reduzierer-Regeln werden Grammatik-Produktionen bei der Analyse solange wiederholt angewandt, wie sich neue Nichtterminale generieren lassen; dadurch können mit ihnen auch rekursive Strukturen beschrieben werden. Dafür sind die Produktionen in ihrer Mächtigkeit stärker eingeschränkt. Ursprünglich ließ DiaGen nur kontextfreie Grammatiken zu; dieser Formalismus genügt auch tatsächlich zur Analyse von Baumdiagrammen. Mit Hilfe der in Abbildung 3.3 gezeigten Graph-Grammatik analysiert der Parser das reduzierte Hypergraph-Modell, indem entsprechend den (inversen) Produktionen Teilbäume, die Kinder desselben Knotens sind, erst untereinander und dann mit dem Elternknoten zu neuen Nichtterminal-Hyperkanten zusammengefaßt werden. Primitive Teilbäume dieser Rekursion sind natürlich die Blattknoten. In Abbildung 3.3 ist auch der zum Beispieldiagramm aus Abbildung 3.1 und 3.2 gehörige Ableitungsbaum zu sehen.

<sup>2</sup>An dieser Stelle sei bereits darauf hingewiesen, dass sich mit Hilfe der Grammatik-Produktion aus dem Startsymbol im Allgemeinen auch Terminalmengen ableiten lassen, die kein korrektes rHGM repräsentieren. Unter Umständen lassen sich so nämlich auch Modelle erzeugen, welche der Reduzierer nie aus einer tatsächlichen HGM-Struktur ableiten könnte. Die in einer Spezifikation beschriebene Diagrammsprache ist dementsprechend nur die Menge aller Diagramme, die sich mit Hilfe der Reduzierer- und Parser-Produktionen korrekt analysieren lassen; die Menge der durch Umkehrung dieser Produktionen generierbaren Diagramme ist im Allgemeinen größer.

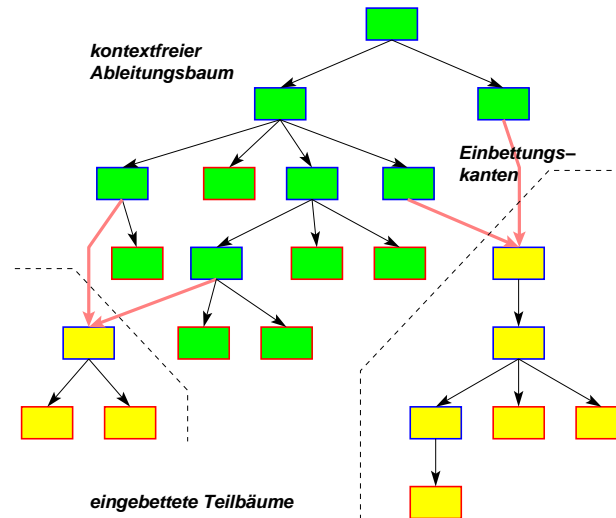


Abbildung 3.4: Beispiel für eine nicht kontextfreie Ableitungsstruktur

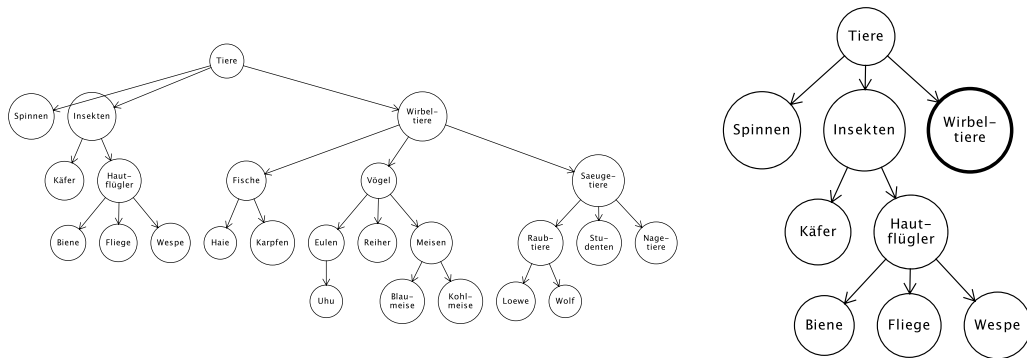
Viele Arten von Diagrammen, besonders alle graph-artigen Strukturen, lassen sich aber unter Beschränkung auf kontextfreie Grammatiken nicht beschreiben. Aus diesem Grund wurden in DiaGen "Einbettungsproduktionen" eingeführt, welche ein zusätzliches Nichtterminal in einen Kontext innerhalb des Hauptableitungsbaums einbetten. Diese Einbettung ist allerdings nicht rekursiv möglich, d. h. Elemente eines Ableitungsteilbaums, der aus einem eingebetteten Nichtterminal abgeleitet wurde, können ihrerseits nicht wieder als Einbettungskontext dienen. Die sich insgesamt ergebende Ableitungsstruktur (in dieser Arbeit als "Parse-DAG" bezeichnet), ist die eines (kontextfrei abgeleiteten) Hauptbaums, in den an einzelnen Stellen weitere Ableitungsteilbäume eingebettet sind; Abbildung 3.4 zeigt ein Beispiel für eine solche Struktur.<sup>3</sup>

Durch Auswertung von mit den Reduzierer- und Parser-Produktionen verknüpften Attributabhängigkeiten wird bei Baumdiagrammen schließlich eine interne Objekt-Baumstruktur erzeugt, welche die gefundene grafische Struktur des Diagramms widerspiegelt. Auf dieser Objektstruktur operiert ein einfacher Standard-Baumlayout-Algorithmus, welcher die Knotenpositionen anpaßt und so für ein regelmäßiges und gut erkennbares Diagrammlayout sorgt.

### 3.3 Abstraktion in Baumdiagrammen

Baumdiagramme bieten sich durch ihre inhärente Hierarchie zum Bearbeiten mit Abstraktionsoperationen an: Eine naheliegende Operation besteht darin, alle

<sup>3</sup>Man beachte, dass die "Knoten" des Parse-DAG eigentlich auch Hyperkanten sind, die nicht nur durch Ableitungsbeziehungen sondern auch durch Tentakel und Knoten miteinander verknüpft sind. Dieser Aspekt ist in der Abbildung nicht dargestellt.



**Abbildung 3.5:** Anwendung einer Abstraktionsoperation in einem Baumdiagramm

Nachkommen eines ausgewählten Knotens auszublenden, so dass dieser Knoten danach nicht nur sich selbst repräsentiert, sondern einen ganzen abhängigen Teilbaum. Abbildung 3.5 zeigt, wie die Auswirkung einer solchen Transformation aussehen kann; hier wurde der Teilbaum unter "Wirbeltiere" (rechts außen) abstrahiert.

Natürlich muss es ebenso möglich sein, einen solchen "abstrahierten" Knoten wieder zu expandieren um den darunterliegenden Teilbaum einzublenden. Für derartige Operationen, benötigen wir offensichtlich eine Möglichkeit, Diagrammkomponenten ein- und auszublenden. Außerdem sollten die Operationen die von der Diagrammanalyse erkannte Hierarchie-Information nutzen können. Diese Information ist zwar in den Datenstrukturen von Reduzierer und Parser vorhanden, war aber bislang für Diagrammtransformationen nicht zugänglich. Wie die Abstraktionstransformationen durch Anwendung neuer Graphtransformationsmuster und -operatoren realisiert werden, wird in Kapitel 5 näher erörtert.

Um erkennen zu können, welche Baumknoten ausgeblendete Teilbäume repräsentieren, ist es sinnvoll, diese Knoten auch visuell zu markieren. Dies lässt sich auf einfache Weise erreichen, da Diagrammtransformationen in DiaGen grundsätzlich beliebigen Java-Code aufrufen dürfen und dabei Zugriff auf die involvierten Hyperkanten des HGM und damit indirekt auf die beteiligten Komponenten haben. Die Operationen zum Abstrahieren und Expandieren eines Teilbaums setzen über diesen Mechanismus ein Flag in der Wurzel-Komponente, welche ihre visuelle Darstellung beeinflusst. Abstrahierte Knoten können so mit einer stärkeren Umrandung markiert werden (vgl. Abb. 3.5).

Die Layoutkorrektur für Baumdiagramme arbeitet sehr restriktiv: Die relative Position der Knoten wird aus der Baumstruktur deterministisch berechnet und kann von der Benutzerin nicht geändert werden, so dass es für jeden gezeichneten Baum nur genau ein mögliches Layout gibt. Im Zusammenhang mit Zoom-Transformationen erweist sich dies als vorteilhaft, da ein "Auseinanderdrängen" bzw. "Zusammenziehen" des Diagramms beim Ein- und Ausblenden von Teilbäumen automatisch erfolgt. Kapitel 6 geht noch genauer auf das Zusammenspiel von Layoutalgorithmen und Zoom-Transformationen ein.

## 3.4 UML-Diagramme

Während der Baumdiagramm-Editor so ein auf das Wesentliche reduziertes Beispiel für die Anwendung von Zoom-Transformationen liefert, sollte an einem zweiten Beispiel-Editor gezeigt werden, wie sich diese Ideen auf eine komplexe und praktisch relevante Diagrammsprache anwenden lassen. Für diese Zwecke bietet sich die Familie von Diagrammsprachen an, die zusammen die "Unified Modeling Language" (UML) bilden. Die UML stellt in ihren verschiedenen Diagrammtypen Notationen zur Beschreibung objektorientierter Programmstrukturen zur Verfügung, welche sich in den letzten Jahren im Bereich des objektorientierten Software-Engineering als Standard etabliert haben. Es stehen eine ganze Reihe von kommerziellen Entwicklungstools zur Verfügung, welche sich dieser Notation bedienen, sowohl zur Dokumentation ("Re-Engineering") als auch zur Konsistenzprüfung und schließlich zur automatischen Codegenerierung. Das Standardisierungskonsortium der "Object Management Group" (OMG) hat mit der offiziellen Referenzdokumentation der UML (derzeit in der Version 1.3, s. [12]) eine verbindliche Spezifikation erarbeitet. Neben der grafischen Notation wurde auch ein zugrundeliegendes abstraktes Klassenmodell für UML-Diagramme, das sogenannte UML-Metamodell definiert, welches als gemeinsamer Nenner für alle mit UML arbeitenden Tools und als Basis für den Datenaustausch zwischen diesen dienen soll.<sup>4</sup> Auf die sich daraus ergebenden Konsequenzen für die vorgestellte Implementierung gehen wir in Kapitel 4 noch näher ein.

Vor allem von akademischer Seite wurde immer wieder das Fehlen einer eindeutigen formalen Semantik für die Umsetzung von UML-Diagrammen in konkrete Programmstrukturen kritisiert. Für unsere Zwecke sind diese Diskussionen glücklicherweise von untergeordneter Bedeutung, da die existierenden Spezifikationen zur Erstellung von UML-Diagrammeditoren und zur Abbildung der Diagramme auf das UML-Metamodell ausreichend sind.

Ein gewisses Problem bei der Verwendung dieser Spezifikationen zur Definition eines Diagrammeditors in DiaGen besteht allerdings darin, dass die Dokumente die Abbildung eines internen Modells auf der Grundlage des UML-Metamodells in eine grafische Darstellung beschreiben. Dies ist auch die Vorgehensweise, welche typischen Modellierungstools zugrundeliegt, die nach dem Prinzip des syntaxgesteuerten Bearbeitens bedient werden. DiaGen beschreitet jedoch, wie in Abschnitt 1.2 beschrieben, den grundsätzlich entgegengesetzten Weg, ein internes Modell erst nachträglich aus der konkreten grafischen Darstellung des Diagramms zu generieren. Das bedeutet, dass diese umgekehrte Abbildung an manchen Stellen nicht unbedingt eindeutig ist (z. B. bei der Zuordnung von Beschriftungen zu manchen Elementen), und wir gezwungen sind, die Konflikte aufzulösen, die sich daraus ergeben.

Aus den verschiedenen Diagrammtypen, die zur UML gehören, haben wir

---

<sup>4</sup>Die Struktur und Funktion des UML-Metamodells wird in Abschnitt 4.1 näher erläutert.

für unseren Beispieleditor die UML-Klassendiagramme herausgegriffen.<sup>5</sup> Zum einen stellen Klassendiagramme den ursprünglichen Kern der UML-Notation (bzw. ihrer Vorgänger) dar und sind daher am weitesten verbreitet und bekannt. Außerdem bieten sie viel hierarchische Struktur und damit naheliegende Möglichkeiten zum Einsatz von Zoom-Transformationen. Der Beispieleditor implementiert nicht den vollen (enorm großen) Umfang der in der Spezifikation enthaltenen Notationsmöglichkeiten,<sup>6</sup> sondern nur eine Untermenge von "wesentlichen" Bestandteilen, die entweder häufig genutzt werden oder besondere Implementierungsprobleme aufwerfen. Ein vollständiger Editor sollte sich mit den dabei verwendeten Techniken realisieren lassen, ohne auf grundlegend neue Probleme zu stoßen.

UML bietet sich gerade deshalb für den Einsatz eines Editor-Generators wie DiaGen an, weil diese Notation so viele unterschiedliche Diagrammtypen und Varianten beinhaltet, die in ihrer grafischen Darstellung und Manipulation oft recht ähnlich sind. Wenn die zugrundeliegenden Semantik- und Layout-Konzepte bereits implementiert sind, ist eine Erweiterung des Editors um zusätzliche Notationsvarianten mit relativ wenig Aufwand möglich, da die Spezifikation einfach ergänzt werden kann. Aus demselben Grund erleichtert es die Verwendung eines Generators, auf Änderungen im Sprachstandard zu reagieren, da die Definition der UML sich gegenwärtig noch mitten in ihrer Entwicklung befindet.

### 3.5 Elemente von UML-Klassendiagrammen

Nach diesen Betrachtungen zu UML im Allgemeinen wollen wir uns nun der implementierten Diagrammsprache genauer zuwenden. In den späteren Erörterungen muss in manchen Details eine gewisse Vertrautheit mit UML-Klassendiagrammen vorausgesetzt werden; jedoch soll ihre grundlegende Struktur hier kurz eingeführt werden.<sup>7</sup> Abbildung 3.6 zeigt die typischen Bestandteile eines solchen Diagramms.

Wie man sieht, handelt es sich bei UML-Klassendiagrammen um eine graph-artige Diagrammsprache. Die Diagramme dienen zur Darstellung der statischen Klassenstruktur eines objektorientierten Programms: Die Knoten des Graphen sind Rechtecke, welche die Klassen des Objektmodells repräsentieren. Diese Rechtecke enthalten Informationen zu Details der Klasse, vor allem ihre Attribute (Member-Variablen) und Operationen (Methoden-Definitionen). Verschiedene Arten von Pfeilen repräsentieren die verschiedenartigen Beziehungen in denen diese Klassen zueinander stehen:

<sup>5</sup>Im Folgenden wird der Beispieleditor für UML-Klassendiagramme kurz als "UML-Editor" bezeichnet.

<sup>6</sup>tatsächlich ist uns z. Zt. auch kein kommerzielles Tool bekannt, das alle Notationsvarianten wie z. B. Mehrfachassoziationen oder die Notation von Diskriminatoren für Generalisierungen unterstützt. Selbst erfahrene Modellierer kennen bzw. verwenden im Allgemeinen nur einen Teil der möglichen Notationsvarianten.

<sup>7</sup>Zur genaueren Beschreibung und exakteren Definition der verwendeten Begriffe siehe [12] oder eines der vielen Bücher zur Einführung in UML.

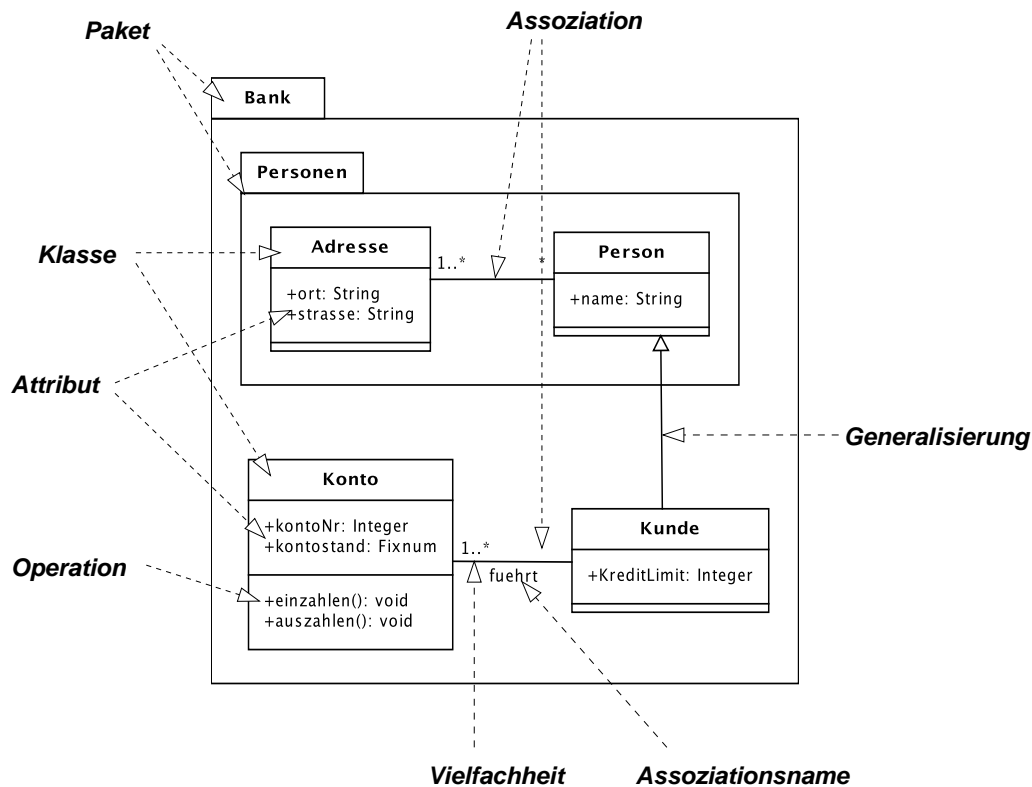


Abbildung 3.6: Bestandteile eines UML-Klassendiagramms

- Assoziationsverbindungen zeigen an, dass Instanzen der jeweiligen Klassen in irgendeiner Form aufeinander zugreifen können; als typische Realisierung enthält eine Instanz der einen Klasse eine Referenz auf eine oder mehrere Instanzen der anderen. Eine Assoziation kann auch als Schleife eine Klasse mit sich selbst verbinden; in diesem Fall besteht ein Zugriffsweg zwischen zwei oder mehr Instanzen derselben Klasse.
- Aggregations- und Kompositionsverbindungen stellen Spezialfälle von Assoziationsverbindungen dar und zeigen an, dass eine Instanz der Klasse am Ausgangsende (mit der Raute) eine oder mehrere Instanzen der Zielklasse als Bestandteile "enthält".
- Generalisierungspfeile repräsentieren Vererbungsbeziehungen zwischen Klassen. Die Klasse, von welcher der Pfeil ausgeht, stellt eine Spezialisierung der Zielklasse dar und erweitert sie um zusätzliche Elemente und Verknüpfungen.
- Abhängigkeitspfeile (gestrichelte Pfeile, in der Abbildung nicht dargestellt) repräsentieren beliebige Abhängigkeiten zwischen Klassen, welche sich nicht einer der anderen Kategorien zuordnen lassen.

Eine hierarchische Strukturierung der Klassen wird dadurch angezeigt, dass sie Bestandteile von Paketen (“Packages”) sind, welche ihrerseits wieder in Paketen höherer Ordnung enthalten sein können. Pakete können ebenfalls verschiedene Beziehungen untereinander (z. B. Abhängigkeiten) haben. Wir haben es also mit einem hierarchischen Graph-Konzept zu tun, bei dem Knoten andere Knoten enthalten können. Diese Hierarchie unterliegt kaum Einschränkungen, z. B. dürfen Kanten beliebig über Hierarchiegrenzen hinweg laufen und Pakete können gleichzeitig Klassen und andere Pakete enthalten.

Die meisten dieser Elemente der UML werden im UML-Editor direkt durch Diagrammkomponenten repräsentiert. Wir haben uns dabei für eine relativ “feine” Aufteilung entschieden; so sind beispielsweise Pfeilbeschriftungen eigene Komponenten, welche unabhängig von einem Pfeil erzeugt und manipuliert werden – in anderen UML-Werkzeugen werden dagegen Beschriftungen typischerweise durch Öffnen einer Dialog-Box zu einem Pfeil hinzugefügt. Durch die Repräsentation als eigene Komponenten bieten sich weitgehende Möglichkeiten zum Bearbeiten durch direkte Manipulation, beispielsweise können Beschriftungen einfach anderen Pfeilen zugeordnet werden, anstatt sie bei einem Pfeil zu löschen und beim anderen neu einzutragen. Natürlich müssen bei einem korrekten Diagramm alle Beschriftungen zu genau einem Pfeil gehören.

Wo immer es sinnvoll war, haben wir versucht, dieselbe Komponente in verschiedenen Rollen zu benutzen: Pfeilbeschriftungen können sowohl als Assoziationsnamen wie auch als Rollenbeschreibungen der beteiligten Klassen dienen; je nachdem, ob sie an einem Ende oder in der Mitte des Assoziationspfeils platziert werden. Stereotypen (UML-Elemente, die Untertypen von anderen Elementen bezeichnen) werden durch *eine* Komponente realisiert, die sowohl in Klassen- oder Paketraumen wie auch an Pfeilen auftreten darf. Schließlich wird nur eine Art von Pfeil-Komponenten eingesetzt, die wahlweise die Form eines Assoziations-, Generalisierungs-, oder Abhängigkeitspfeils annehmen kann.

### 3.6 Abstraktion in UML-Klassendiagrammen

In der beschriebenen Notation für UML-Klassendiagramme sind auch schon Möglichkeiten für Abstraktionsdarstellungen vorgesehen (s. Abb. 3.7):

- Zum einen muss eine Klasse nicht immer mit allen in ihr enthaltenen Detailinformationen dargestellt werden. Es ist so z. B. möglich, Attribute nur dann anzuzeigen, wenn ihr Sichtbarkeitsbereich als öffentlich (“public”) definiert wurde. Insbesondere können auch Segmente der Klassendarstellung komplett weggelassen werden. Für den UML-Editor wurde deshalb eine Abstraktionsoperation vorgesehen, welche eine Klasse vereinfacht, indem die Segmente für Attribute und Operationen mit ihrem Inhalt ausgeblendet werden.

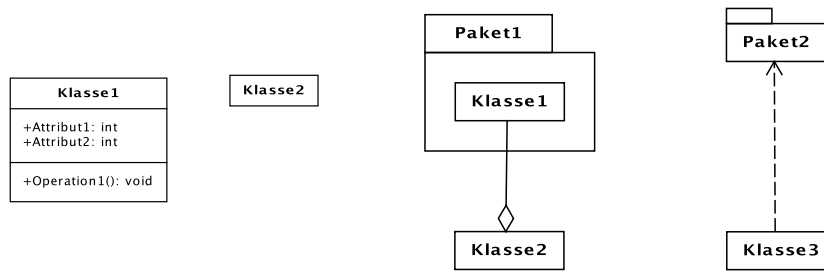


Abbildung 3.7: Abstraktion von Klassen und Paketen

- Auch für Pakete definiert der UML-Standard eine vereinfachte Notation. Dabei wird nur der Paketname<sup>8</sup> angezeigt, der Inhalt des Pakets bleibt unsichtbar. Der Name wandert dabei aus dem "Reiter" der Paketdarstellung in den Paketrumpf. Eine Besonderheit bei dieser Form der Diagrammabstraktion besteht darin, dass Beziehungen anderer Elemente, die von außen in das Paket hineingreifen, "umgelenkt" werden müssen. Die Beziehungspfeile müssen dabei auch ihren Typ ändern, da z. B. eine Assoziation einer Klasse c1 im Paket p mit einer anderen Klasse c2 außerhalb dieses Pakets nicht gleichbedeutend mit einer Assoziation von p zu c2 ist. (Tatsächlich dürfen Pakete entsprechend den Regeln für UML-Diagramme überhaupt keine Assoziationsverbindungen besitzen.) Stattdessen müssen Abhängigkeitspfeile als "Ersatzdarstellungen" verwendet werden. Wir sehen hier also ein typisches Beispiel für eine Diagrammabstraktion, die mit einem allgemeinen Mechanismus ohne Berücksichtigung der Diagrammsprache nicht realisiert werden kann.

Bei dieser Abstraktionsoperation können u. U. mehrere gleiche Ersatzpfeile zwischen denselben Elementen entstehen. Diese repräsentieren im Grunde dieselbe Information, nämlich dass eine Detail-Abhängigkeit besteht, die auf dem aktuellen Abstraktionsniveau nicht sichtbar ist. Deshalb wird außerdem noch eine "Nachbehandlung" des Diagramms durchgeführt, die solche gleichen Pfeile gegebenenfalls zusammenfasst. Auch dazu sind spezielle Diagrammtransformationen erforderlich.

Die Implementierung dieser beiden Zoom-Transformationen demonstriert auch zwei mögliche Alternativen für die visuelle Darstellung von abstrahierten Elementen. Abstrahierte Klassen werden wie die Knoten im Baum-Beispiel nur mit einem Flag markiert, welches bewirkt, dass die entsprechende Komponente anders visualisiert wird. Die Diagrammkomponente und die dazugehörige Hyperkante des HGM bleiben jedoch dieselben.

Im Gegensatz dazu unterscheiden sich "expandierte" und "gefaltete" Pakete hinsichtlich ihrer Behandlung auch durch die Diagrammanalyse erheblich stärker voneinander. Deshalb wurde hier ein eigener Komponenten-Typ für gefaltete

<sup>8</sup>und eventuell der Stereotyp des Pakets

Pakete definiert. Die Abstraktions- und Expansionsoperationen für Pakete entfernen dann die gewählte Komponente und ersetzen sie durch eine des jeweils anderen Typs. Die Auswirkungen dieses Ersetzungsvorgangs auf das HGM werden in Kapitel 5 näher betrachtet.

## Kapitel 4

# Analyse von UML-Diagrammen

Wie bereits angesprochen, existiert zwar (noch?) keine formale Semantik für die UML, aber der Sprachstandard spezifiziert eine Art "abstrakte Syntax", welche zur programminternen Darstellung von Diagrammen und zum Datenaustausch zwischen verschiedenen Tools genutzt werden soll. Diese interne Darstellung erfolgt wiederum in Form eines Objektmodells, dessen Struktur durch das *UML-Metamodell* definiert wird.

### 4.1 Die Metamodell-Hierarchie der UML

Insgesamt gehen die Dokumente der OMG von einer vierstufigen objektorientierten Modell-Hierarchie aus. Dabei besteht jede Ebene aus Instanzen von Objekten, deren Klassenstruktur durch die nächsthöhere Ebene beschrieben wird:

<i>Ebene</i>	<i>Element</i>	<i>System</i>
Meta-Metamodell	Klasse	Meta-Object Facility
Metamodell	Klasse	UML-Metamodell
Modell	Kunde	UML-Klassendiagramm
Instanz	Meier42	konkretes System

Auf der untersten Ebene (Instanzebene) finden sich Instanzen von Objekten, wie sie in einem speziellen Programmkontext vorkommen, beispielsweise das Objekt "Meier42", welches einen konkreten Kunden in einer Bank-Transaktions-Anwendung darstellen könnte. Die zweite Ebene (Modellebene) enthält die Klassen, welche die Struktur der Instanzebene definieren. Beispielsweise ist das Objekt "Meier42" eine Instanz der Klasse "Kunde" mit Attributen wie "Name", "Adresse" usw. Auf dieser Ebene sind UML-Klassendiagramme angesiedelt, wie sie normalerweise in der Software-Entwicklung verwendet werden und auf ihr bewegen sich typischerweise Anwendungsprogrammiererinnen.

Die Elemente der zweiten Ebene (Klassen, Attribute, Assoziationen) lassen sich nun selbst wieder als (Meta-)Objekte auffassen, so dass die Klassenstruktur eines solchen Objektmodells ihrerseits auf der nächsthöheren Ebene (Metamodell-Ebene, M2-Ebene) beschrieben werden kann. Auf dieser Ebene können wir z. B. Elemente des Bank-Transaktions-Klassenmodells "Kunde" und "Name" als Instanzen der "Meta"-Klassen "Klasse" bzw. "Attribut" auffassen. Während also Modelle der zweiten Ebene Klassenstrukturen für spezifische Anwendungsbereiche beschreiben, definieren Metamodelle die Regeln, denen solche Modelle unterworfen sind, und gelten damit übergreifend für beliebige Anwendungen. Auf der Metamodell-Ebene bewegen sich normalerweise keine Anwendungsprogrammiererinnen sondern nur Entwicklerinnen von Tools zur Software-Entwicklung. Das einzige Metamodell, welches in der vorliegenden Arbeit betrachtet wird, ist das UML-Metamodell.

Auch Metamodelle wie das UML-Metamodell werden in ihrer Struktur auf der Basis einer höheren Metamodell-Ebene (M3-Ebene) beschrieben. Während auf der M2-Ebene eventuell eine Notwendigkeit für grundsätzlich verschiedene Ausdrucksmöglichkeiten und Objekt-Konzepte und damit verschiedene Metamodelle bestehen könnte,<sup>1</sup> existiert auf der M3-Ebene nur noch ein einziges Modell, welches durch die Meta Object Facility (MoF) [13] der OMG definiert ist. Da man trotzdem auch Metamodell-Elemente wieder als Instanzen eines darüber liegenden Klassenmodells ansehen kann, lässt sich diese Schichtung von Modellierungsebenen natürlich beliebig weiter treiben. Die OMG hat dieses Definitionsproblem dadurch gelöst, dass die Struktur der vierten Ebene rekursiv definiert wird, d.h. das MoF-Objektmodell ist eine Instanz seiner selbst.

Da die Beschreibungselemente des MoF-Modells außerdem zu einer Teilmenge des UML-Klassenmodells äquivalent sind, könnte man im Grunde auch bereits das UML-Metamodell durch sich selbst beschreiben; allerdings kann man durch die geschilderte Struktur mit vier Ebenen auch zur UML alternative Metamodelle einbeziehen. Die sich daraus ergebenden praktischen Anwendungsmöglichkeiten für Metamodell-Repositories und Online-Datenaustausch über Metamodell-Grenzen hinweg, welche in [13] als Motivation angeführt werden, kann man aber wohl als eher esoterisch ansehen. Selbst auf UML-Basis ist eine Unterstützung für tool-übergreifende Repositories nicht in Sicht. Immerhin setzt sich aber allmählich ein aus dem UML-Metamodell abgeleitetes Dateiformat (XMI) als Möglichkeit zum Offline-Austausch von abstrakten Diagrammbeschreibungen zwischen verschiedenen UML-Tools durch. Mit den im Folgenden beschriebenen Techniken kann der UML-Editor tatsächlich solche XMI-Dateien erzeugen, so dass die bearbeiteten Diagrammen in andere UML-Tools exportiert werden können.

---

<sup>1</sup>obwohl das UML-Metamodell allein bereits sehr flexibel und ausdrucksstark ist und außerdem Erweiterungsmöglichkeiten wie Stereotypen vorsieht

## 4.2 Objektmodelle und Hypergraphen

Auf jeden Fall möchten wir die Analyse-Funktionalität von DiaGen so benutzen, dass aus den im UML-Editor bearbeiteten Diagrammen ein abstraktes Modell generiert wird, welches in seiner Struktur dem UML-Metamodell möglichst ähnlich ist. Die entscheidende Idee für die Kombination der DiaGen-Analyse mit dem UML-Metamodell besteht darin, dass die intern von DiaGen verwendeten Hypergraph-Strukturen viele Aspekte mit Objekt-“Graphen”(!) gemeinsam haben, wie sie als Datenstrukturen von objektorientierten Programmen verwendet werden: Wir haben es in beiden Fällen mit einer Menge von typisierten Elementen zu tun, welche mit typspezifischen Attributen versehen und untereinander verknüpft sind.<sup>2</sup> Da nun DiaGen viele Fähigkeiten zur Analyse und Manipulation solcher Hypergraph-Strukturen bietet, liegt es nahe, eine (teilweise) Abbildung des MoF-Objektmodells auf DiaGen-Hypergraphen zu definieren, um das (mittels der MoF spezifizierte) UML-Metamodell in ein abstraktes Hypergraph-Modell überführen zu können (nicht zu verwechseln mit dem HGM eines Diagramms, das seine konkrete Syntax repräsentiert).<sup>3</sup> Die Analyse-Komponente von DiaGen kann dann mittels einer geeigneten Spezifikation abstrakte Syntaxgraphen (ASGn) erzeugen, die diesem Hypergraph-Modell genügen und sich wieder in zum UML-Metamodell konforme Objektstrukturen zurücktransformieren lassen. Die folgende Tabelle zeigt, wie die gewünschte Abbildung aussieht:

<i>Objektmodell</i>	<i>abstraktes Hypergraph-Modell</i>
Objekt	Hyperkante
Klasse des Objekts	Typ (Markierung) der Hyperkante
Attribut des Objekts	Attribut der Hyperkante
Anzahl der mit der Klasse verbundenen Assoziationsenden	Stelligkeit der Hyperkante
(mögliche) Assoziationen des Objekts	Tentakel der Hyperkante
Assoziation zwischen Objekten	Entsprechende Tentakel der Hyperkanten führen zum selben Knoten

<sup>2</sup>Im Unterschied zu typischen bekannten Graph-Konzepten stehen bei DiaGen-Hypergraphen nicht die Knoten sondern die Hyperkanten als typisierte und attributierte Elemente im Vordergrund, während die Knoten als anonyme Verbindungselemente dienen. Dieses Graph-Modell wurde gewählt, da es bessere Voraussetzungen für den Einsatz effizienter Graph-Parser bietet, worauf aber in dieser Arbeit nicht näher eingegangen werden kann.

<sup>3</sup>Heckel und Engels entwickeln in [24] weitere Ideen zu den Möglichkeiten und Vorteilen, welche Graphen als Modell für die Syntax und Semantik von visuellen Sprachen bieten.

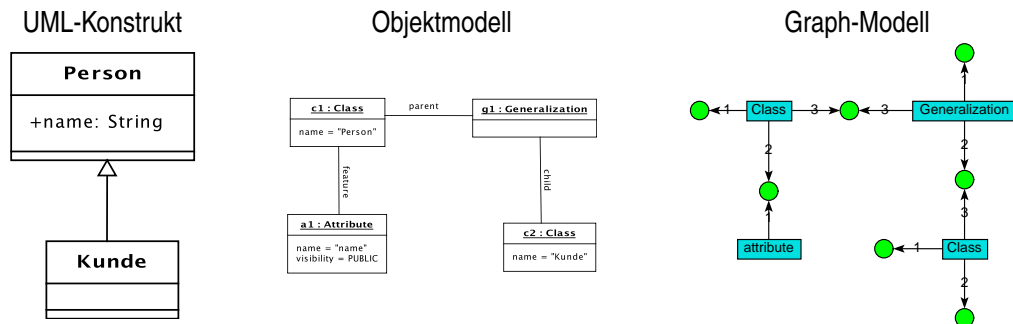


Abbildung 4.1: Metamodell und abstrakter Syntaxgraph eines Diagrammausschnitts

Abbildung 4.1 zeigt als Beispiel einen Ausschnitt des UML-Diagramms aus Abbildung 3.6, das dazugehörige (Meta-)Objektmodell<sup>4</sup> und den entsprechenden ASG, wie er vom UML-Editor aus dem Diagramm erzeugt wird. Aus einem solchen ASG kann dann auch mit wenig Aufwand eine XMI-Datei zum Export des Diagramms in andere UML-Tools erzeugt werden.

Da Assoziationen von Klassen, welche im MoF-Objektmodell grundsätzlich zwei Enden haben,<sup>5</sup> in der Hypergraph-Repräsentation auf einen einzigen Knoten abgebildet werden, scheint hier auf den ersten Blick ein irreversibler Informationsverlust vorzuliegen. Dies ist aber tatsächlich nicht der Fall, da wir bei der Betrachtung jeder über eine Tentakel mit einem solchen Knoten verbundenen Hyperkante feststellen können, zu welcher Seite der Assoziation das entsprechende Objekt zugeordnet werden muss:

- Normalerweise ergibt sich die Zuordnung der Hyperkante aus ihrem Typ (der Kantenmarkierung). Wenn wir beispielsweise in Abbildung 4.1 den Knoten betrachten, der die Assoziation "feature" repräsentiert, dann verbindet dieser eine einer Klasse am einen Ende der Assoziation mit einem oder mehreren Attributen und Operationen am anderen Ende; jede verbundene Hyperkante muss einen dieser Typen besitzen und kann so zugeordnet werden.
- Unter Umständen erlaubt der Typ der Hyperkante keine Zuordnung zu einem Ende der Assoziation. Dieser Fall tritt bei Assoziationen auf, die als Schleife eine Klasse mit sich selbst verbinden und er kann auch dann eintreten, wenn die Assoziation eine Klasse mit einer ihrer Oberklassen verbindet, da die konkrete Hyperkante, welche die "Instanziierung" der Oberklasse darstellt, eventuell auch zum Typ der Unterklasse konform ist. In diesen beiden Fällen lässt sich die Zuordnung aber daraus ableiten, an welchem Tentakel die Hyperkante mit dem Assoziationsknoten verbunden ist,

<sup>4</sup>Dabei handelt es sich um ein Objektdiagramm, das Objekt-Instanzen und konkrete Ausprägungen von Assoziationen zeigt.

<sup>5</sup>Assoziationen zwischen mehr als zwei Klassen sind zwar im Objektmodell der UML, nicht aber in dem der MoF zulässig.

da ja jedes mit einer Klasse verbundene Ende einer Assoziation auf genau einen Tentakel der entsprechenden Hyperkante abgebildet wird.

Als Konsequenz aus diesen Überlegungen sehen wir, dass sich Hypergraphen, welche die durch unsere Abbildung definierte Struktur aufweisen, mit Hilfe von Mustervergleichen genauso traversieren lassen, wie man in Objektstrukturen navigieren kann, welche einem entsprechenden Klassenmodell genügen. Wenn sich ein Objekt von einem anderen durch Verfolgen von Referenzen erreichen lässt, dann existiert auch ein äquivalenter Pfad im entsprechenden Hypergraphen.

Dagegen ließe sich eine Assoziation übrigens nicht als eine einzelne Hyperkante darstellen, da eine solche immer nur eine feste Anzahl von Objekten (Knoten) verbinden kann, während binäre Assoziationen auch 1-zu-n oder m-zu-n-Beziehungen repräsentieren können. In einem derartigen Graphmodell müsste eine solche Assoziation in eine Menge von Hyperkanten übersetzt werden; genauso reichen bei der programmiersprachlichen Implementierung derartiger Assoziationen einfache Objektreferenzen nicht aus, sondern es müssen Container-Objekte zu Hilfe genommen werden.

### 4.3 Generierung von abstrakten Syntaxgraphen

Nachdem wir die Struktur der ASGn definiert haben, die der UML-Editor erzeugen soll, wenden wir uns nun der Frage zu, wie sich die Möglichkeiten, die DiaGen zur Graphmanipulation bietet, am besten nutzen lassen, um solche Graphen aus den bearbeiteten Diagrammen abzuleiten. Eine naheliegende Idee wäre es, sie mit Hilfe der Reduzierer-Regeln von DiaGen zu erzeugen, so dass der abstrakte Syntaxgraph eines Diagramms dem rHGM oder einer Teilmenge desselben entspräche. Da mit diesen Graphen aber rekursive Strukturen wie z. B. die Pakete-Schachtelung des Diagramms beschrieben werden müssen, reicht die Mächtigkeit der Reduzierer-Regeln dazu nicht aus (vgl. Abschnitt 3.2). Dagegen hat sich eine Kombination aus Produktionen des Reduzierers und der Grammatik für unsere Zwecke als ausreichend leistungsfähig erwiesen. Die Hyperkanten des ASG sind somit eine Teilmenge der Nichtterminal-Hyperkanten, welche der Parser aus dem HGM des Diagramms erzeugt.

Für das in Abbildung 4.1 angeführte Beispiel zeigt Abbildung 4.2 den entsprechenden (vereinfachten) Parse-DAG; die markierten Nichtterminale sind Bestandteile des ASG. Sinnvollerweise sollte man zwei zusätzliche Forderungen an die Struktur dieser Teilmenge des Parse-DAG stellen:<sup>6</sup>

<sup>6</sup>Man könnte hier weitere Untersuchungen anstellen, ob sich diese Eigenschaften statisch, d.h. vor der Generierung eines Editors, durch Vergleich der Grammatik und des Klassenmodells der abstrakten Diagrammstruktur (des Metamodells) überprüfen ließen. In Verbindung mit dynamischen Anwendbarkeitsbedingungen für Grammatik-Produktionen dürfte sich dies allerdings schwierig gestalten.

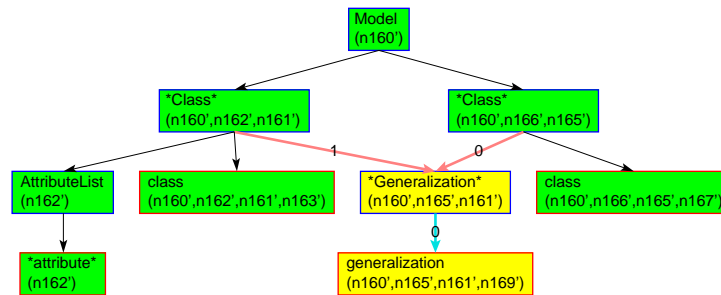


Abbildung 4.2: Generierung des ASG aus dem Parse-DAG

- Die Objekthierarchie des Objektmodells sollte sich in der Ableitungshierarchie des Parse-DAGs widerspiegeln. Das bedeutet, dass eine Kompositionsbeziehung zwischen zwei Objekten des abstrakten Objektmodells auch zu einer (evtl. indirekten) kontextfreien Ableitungsbeziehung zwischen den entsprechenden ASG-Hyperkanten führt, wobei die übergeordnete Hyperkante in der Parse-Struktur dem “Besitzerobjekt” entspricht und die abgeleitete Hyperkante dem Teilobjekt.
- Der ASG sollte tatsächlich eine Abstraktion des konkreten Diagramms darstellen, und darum sollte jede Hyperkante des HGM, die bedeutungstragend ist und als korrekt erkannt wurde, auch im ASG repräsentiert sein. Das bedeutet, dass auf jedem Pfad, der im Parse-DAG eines korrekten Diagramms vom Startsymbol zu einem Terminalsymbol führt, (mindestens) ein Nichtterminal liegt, welches Bestandteil des ASG ist.

Als Konsequenz daraus kann aus jedem korrekten ASG durch Anwendung der Grammatik-Produktionen und der umgekehrten Reduzierer-Regeln prinzipiell ein (mögliches) HGM abgeleitet werden; aus diesem ließe sich mit einem geeigneten Layout-Algorithmus dann wieder ein Diagramm generieren.<sup>7</sup> Natürlich wäre diese Generierung eines Diagramms aus einer abstrakten Beschreibung nicht eindeutig, aber da alle erzeugbaren Diagramme demselben ASG entsprechen, sind sie per definitionem äquivalent. Hier liegen noch interessante Anwendungsmöglichkeiten für die Zukunft; die Behandlung des damit verbundenen “Unparsing-Problems” für die DiaGen-Analyse erscheint allerdings relativ aufwändig, daher haben wir uns bislang mit dieser Fragestellung noch nicht weiter beschäftigt.

Aufgrund unserer Erfahrungen mit der Spezifikation des UML-Editors glauben wir, dass die vorgestellten Überlegungen zur Beschreibung und Generierung von abstrakten Syntaxgraphen auch eine allgemeinere Anwendung finden können;

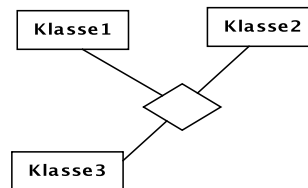
<sup>7</sup>Außerdem muss sich auch der ASG seinerseits aus dem Startsymbol ableiten lassen; d.h. der DiaGen-Parser könnte zu einer eingeschränkten Korrektheitsprüfung für ASGn genutzt werden. Da allerdings die Ableitbarkeit eines Terminalgraphen aus dem Startsymbol alleine noch nicht seine Korrektheit garantiert (vgl. Abschnitt 3.2), gilt dies auch für einen ASG.

dies muss aber erst durch weitere Beispiele belegt werden. Wir haben es daher zunächst auch unterlassen, die vorgestellten Konzepte exakt zu formalisieren, da für andere Anwendungen noch Modifikationen oder Verallgemeinerungen notwendig sein könnten.

## 4.4 Probleme mit abstrakten Syntaxgraphen

Die beschriebene enge Koppelung von Diagrammanalyse und Generierung des ASG in Verbindung mit unseren Konsistenz-Forderungen stellt natürlich eine wesentliche Einschränkung beim Spezifizieren der Diagrammanalyse da. Das Metamodell des Diagramms und damit die Syntax des ASG sind ja extern vorgegeben (durch die UML-Referenz [12]) und müssen bei der Formulierung der Grammatik berücksichtigt werden. Das kann zu Problemen führen, z. B. wenn stark voneinander abweichende Notationsvarianten für dasselbe abstrakte Konzept vorgesehen sind, da dann unterschiedliche HGM-Strukturen in dieselben Nichtterminale überführt werden müssen. Bei der Spezifikation des UML-Editors erwiesen sich die Konstrukte, die DiaGen zu Diagrammanalyse erlaubt, aber immer als ausreichend leistungsfähig und flexibel, um solche Probleme zu lösen.

Beispielsweise enthält die UML eine Notation für Assoziationen zwischen mehr als zwei Klassen, die sich visuell stark von der Darstellung binärer Assoziationen als einfachem Strich unterscheidet. Beide Notationen werden aber auf dasselbe Metamodell-Konstrukt abgebildet, welches die Assoziation und beliebig viele "Assoziationsenden" als separate Objekte repräsentiert. Entsprechend ist auch die Grammatik für UML-Diagramme so aufgebaut, dass beide Notationen auf höherer Ebene in dieselben Analyse-Konstrukte (Nichtterminale) überführt werden.



**Abbildung 4.3:** UML-Notation für eine Assoziation zwischen drei Klassen

Ein grundsätzliches Problem bei der Abbildung von Objektmodellen auf ASGn und deren Generierung aus dem HGM eines Diagramms besteht darin, dass sich die Produktionen der DiaGen-Analyse nur schlecht dazu eignen, neue Knoten zu "erzeugen": Die Knoten des HGM entsprechen den Konnektoren der Komponente (vgl. Abschnitt 3.1) und werden durch Reduzierer-Regeln direkt auf rHGM-Knoten abgebildet.<sup>8</sup> Zwar dürfen Reduzierer-Regeln auch neue Knoten erzeugen; da sie jedoch parallel unabhängig sein müssen, können solche Knoten aber nur innerhalb einer Regel referenziert werden, was ihre Verwendbarkeit enorm einschränkt. Grammatik-Produktionen schließlich können nur Knoten aus Hyperkanten ihrer rechten Seite in die (vom Parser erzeugten) Hyperkanten der linken Seite übernehmen; außerdem "verbrauchen" sie die Hyperkanten der rechten Seite, da diese ja nicht mehrfach im Parse-DAG vorkommen dürfen, daher

<sup>8</sup>wobei das Verschmelzen mehrerer HGM-Knoten zu einem rHGM-Knoten möglich ist

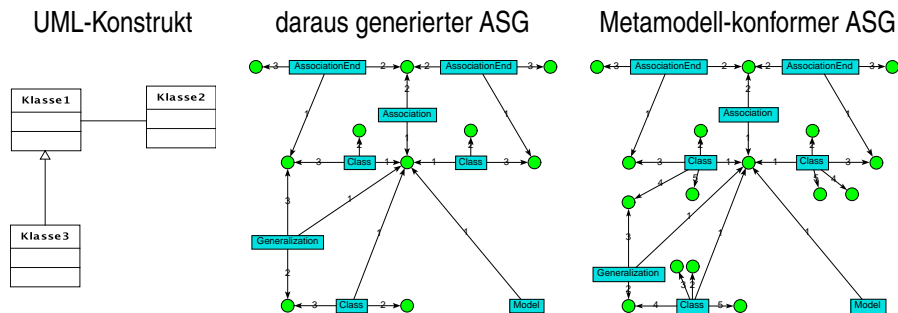


Abbildung 4.4: Abweichen des ASG vom UML-Metamodell

eignen sie sich nicht, um grundsätzliche strukturelle Modifikationen des Hypergraph-Modells zu formulieren.

Wenn wir jedoch, wie in Abschnitt 4.2 beschrieben, Assoziationen des Objektmodells auf Knoten des ASG abbilden wollen, kann es erforderlich sein, diese Knoten bei der Generierung des ASG zu erzeugen, da im HGM keine entsprechenden Knoten existieren. Die einfachste Möglichkeit, mit diesem Problem umzugehen, besteht in der Modifikation der ASG-Struktur. Im Fall des UML-Editors sieht z. B. das HGM so aus, dass eine "ClassFrame"-Komponente einen Konnektor besitzt, über den sie mit Pfeilen in Verbindung stehen kann (vgl. Abschnitt 3.5). Diese Pfeile können aber verschiedene Elemente im Metamodell repräsentieren – nämlich Assoziationen, Generalisierungen und (sonstige) Abhängigkeiten – und diese sind im Metamodell über verschiedene Assoziationen mit der Meta-Klasse "Klasse" verbunden.

Aufgrund der HGM-Struktur ist die Analyse einfacher, wenn man diese verschiedenen Assoziationen in einem Knoten zusammenfasst, weshalb die ASG-Struktur des UML-Editors an dieser Stelle vom UML-Metamodell abweicht. Die unterschiedlichen Typen (Labels) der mit solchen Knoten verbundenen Hyperkanten macht es im beschriebenen Fall weiterhin möglich, jede Hyperkante der richtigen Assoziation zuzuordnen.<sup>9</sup>

Abbildung 4.4 zeigt ein Beispiel für dieses Abweichen. Man beachte, dass der Hyperkanten-Typ "Class" im Metamodell-konformen ASG zwei Tentakel mehr besitzt und an unterschiedlichen Tentakeln mit den Hyperkanten für Assoziationsenden und Generalisierungen verbunden ist.

Alternativ kann man auch redundante Konnektoren für Komponenten einführen; im beschriebenen Fall sähe das so aus, dass z. B. die "ClassFrame"-Komponente nicht nur einen Konnektor besitzt, der mit allen Arten von Pfeilen in Verbindung steht, sondern mehrere (mit identischen zugeordneten Bereichen). Dadurch steht für jede Art von Pfeil ein eigener Konnektor zur Verfügung und jede Assoziation einer ASG-Hyperkante vom Typ "Class" kann auf einen eigenen Knoten abgebildet werden. Ein solches Vorgehen würde aber bedeuten, dass die

<sup>9</sup>z. B. beim in Abschnitt 18 erwähnten XMI-Export

Struktur der Analysekonstrukte diejenige der zugrundeliegenden Diagrammkomponenten bestimmen muss, und eine derartige Abhängigkeit möchten wir grundsätzlich vermeiden. Eine befriedigende Lösung bietet sich erst durch die Verwendung eines "mehrstufigen Reduzierers", wie er in Abschnitt 5.9 beschrieben wird.

Falls sich derartige Probleme nicht lösen lassen sollten, böte sich noch der Ausweg, den ASG nicht direkt als Teilgraphen der Parse-Struktur zu generieren, sondern bei der Diagrammanalyse zunächst nur eine geeignete "Zwischendarstellung" zu erzeugen, welche dem ASG zwar ähnlich ist, aber an den notwendigen Punkten von dessen (extern vorgegebener) Struktur abweicht. Die erforderlichen letzten Transformationen müssten dann mit einem leistungsfähigeren Mechanismus explizit programmiert werden. Ein solcher weiterer Transformationsschritt würde aber die Spezifikation einer Diagrammsprache weiter komplizieren und sollte daher nur eingeführt werden, wenn dies wirklich nicht zu vermeiden ist.

Wie schon gesagt glauben wir, dass die vorgestellte Technik zur Abbildung von objektorientierten Modellen auf Hypergraphen sich auch auf andere Diagrammtypen übertragen ließe, bei denen ein Diagrammeditor eine "semantische Repräsentation" erzeugen soll, welche als Klassenmodell beschrieben werden kann. Da die objektorientierte Modellierung prinzipiell zur Repräsentation beliebiger Anwendungsbereiche verwendet werden kann, bietet sich hier ein weites Feld für Experimente. Allerdings wäre es wohl für viele Einsatzzwecke nötig, die generierte Graphstruktur tatsächlich in einen Objektgraphen der Anwendungssprache (Java) überführen zu können. Eine allgemeine Programmkomponente für diesen Zweck ließe sich möglicherweise unter Verwendung des Java-Reflection-Mechanismus realisieren, da dieser zur Laufzeit die Instantiierung von Klassen erlaubt, die beim Compilieren des Codes noch nicht bekannt sind.

Für den in dieser Arbeit entwickelten UML-Editor ist eine solche Funktionalität aber nicht erforderlich, da die generierten abstrakten Syntaxbeschreibungen (UML-Modelle) nicht direkt von anderen Programmkomponenten verarbeitet werden. Wir haben daher lediglich, wie bereits erwähnt, eine Umwandlung in das standardisierte XMI-Dateiformat zum Off-Line-Datenaustausch von UML-Modellen realisiert.



## Kapitel 5

# Graphtransformation

In Kapitel 2 haben wir vorgeschlagen, die Idee einer semantischen Fokus-und-Kontext-Darstellung in DiaGen mit Hilfe von Zoom-Transformationen zu realisieren. Kapitel 3 hat gezeigt, wie solche Zoom-Transformationen für zwei beispielhafte Diagrammsprachen aussehen können und welchen Effekt sie haben sollen. Wir wenden uns nun der Frage zu, wie Zoom-Transformationen in der Spezifikation einer Diagrammsprache beschrieben werden können und wie die gewünschten Effekte im Programm realisiert werden.

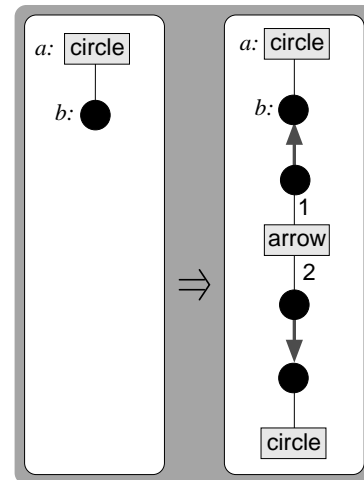
Zoom-Transformationen werden in DiaGen grundsätzlich genauso spezifiziert und ausgeführt wie andere Diagrammtransformationen auch. Sie stellen nur eine Untergruppe dieser Transformationen dar, die durch die besondere Art ihrer Effekte charakterisiert wird, und sie benutzen dazu spezielle Sprachmittel, die im Folgenden beschrieben werden. Als Voraussetzung dafür müssen wir aber kurz erläutern, wie die Spezifikation und Ausführung von Diagrammtransformationen in DiaGen erfolgt.

### 5.1 Beschreibung von Diagrammtransformationen in DiaGen

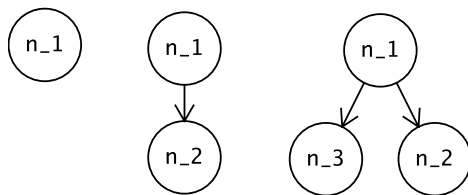
Diagramm-Transformationen werden in DiaGen-Spezifikationen als Graphtransformationen auf dem Hypergraph-Modell des Diagramms (vgl. S. 26) formuliert. Basis dieser Transformationen sind einzelne Transformationsregeln: Eine solche Regel beschreibt ein Muster im HGM und einen Ersetzungsgraphen, der durch Hinzufügen und Entfernen von Hyperkanten aus dem Muster entsteht; bei ihrer Anwendung wird eine Passung ("Match") des Musters im HGM gesucht und entsprechend modifiziert. Werden dabei Komponenten-Hyperkanten erzeugt oder gelöscht, so geschieht dies auch mit den entsprechenden Komponenten in der grafischen Darstellung.

Abbildung 5.1 zeigt eine Transformationsregel für den Baumdiagramm-Editor, die zu einem vorgegebenen Baumknoten einen neuen Kindknoten hinzufügt. Beim Erzeugen von grafischen Komponenten ist es zusätzlich noch notwendig, diese durch Aufruf von zusätzlichem (handgeschriebenen) Java-Code mit geeigneten Initialisierungswerten (Konstruktor-Parametern) zu versorgen. Durch derartige Methodenaufrufe können Transformationsregeln auch beliebige sonstige Effekte auslösen.

Solche "primitiven" Transformationsregeln können auch als Bausteine bei der Beschreibung komplexerer Transformationen dienen: Statt die Modifikation des Musters direkt zu beschreiben, können "zusammengesetzte" Regeln auch die Anwendung anderer Regeln veranlassen. Dabei wird normalerweise ein Teil der Passung für das Muster der untergeordneten Regel von der aufrufenden Regel vorgegeben und, falls nötig, durch Mustervergleich ergänzt; auf diese Weise erfolgt eine Art Parameterübergabe wie beim Aufruf von Funktionen in klassischen Programmiersprachen. Natürlich kann die Anwendung einer Regel auch scheitern, wenn keine geeignete Passung gefunden werden kann. Die Ausführung einer Diagrammtransformation entspricht der Anwendung ihrer "Haupt"-Regel und muss von der Benutzerin durch das GUI explizit veranlasst werden. Dabei kann die Benutzerin Komponenten des Diagramms auswählen, die als Vorgabe für die Passung an die Anwendung der Hauptregel übergeben werden.



**Abbildung 5.1:** Transformation zum Einfügen eines Baumknotens



**Abbildung 5.2:** Wiederholtes Einfügen eines Baumknotens

möglichen Passungen anzuwenden. Gegenwärtig lässt das Transformationsmodul von DiaGen kein Backtracking beim Scheitern von Regelanwendungen zu, da sich dies bislang nicht als notwendig erwiesen hat. In [32] wird die Spezifikation und die Ausführung von Diagrammtransformationen genauer erörtert und auf eine formale Grundlage gestellt.

Diagrammtransformationen brauchen sich grundsätzlich nicht um die Anpassung der Analysestrukturen zu kümmern: Nach der Ausführung einer Transformation erfolgt immer eine vollständige Analyse des Diagramms und darauf folgend eine Layoutkorrektur. Über diese Layoutkorrektur können Transformationen auch indirekt die visuelle Erscheinung des Diagramms beeinflussen, indem

Der Kontrollfluss beim Abarbeiten von zusammengesetzten Regeln kann durch verschiedene Operatoren gesteuert werden. Vor allem kann abhängig vom Erfolg oder Scheitern einer Regelanwendung in unterschiedliche Ausführungspfade verzweigt werden, und es besteht die Möglichkeit, eine Regel der Reihe nach auf alle

sie z. B. Relationen-Hyperkanten einfügen oder entfernen und so entsprechende Layoutanpassungen herbeiführen. Abbildung 5.2 zeigt die zweimalige Anwendung der Transformation aus Abbildung 5.1 auf einen einzelnen Baumknoten: Ohne dass die Transformation direkt ins Layout eingreift, erfolgt eine Anpassung der Knotenpositionen. Genauso können durch einfaches Ändern der “inside”-Relationen Baumknoten oder Teilbäume an eine andere Position im Baum verschoben werden.

Falls der gewünschte Effekt über den automatischen Layoutmechanismus nicht erreicht werden kann, ist es auch möglich, aus Transformationsregeln heraus direkt Java-Code zu aktivieren, der dann die Positionsattribute von Komponenten verändert oder das Layout-Modul explizit aufruft. Auf diese Art werden auch explizite Layoutkorrekturen nach Zoom-Transformationen realisiert, die in Kapitel 6 genauer beschrieben werden.

## 5.2 Aus- und Einblenden von Diagrammteilen

Um die Zoom-Transformationen für die gewünschte semantische Fokus- und Kontext-Darstellung auf diese Weise beschreiben zu können, ist es notwendig, neue Operatoren für primitive Regeln einzuführen:

Auszublendende HGM-Hyperkanten und die entsprechenden Komponenten können genauso aus dem Diagramm entfernt werden, wie dies beim Löschen geschieht. Es ist aber notwendig, eine geeignete Referenz auf diese Hyperkanten zu behalten, um sie später wieder einblenden zu können. Wir haben dies so realisiert, dass beim Ausblenden einer Hyperkante stets eine andere Hyperkante als “Anker” angegeben werden muss. Dieser Anker kann dann später durch eine andere Diagrammtransformation wieder “expandiert” werden, was bewirkt, dass alle von ihm referenzierten ausgeblendeten Hyperkanten (und ggf. die von ihnen besuchten Knoten) wieder zum HGM hinzugefügt werden. Normalerweise wird dieser Mechanismus so genutzt, dass die Komponenten-Hyperkante einer Abstraktionsdarstellung als Anker für den von ihr repräsentierten ausgeblendeten Diagrammbereich (bzw. die entsprechenden HGM-Hyperkanten) dient. Anker-Hyperkanten können gegebenenfalls auch ihrerseits wieder ausgeblendet werden. Wird ein Anker endgültig gelöscht, so gehen auch alle von ihm referenzierten ausgeblendeten Hyperkanten verloren.

Beim Ausblenden von Diagrammteilen müssen im Allgemeinen nicht nur die Komponenten-Hyperkanten gespeichert werden, sondern meist auch die Relationen-Hyperkanten, welche ihre Beziehung zueinander und zum Rest des Diagramms beschreiben: Das Diagramm kann ja modifiziert werden, während Teile ausgeblendet sind, und wieder eingeblendete Teile sollten in ihrer Lage entsprechend angepasst werden. Normalerweise erledigt dies die Layoutkorrektur, aber dazu muss das Layout-Modul natürlich über die notwendige Information über den Zusammenhang des Diagramms in Form von Relationen-Hyperkanten und der daraus abgeleiteten Analysestruktur verfügen.

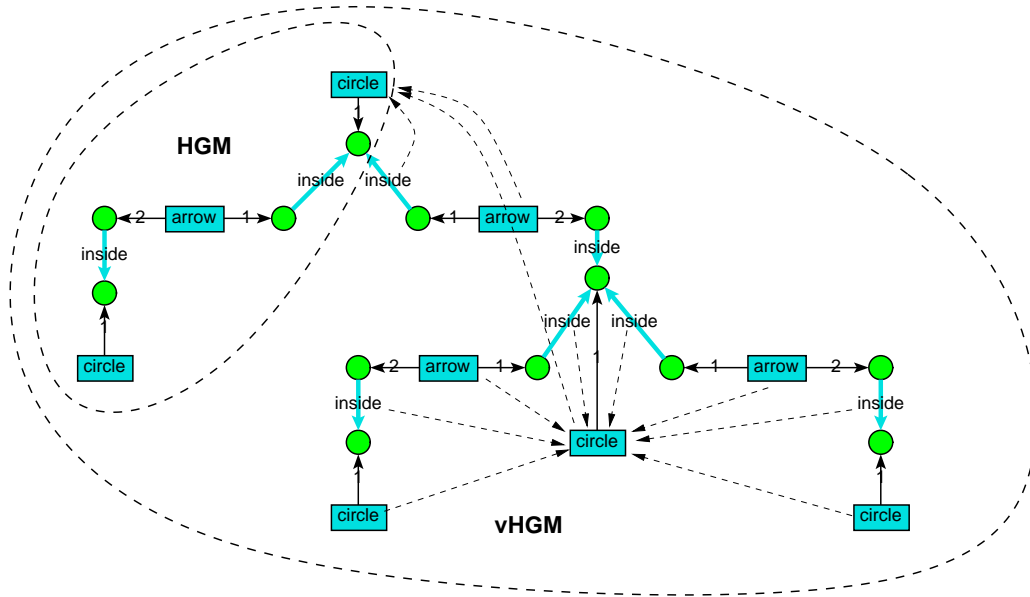


Abbildung 5.3: Graph-Modell eines Diagramms mit ausgeblendeten Teilen

Anschaulich kann man sich die Programmstrukturen für ausgeblendete HGM-Hyperkanten so vorstellen, dass das HGM, welches visualisiert wird, nur einen Teilgraphen eines größeren "virtuellen Hypergraph-Modells" darstellt. Dieses besteht neben dem HGM auch noch aus ausgeblendeten Diagrammteilen; dabei müssen beide Modelle der Konsistenzbedingung genügen, dass eine Relationen-Hyperkante nur Knoten besuchen darf, welche mit Komponenten-Hyperkanten im selben Modell verbunden sind (vgl. S. 26). Das bedeutet, sichtbare Relationen-Hyperkanten müssen immer auch sichtbare Komponenten-Hyperkanten verbinden; ausgeblendete dürfen dagegen sowohl mit sichtbaren als auch mit ausgeblendeten Komponenten-Hyperkanten verbunden sein, aber sie dürfen nicht frei "hängen". Deshalb müssen beim Löschen einer Komponenten-Hyperkante auch alle ausgeblendeten Relationen-Hyperkanten entfernt werden, die mit ihr verbunden sind. (Tatsächlich erfolgt eine entsprechende Prüfung erst beim Wiedereinblenden, da auf ausgeblendete Hyperkanten ohnehin nicht zugegriffen werden kann.)

Alle ausgeblendeten Kanten sind außerdem über eine Meta-Kante<sup>1</sup> mit ihrer Anker-Hyperkante verbunden. Diese kann selbst wieder ausgeblendet sein, wodurch eine Baumstruktur dieser Meta-Kanten entsteht; die Wurzeln dieser Bäume liegen immer bei Hyperkanten des sichtbaren HGMs. Abbildung 5.3 zeigt diese Datenstrukturen für ein Baumdiagramm, bei dem der rechte Teilbaum in zwei Stufen ausgeblendet wurde; die Anker-Verbindungen sind als gestrichelte Pfeile dargestellt.

<sup>1</sup>Wir verwenden die Bezeichnung "Meta-Kante" für eine Kante, deren Ausgangs- und Endpunkte keine Knoten, sondern andere Kanten sind.

Wir haben uns entschieden, den meisten Programm-Modulen von DiaGen nur das (sichtbare) HGM und nicht das volle virtuelle HGM zugänglich zu machen, einerseits, weil so tiefgreifende Änderungen der Programmstruktur vermieden werden konnten, aber vor allem, weil die Dauer der Diagrammanalyse bei den meisten Diagrammsprachen erheblich stärker als linear in der Anzahl der HGM-Kanten ansteigt. (In [32] finden sich empirische Ergebnisse zur Effizienz der Analyse.) Es wäre für die Benutzerin störend und nicht nachvollziehbar, wenn die Analyse eines scheinbar wenig komplexen Diagramms mit vielen ausgeblendeten Teilen trotzdem ausgesprochen lang dauern würde, da ein großes virtuelles HGM verarbeitet werden muss.

Allerdings lassen sich so keine Diagrammtransformationen formulieren, welche selektiv auf ausgeblendete Kanten zugreifen. Es ist allerdings möglich, Hyperkanten innerhalb derselben Transformation erst ein- und dann wieder auszublenken. Falls sich dieser Umweg als zu unpraktisch erweisen sollte, könnte man auch über andere Zugriffsmöglichkeiten nachdenken.

### 5.3 Auswahl von Abstraktionsbereichen

Mit den beschriebenen Basis-Operatoren ist es möglich, beliebige Hyperkanten des HGM und die dazugehörigen Komponenten aus- und wieder einzublenden. Um sie sinnvoll einsetzen zu können, ist es notwendig, diejenigen Hyperkanten, welche eine Abstraktionseinheit darstellen, mit Hilfe von Mustern der Transformationsregeln auszuwählen.

In vielen Fällen reichen die zu Beginn der vorliegenden Arbeit verfügbaren Regelmuster in Form von Teilgraphen des HGM für diese Zwecke schon aus, wenn die Regelanwendungen durch die oben erwähnten Kontrollstrukturen geeignet iteriert werden. Die Formulierung von Zoom-Transformationen ist mit diesen Mitteln aber alles andere als einfach und folgt auch keinem allgemeinen Schema. Zusätzliche Probleme entstehen dadurch, dass beim Ausblenden von Diagrammteilen in mehreren Schritten (mit Hilfe von Kontrollstrukturen) genau darauf geachtet werden muss, dass sich diese nicht ungewollt durch Seiteneffekte beeinflussen.

Zum Beispiel ist es in Baumdiagrammen recht einfach möglich, mit Hilfe von rekursiven Transformationsregeln einen Teilbaum ausgehend von dessen Wurzel zu durchlaufen, wie dies für die in Abschnitt 3.3 vorgestellte Abstraktionsoperation notwendig ist. Beim Ausblenden muss man aber beachten, von den Blättern zur Wurzel vorzugehen, da sonst Hyperkanten vorzeitig entfernt werden, die noch zum weiteren Durchlaufen benötigt werden. In anderen Fällen sind ähnliche Probleme schwieriger zu lösen, daher bestand hier ein Bedarf für einfachere mächtigere Möglichkeiten zur Formulierung von Transformationsregeln.

Da die hierarchische Struktur einer Diagrammsprache, die wir zur Abstraktion nutzen wollen, normalerweise auch in ihrer Grammatik zum Ausdruck kommt,

liegt es nahe, die Ergebnisse der Diagrammanalyse auch zur Bestimmung von Abstraktionsbereichen zu benutzen. Dies wird durch zusätzliche Pfadausdrücke<sup>2</sup> für Regelmuster ermöglicht:

Neben bereits implementierten Pfadausdrücken für das Verfolgen von “normalen” Hyperkanten- und Knoten-Verbindungen im HGM wurden Ausdrücke für Pfade eingeführt, welche aus dem HGM herausführen und die Analysestrukturen (rHGM und Parse-DAG, vgl. Abschnitt 3.2) durchlaufen können. Die Analysestrukturen werden dabei nur zum Mustervergleich herangezogen, dürfen aber von Transformationsregeln nicht direkt modifiziert werden. Die Notwendigkeit solcher Pfadausdrücke wurde bereits in [32] diskutiert; im Zusammenhang mit Zoom-Transformationen hat sich ihre Verwendung als ausgesprochen nützlich erwiesen.

Die Pfadausdrücke enthalten Muster für (virtuelle) Meta-Kanten, die HGM-, Terminal- und Nichtterminal-Hyperkanten untereinander verbinden. Folgende Muster für Pfad-Verbindungen sind zulässig:

- Verbindungen von einer HGM-Hyperkante  $k$  zu allen reduzierten Terminal-Hyperkanten  $t$ , wenn  $t$  durch die Anwendung einer Reduzierer-Regel auf eine Passung im HGM erzeugt wurde, die  $k$  enthält (nicht aber, falls  $k$  nur zum Kontext der Passung gehört). Wir definieren dies für später als Relation  $\text{lex} \subseteq (k, t)$ .<sup>3</sup> Diese Verbindungen können natürlich auch in der umgekehrten Richtung verfolgt werden.
- Verbindungen von einer Hyperkante des rHGM zu der entsprechenden Terminal-Hyperkante im Parse-DAG und umgekehrt. Hier besteht immer eine 1-zu-1 Beziehung, d. h. die Kanten sind im Grunde identisch, aber sie befinden sich in unterschiedlichen internen Hypergraph-Strukturen.
- Ableitungsverbindungen (nach “oben” und “unten”) im Parse-DAG unter Berücksichtigung der zur Ableitung benutzten Produktionen der Grammatik.
- Kontextfreie Ableitungsverbindungen nach “oben” (in Richtung zum Startsymbol) ohne Berücksichtigung der benutzten Produktionen, entweder zur direkten “Eltern“-Kante oder zum nächsten “Vorfahren” mit einem bestimmten Label.
- Verbindungen von einer Nichtterminal-Hyperkante zu allen kontextfrei daraus abgeleiteten Terminal-Hyperkanten.

<sup>2</sup>Pfadausdrücke beschreiben komplexe Verbindungen zwischen zwei Elementen (Hyperkanten oder Knoten) eines Graph-Musters die über einen “Pfad” von dazwischenliegenden Elementen hergestellt werden. Solche Pfade können in DiaGen auch durch zusammengesetzte Muster (ähnlich den bekannten regulären Ausdrücken) beschrieben werden.

<sup>3</sup>Es hat sich herausgestellt, dass bei der Formulierung von Transformationen oftmals Probleme auftreten, wenn  $\text{lex}$  nicht rechtseindeutig ist, d. h. wenn eine Hyperkante des HGM zu mehreren Terminal-Hyperkanten des rHGM “reduziert” wird. Solche Fälle sollten durch geeignete Formulierung von Teilmustern in Reduzierer-Regeln als “positiver Kontext” vermieden werden.

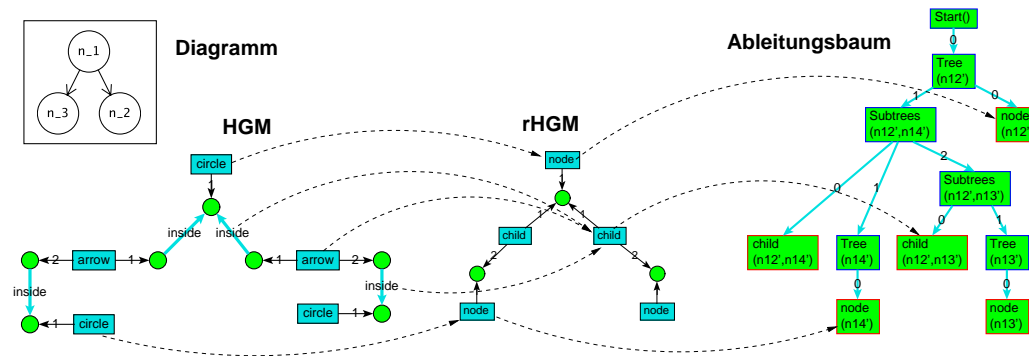


Abbildung 5.4: Verbindungen zwischen den Modellen der verschiedenen Analysestufen

- Verbindungen von einer Nichtterminal-Hyperkante zu allen in ihren kontextfreien Ableitungsteilbaum eingebetteten Parse-Hyperkanten; dabei kann entweder eine "vollständige" Einbettung gefordert werden (alle Einbettungskanten der Ziel-Hyperkante führen im DAG zum bezeichneten Teilbaum) oder eine "partielle" Einbettung (mindestens eine Einbettungskante führt zu einem anderen Bereich des Hauptbaums, vgl. Abbildung 3.4).

Außerdem können Pfadausdrücke Verbindungen zwischen Terminal-Hyperkanten über deren Tentakel und die Knoten des rHGM beschreiben.<sup>4</sup> Wenn sich das in Kapitel 4 beschriebene Konzept zur Generierung eines ASG aus diesen Ableitungsstrukturen bewähren sollte und besser in das Gesamtsystem integriert wird, dann wäre es sinnvoll, auch diesen Graphen, dessen Hyperkanten ja mit Teilen des Parse-DAG korrespondieren, über Pfadausdrücke in den Mustervergleich für Transformationen einzubinden.

Abbildung 5.4 zeigt für ein einfaches Baumdiagramm die verschiedenen daraus erzeugten Hypergraph-Strukturen (HGM, rHGM und Ableitungsbaum) und einige der virtuellen Meta-Kanten zwischen ihnen. Alle sichtbaren Verbindungen können durch geeignete Pfad-Muster verfolgt werden, so dass durch Kombination von solchen Mustern und Regeln leistungsfähige Möglichkeiten bestehen, Teile des Diagramms (bzw. des HGM) auszuwählen und zu manipulieren.

Obwohl in einzelnen Fällen komplexe derartige Muster verwendet werden müssen, hat sich gezeigt, dass zur Formulierung von Abstraktionsoperationen meist ein einfaches Schema ausreicht: Der Abstraktionsbereich wird meistens durch die Auswahl einer "Toplevel"-Komponente  $k$  bezeichnet; in Baumdiagrammen ist dies z. B. die Wurzel des zu abstrahierenden Teilbaums, bei der Abstraktion von UML-Klassen die Klassenrahmen-Komponente. Wenn nun die kontextfreie Ableitungshierarchie der Grammatik der Abstraktionshierarchie entspricht, so

<sup>4</sup>Für nichtterminale Hyperkanten ist dies aus implementierungstechnischen Gründen nicht möglich.

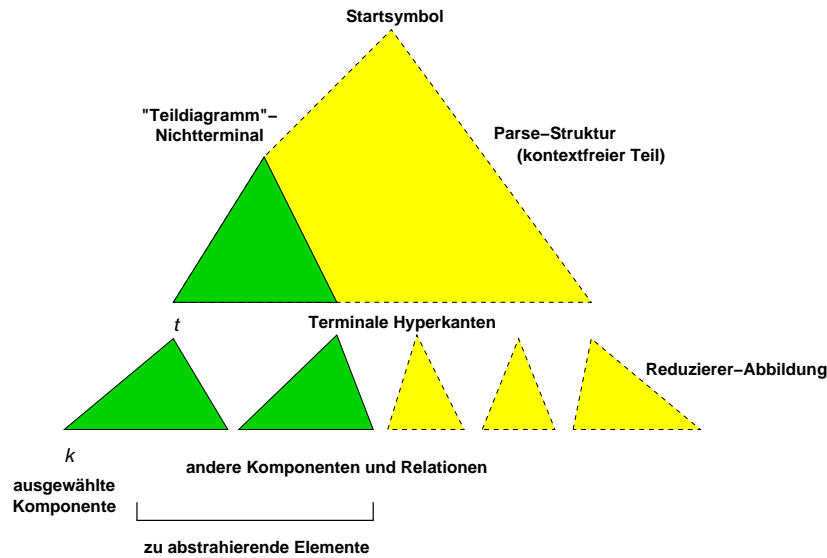


Abbildung 5.5: Schema zur Auswahl eines Abstraktionsbereichs

existiert außerdem ein Nichtterminal-Symbol, das den gewünschten Abstraktionsbereich beschreibt. Im Baum-Beispiel wäre dies z. B. das "Tree"-Nichtterminal.

Zur Auswahl dieses Diagrammbereichs genügt es nun, von der Terminal-Kante  $t$  mit  $\text{lex}(k, t)$  ausgehend<sup>5</sup> im kontextfreien Ableitungsbaum so weit entgegen der Ableitungsrichtung zu laufen, bis ein Nichtterminal des gewünschten Typs gefunden wird. Alle aus diesem Nichtterminal abgeleiteten HGM-Hyperkanten stellen dann den Abstraktionsbereich dar und können mit einem einfachen Pfadausdruck gefunden und ausgeblendet werden. Abbildung 5.5 illustriert dieses Schema. Die Hyperkante  $k$  selbst soll normalerweise nicht ausgeblendet werden, sondern als Abstraktionsdarstellung zurückbleiben (und damit als Anker zum Wiedereinblenden dienen). Günstigerweise bewirken Injektivitätsbedingungen<sup>6</sup> beim Mustervergleich, dass der Pfadausdruck die Hyperkante  $k$  nicht mit beschreibt. Falls nötig, können durch Ergänzung des Auswahl-Pfadausdrucks oder durch weitere Regeln noch weitere HGM-Hyperkanten ausgeblendet werden; so wird man z. B. meist auch alle vollständig in den gefundenen Teilbaum eingebetteten Ableitungsstrukturen einschließen wollen.

Ein Problem bei der Verwendung von Analysestrukturen in Mustern zur Graphtransformation besteht darin, dass diese Strukturen erst nach Beendigung einer kompletten Diagrammtransformation aktualisiert werden können. Daher führen Transformationen mit mehreren Einzelschritten zwangsläufig dazu, dass Inkonsistenzen zwischen dem HGM und den Analysestrukturen auftreten. So ist es

<sup>5</sup>Wir setzen hier wieder voraus, dass  $t$  eindeutig ist.

<sup>6</sup>Eine "injektive" Passung eines Musters erfordert, dass ein Element des Wirtsgraphen nicht gleichzeitig zwei Elementen des Musters zugeordnet werden kann; so kann hier z. B.  $k$  nicht als Ausgangspunkt und Ziel eines Pfades fungieren.

z. B. möglich, mit einer (untergeordneten) Regel eine HGM-Hyperkante zu löschen und anschließend eine andere Regel zu aktivieren, deren Muster mittels eines Pfadausdrucks über das nicht aktualisierte rHGM auf diese gelöschte Kante zugreift. Eine derartige Verwendung von Regeln betrachten wir als einen Spezifikationsfehler, der zur Laufzeit einen Abbruch der Regelauswertung nach sich zieht. Bei der Komposition von Regeln muss sich die Programmiererin daher derartiger Seiteneffekte bewusst sein.

Obwohl dieser Zustand relativ unbefriedigend erscheint, haben wir noch keinen geeigneten Weg gefunden, um solche Inkonsistenzen grundsätzlich zu vermeiden. Die entstehenden Probleme lassen sich z. T. dadurch umgehen, dass eine Diagrammtransformation in Teilschritte zerlegt wird, zwischen denen eine neue Analyse des veränderten HGM erfolgt. Ein solches Vorgehen ist z. B. notwendig, wenn Abstraktions- und Expansionstransformationen nacheinander ausgeführt werden sollen (vgl. Kapitel 7). Das explizite Auslösen der Diagrammanalyse darf aber nicht zu häufig erfolgen, da diese, wie besprochen, sehr zeitaufwendig ist.

Um Inkonsistenzen zwischen HGM und rHGM prinzipiell auszuschließen, könnte man zu einem veränderten Ausführungsmodell für Transformationsregeln übergehen, bei dem Modifikationen des HGM während einer Transformation nur "gesammelt" werden und erst nach ihrem Abschluss tatsächlich ausgeführt werden. Ein derartiges Ausführungsmodell erscheint aber zur Programmierung nicht intuitiv und bringt außerdem zahlreiche Einschränkungen mit sich; so können z. B. keine Markierungshyperkanten mehr (wie im folgenden Abschnitt beschrieben) zur Ablaufsteuerung verwendet werden.

## 5.4 Austauschen von Diagrammkomponenten

Die beiden beschriebenen Erweiterungen des Graphtransformationsmoduls von DiaGen bilden die Basis für die Spezifikation von Zoom-Transformationen. Um diese komfortabel formulieren zu können, wurden aber noch weitere neue Spezifikationskonstrukte eingeführt, die hier kurz beschrieben werden sollen.

Zunächst erfordert die Spezifikation einer Zoom-Transformation oft, dass eine Diagramm-Komponente durch eine Abstraktionsdarstellung ersetzt wird (vgl. z. B. Abbildung 3.7). Dies lässt sich durch Löschen der alten Komponente und Einfügen der neuen leicht realisieren; die neue Komponente muss dabei aber auch die Relationen-Hyperkanten der alten "übernehmen", welche ihre Beziehung zum Rest des Diagramms beschreiben, da diese sonst mit dem Löschen der alten Komponente verlorengehen. Dieses Umsetzen oder Kopieren der Relationen-Hyperkanten kann zwar prinzipiell mit den vorhandenen Mitteln durch Iteration von Kopierregeln für alle relevanten Relationen-Kantentypen beschrieben werden; ein solches Vorgehen führt aber zu sehr umständlichen Spezifikationen.

Deshalb haben wir einen speziellen Relationen-Kopieroperator für Transformationsregeln eingeführt, der jeweils mit einem Paar von entsprechenden Knoten

der alten und der neuen Komponente aufgerufen wird. Bei seiner Anwendung wird für jede Relationen-Hyperkante, die den Quellknoten besucht, eine neue Kante gleichen Typs erzeugt, deren entsprechende Tentakel den Zielknoten besucht.<sup>7</sup> So lässt sich das Ersetzen von Komponenten unter Beibehaltung ihres Kontexts einfach spezifizieren.

Komplexe mehrstufige Graphtransformationen erfordern auch oft die Verwendung von Markierungselementen, um Informationen zwischen verschiedenen Teil-Transformationen weitergeben zu können. Um dies zu realisieren, haben wir zusätzlich zu den Komponenten- und Relationen-Hyperkanten noch eine Klasse von Markierungshyperkanten im HGM eingeführt. Solche Hyperkanten verbinden, wie Relationen-Hyperkanten auch, die Konnektoren von Komponenten (bzw. die entsprechenden Knoten) und sie werden auch automatisch entfernt, wenn eine der entsprechenden Komponenten gelöscht wird. Im Unterschied zu Relationen-Hyperkanten können Markierungshyperkanten beliebig viele Tentakel besitzen; sie sind in der Visualisierung des Diagramms nicht sichtbar und können nur durch Diagrammtransformationen erzeugt oder gelöscht werden.

Als einfaches Anwendungsbeispiel erlauben solche Markierungshyperkanten das Weiterreichen von einzelnen Knoten zwischen Transformationsregeln: Knoten können zwar mit der Graphtransformationssprache von DiaGen nicht direkt manipuliert werden, aber es ist möglich, eine unäre Markierungshyperkante zu erzeugen, deren einziger Tentakel den gewünschten Knoten besucht. Diese Kante kann dann als Passungsvorgabe (vgl. S. 47) bei der Anwendung einer Transformationsregel benutzt werden.

Markierungshyperkanten können z. B. auch dazu genutzt werden, Informationen über ausgeblendete Elemente im HGM zu speichern. Wie auf Seite 34 beschrieben, ist es beim Abstrahieren von Paketen in UML-Diagrammen notwendig, Pfeile zu oder von inneren Elementen des Pakets in Ersatzdarstellungen umzuwandeln und zur Abstraktionsdarstellung des Pakets "umzuleiten". Dies kann natürlich an beiden Enden eines Pfeils geschehen. Der Pfeil muss genau dann wieder zu seiner Originalform zurückgesetzt werden, wenn beide "echten" Endpunkte sichtbar sind, unabhängig von der Reihenfolge, in der Abstraktions- und Expansionstransformationen ausgeführt worden sind. Um diese Informationen über die Umleitung zu speichern, müssen Markierungshyperkanten verwendet werden.

Die entsprechenden Transformationen sind zu komplex, um hier erörtert zu werden, Abbildung 5.6 zeigt jedoch ihren Effekt auf ein (sehr einfaches) Beispieldiagramm: Immer wenn eine oder beide Klassen, die der Generalisierungspfeil verbindet, ausgeblendet sind, dann wird der Pfeil als Abhängigkeit dargestellt und endet an der Abstraktionsdarstellung des jeweiligen Pakets. Sobald beide Klassen wieder expandiert sind, wird auch der Pfeil wieder als Generalisierung sichtbar.

---

<sup>7</sup>Die Notwendigkeit für Spezialkonstrukte zum "Umhängen" von Kanten wurde auch schon bei der Formulierung anderer Graphtransformationssprachen erkannt, die das Löschen von Knoten mit nicht voll spezifiziertem Kontext erlauben. PROGRES [26] bietet z. B. durch seine "path redirection" einen ähnlichen (leistungsfähigeren) Mechanismus.

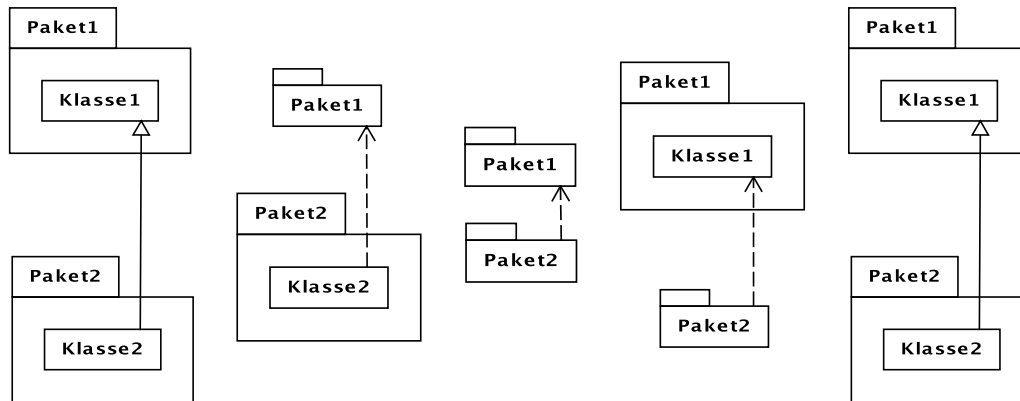


Abbildung 5.6: Veränderung eines Pfeils bei Abstraktion und Expansion

## 5.5 Generische Transformationsregeln

Eine andere Erweiterung, welche die Spezifikation von Diagrammtransformationen und insbesondere Zoom-Transformationen vereinfacht, ist die Einführung einer Typen-Hierarchie für HGM-Hyperkanten: Bei der Deklaration eines Typs einer HGM-Hyperkante mit den entsprechenden Tentakeln und Attributen können optional ein oder mehrere Obertypen angegeben werden. Diese Typen-Beziehung gilt natürlich auch transitiv. Vergleichbar zur Typkonformität in objektorientierten Programmiersprachen wurde der Mustervergleich für Regeln in DiaGen so erweitert, dass für eine Kante des Typs  $t_1$  im Muster auch eine Kante eines Typs  $t_2$  eine korrekte Passung darstellen kann, falls  $t_2$  ein (indirekter) Untertyp von  $t_1$  ist (und alle sonstigen Bedingungen erfüllt sind).<sup>8</sup>

Zur Konformität von Typen fordern wir, dass ein Kantentyp immer mindestens soviele Tentakel definieren muss, wie alle seine Obertypen, so dass stets alle von einem Obertyp im Muster spezifizierten Verbindungen abgedeckt werden können. Eventuelle zusätzliche Tentakel werden beim Mustervergleich ignoriert. Das bedeutet, dass ein Tentakel eines Obertyps immer genau dem Tentakel seiner Untertypen mit derselben Ordnungsposition entspricht.

Ein Obertyp muss immer von der gleichen Art (Komponenten-, Relations- oder Markierungshyperkante) sein wie seine Untertypen; d. h. wir haben es im Grunde mit drei separaten Typenhierarchien zu tun. Da der Generator abhängig von der Art der betroffenen HGM-Hyperkanten unterschiedlichen Code zur Ausführung von Graphtransformationen erzeugen muss, ist es (vorläufig) nicht möglich, einen allgemeinen Basistyp anzugeben. Dieser Punkt kann erst bei einer Neuimplementierung der zugrundeliegenden Graphtransformationsmaschine berücksichtigt werden.

<sup>8</sup>Solche Typen-Hierarchie-Konzepte wurde ebenfalls schon in anderen Graphtransformationssprachen implementiert. PROGRES [26] erlaubt beispielsweise eine vergleichbare Deklaration von Ober- bzw. Untertypen für Knoten, da in PROGRES die Knoten eines Graphen die bedeutungstragenden Elemente sind.

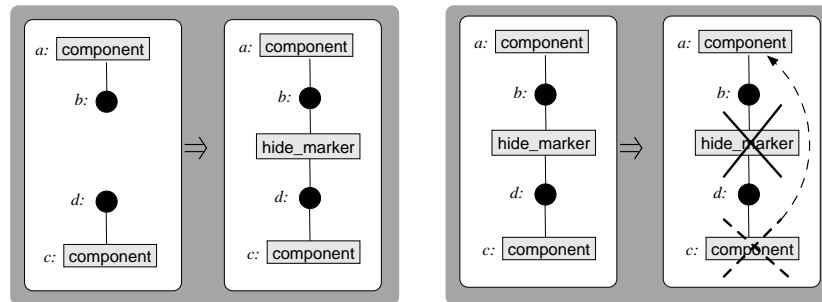


Abbildung 5.7: Generische Regeln zum verzögerten Ausblenden von Komponenten

Durch diese Typhierarchie ist es möglich, Transformationsregeln “allgemein” (nämlich auf Obertypen) zu formulieren, statt für jede Kombination von Typen, die im HGM tatsächlich vorkommen kann, ein eigenes Regelmuster zu schreiben. Zoom-Transformationen bieten ein gutes Anwendungsbeispiel für solche allgemeinen Regeln:

Die Eigenschaft des Transformationsmoduls, beim Entfernen oder Ausblenden von Komponenten-Hyperkanten auch alle verbundenen Relationen-Hyperkanten zu löschen, ist zwar für die Konsistenz des Modells erforderlich, aber dann nachteilig, wenn manche von diesen eigentlich auch ausgeblendet und nicht gelöscht werden sollten. Ein Problem tritt so z. B. bei Iteration über alle Passungen der oben beschriebenen Muster für Zoom-Transformationen auf: Diese liefert die Hyperkanten des gewünschten Abstraktionsbereichs in einer “zufälligen” Reihenfolge, deshalb können Komponenten-Hyperkanten in einer solchen Iteration nicht unmittelbar ausgeblendet werden; dabei würden als Seiteneffekt möglicherweise Relationen-Hyperkanten vorzeitig gelöscht, die die Iteration später noch besuchen soll.

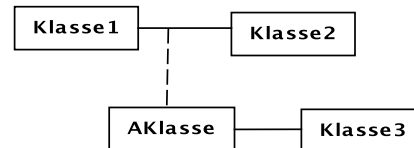
Eine einfache Lösung ergibt sich, wenn die Komponenten-Hyperkanten nicht direkt ausgeblendet werden, sondern zunächst nur durch spezielle Markierungshyperkanten mit der Anker-Hyperkante verbunden werden. Am Ende der Zoom-Transformation kann mit einer Iteration über alle diese Markierungshyperkanten das Ausblenden der Komponenten nachgeholt werden. Abbildung 5.7 zeigt die Regeln zum verzögerten Ausblenden von Komponenten;<sup>9</sup> für Relationen muss ein analoges Regelpaar definiert werden. Alle in diesem Schema verwendeten Transformationsregeln sind generisch, d. h. es kommt nur darauf an, ob die auszublende Hyperkante eine Komponente oder eine Relation repräsentiert. Der konkrete Typ der Hyperkante und auch des Ankers ist dagegen ohne Bedeutung. Das bedeutet, dass zum Ausblenden eines Abstraktionsbereichs nur ein oder mehrere Muster spezifiziert werden müssen, welche diesen Bereich aus dem HGM auswählen; die Transformation braucht dann nur noch die generischen “Bibliotheks”-Regeln aufzurufen.

<sup>9</sup>Durchgestrichene Hyperkanten werden gelöscht; als Notation für das Ausblenden dient ein gestricheltes Kreuz mit einem Pfeil zur Anker-Hyperkante.

## 5.6 Diagrammanalyse bei mehrstufigen Einbettungen

Die bislang besprochenen Erweiterungen der Graphtransformationfähigkeiten von DiaGen beziehen sich alle auf die Spezifikation von Diagrammtransformationen, besonders Zoom-Transformationen. Bei der Spezifikation einer komplexen Diagrammsprache wie UML haben sich aber auch Defizite der Beschreibungsmittel gezeigt, die DiaGen für die Diagrammanalyse bietet. Zu ihrer Behebung waren zusätzliche Ergänzungen des DiaGen-Systems notwendig; diese stehen zwar nicht in direktem Zusammenhang mit Zoom-Transformationen, aber im Rahmen dieser Arbeit stellten sich mehrere generelle Probleme, deren Lösung im Folgenden kurz dargestellt werden soll.

Eine erste Schwierigkeit ergab sich daraus, dass der in DiaGen benutzte Formalismus einer kontextfreien Grammatik mit Einbettungen (vgl. S. 26) es nicht erlaubt, dass ein eingebettetes Diagrammelement seinerseits als Einbettungskontext fungiert. Wie schon in [25] argumentiert wurde, ist es damit prinzipiell nicht möglich, die UML-Notation für Assoziationsklassen geeignet zu parsen: Eine Assoziationsklasse *ac* beschreibt eine Assoziation *a* zwischen zwei Klassen *c*<sub>1</sub> und *c*<sub>2</sub>, die selbst noch über zusätzliche Attribute verfügt. Die Assoziationsklasse *ac* kann dabei ihrerseits auch wieder Assoziationen zu anderen Klassen wie *c*<sub>3</sub> haben.



**Abbildung 5.8:** Eine Assoziationsklasse, die selbst eine Assoziation hat

Die Verbindungen zwischen *a* und *c*<sub>1</sub> bzw. *c*<sub>2</sub> sowie die zwischen *a* und *ac* müssen als Einbettung beschrieben werden, da eine kontextfreie Ableitung die entsprechenden Nichtterminale "verbrauchen" und damit verhindern würde, dass *c*<sub>1</sub>, *c*<sub>2</sub> und *ac* mit weiteren Assoziationen verbunden sind. Dies verstieße aber gegen die Regeln für Einbettungen, da die Klasse *ac* als eingebettetes Element nicht gleichzeitig wieder Einbettungskontext für die Assoziation zu *c*<sub>3</sub> sein kann.

Die Aufteilung der DiaGen-Analyse in zwei Phasen (Reduzierer und Parser) erlaubt es aber, derartige Probleme mit einem "Trick" zu lösen: Die Verbindungen zwischen einer Assoziation und den beteiligten Klassen wird nämlich bereits vom Reduzierer verifiziert; nur korrekt verbundene Assoziationen werden so überhaupt zu Nichtterminalen reduziert. Der Parser kann daher jedes Assoziationen-Nichtterminal einfach mit einer *kontextfreien* Produktion in ein korrektes Diagramm integrieren, ohne seine Verbindungen überhaupt zu prüfen.

Deshalb können Assoziationen auch als Einbettungskontext für weitere Pfeile dienen, da UML z. B. auch Generalisierungen zwischen Assoziationen erlaubt. Man beachte, dass dieser "Trick" nur funktioniert, weil *jeder* Klassenrahmen in einem UML-Diagramm Bestandteil einer korrekten Klasse ist; sonst könnte die Korrektheit der Verbindungen nicht vom Reduzierer geprüft werden.

## 5.7 Effizientes Parsen von Mengen

Ein Punkt, der bei der Formulierung effizient zu parsender Grammatiken mit DiaGen immer Probleme bereitet hatte, war die Spezifikation von Produktionen, die eine Menge gleichberechtigter Elemente zusammenfassen. Derartige Mengen treten in praktisch jeder Diagrammsprache auf; bei UML-Diagrammen bildet die Menge aller Bestandteile eines Pakets (Klassen und Unterpakete) ein geeignetes Beispiel. Eine naheliegende Möglichkeit, solche Mengen zu beschreiben, bieten Produktionen nach dem folgenden Schema

```
set ::= element
set ::= set element
```

wie sie häufig in String-Grammatiken auftreten. Während aber Strings durch ihre implizite Ordnung mit solchen Produktionen deterministisch analysiert werden können, bieten sich bei ungeordneten Graphen für  $n$  Elemente  $n!$  verschiedene Reihenfolgen, in denen die Elemente zusammengefasst werden können und es werden entsprechend auch verschiedene Nichtterminale erzeugt, die die gesamte Menge repräsentieren, d. h. die Grammatik ist nichtdeterministisch. Da der Parser aus jedem der erzeugten Nichtterminalen (und außerdem meist auch noch aus Nichtterminalen, die Teilmengen repräsentieren) wieder größere Einheiten konstruieren kann, erhält man bei rekursiven Strukturen eine hyper-exponentiell steigende Anzahl der möglichen Ableitungsstrukturen, so dass Diagramme nicht mehr in akzeptabler Zeit analysiert werden können.

Ursprünglich hatten wir versucht, dieses Problem dadurch zu beheben, dass der Reduzierer zusätzliche "Ordnungskanten" einfügt, welche die Elemente nach einem beliebigen Kriterium (z. B. ihrer Position) sortieren und beim Parsen eine deterministische Reihenfolge der Zusammenfassung erzwingen.<sup>10</sup> Die bei diesem Schema erforderlichen Hilfsproduktionen verschleiern aber, gerade bei komplexen Diagrammsprachen, die eigentliche Struktur der Sprache und ihrer Grammatik und komplizieren auch die erzeugten Analyse-Strukturen (rHGM, Parse-DAG) unnötig. Ausserdem verlangt eine solche "Sortierung" durch den Reduzierer, dass dieser bereits entscheiden kann, ob alle Elemente korrekt geparkt werden können; andernfalls scheitert der Parser beim Zusammenfassen der Menge, sobald eine Ordnungskante nicht auf ein korrektes Nichtterminal verweist.

Stattdessen hat es sich als sinnvoller erwiesen, die Grammatik (und den Parser) um spezielle "Mengenproduktionen" zu erweitern. Abbildung 5.9 zeigt ein Beispiel; das neu eingeführte Konstrukt wird durch den Kasten auf der rechten Seite symbolisiert, der andeutet, dass sein Inhalt mehrfach auftreten kann. Eine solche Produktion bedeutet, dass ein Nichtterminal "Set" zu einer beliebigen Anzahl von Nichtterminalen des Typs "Element" abgeleitet werden kann, welche die Knoten außerhalb des Mengen-Kastens gemeinsam haben. Knoten innerhalb

<sup>10</sup>Dieses Erzwingen einer Ordnung weist Parallelen zu anderen Grammatik-Formalismen für visuelle Sprachen auf [28], die aus Effizienzgründen vor dem Parse-Vorgang eine Linearisierung aller Terminal-Elemente verlangen.

des Kastens existieren für jedes Element der Menge separat, daher können sie auch nicht auf der linken Seite der Produktion erscheinen. Mit solchen Produktionen können Strukturen der beschriebenen Art auf intuitive Weise spezifiziert und hinreichend effizient geparkt werden, da der Parser immer nur ein gültiges Nichtterminal für jede derartige Menge erzeugt; auch ihre Teilmengen führen nicht zu zusätzlichen Nichtterminalen.

Auf die dazu erforderliche Erweiterung des Parsers soll hier nicht näher eingegangen werden (diese war nicht Bestandteil der vorliegenden Arbeit); es sei nur erwähnt, dass diese ein mehrstufiges Vorgehen verlangt, bei dem Mengen durch neu generierte Elemente erweitert werden können und damit auch bereits gefundene Nichtterminale für Mengen wieder ungültig werden können. Auf eine ähnliche Weise sollte es auch möglich sein, "mehrstufige" Einbettungen zuzulassen und so die im vorangehenden Abschnitt besprochenen Probleme grundsätzlich zu vermeiden. Vielleicht ließe sich auch ein allgemeinerer Mechanismus formulieren, der die Anwendungsfälle von Einbettungs- wie auch Mengenproduktionen abdeckt, aber (im Gegensatz zu allgemeinen kontextsensitiven Grammatiken) trotzdem eine effiziente Analyse gängiger Diagrammsprachen erlaubt.

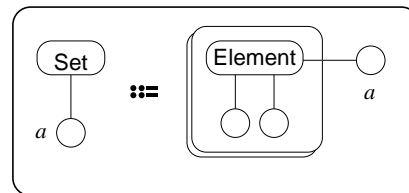


Abbildung 5.9: Eine Mengenproduktion

## 5.8 Berücksichtigung der Bearbeitungsabfolge

Die Struktur von UML-Diagrammen in Verbindung mit der automatischen Layoutkorrektur macht auch noch eine grundsätzliche Erweiterung der Analyse-Mechanismen von DiaGen notwendig: In manchen Fällen kann ein Diagramm nämlich nicht allein auf der Basis seines aktuellen Zustands analysiert werden, sondern es ist notwendig, die "Vorgeschichte" zu kennen, welche zu diesem Zustand geführt hat.

Als Motivation zu dieser Überlegung soll die Zuordnung von Pfeilbeschriftungen in UML-Diagrammen dienen: Die Spezifikation der UML sagt hier nur aus, dass eine Beschriftung "nahe" am dazugehörigen Pfeil liegen muss. Diese Spezifikation reicht zur Erzeugung eines Diagramms aus einer abstrakten Repräsentation (im Rahmen des UML-Metamodells); sie genügt aber bei der Analyse von UML-Diagrammen nicht immer, um eine Beschriftung eindeutig zuzuordnen, da sich ja auch andere Pfeile in ihrer Nähe befinden können. Natürlich dürfen mehrdeutige Diagramme in DiaGen-Editoren auftreten; in diesem Fall sollte die Analyse-Spezifikation dafür sorgen, dass die mehrdeutigen Teilbereiche überhaupt nicht analysiert werden und so die "Fehlerquelle" sichtbar wird. Die Benutzerin muss dann aber eine Möglichkeit haben, das Diagramm tatsächlich eindeutig zu machen, um die gewünschte "Semantik" ausdrücken zu können.

Man könnte Konflikte nun so auflösen, dass eine Beschriftung immer dem geometrisch nächstliegenden Pfeil zugeordnet wird. Diese Beziehungen müssten dann aber bei Verschiebungen der Pfeile durch die Layoutkorrektur bewahrt werden, und ein entsprechender Layout-Algorithmus wäre unvertretbar aufwendig. Die von uns ursprünglich gewählte Alternative war, beim Erzeugen jeder Beschriftung von der Benutzerin eine explizite Zuordnung zu verlangen; diese kann dann mit Hilfe der auf Seite 56 vorgestellten Markierungshyperkanten im HGM festgehalten werden. Dieser Ansatz widerspricht aber dem Bearbeitungskonzept von DiaGen, da eine solche (nicht sichtbare) Zuordnung nur durch Diagrammtransformationen und nicht durch direkte Manipulation erzeugt und geändert werden kann.

Wir haben uns deshalb für eine Lösung entschieden, bei der solche Zuordnungskonflikte aus der Bearbeitungsvorgeschichte des Diagramms gelöst werden, d. h. der Reduzierer soll sich, falls er mehrere anwendbare Passungen für eine Zuordnungsregel findet, für die Zuordnung entscheiden, die bei der letzten Analyse des Diagramms gewählt wurde (sofern dies möglich ist; andernfalls ist das Diagramm mehrdeutig und die Regel wird, nach dem oben erwähnten Prinzip, gar nicht angewandt). Damit wird eine einmal erkannte Zuordnung solange wie möglich beibehalten; die Benutzerin kann aber durch direkte Manipulation auch eine andere eindeutige Zuordnung herbeiführen.

Zur Realisierung dieser Idee wurde eine zusätzliche Rückkopplung von der Analyse zum Diagramm eingeführt: Der Reduzierer kann nun nicht nur Terminal-Hyperkanten für das rHGM erzeugen sondern zusätzlich auch "Erinnerungs"-Hyperkanten, die nach Abschluss der Analyse zum HGM hinzugefügt werden. Diese Hyperkanten repräsentieren eine einmal getroffene Zuordnung und werden von den Reduzierer-Regeln zur Konfliktauflösung benutzt. Sie stellen einen Spezialfall von Markierungshyperkanten dar, sind aber temporär und werden nach dem nächsten Reduzierungsvorgang wieder gelöscht. Wenn die Zuordnung allerdings beibehalten wird, sorgt die erneute Anwendung der Reduzierer-Regel dafür, dass die verwendete "Erinnerungs"-Hyperkante anschließend wieder erzeugt wird und so erhalten bleibt.

Abbildung 5.10 zeigt das Schema, nach dem diese Hyperkanten benutzt werden:<sup>11</sup> Zum Erkennen einer Zuordnung sind zwei Reduzierer-Regeln zuständig; eine, die nur dann angewandt wird, wenn kein Konflikt vorliegt (negativer Kontext) und eine, die bei einem auflösbaren Konflikt angewandt wird (positiver Kontext und Erinnerungshyperkante). Beide Regeln erzeugen mit ihren Ersetzungsgraphen neben einer Terminalkante auch eine Erinnerungshyperkante. Beim Formulieren der Regeln muss darauf geachtet werden, dass nie beide gleichzeitig angewendet werden, da sonst Teile des rHGM dupliziert würden; dazu dient der positive Kontext in der zweiten Regel.

Natürlich müssen diese Erinnerungshyperkanten auch mit dem Diagramm gespeichert werden, damit es nach dem erneuten Laden korrekt analysiert werden

<sup>11</sup>Grau hinterlegte Musterbereiche bezeichnen Kontext-Elemente; wenn sie durchgestrichen sind, handelt es sich um negativen Kontext, sonst um positiven.

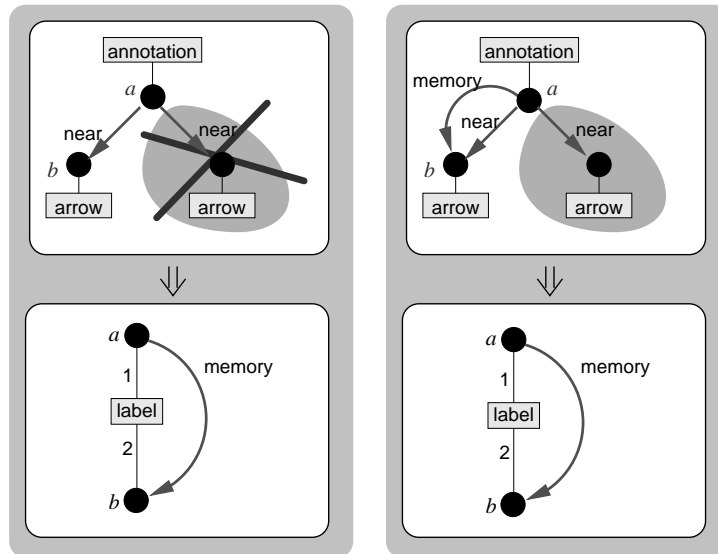


Abbildung 5.10: Konfliktauflösung beim Zuordnen von Pfeilbeschriftungen

kann. Durch die Konsistenz-Prüfung bei Modifikationen des HGM werden sie wie andere Markierungshyperkanten automatisch entfernt, wenn eine der verbundenen Komponenten entfernt wird.

Dieser Zuordnungsmechanismus wird im UML-Editor z. B. auch dazu benutzt, zu verhindern, dass Attribute von einer Klasse zu einer anderen “wandern”, wenn sich die Klassen nach einer Manipulation des Diagramms überlappen sollten.

## 5.9 Zusätzliche wünschenswerte Erweiterungen

Mit den vorgestellten Mitteln konnte eine verständliche und hinreichend kurze Spezifikation (ca. 1500 Zeilen) der Struktur und Transformation von UML-Klassendiagrammen entwickelt werden. Im Laufe dieses Prozesses ergaben sich auch noch weitere Ideen, wie die Spezifikationsprache von DiaGen erweitert werden könnte, um die Beschreibung von Diagrammsprachen komfortabler zu gestalten. Ein solches Konzept, das bislang nicht realisiert wurde und hier nur kurz angesprochen werden soll, ist die Einführung von mehreren Reduzierer-Schritten.

Das rHGM wird dann nicht durch die “gleichzeitige” Anwendung aller Reduzierer-Regeln konstruiert, sondern indem in mehreren Schritten unterschiedliche Regelmengen angewendet werden. So können Vorverarbeitungsregeln Muster finden, die dann in mehreren anderen Regeln noch weiterverarbeitet werden, bevor das rHGM feststeht; d. h. die Reduzierer-Regeln können in “Unterprogramme” zerlegt und so vereinfacht werden.

Beispielsweise tritt eine "enthalten-in"-Relation, die in vielen Diagrammsprachen in tieferen Schachtelungen bestehen kann (z. B. bei Paketen im UML) bei der Hypergraph-Modellierung immer als transitive Hülle auf. Wenn also ein Paket  $p_1$  ein anderes Paket  $p_2$  enthält und dieses wiederum eine Klasse  $c$ , dann existiert auch eine Relationen-Hyperkante zwischen  $p_1$  und  $c$ , aber diese interessiert normalerweise für die Analyse der Sprache nicht und ist oft sogar störend, da  $c$  ja schon Teil von  $p_2$  ist, und deshalb nicht auch noch direkt in  $p_1$  integriert werden soll. Deshalb müssen Muster, die mit den entsprechenden Elementen arbeiten, meist auf "direktes Enthalten-Sein" prüfen (mit einem negativen Kontext), was bei komplizierteren Aufgaben schnell zu verwickelten Mustern mit mehrfachen oder geschachtelten Kontexten führen kann.<sup>12</sup>

Bei mehreren Reduzierer-Schritten kann stattdessen eine Vorverarbeitung stattfinden, die aus allen Kanten der transitiven Hülle neue Hyperkanten erzeugt, die nur noch das direkte Enthalten-Sein repräsentieren. Alle anderen Reduzierer-Regeln können dann in ihren Mustern auf diese Hyperkanten Bezug nehmen.

Mit einem mehrstufigen Reduzierer kann z. B. auch das auf Seite 43 angesprochene Problem des Erzeugens von Knoten für die ASG-Repräsentation gelöst werden. In einem ersten Schritt wird aus einer HGM-Kante eine Hyperkante erzeugt, die über zusätzliche Tentakel und Knoten verfügt. Die zusätzlichen Knoten wären dann aber nicht nur in dieser einen Regel zugänglich, sondern die erzeugte Hyperkante kann wieder auf der linken Seite von anderen Reduzierer-Regeln erscheinen, welche sie geeignet mit anderen Konstrukten verbinden.

Um eine feste Anzahl von Reduzierer-Schritten zu garantieren, müssen alle in Reduzierer-Regeln auftretenden Hyperkanten-Typen partiell geordnet werden, wobei eine Typ  $t_1$  "kleiner" als ein anderer Typ  $t_2$  ist, wenn es eine Regel gibt, in deren Muster eine Hyperkante vom Typ  $t_1$  auftritt und die eine Hyperkante vom Typ  $t_2$  erzeugt; die kleinsten Hyperkanten dieser Ordnung sind die des HGMS. Anhand dieser Ordnung kann der Generator entscheiden, in welcher Reihenfolge die Reduzierer-Regeln abgearbeitet werden, ohne dass dies explizit spezifiziert werden müsste. Außerdem werden damit (evtl. indirekt) rekursive Regeln ausgeschlossen.

Insgesamt haben wir festgestellt, dass viele der Möglichkeiten, die große Graphtransformationspakete wie PROGRES [26] bieten, und die in DiaGen (noch) nicht implementiert sind, sich in bestimmten Fällen als sehr nützlich erweisen könnten, um Spezifikationen zu vereinfachen oder verständlicher zu machen. Beispiele dafür wären leistungsfähigere Kontrollstrukturen für Graphtransformationen oder das selektive Abschalten von Injektivitätsprüfungen beim Mustervergleich; da dies in DiaGen nicht möglich ist, müssen hier Spezialfälle bei der Anwendung von Reduzierer- und Transformationsregeln, bei denen ein Element des Wirtsgraphen mehrere Rollen im Muster übernehmen kann, durch duplizierte Regeln mit leicht abgewandelten Mustern ausgedrückt werden.

---

<sup>12</sup>Diese lassen sich in DiaGen nur eingeschränkt benutzen und wären bei einer grafischen Spezifikation sehr unhandlich.

Aus dieser Beobachtung ergibt sich die Überlegung, dass es sinnvoller sein könnte, nicht für einzelne Anwendungen wie DiaGen spezialisierte Graphtransformationssprachen und -systeme zu programmieren, sondern stattdessen eine möglichst weitreichende und erweiterbare allgemeine Standard-Bibliothek zu entwickeln, analog zu Bibliotheken für Datenstrukturen wie STL<sup>13</sup> oder LEDA<sup>14</sup>. Allerdings ist diese Idee bislang daran gescheitert, dass die vielen existierenden Modelle für Graphen und Graphtransformationen nicht in einem System zu fassen waren. Das generellste uns bekannte Graphtransformationspaket, PROGRES, ist beispielsweise sehr allgemein und leistungsfähig, aber mit dem DiaGen-Ansatz grundsätzlich unvereinbar, da es keine Hypergraphen kennt. Trotzdem scheint es uns möglich, dass man unter Ausnutzung von objektorientierten Konzepten ein allgemeines System zur Bearbeitung von Graphstrukturen realisieren könnte, das die meisten Anwendungsfälle abdeckt und gegebenenfalls angepasst und erweitert werden kann. Die Verfügbarkeit eines solchen Systems würde die Verwendung von Graphen und Graphtransformationen als Modell in der Programmierung sicher enorm fördern.

---

<sup>13</sup>siehe z. B. unter <http://www.cs.rpi.edu/projects/STL/htdocs/stl.html>

<sup>14</sup><http://www.mpi-sb.mpg.de/LEDA/leda.html>



## Kapitel 6

# Diagrammlayout

Ein wesentlicher Bestandteil eines jeden mit DiaGen erzeugten Editors ist die automatische Layoutkorrektur. Ohne einen solchen Mechanismus wäre ein sinnvolles Arbeiten mit direkter Manipulation der Diagrammkomponenten kaum möglich, da jede kleine Änderung weitere Korrekturen nach sich zieht, um die Korrektheit des Diagramms zu erhalten. Das Verschieben einer Klasse im UML-Editor macht beispielsweise eine entsprechende Verschiebung der daran hängenden Pfeilenden erforderlich. Aufgabe der Layoutkorrektur ist es, solche Folgekorrekturen automatisch vorzunehmen. Falls die Benutzerin in einzelnen Fällen explizit Diagrammänderungen vornehmen möchte, welche die Korrektheit des Diagramms zerstören und von der Layoutkorrektur nicht zugelassen werden, ist es auch möglich, diese vorübergehend abzuschalten.

Das Layout eines Diagramms in einem DiaGen-Editor wird durch ausgezeichnete Attribute seiner Diagrammkomponenten (in Form von Fließkomma-Zahlen) festgelegt; diese Attribute definieren Position und Form der jeweiligen Komponente und können zur Anpassung des Layouts von außen (durch das Layoutmodul, vgl. Abb. 2.7) modifiziert werden. Eine solche Änderung hat dann eine automatische Neuberechnung der visuellen Darstellung der betroffenen Diagrammkomponente zur Folge. Wie schon erwähnt (vgl. S. 22), können diese Layoutveränderungen auch animiert werden, um die Effekte verständlicher zu präsentieren.

Im Zusammenhang mit Zoom-Transformationen kommt der Layoutkorrektur die wichtige Rolle zu, das Gesamtdiagramm so anzupassen, dass Größenänderungen einzelner Komponenten Rechnung getragen wird und keine zu großen Leerräume oder Überlappungen entstehen; die Gesamtfläche des Diagramms soll sich ungefähr so verändern, wie die der modifizierten Komponenten. Wie wir anhand der Beispieleditoren sehen werden, kann dies entweder einfach im Rahmen der ohnehin nach jeder Diagrammtransformation erforderlichen Layoutkorrektur geschehen, oder indem die Zoom-Transformation das Layoutmodul explizit anweist, noch zusätzliche Korrekturen vorzunehmen.

## 6.1 Techniken zur Layoutspezifikation

Die Layoutkorrektur für einen Diagrammeditor muss natürlich speziell auf die jeweilige Diagrammsprache zugeschnitten sein, daher ist es erforderlich, in das Dokument, welches die Diagrammsprache beschreibt, auch eine Spezifikation des Diagrammlayouts aufzunehmen. Grundsätzlich existieren zwei verschiedene Grundprinzipien, um das Layout von Diagrammen zu spezifizieren[15]:

- Eine *algorithmische* Spezifikation gibt unmittelbar an, wie aus der Struktur eines Diagramms mittels eines bestimmten Algorithmus ein geeignetes Layout konstruiert wird. In der Literatur der entsprechenden Fachgebiete sind viele derartige Layout-Algorithmen für spezielle Diagrammtypen wie z. B. Bäume oder Graphen beschrieben, beispielsweise in den Konferenzbänden zu “Graph Drawing” (GD).
- Eine *deklarative* Spezifikation legt dagegen nur fest, welche Eigenschaften ein korrektes Layout auszeichnen, ohne aber genau darauf einzugehen, wie dieses berechnet werden soll. Typischerweise werden die gewünschten Layout-Eigenschaften in Form von Constraints (Bedingungen) beschrieben, welche z. B. einen Mindestabstand zwischen zwei Diagrammelementen festlegen oder fordern, dass die Spitze eines Pfeils den Rand eines bestimmten Elements berühren muss. Ein Constraint-Solver ist dann dafür zuständig, die Diagrammelemente so zu positionieren und anzupassen (z. B. in ihrer Größe), dass alle Constraints erfüllt sind. Dazu stehen eine Reihe von unterschiedlich effizienten und ausdrucksstarken Mechanismen zur Verfügung, [17] bietet eine ausführliche Klassifikation.

DiaGen erlaubt grundsätzlich beide Möglichkeiten der Layoutspezifikation. Ursprünglich wurde nur der deklarative Ansatz unterstützt, der, wie wir gleich sehen werden, grundsätzlich auch besser zum Konzept des DiaGen-Systems passt. Effizienzprobleme mit den verwendeten Constraint-Solvern<sup>1</sup> haben es jedoch notwendig gemacht, auch eine Verwendung von direkt programmierten Layoutalgorithmen vorzusehen. Durch die enge Integration von DiaGen mit der “Wirtschaftssprache” Java war dies auch einfach zu erreichen.

## 6.2 Layout in Diagrammeditoren

Im Folgenden sollen nun die unterschiedlichen Vor- und Nachteile beider Techniken für die Anwendung in Diagrammeditoren genauer betrachtet werden:

---

<sup>1</sup>ParCon von der Universität Paderborn und Qoca von der Monash University,  
<http://www.uni-paderborn.de/cs/ag-szwillus/forschung/projekte/parcon/>  
<http://www.csse.monash.edu.au/projects/qoca/>

Die deklarative Spezifikation des gewünschten Layouts ist im Allgemeinen einfacher als die Entwicklung eines Algorithmus, der dieses Layout erzeugt. Der deklarative Ansatz eignet sich daher prinzipiell besonders für Rapid-Prototyping-Anwendungen, für die z. B. auch das DiaGen-System gedacht ist.

Deklarative Layout-Techniken können auch gut zum inkrementellen Layout von Diagrammen eingesetzt werden. In Diagrammeditoren ist es im Allgemeinen nicht nötig, ein Layout für das komplette Diagramm zu berechnen, ohne dass Vorinformationen zur Verfügung stehen. Meist verändern Editieroperationen nur eingeschränkte Bereiche des Diagramms und die Layoutberechnung kann gegebenenfalls auch auf das (korrekte) vorherige Layout zurückgreifen. Dieser Anwendungsfall eignet sich sehr gut zur Umsetzung mittels Constraint-Solving-Techniken: Eine Änderung am Diagramm führt dazu, dass einige Constraints nicht mehr erfüllt sind; nun kann nach einer Möglichkeit gesucht werden, das System zu "reparieren", wobei die vorherige Lösung des Gesamtsystems als Ausgangspunkt zur Verfügung steht. Deshalb müssen nur direkt oder indirekt von der Änderung betroffene Diagrammbereiche modifiziert werden. Viele Standard-Layoutalgorithmen können dagegen keinen Nutzen daraus ziehen, dass sich nur Teile des Diagramms geändert haben und müssen in jedem Fall das komplette Layout neu berechnen.

Zum Ausgleich bieten sich dafür aber natürlich auch mehr Ausdrucksmöglichkeiten, wenn der komplette Layoutalgorithmus von Grund auf neu geschrieben wird. Bei einer deklarativen Spezifikation ist die Programmiererin dagegen durch die Ausdrucksfähigkeit der Constraints eingeschränkt, welche der verwendete Constraint-Solver unterstützt; normalerweise besteht nicht die Möglichkeit, diesen zu erweitern und an spezielle Gegebenheiten anzupassen.

Algorithmen zur Generierung eines Diagrammlayouts sind oft sehr restriktiv, d.h. sie bieten kaum die Möglichkeit, Darstellungspräferenzen der Benutzerin zu berücksichtigen. Typischerweise wird das Layout direkt aus einer abstrakten Repräsentation des Diagramms generiert, so dass für eine solche Repräsentation nur genau ein korrektes Layout existiert. Manipulationen des Diagramms, welche die abstrakte Repräsentation nicht verändern, gehen daher beim nächsten Layoutvorgang wieder verloren. Eine derartige Layouttechnik widerspricht also dem Konzept von DiaGen, das darauf abzielt, der Benutzerin möglichst viele Freiheiten beim Bearbeiten des Diagramms zu lassen und nur unterstützend einzugreifen und Fehler anzuzeigen; diese Idee kommt z. B. darin zum Ausdruck, dass die "direkte Manipulation" des Diagramms die grundlegende Bearbeitungstechnik ist (vgl. Abschnitt 1.2).

Ein constraint-basiertes Layout ist demgegenüber wesentlich flexibler; hier sind für dieselbe abstrakte Diagrammstruktur normalerweise viele verschiedene Layouts möglich, welche alle gleichermaßen die vorgegebenen Bedingungen erfüllen. Ein geeigneter Constraint-Solver wird das Layout nur soweit verändern, wie es zur Erfüllung aller Constraints notwendig ist und ansonsten die von der Benutzerin vorgenommenen Änderungen respektieren.

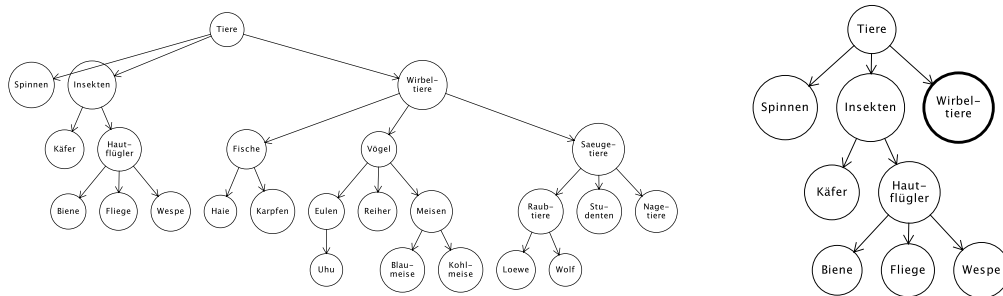
Es kann allerdings auch vorkommen, dass der Constraint-Solver Lösungen findet, die von der Programmiererin beim Spezifizieren der Constraints nicht beabsichtigt wurden. So trat bei einem constraint-basierten Baumlayout in DiaGen beispielsweise der Fall auf, dass der Constraint-Solver beim Verschieben eines Knotens den Knotenradius vergrößerte, um den Kontakt zu daran hängenden Pfeilen nicht zu verlieren; ein Verschieben des Pfeilendes wäre natürlich die sinnvollere Lösung gewesen.

In solchen Fällen ist eine geeignete Anpassung des Constraint-Systems notwendig: Beispielsweise erlauben es leistungsfähige Constraint-Solver, den Constraints Prioritäten zuzuordnen, so dass bei Konflikten wichtigere Constraints bevorzugt erfüllt werden. Wenn man dann geeignete Variablen mit sogenannten "Weak-Stay-Constraints" belegt – Constraints, welche ein Beibehalten des aktuellen Wertes verlangen, dabei aber nur die geringstmögliche Priorität besitzen – kann man so steuern, welche Lösungen der Constraint-Solver bevorzugen soll [16]. Da ein solches Vorgehen von der Programmiererin verlangt, das prozedurale Verhalten des Constraint-Solvers in gewissem Umfang vorherzusehen, wird dadurch natürlich der Vorteil einer einfacheren deklarativen Spezifikation wieder eingeschränkt.

Ein entscheidender Punkt bei der Wahl eines geeigneten Layoutalgorithmus ist natürlich auch seine Effizienz. Hier sind optimierte Spezialalgorithmen oft den allgemeinen Mechanismen des deklarativen Layouts klar überlegen. Beim Layout von Diagrammen entstehen schnell sehr große Constraint-Systeme und das interaktive Arbeiten mit Diagrammeditoren verlangt ein sehr schnelles Layout, das mit einem allgemeinen Constraint-Solver eventuell nicht zu erreichen ist. Dies war ein wesentlicher Grund, weshalb für das DiaGen die ursprüngliche Idee fallengelassen werden musste, sich nur auf eine allgemeine Technik zur Spezifikation des Layouts auf Basis von Constraints zu beschränken [32]. Allerdings wurden dabei Constraint-Solver für allgemeine lineare (Un)gleichungssysteme verwendet, die insbesondere auch zirkuläre Abhängigkeiten verarbeiten können, dafür aber aufwendige Algorithmen benutzen. Wenn man zirkuläre Abhängigkeiten ausschließt, könnten dagegen schnellere Verfahren angewandt werden [17].

### 6.3 Layout für Baumdiagramme

Der Editor für Baumdiagramme bietet ein Beispiel für eine sehr restriktive algorithmische Layoutspezifikation. Hier wird ein einfacher Standard-Layoutalgorithmus für Bäume verwendet, bei dem ausgehend von den Blattknoten das Layout für zunehmend größere Teilbäume berechnet wird. Für jeden Knoten werden seine Kindknoten (falls er solche besitzt) so nebeneinander angeordnet, dass die umschließenden Rechtecke der jeweiligen Unterbäume (deren internes Layout bereits berechnet worden ist) genau aneinander stoßen. Die Knoten der einzelnen Ebenen des Baums werden jeweils horizontal auf derselben Linie ausgerichtet, wobei der Abstand zwischen den Ebenen ebenfalls vorgegeben ist.



**Abbildung 6.1:** Layout eines Baumdiagramms, automatische Korrektur nach Abstraktion

Damit ist es z. B. nicht möglich, zwei “Geschwisterknoten” näher zusammenzuziehen oder einen Unterbaum auf eine andere Ebene zu bewegen. Das Verschieben eines Knotens hat immer zur Folge, dass sich der gesamte Baum mit diesem Knoten bewegt, da sich die relative Lage der Knoten zueinander nicht ändern kann.

Ein derart restriktiver Layoutalgorithmus ist, wie bereits diskutiert, für Freihand-Diagrammeditoren wie die von DiaGen generierten eigentlich nicht wünschenswert. Diese starke Einschränkung bietet jedoch den Vorteil, dass die Realisierung von Zoom-Transformationen wesentlich vereinfacht wird: Dadurch dass das Layout des Diagramms durch die abstrakte Struktur eindeutig vorgegeben ist, sind zusätzliche Layoutkorrekturen weder notwendig noch möglich. Das “Zusammenziehen” und “Auseinanderdrängen” des Diagramms nach dem Abstrahieren bzw. Wiedereinblenden von Diagrammteilen, das ansonsten erforderlich wäre, kann somit entfallen (bzw. es erfolgt von selbst). Wenn wir beispielsweise im Baumdiagramm aus Abbildung 6.1 einen Teilbaum ausblenden, sollte der entstehende Leerraum genutzt werden, um das Diagramm zu kompaktieren. Tatsächlich geschieht dies automatisch, wie die Abbildung zeigt.

Der abstrahierte Knoten rechts außen ist näher an den Rest des Diagramms gerückt, während er bei einem “besseren” Layoutalgorithmus, der das vorherige Layout nach Möglichkeit beibehält, in seiner relativen Lage bleiben müsste, da die Korrektheit des Diagramms nicht beeinträchtigt wäre. In diesem Fall müsste die Zoom-Transformation eine explizite Layoutkorrektur veranlassen.

## 6.4 Layout für UML-Klassendiagramme

Während also der existierende Layoutalgorithmus für Baumdiagramme, wie dargestellt, ohne Veränderungen übernommen werden konnte, musste für den UML-Beispieleeditor eine geeignete Layouttechnik neu entwickelt werden. Unsere Absicht war es dabei, ein möglichst flexibles Layout zuzulassen und Änderungen durch die Benutzerin so weit wie möglich zu respektieren. Deshalb mussten besondere Vorkehrungen getroffen werden, um explizite Layoutkorrekturen nach Zoom-Transformationen zu unterstützen.

Die geometrische Grundstruktur von UML-Klassendiagrammen ist, wie in Abschnitt 3.5 beschrieben, die eines hierarchischen Graphen. Zwar wurden Layoutalgorithmen für solche Strukturen in der Literatur bereits ausführlich behandelt; unser spezieller Anwendungsfall weist jedoch Besonderheiten auf, die ihn von den sonst dargestellten Beispielen unterscheiden.

Zunächst haben UML-Klassendiagramme zwar eine graph-artige Grundstruktur, sie sind jedoch in ihrer Visualisierung erheblich komplexer als die in der Literatur untersuchten Graphen: Knoten (z. B. Klassen) besitzen eine (variable) räumliche Ausdehnung und sollten sich nicht überlappen, sie haben auch eine interne Struktur mit verschiedenen Bestandteilen (Attribute, Stereotypen etc.), für deren Platzierung ebenfalls gesorgt werden muss. Zwischen zwei Knoten verlaufen oft mehrere Pfeile (Kanten), die auch sichtbar nebeneinander angeordnet werden sollten, diese Pfeile können verschiedene Textbeschriftungen besitzen und schließlich sind auch Schleifen von einem Knoten zu sich selbst zulässig.

Der zu realisierende Layoutalgorithmus soll natürlich nicht nur im Kontext eines speziellen Editors anwendbar sein, sondern er soll einfach anpassbar sein, damit er auch für ähnliche (oder einfachere) graph-artige Diagrammsprachen nutzbar ist. Solche Diagrammsprachen, die aus mehr oder weniger komplexen Objekten und Verbindungspfeilen dazwischen bestehen, kommen in der Praxis auch relativ häufig vor, man denke beispielsweise an Statecharts oder (evtl. hierarchische) Petri-Netze.

Wie erwähnt, soll das Layout flexibel sein und Benutzerwünsche berücksichtigen. Korrekturen sollen nur erfolgen, wenn sie für die Korrektheit des Diagramms notwendig sind. Dies stellt einerseits eine zusätzliche Anforderung dar, andererseits erleichtert es aber auch die Aufgabe der Layoutberechnung: Während in der Literatur beschriebene Graphlayout-Algorithmen fast immer nur von einer abstrakten Graphstruktur ausgehen und die Knotenpositionen völlig frei wählen (müssen), verfügen wir in unserem Anwendungsfall immer über ein Ausgangslayout, das nur korrigiert werden soll. Komplexe Optimierungen wie die Minimierung von Kantenkreuzungen, die das allgemeine Graphlayout-Problem so schwierig machen, können wir daher vernachlässigen; diese muss die Benutzerin selbst vornehmen. Eventuell könnten ja auch inhaltliche Kriterien, die dem Layoutmechanismus überhaupt nicht zugänglich sind, bei der Positionierung der Knoten den Ausschlag geben, so dass die Layoutkorrektur hier nicht unnötig eingreifen sollte.

## 6.5 Inkrementelles und vollständiges Layout

Das Layout muss schließlich auch schnell genug für interaktives Arbeiten sein. Um Richtwerte für die Geschwindigkeitsanforderungen geben zu können, müssen wir berücksichtigen, dass DiaGen-Editoren prinzipiell zwei Arten von Layoutberechnungen unterscheiden. Obwohl es möglich ist, für beide Arten denselben Layoutalgorithmus zu benutzen (wie dies z. B. beim oben beschriebenen Baumlayout geschieht), stellen sie doch unterschiedliche Anforderungen:

- Das *inkrementelle Layout* erfolgt, während die Benutzerin eine interaktive Editieroperation am Bildschirm durchführt, z. B. während sie eine Diagrammkomponente mit der Maus verschiebt. Um hier einen flüssigen Bewegungsablauf darstellen zu können, sollte ein solcher Layoutvorgang nicht länger als 100ms dauern (“no perceived delay”, [22]). Da während einer interaktiven Editieroperation keine erneute Diagrammanalyse erfolgt, kann der Layoutalgorithmus davon ausgehen, dass sich die abstrakte Struktur des Diagramms nicht geändert hat.
- Ein *vollständiges Layout* des Diagramms wird erst nach Abschluß einer interaktiven Editieroperation berechnet (d.h. beispielsweise wenn die verschobene Komponente, bzw. die Maustaste, losgelassen wurde). Nach der Ausführung einer programmgesteuerten Diagrammtransformation erfolgt ebenfalls ein vollständiges Layout. Vor einem solchen Layoutvorgang wird jedesmal eine komplette Diagrammanalyse durchgeführt und dabei erkannte Strukturänderungen müssen berücksichtigt werden. Dafür ist das vollständige Layout weniger zeitkritisch, da es seltener nötig ist und außerdem immer in Zusammenhang mit einer (zeitaufwendigen) Diagrammanalyse auftritt. Wir betrachten deshalb eine Berechnungsdauer von bis zu 1s (“repetitive task”, [22]) noch als tolerabel.

Für das Layout von UML-Diagrammen hat es sich als zweckmäßig erwiesen, tatsächlich zwischen diesen beiden Layoutvorgängen zu unterscheiden: Um das interaktive Layout möglichst effizient zu gestalten, werden dabei nur diejenigen Layoutkorrekturen ausgeführt, die unbedingt erforderlich sind. Dies sind:

- Verschieben von angehängten Pfeilenden, wenn ein Knoten bewegt wird,
- Verschieben von Pfeilbeschriftungen mit dem zugehörigen Pfeil,
- Verschieben des Knoteninhalts mit dem umgebenden Knoten,
- Verhindern, dass Klassenrahmen (oder ihre Segmente) zu klein werden, um die enthaltenen Elemente zu umschließen.<sup>2</sup>

Alle aufwendigeren Layoutvorgänge werden nur im Rahmen des vollständigen Layouts ausgeführt. Dies sind insbesondere:

- Vollständiges Layout der Knoteninhalte (z. B. Anordnung der Attribute in Zeilen),
- Vermeidung von Knotenüberlappungen unter Berücksichtigung der hierarchischen Knotenstruktur,
- Korrektur von Pfeilenden, die auf der falschen Seite des zugehörigen Knotens oder im Knoteninneren enden.

---

<sup>2</sup>Dies ließe sich auch für Paketrahmen realisieren, aber das Layout lässt es explizit zu, Paketrahmen so zu verkleinern, dass Klassen aus dem Paket herausfallen. Dies beeinträchtigt nämlich die Korrektheit des Diagramms nicht, während Klassenbestandteile nur innerhalb von Klassen auftreten dürfen.

## 6.6 Layout durch Constraint-Propagation

Wenn wir die Aufgaben des inkrementellen Layouts analysieren, erkennen wir, dass sich diese als ein System von funktionalen Abhängigkeiten der Layout-Attribute der Diagrammkomponenten ausdrücken lassen. Dabei fällt besonders auf, dass diese Abhängigkeiten immer nur in einer Richtung ausgewertet werden müssen und dass die Reihenfolge ihrer Berechnung explizit vorgegeben werden kann:

1. Abhängigkeiten von den Knotenpositionen zu den Positionen der Knoteninhalte
2. Abhängigkeiten von den Knotenpositionen zur Lage der Pfeilenden
3. Abhängigkeiten von der Lage der Pfeilenden zu den Positionen der Pfeilbeschriftungen

Diese Beobachtung führt zu der Idee, das inkrementelle Layout für den UML-Editor auf der Basis von Constraints zu spezifizieren, da ein constraint-basiertes Layout, wie oben gezeigt vor allem zur inkrementellen Anpassung besonders geeignet ist. Die genannten Eigenschaften machen es nämlich möglich, sich auf einen ausgesprochen primitiven und dementsprechend schnellen Mechanismus zur Propagierung funktionaler Abhängigkeiten zu beschränken, der in weniger als 100 Zeilen Programmcode implementiert wurde und sich auf folgende Funktionalität beschränkt:

Gegeben:

- eine Menge  $V$  von Variablen (reellwertige Layout-Attribute)
- eine geordnete Folge von funktionalen Abhängigkeiten zwischen diesen Variablen und dazugehörigen Berechnungsvorschriften

Eingabe: Eine Menge  $C \subset V$  von geänderten Variablen

Berechnung:

Für alle Abhängigkeiten in ihrer vorgegebenen Reihenfolge  
 Sei  $I$  die Menge der Eingangsvariablen der Abhängigkeit  
 Wenn  $C \cap I \neq \emptyset$   
     Berechne die Ausgangsvariable  $o$  durch Anwendung  
     der Berechnungsvorschrift  
 Setze  $C = C \cup \{o\}$

Der Algorithmus ist also kein Constraint-Solver, sondern lediglich ein Mechanismus zur Wertepropagierung, da es ja vorkommen kann, dass der Effekt von bereits ausgewerteten Abhängigkeiten durch später berechnete wieder überschrieben wird. Die Programmiererin muss deshalb mit der Festlegung der Auswertungsreihenfolge dafür Sorge tragen, dass dies nicht ungewollt geschieht. Da für

jede Abhängigkeit nur eine Berechnungsvorschrift zulässig ist und Eingangs- sowie Ausgangsvariablen explizit festgelegt werden müssen (“One-Way-Constraints” [16]), ist es unter Umständen auch notwendig, Abhängigkeiten aufzuspalten, welche in mehreren Richtungen berechnet werden können (je nachdem, welche der beteiligten Variablen sich verändert hat, “Multi-Way-Constraints”). Diese müssen dann durch mehrere Abhängigkeiten, eine für jede Auswertungsrichtung, repräsentiert werden.

Der zusätzliche Programmieraufwand wird dafür durch einen entsprechenden Geschwindigkeitszuwachs ausgeglichen, da eine aufwendige Planung der Auswertungsreihenfolge und -richtung von Abhängigkeiten, wie er bei echten Constraint-Solvern (vgl. [16]) nötig ist, entfallen kann. Außerdem können auch leistungsfähigere Systeme nicht verhindern, dass sich die Programmiererin explizit über Prioritäten bei der Auswertung von Constraints Gedanken machen muss (vgl. S. 70), daher scheint es durchaus vernünftig, zu verlangen, dass die Auswertungsreihenfolge explizit vorgegeben werden muss.

Zirkuläre Abhängigkeiten dürfen dabei auch auftreten; sie werden dann natürlich nicht korrekt gelöst – dies ist überhaupt auch nur bei einer Einschränkung der Constraints auf lineare (Un)gleichungen sinnvoll möglich und ausgesprochen ineffizient – sondern nur in der vorgegebenen Reihenfolge maximal einmal ausgewertet; das kann aber durchaus ein erwünschtes Verhalten sein.

In der objektorientierten Implementierung dieses Algorithmus sind Variablen-Abhängigkeiten als abstrakte Klassen realisiert und Berechnungsvorschriften als davon abgeleitete konkrete Klassen. Für die Formulierung der Berechnungsvorschriften steht deshalb der volle Sprachumfang der verwendeten Programmiersprache (Java) zur Verfügung, insbesondere lassen sich auch nichtlineare Abhängigkeiten ausdrücken, wie sie in geometrischen Anwendungen z. B. bei Abstandsberechnungen sehr häufig auftreten. Constraints, die Ungleichungen beschreiben, lassen sich als bedingte Ausdrücke (min- oder max-Funktion) realisieren. Durch den flexiblen Grundmechanismus der Propagierung von Abhängigkeiten dürfte es verhältnismäßig einfach sein, den Layoutalgorithmus für UML-Diagramme, wie gefordert, für ähnliche Diagrammsprachen anzupassen, die dann natürlich z. B. ein anderes internes Layout der Graph-Knoten erfordern würden.

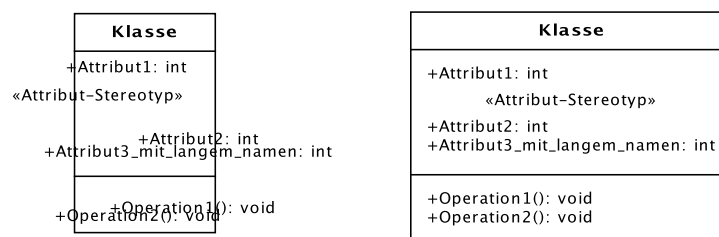
Es hat sich herausgestellt, dass das skizzierte primitive System zur Propagierung von Abhängigkeiten tatsächlich ausreicht, um beim inkrementellen Layout von UML-Diagrammen befriedigende Ergebnisse zu erzielen. Die Einschränkung auf One-Way-Constraints macht es sogar möglich, Effekte zu erzielen, die mit Multi-Way-Constraints nicht realisierbar wären: Beispielsweise existiert zwar eine Abhängigkeit von der Position eines Klassenrahmens zu den Positionen der Attribute der Klasse, nicht jedoch in umgekehrter Richtung. Das bedeutet, dass sich die Attribute beim Verschieben des Rahmens mit diesem bewegen; wenn jedoch die Attribute selbst verschoben werden, bewegt sich der Klassenrahmen nicht mit. Dadurch ist es auf einfache Weise möglich, Attribute von einer Klasse zu

einer anderen zu ziehen; beim anschließenden vollständigen Layout wird dann die Größe der beteiligten Klassen entsprechend angepaßt. Bei der Verwendung von Multi-Way-Constraints wäre diese Form von "Drag & Drop" nicht möglich; hier müssten die Attribute relativ zum Klassenrahmen fixiert werden.

## 6.7 Erzeugung der Constraints

Das vollständige Layout des Diagramms muss, wie eingangs erwähnt, erheblich kompliziertere Aufgaben übernehmen als das inkrementelle Layout, deshalb haben wir uns entschieden, diesen Teil des Layoutvorgangs algorithmisch zu spezifizieren, d.h. explizit zu programmieren. Tatsächlich ist es auch Aufgabe des vollständigen Layouts, das Abhängigkeitssystem für das inkrementelle Layout überhaupt erst zu erzeugen.

Dies sieht z. B. beim internen Layout von UML-Klassen so aus, dass durch die Diagrammanalyse zunächst einmal nur die Information zur Verfügung steht, welche Attribute zu einer Klasse gehören und in welcher vertikalen Reihenfolge sie angeordnet sind. Der Layoutalgorithmus konstruiert nun daraus ein System von Abhängigkeiten, welches die Positionen der Attribute aus der Position des Klassenrahmens so berechnet, dass sie passend in gleichmäßigem Abstand innerhalb des Rahmens ausgerichtet werden. Dieses Abhängigkeitssystem wird dann einmal vollständig ausgewertet (d.h. der beschriebene Algorithmus wird so ausgeführt, als ob alle Eingangsvariablen geändert worden wären), so dass die Attribute korrekt platziert sind. Das Layout von Pfeilen und Pfeilbeschriftungen erfolgt analog. Nach Abschluß des im Folgenden beschriebenen hierarchischen Knotenlayouts muss das erzeugte Abhängigkeitssystem für das inkrementelle Layout dann nochmals ausgewertet werden, um Knoteninhalte und Pfeile an die möglicherweise veränderten Knotenpositionen anzupassen.



*Abbildung 6.2: Vollständiges Layout einer Klasse*

Abbildung 6.2 zeigt, wie das Erscheinungsbild einer Klasse durch Constraints korrigiert wird, wenn die Bestandteile von der Benutzerin nicht exakt positioniert worden sind. Das dabei generierte Abhängigkeitssystem sieht folgendermaßen aus:

$$\text{attr}_1.x = \text{klasse}.x + [\text{rand}]$$

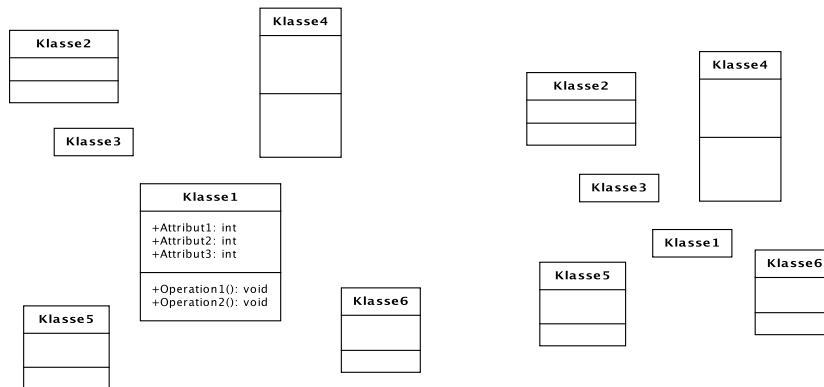


Abbildung 6.3: Layoutkorrektur beim Verkleinern einer Klasse

$$\begin{aligned}
 \text{attr}_1.y &= \text{klasse.y1} + [\text{rand}] \\
 \text{stype.x} &= \text{klasse.x} + 0.5 \cdot \text{klasse.w} \\
 \text{stype.y} &= \text{klasse.y1} + [\text{rand} + \text{attr}_1.h + \text{zeilenabst}] \\
 \text{attr}_2.x &= \text{klasse.x} + [\text{rand}] \\
 \text{attr}_2.y &= \text{klasse.y1} + [\text{rand} + \text{attr}_1.h + \text{stype.h} + 2 \cdot \text{zeilenabst}] \\
 &\dots \\
 \text{klasse.w} &= \max(\text{klasse.w}, [\max(\text{attr}_n.\text{breite}) + 2 \cdot \text{rand}]) \\
 \text{klasse.h}_1 &= \max(\text{klasse.h}_1, [\text{sum}(\text{attr}_n.h + 2 \cdot \text{rand} + n \cdot \text{zeilenabst}])
 \end{aligned}$$

(Die Teilausdrücke in eckigen Klammern werden dabei schon bei der Generierung des Abhängigkeitssystems zu Konstanten ausgewertet.)

## 6.8 Layout durch Kräftesimulation

Das schwierigste Teilproblem bei der Implementierung des Layouts für den UML-Editor bestand in der Entwicklung einer geeigneten Technik zur Knotenpositionierung, die Überlappungen von Knoten verhindern soll und auch Größenänderungen der Knoten durch Zoom-Transformationen angemessen berücksichtigen kann. Dazu müssen die Knoten – falls notwendig – geeignet auseinandergeschoben oder zusammengezogen werden, sie sollen jedoch, entsprechend den oben formulierten Anforderungen, ihre relative Lage zueinander möglichst beibehalten, um keine unnötigen Veränderungen der Diagrammstruktur zu bewirken. Intuitiv kann man sich leicht vorstellen, welche Auswirkungen eine derartige Layoutkorrektur haben sollte; Abbildung 6.3 zeigt als Beispiel das Abstrahieren (d.h. Verkleinern) einer Klasse.

Wie lässt sich nun ein derartiges Verhalten als Algorithmus formulieren? Eine erprobte Technik zur Positionierung von Knoten eines Graphen, die sich für diese Zwecke eignet, ist das Layout durch Simulation von Kraftfeldern (“Force-Directed Layout” [18]). Die Grundidee dieser Layout-Technik ist es, die Knoten des

Graphen als bewegliche Objekte zu betrachten, die aufeinander Anziehungs- und Abstoßungskräfte ausüben. Unter dem Einfluß dieser Kräfte werden die Knoten iterativ bewegt, was zu einem Layout führen soll, bei dem die Kräfte insgesamt minimiert werden.

In der einfachsten Form des Algorithmus werden die Knoten nur als Punkte betrachtet; zwischen zwei mit einer Kante verbundenen Knoten wirken Kräfte, die nur bei einer (vorgegebenen) optimalen Kantenlänge verschwinden. Sind die Knoten weiter voneinander entfernt, so ziehen sie sich mit steigendem Abstand immer stärker an; wird dagegen die optimale Entfernung unterschritten, dann stoßen sie sich um so stärker ab, je näher sie einander sind.

Zwischen allen übrigen Knotenpaaren wirken noch betragsmäßig schwächere Abstoßungskräfte, die dazu führen sollen, dass der Graph nicht "verklumpt" sondern sich so weit in der Fläche ausbreitet, wie es die Kantenverbindungen zulassen. Wenn man nun, ausgehend von einer zufälligen Anfangsposition der Knoten, durch ausreichend lange Iteration die Kräfte minimiert, so sollte im Ergebnis ein zufriedenstellendes Layout des Graphen erreicht werden.

Dieser ursprüngliche Algorithmus wurde in vieler Hinsicht verbessert und erweitert; so kann man beispielsweise eine bevorzugte Orientierung für manche Kanten realisieren [19] oder statt des beschriebenen simplen Gradientenabstiegs zur Minimierung der Kräfte auf effizientere Optimierungstechniken wie das "Simulated Annealing" zurückgreifen [20]. Brandenburg et al. [18] bieten eine Übersicht und einen Vergleich von verschiedenen Variationen von Kraftsimulationstechniken zum Graph-Layout.

Das Prinzip der Kräftesimulation wurde vor allem in der beschriebenen Form zum Layout von Graphen benutzt, von denen nur die abstrakte Struktur bekannt ist und zufällige Ausgangspositionen gewählt werden müssen. Dabei hat diese Technik mit allen Problemen zu kämpfen, mit denen heuristische Optimierungsverfahren grundsätzlich behaftet sind; so sind die Ergebnisse nach vielen Iterationen nur schlecht vorhersehbar und der Algorithmus tendiert dazu, in Nebenminima steckenzubleiben, so dass z. B. Kreuzungen des Graphen nicht gut minimiert werden.

Diese Probleme treten aber in den Hintergrund, sobald es nicht mehr um die Generierung eines Layouts ohne Vorinformationen geht ("Layout Creation"), sondern lediglich um die Korrektur eines bereits vorliegenden Layouts des Graphen ("Layout Adjustment", vgl. [21]), wie im Fall von Diagrammeditoren. Hier steht ja schon ein weitgehend geeignetes Ausgangslayout zur Verfügung und schwierige Optimierungen wie das Minimieren von Kantenkreuzungen müssen nicht erfolgen, sondern können der Benutzerin überlassen werden.

Eine große Stärke der Kraftsimulationstechnik ist es, dass sie einfach zu implementieren ist und ein sehr flexibles Grundgerüst bietet, um durch geeignete Wahl der Kräfte unterschiedliche Effekte zu erzielen. Beim Layout von graph-artigen Diagrammen können die Kräfte beliebige Eigenschaften der Diagrammelemente

(Größe der beteiligten Knoten, Art der Verbindung) berücksichtigen. Auch kann man, falls gewünscht, bestimmte Knoten fixieren und nur einen Teil des Graphen durch die Krafteinwirkung verschieben lassen.

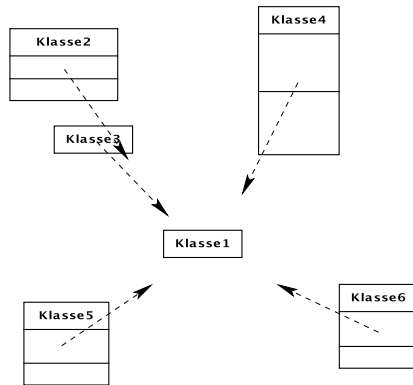


Abbildung 6.4: Kräfte für eine Zoom-Korrektur

Es leuchtet z. B. unmittelbar ein, dass die Art, wie sich UML-Diagramme bei Zoom-Korrekturen zusammenziehen oder expandieren sollen, gut mit Hilfe solcher Kraftfelder zu beschreiben ist. Abbildung 6.4 zeigt, wie durch Anziehungskräfte die in Abbildung 6.3 dargestellte "kontrahierende" Zoom-Korrektur erreicht wird.

Der Layoutalgorithmus für UML-Klassendiagramme verwendet die Kräftesimulation außerdem, um Überlappungen von Knoten zu vermeiden. Da diese Funktionalität auch bei der kontrahierenden Zoom-Korrektur benötigt wird, soll das entsprechende Kraftsystem zuerst besprochen werden.

## 6.9 Ein Kraftsystem zu Abstandskorrektur von Knoten

Ziel der beschriebenen Layoutkorrektur ist es, die Knoten unter weitgehender Beibehaltung ihrer relativen Lage so zu positionieren, dass zwischen zwei Knoten ein benutzerdefinierter Mindestabstand  $d$  eingehalten wird. Offensichtlich müssen hierbei Abstoßungskräfte zwischen benachbarten Knotenpaaren eingesetzt werden, die um so stärker sind, je weiter der Mindestabstand unterschritten wird. Wenn die Knoten dagegen den Mindestabstand einhalten, sollten keine Kräfte wirksam sein, um unnötige Layoutveränderungen zu vermeiden. Dadurch ist es auch möglich, die Kräfte einfach zwischen allen Knotenpaaren zu berechnen (für weit auseinanderliegende Paare ist ihr Betrag dann 0) und sich so die Bestimmung benachbarter Knoten zu ersparen.

Dagegen ist es unbedingt nötig, die Knoten nicht nur als Punkte zu behandeln, sondern auch ihre Größe zu berücksichtigen, da diese sehr stark variieren kann. Die Knotenform wird dabei durch (umschließende) Rechtecke angenähert, wobei alle Kräfte im Mittelpunkt des Rechtecks angreifen. Misue et al. [21] schlagen vor, die Rechtecke nur auf der Geraden zwischen ihren Mittelpunkten zu verschieben, um die Proportionen des Layouts beizubehalten. Wenn die am nächsten beieinanderliegenden Kanten den Abstand  $d$  haben sollen, berechnet sich der optimale Abstand  $d_o$  der Mittelpunkte auf der (durch den Einheitsvektor  $(u_x, u_y)$  bestimmten) Geraden folgendermaßen:

$$d_o = \min \left( \frac{1}{u_x} \left( \frac{w_1 + w_2}{2} + d \right), \frac{1}{u_y} \left( \frac{h_1 + h_2}{2} + d \right) \right)$$

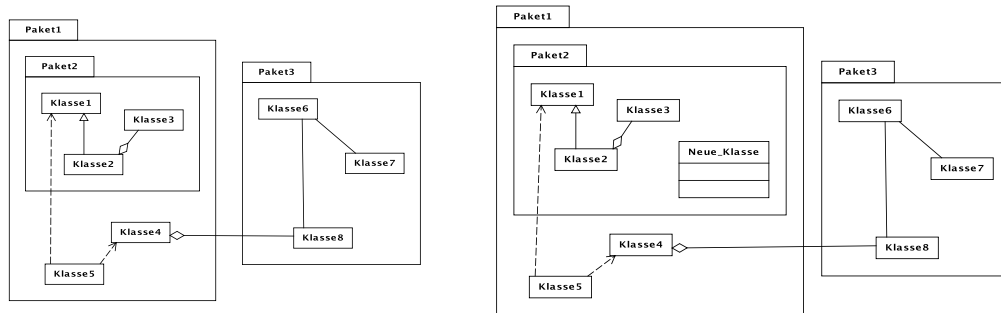


Abbildung 6.5: Berücksichtigung der Pakethierarchie beim Layout

( $w_{1,2}$  und  $h_{1,2}$  entsprechen dabei der Breite bzw. Höhe der beteiligten Knoten.)

Die notwendige Abstoßungskraft berechnet sich dann einfach aus einem Skalierungsfaktor und der Differenz zwischen aktuellem Abstand  $d_a$  und dem optimalen Abstand, falls dieser zu klein sein sollte:  $f = s \cdot \max(d_a - d_o, 0)$

Die Layoutkorrektur iteriert nun über alle Knotenpaare und summiert für jeden Knoten alle berechneten Kräfte. Anschließend wird jeder Knoten um den skalierten Summen-Kraftvektor verschoben. Kraftberechnung und Verschiebung der Knoten werden solange wiederholt, bis alle Knotenpaare den Mindestabstand  $d$  einhalten. Dieser Mindestabstand wird bei der Kraftberechnung noch mit einem Faktor  $\varepsilon > 1$  multipliziert, da die Iteration des Layouts sonst nur asymptotisch konvergieren würde. Es hat sich herausgestellt, dass bei geeigneter Skalierung der Kräfte im Allgemeinen 5–10 Iterationen zu einem befriedigenden Ergebnis führen.<sup>3</sup> Ob zwischen bestimmten Knotenpaaren Kanten existieren, wird in der hier beschriebenen Anwendung völlig vernachlässigt, da auch ohne Verwendung dieser Information zufriedenstellende Ergebnisse erzielt werden konnten.

Dieser Algorithmus kann bei der hierarchischen Graphstruktur der UML-Klassendiagramme nicht einfach auf alle Knoten gleichzeitig angewandt werden. Stattdessen erfolgt das Layout im Hierarchiebaum aufsteigend einzeln für jede Gruppe von Knoten, die direkt im selben Knoten der nächsthöheren Hierarchieebene enthalten sind. Die relativen Positionen werden in jeder Gruppe anschließend fixiert und das später folgende Layout der nächsthöheren Ebene kann nur noch den umschließenden Knoten als Ganzes verschieben.

Damit wird erreicht, dass Veränderungen des Diagramms sich nur auf das interne Layout von direkt betroffenen Elementen auswirken. Wird beispielsweise eine Klasse in ein Paket eingefügt, so kann das dazu führen, dass die Fläche des Paketrahmens vergrößert werden muss und benachbarte Pakete zur Seite geschoben werden müssen. Die Form und das interne Layout dieser Pakete wird dabei aber nicht verändert (vgl. Abb. 6.5).

<sup>3</sup>Bei zu großen Kräften und weniger Iterationen kann es dagegen vorkommen, dass die einzelnen Schritte "über das Ziel hinausschießen" und Knoten ihre relative Lage zu stark verändern.

## 6.10 Ein Kraftsystem zur Zoom-Korrektur

Das Kraftsystem zur Berechnung von Zoom-Korrekturen muss demgegenüber anders aufgebaut werden. Naheliegenderweise wirken Anziehungskräfte von “geschrumpften” Knoten auf andere und Abstoßungskräfte von “gewachsenen”. Da sich beim Abstrahieren die Gesamtfläche des Diagramms verkleinern sollte, um den gewünschten semantischen Fokus-und-Kontext-Effekt durch Erhöhen der Vergrößerungsstufe erzielen zu können (vgl. S. 18), wirken die Kräfte gleichmäßig radial auf alle anderen Knoten, unabhängig von deren Entfernung. Der jeweilige Betrag muss proportional zur Größenänderung des kontrahierten Knotens sein. Das Kraftfeld wird insgesamt folgendermaßen berechnet:

Gegeben:

- eine Menge  $N$  von Knoten
- eine Menge  $K \subset N$  von kontrahierten Knoten

Berechnung:

Für jedes Paar  $(n_1 \in K, n_2 \in N)$

Falls  $n_1 \neq n_2$

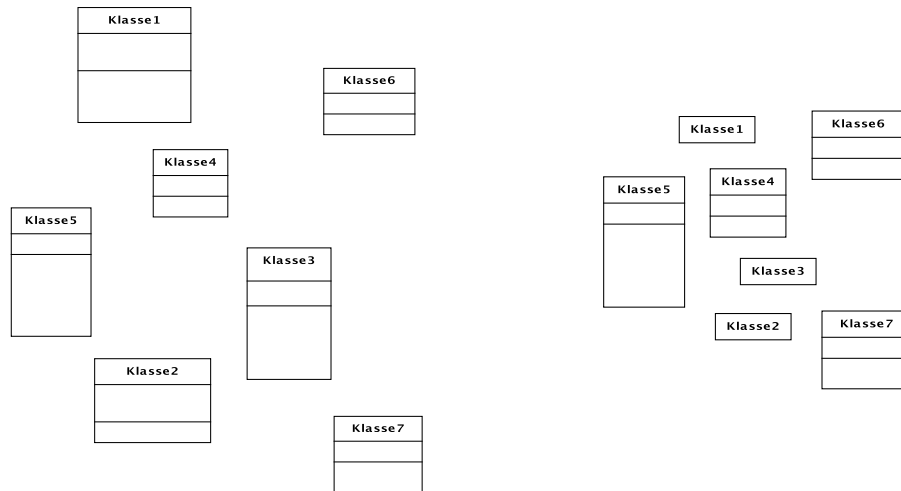
Lasse  $f = \text{oldsize}(n_1) - \text{size}(n_1)$  auf  $n_2$  wirken.

Dabei ist “size” eine geeignete Größenfunktion für rechteckige Knoten (z. B. ihre Diagonale); “oldsize” liefert die vorherige Größe eines nun kontrahierten Knotens.

Kräfte wirken sich nur von einem Knoten in  $K$  auf alle anderen aus (auch auf die anderen Elemente von  $K$ !) und nicht umgekehrt – im Gegensatz zum Kraftfeld bei der Abstandskorrektur, bei dem die Abstoßung zwischen jedem Knotenpaar immer beide beteiligten Knoten betrifft. Ansonsten würden sich die Knoten aus  $K$ , die mit allen Knoten aus  $N$  in Wechselwirkung stehen, viel stärker bewegen als die aus  $N \setminus K$ , die nur von den wenigen Knoten aus  $K$  beeinflusst werden, was ungewollte Effekte zur Folge hätte.

Die berechneten Anziehungskräfte werden auch nicht nur einmal angewandt, sondern geeignet skaliert und dafür mehrfach iteriert, um beim Zusammenwirken mehrerer Kontraktionsknoten einen glatten Verlauf zu erzielen. Das Zusammenziehen des Layouts soll natürlich auch nicht dazu führen, dass der Mindestabstand zwischen zwei Knoten unterschritten wird, deshalb werden die beschriebenen Kontraktionskräfte zusätzlich bei jeder Iteration mit Abstoßungskräften kombiniert, die nach dem oben beschriebenen Prinzip zur Abstandskorrektur berechnet werden.

Den Effekt einer gleichzeitigen Kontraktion von drei Layoutobjekten sehen wir in Abbildung 6.6: Alle Klassen behalten ihre relative Lage ungefähr bei, aber der durch die Kontraktion freiwerdende Platz wird genutzt, um das Diagramm zu



**Abbildung 6.6:** Layoutkorrektur bei gleichzeitiger Abstraktion mehrerer Klassen

kompaktieren. Auch diese Layoutkorrekturen bei Zoom-Transformationen erfolgen gegebenenfalls nach dem oben beschriebenen hierarchischen Schema. Beim "Vergrößern" von Diagrammelementen kann genau derselbe Mechanismus eingesetzt werden; das geänderte Vorzeichen der Größendifferenz führt dann automatisch zu Abstoßungskräften.

Die vorgestellte Verwendung der Kräftesimulation hat dabei längst nicht alle Möglichkeiten ausgereizt, die die zugrundeliegende Technik zum Layout von UML-Klassendiagrammen bietet. Um das Layout weiter zu verbessern, könnte man zusätzlich bei der Berechnung der Kräfte berücksichtigen, welche Knoten durch Kanten verbunden sind. Auch die Verwendung von Variationen, die eine bevorzugte Orientierung bestimmter Kanten bewirken [19] ist denkbar (Generalisierungspfeile sollten z. B. nach einer üblichen Konvention von unten nach oben verlaufen), oder die Möglichkeit zum expliziten Aufrufen von Kraftfeld-Iterationen zur Korrektur ausgewählter Diagrammteile (als Diagrammtransformation). Hier bietet sich also noch ein weites Feld für zukünftige Erweiterungen.

Zur Implementierung des kompletten Layoutalgorithmus für hierarchische graph-artige Diagramme muss abschließend noch ergänzt werden, dass die beschriebenen Basistechniken durch eine "Fassade" aus Objekten höherer Abstraktionsstufe – Layout-Knoten, Pfeilen und Pfeilbeschriftungen – verborgen werden. Um den Layoutalgorithmus bei der Realisierung eines Diagrammeditors verwenden zu können, muss die Programmiererin sich deshalb nicht direkt mit diesen tiefen Schichten der Implementierung befassen, sondern kann das Layout auf der Ebene dieser Objekte spezifizieren. Zur Anpassung an eine konkrete Diagrammsprache müssen nur wenige abstrakte Methoden definiert werden, welche Form und internes Layout der Knoten und die Projektion von Pfeilenden auf den Knotenrand berechnen.

## Kapitel 7

# Vereinfachte Benutzung von Zoom-Transformationen

In den vorangehenden Kapiteln haben wir dargestellt, wie sich eine semantische Fokus-und-Kontext-Darstellung in DiaGen-Editoren durch Zoom-Transformationen realisieren lässt. Mit solchen Transformationen lassen sich Diagrammbereiche flexibel abstrahieren und wieder einblenden; um die gewünschte Darstellung zu erreichen, muss die Benutzerin allerdings meist mehrere Transformationen ausführen und alle davon betroffenen Bereiche von Hand selektieren. Im Folgenden Kapitel soll nun noch eine Benutzungsschnittstellen-Komponente vorgestellt werden, welche für typische Anwendungsfälle einen komfortablen Weg bietet, das Diagramm durch die Ausführung von einer oder mehreren Zoom-Transformationen zu verändern.

### 7.1 Strukturbäume

Um Zoom-Transformationen manuell oder automatisch in geeigneter Weise kombinieren zu können, ist zunächst eine passende Sicht auf das Diagramm notwendig. Wenn die Transformationen zur Abstraktion von hierarchisch strukturierten Diagrammbereichen verwendet werden sollen, bietet sich dazu eine baumartige Repräsentation der Struktur des Diagramms an. Ein solcher "Strukturbaum" bildet deshalb die Basis der vorgestellten Schnittstellen-Komponente; er stellt die hierarchische Ordnung der Diagrammteile dar.

Im Fall von Baumdiagrammen ist der Strukturbaum natürlich äquivalent zum tatsächlichen Diagramm (bzw. seinen korrekten Teilen), da die Beziehung von über- und untergeordneten Diagrammteilen (Teilbäumen) ja aus ihrer Baumanordnung abgeleitet wird. Interessanter ist der Fall bei UML-Diagrammen; hier betrachten wir Attribute/Operationen, Klassen, Pakete und schließlich das Gesamtmodell als hierarchische Einheiten, deren Ordnung dadurch ausgedrückt ist, dass diese Elemente in anderen enthalten sind. Die Abbildung dieser Hierarchie

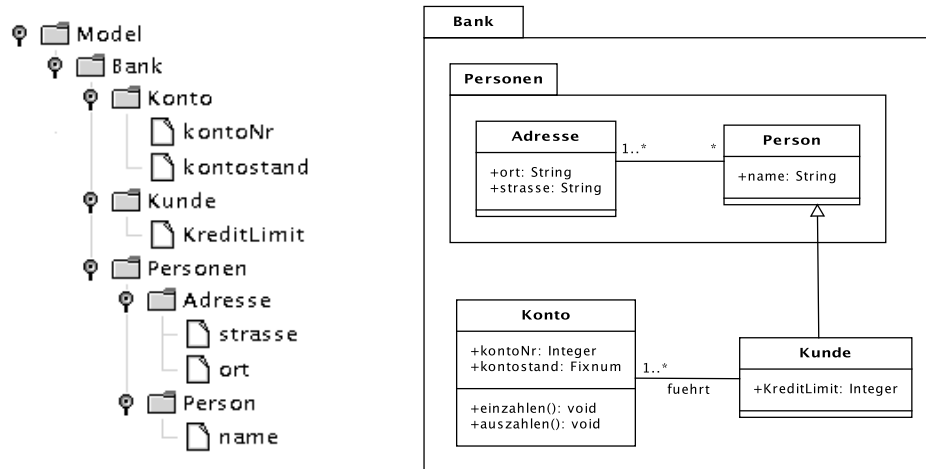


Abbildung 7.1: Baumstruktur eines UML-Diagramms

auf eine Baumstruktur bietet eine völlig andere Sicht des Diagramms. Abbildung 7.1 zeigt ein Beispiel. (Pfeile passen nicht in die gewünschte hierarchische Ordnung und können deshalb im Baum nicht dargestellt werden.) Bei anderen Diagrammtypen orientiert sich die hierarchische Ordnung normalerweise ebenfalls am "Enthalten-Sein" von Diagrammkomponenten.

Strukturbäume werden auch im System als baumartige Objektstruktur repräsentiert. Der Strukturbaum kann als eigenes Element der Benutzungsschnittstelle visualisiert werden; Java stellt dazu eine fertige Komponente des Swing-Toolkits zur Verfügung. Wie diese Komponente in die Benutzungsschnittstelle des Editor eingebunden ist, bleibt der Programmiererin überlassen; möglich ist z. B. eine Aufteilung des Editorfensters (wie im UML-Editor, vgl. Abb. 7.2) oder die Verwendung eines eigenen Bildschirmfensters. Da derartige Baum-Elemente zum Standard-Repertoire jeder grafischen Oberfläche gehören, kann davon ausgegangen werden, dass die Benutzerin mit ihrer Bedienung (die z. B. auch das Aus- und Einblenden von Teilbäumen erlaubt) vertraut ist.

Jedem Knoten des Strukturbaumes muss genau eine Komponente des Diagramms zugeordnet werden, welche die entsprechende Abstraktionsstufe grafisch repräsentiert. Diese Komponente wird im Folgenden als "Hauptkomponente" bezeichnet; bei UML-Diagrammen sind das beispielsweise Klassenrahmen und Paketrahmen (ohne die jeweiligen Inhalte). Wenn die Benutzerin eine solche Diagrammkomponente selektiert, wird auch die Selektion der Baumdarstellung entsprechend angepasst und umgekehrt, wie in Abbildung 7.2 zu sehen ist. Damit dient eine solche visuelle Baum-Sicht des Diagramms auch als Navigations- und Orientierungshilfe, wie sie auch bei anderen UML-Tools zur Verfügung steht (vgl. S. 10).

Alle im Folgenden vorgestellten Modifikationen des Diagramms, die mit Hilfe des Strukturbaums durchgeführt werden, müssen natürlich auch in der Benutzungsschnittstelle zugänglich sein. Wenn der Strukturbaum als Swing-Komponente visualisiert wird, dann können sie durch das Kontextmenü dieser Komponente aufgerufen werden. Dasselbe Menü kann auch in die Hauptmenüleiste des Editors integriert werden. Da Knoten eines Strukturbaums nicht nur direkt ausgewählt werden können, sondern auch implizit durch Selektion der Hauptkomponente im Diagramm, kann der Strukturbaum auch zur Diagrammbearbeitung genutzt werden, wenn er zwar in den Editor eingebunden, aber nicht visualisiert ist, z. B. weil die Benutzerin ein entsprechendes Bildschirmfenster geschlossen hat.

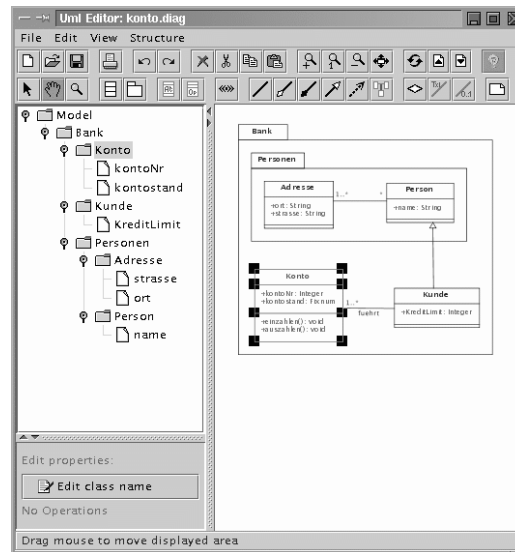


Abbildung 7.2: Benutzungsschnittstelle des UML-Editors mit Strukturbaum

Es gibt mehrere Möglichkeiten, einen Strukturbaum für ein Diagramm zu generieren. Wenn die von der Diagrammanalyse erzeugte “Semantik” die erforderlichen Informationen enthält, kann der Strukturbaum direkt aus ihr erzeugt werden; beispielsweise bildet der Editor für Baumdiagramme diese intern auf Bäume von Objekten ab, so dass zu jedem Knoten dieses Objektbaums auch ein Knoten im Strukturbaum generiert wird. Im UML-Editor wird der Strukturbaum aus dem ASG durch baumförmiges Traversieren erzeugt. Bei Änderungen am Diagramm veranlassen entsprechende Benachrichtigungen (“Events”) des Analyse-Moduls, dass der Strukturbaum automatisch neu generiert wird.

Wenn eine so enge Verknüpfung von Diagramm-“Semantik” und Strukturbaum nicht möglich oder nicht wünschenswert ist, z. B. weil die Hierarchie der Grammatik nicht genau der gewünschten Struktur-Hierarchie entspricht, dann könnte ein Strukturbaum auch getrennt von der eigentlichen “Semantik” durch das Diagrammanalyse-Modul konstruiert werden; die Auswertung der Attribute von Grammatik-Produktionen in DiaGen stellt dazu einen flexiblen Mechanismus bereit (vgl. [32]).

## 7.2 Operationen auf Baumknoten

Um den Strukturbaum zum Bearbeiten nutzen zu können, muss jeder Knoten drei Basis-Operationen bereitstellen:

1. Die Auswahl des vollständigen Diagrammbereichs (im HGM), der seinem Abstraktionsniveau entspricht
2. die Abstraktion dieses Bereichs, d. h. das Ausblenden aller Details
3. das Wiedereinblenden dieses Bereichs, wenn er zuvor abstrahiert wurde

Die zweite Operation wird dabei normalerweise auf die erste zurückgreifen.

Zur Auswahl des Abstraktionsbereichs werden für jeden Knoten ein oder mehrere Regelmuster ("Auswahlmuster") definiert. Alle Passungen dieser Muster, die im HGM mit der Hauptkomponente des Knotens als Vorgabe gefunden werden können, bilden zusammen den Abstraktionsbereich. Dabei kann normalerweise das auf Seite 54 skizzierte Schema angewandt werden, so dass diese Muster einfach zu beschreiben sind; wenn bereits Zoom-Transformationen für eine Diagrammsprache definiert sind, dann können die Muster der entsprechenden Regeln direkt bei der Konstruktion des Strukturbaums als Auswahlmuster angegeben werden.<sup>1</sup>

Wenn die Auswahl des Diagrammbereichs genau dem Basis-Schema folgt, dann kann sogar auf die explizite Definition von Auswahlregeln verzichtet werden: In diesem Fall muss nur jedem Knoten des Strukturbaums ein Nichtterminal-Symbol zugeordnet werden; mit dessen Hilfe kann dann ohne weitere Programmierung ein automatisches "Default"-Selektionsmuster benutzt werden, das den kontextfrei aus diesem Nichtterminal abgeleiteten Diagrammbereich sowie alle vollständig darin eingebetteten Teilbäume (vgl. Abb. 3.4) auswählt. Für UML-Diagramme ist dieses einfache Schema leider nicht immer ausreichend, daher müssen die Auswahlmuster in der Spezifikation durch Transformationsregeln definiert werden. Bei der Definition der Auswahlmuster muss darauf geachtet werden, dass die selektierten Bereiche konsistent mit der Hierarchie des Strukturbaums sind: Der zu einem Knoten gehörige Diagrammbereich (die HGM-Hyperkanten) muss immer eine Teilmenge vom Bereich seines Elternknotens sein. (Die weiter unten beschriebenen zusammengesetzten Zoom-Transformationen gehen von dieser Annahme aus.)

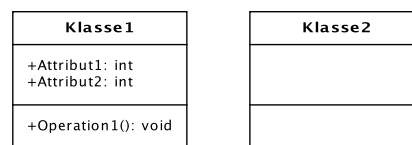
Die erste Erleichterung, die der Strukturbaum somit bei der Bedienung des Editors bietet, ist eine einfache Möglichkeit zur Auswahl einer hierarchischen Einheit im Diagramm. So können z. B. mit einem Mausklick alle zu einem Paket gehörigen Diagrammkomponenten selektiert werden. Der so selektierte Diagrammbereich kann dann mit der "Cut & Paste"-Funktionalität von DiaGen weiter bearbeitet werden.

Zusätzlich zu den Auswahlmustern müssen für jeden Knoten des Strukturbaums noch zwei Zoom-Transformationen zum Aus- und Einblenden des entsprechenden Diagrammbereichs definiert werden. Die Benutzerin kann so einen Kno-

<sup>1</sup>Transformationsregeln sind im DiaGen-System als Java-Objekte repräsentiert, deren Definitionen vom DiaGen-Generator erzeugt werden. Deshalb können Regeln und Regelmuster auch im Code referenziert und an Methodenaufrufe übergeben werden, und zusätzliche Programm-Module wie das Strukturbaum-Modul können sie genauso verwenden wie das DiaGen-Kernsystem.

ten des Strukturbaums auswählen und alle darunterliegenden Details “einfalten” bzw. wieder darstellen. Ausgeblendete Teile bleiben dabei im Strukturbaum erhalten, obwohl die entsprechenden Diagrammkomponenten aus dem Diagramm entfernt werden. Auch für diese Operationen stehen automatisch definierte “Default”-Transformationen zur Verfügung: Die Standard-Abstraktionstransformation blendet den gesamten von den Auswahlmustern selektierten Diagrammbereich aus; nur die Hauptkomponente des Baumknotens bleibt sichtbar und dient als Anker (vgl. S. 49). Die Standard-Transformation zum Wiedereinblenden fügt entsprechend alle ausgeblendeten Diagrammteile mit der Hauptkomponente als Anker wieder ein.

Abbildung 7.3 zeigt den Effekt, den die Standard-Transformation auf eine UML-Klasse haben würde: alle Bestandteile der Klasse werden ausgeblendet, aber der Klassenrahmen selbst bleibt unverändert. Eine automatische visuelle Markierung der abstrahierten Komponente findet so nicht statt; dazu müssen die Zoom-Transformationen explizit programmiert werden, wie dies im UML-Editor auch geschieht.<sup>2</sup>



**Abbildung 7.3:** Anwendung der Standard-Abstraktionstransformation auf eine UML-Klasse

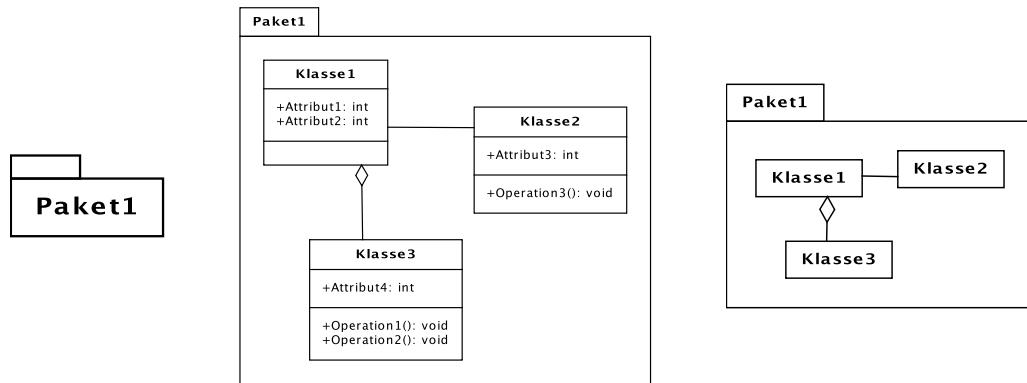
## 7.3 Kombination von Zoom-Transformationen

Bislang haben wir beschrieben, wie Strukturbäume als Navigationshilfe und als Hilfe beim Aufruf einzelner Zoom-Transformationen benutzt werden können. Ihre wesentliche Stärke liegt aber darin, dass sie ein gutes Modell bieten, um Zoom-Transformationen programmgesteuert zu kombinieren und so einfach verschiedene Ansichten eines Diagramms zu erzeugen. Ein Strukturbaum enthält nämlich für das gesamte Diagramm alle Informationen, welche Teile abstrahiert wurden (auch über mehrere Stufen) und welche Elemente expandiert werden müssen, um einen bestimmten Diagrammteil wieder sichtbar zu machen.

Das einfachste Beispiel für eine solche zusammengesetzte Transformation ist das vollständige Expandieren eines Knotens. Dabei werden alle Knoten im darunterliegenden Teilbaum (und der Knoten selbst) expandiert, soweit dies möglich ist. Die Anwendung dieser Transformation auf den Wurzelknoten des Strukturbaums bewirkt, dass sämtliche Details des Diagramms angezeigt werden.

Eine andere Transformation, die sich so einfach realisieren lässt, ist das schrittweise Verfeinern einer abstrahierten Diagrammkomponente: Statt alle darunterliegenden Details wieder einzublenden, soll nur eine weitere Detailstufe angezeigt werden. Dies könnte bei einem UML-Diagramm z. B. bedeuten, dass ein abstrahiertes Paket soweit expandiert wird, dass die enthaltenen Klassen wieder

<sup>2</sup>In Abb. 3.7 sieht man den Effekt der tatsächlich verwendeten Transformation: die Attribut- und Operationssegmente der Klasse werden nicht mehr angezeigt.



**Abbildung 7.4:** Verfeinern eines abstrahierten Pakets über einen Zwischenschritt

sichtbar sind, nicht jedoch deren Attribute. Diese Transformation wird realisiert, indem der gewählte Knoten des Strukturbaums expandiert wird und anschließend alle seine Kind-Knoten wieder abstrahiert werden. Abbildung 7.4 zeigt die beiden Schritte.

Analog ließe sich auch eine Transformation formulieren, die das gesamte Diagramm gleichmäßig in einer bestimmten Detailstufe anzeigt und damit Darstellungen mit semantischen Zoom ohne Fokuspunkt erzeugt (vgl. S. 18).

Diese Transformationen erfolgen, wie in Abbildung 7.4 gezeigt, in zwei Phasen: Zuerst werden alle Expansionen von Knoten des Strukturbaums durchgeführt, danach alle notwendigen Abstraktionstransformationen. Wie auf Seite 54 erwähnt, können beim Kombinieren mehrerer Zoom-Transformationen Probleme auftreten, weil die Hypergraph-Strukturen des Analyse-Moduls zwischen einzelnen Schritten nicht an Veränderungen im HGM angepasst werden. Solche Schwierigkeiten können wir weitgehend dadurch vermeiden, dass zwischen den beiden Phasen eine erneute Analyse des maximal expandierten Diagramms veranlasst wird, so dass die Datenstrukturen zu Beginn der Abstraktionsphase wieder konsistent sind. Expansionstransformationen müssen ohnehin keine Diagrammbereiche auswählen, daher benötigen sie keine Analyse-Information und sind in dieser Beziehung unproblematisch. Die Abstraktionsphasen sind bei allen zusammengesetzten Transformationen des Strukturbaums so konstruiert, dass sich die betroffenen Bereiche des Diagramms (bzw. des HGMs) nicht überlappen, daher sollten hier auch keine Probleme auftreten. Beeinflussungen zwischen Expansions- und Abstraktionstransformationen werden durch die erwähnte zusätzliche Diagrammanalyse vermieden.

Auf der Grundlage dieses Zwei-Phasen-Mechanismus haben wir schließlich auch ein allgemeines Schema realisiert, das beliebige (programmgesteuert vorgegebene) Abstraktionsansichten erzeugt bzw. annähert. Diese Ansichten werden durch eine Menge von "wesentlichen" Knoten des Strukturbaums definiert, die im Diagramm sichtbar sein sollen; alle übrigen Diagrammteile sollen, falls möglich, ausgeblendet werden. Um nun von der aktuellen Darstellung des Diagramms zu dieser Ansicht zu gelangen, werden die Knoten des Strukturbaums

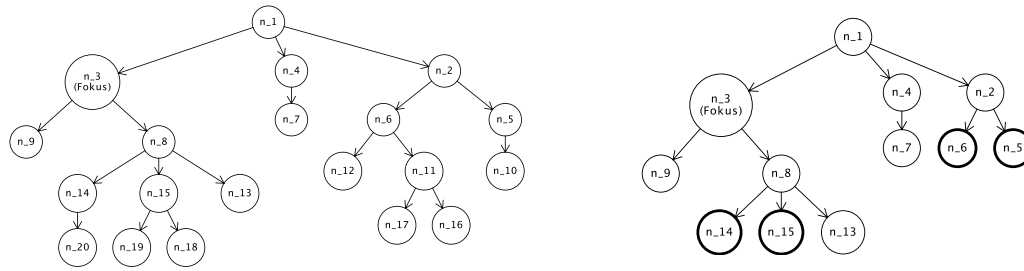


Abbildung 7.5: Größerer Baum mit Fokus-und-Kontext-Ansicht

zuerst so weit expandiert, bis alle gewünschten Elemente sichtbar sind, danach werden alle “überflüssigen” sichtbaren Diagrammteile soweit möglich durch Anwendung von Abstraktionstransformationen entfernt.

Da Knoten des Strukturbaums immer nur als Ganzes abstrahiert oder expandiert werden können, enthält die resultierende Ansicht unter Umständen mehr Elemente, als in der Definition der Ansicht vorgegeben waren;<sup>3</sup> die Menge dieser zusätzlichen Elemente ist aber minimal, d. h. es wird kein Diagrammteil angezeigt, der abstrahiert werden könnte, ohne dabei auch ein wesentliches Element zu verlieren.

Nach diesem Schema lassen sich fast beliebige Abstraktionsansichten eines Diagramms darstellen; wir benötigen aber noch einen einfachen Weg für die Benutzerin, “interessante” Ansichten zu bestimmen und müssen daraus die richtige Menge von wesentlichen Elementen berechnen. Es bietet sich an, für diese Aufgabe auf die ursprüngliche Definition von Fokus-und-Kontext-Ansichten zurückzugreifen und für jeden Knoten des Strukturbaums einen Relevanzwert zu berechnen, wie dies in Abschnitt 2.6 besprochen wurde. Mit Hilfe dieser Relevanzwerte kann dann entschieden werden, ob ein Knoten “wesentlich” ist oder nicht. Die Benutzerin muss dabei nur ein Diagrammelement (bzw. den entsprechenden Knoten des Strukturbaums) wählen und einen Detailgrad angeben, der bestimmt, ab welchem Relevanzwert ein Knoten des Strukturbaums angezeigt werden soll.<sup>4</sup>

Basierend auf der Information des Strukturbaums können die DOI und LOD-Werte nach den Vorschlägen von Furnas berechnet werden (vgl. S. 13); bei Baumdiagrammen kann man damit schon sehr befriedigende Ergebnisse erzielen. Abbildung 7.5 zeigt einen etwas größeren Baum und eine Fokus-und-Kontext-Ansicht, bei welcher der entsprechend beschriftete Knoten links außen als Fokus dient und ein “Radius” von 2 gewählt wurde; d. h. alle Knoten, die weiter als zwei Kanten vom Fokus oder einem seiner (nach der Formel genauso wichtigen) direkten Vorfahren entfernt liegen, wurden ausgeblendet.

<sup>3</sup>nämlich alle “Geschwister” von Knoten aus der Definitionsmenge

<sup>4</sup>Die numerische Eingabe des Detailgrads ist nicht intuitiv, da Relevanzwerte keine direkte grafische Entsprechung im Diagramm haben; eine andere Art der Bestimmung wäre hier wünschenswert.

Für andere Diagrammtypen sollten sinnvollerweise nicht nur strukturelle, sondern auch geometrische Faktoren in die Relevanzberechnung einfließen: Der DOI-Wert sollte die geometrische Entfernung zum Fokus berücksichtigen und der LOD-Wert die visuelle Größe einer Struktur-Komponente. Hier bietet sich wieder ein weites Feld für Experimente mit unterschiedlichen Diagrammsprachen und Berechnungsformeln.

Zuletzt sollte noch darauf hingewiesen werden, dass Strukturbäume und alle in diesem Kapitel vorgestellten Möglichkeiten zur Manipulation von Diagrammen nur eine mögliche Anwendung für Zoom-Transformationen und sFK-Darstellung bilden. Die Abstraktion von Diagrammteilen muss sich nicht immer auf eine inhärente hierarchische Struktur des Diagramms stützen; als alternatives Beispiel könnte man sich z. B. eine Transformation zur Vergrößerung von "flachen" Petri-Netzen vorstellen, bei der ein geeignetes stellen- oder transitionsberandetes Teilnetz durch eine einzige Stelle bzw. Transition ersetzt wird.

## Kapitel 8

# Schlussbemerkungen

Die vorangehenden Kapitel haben das Konzept der semantischen Fokus-und-Kontext-Darstellung durch Zoom-Transformationen eingeführt, Beispiele für seine Anwendung gegeben und die Details der Realisierung genauer beleuchtet. Zum Abschluss wollen wir noch einmal die wesentlichen Ergebnisse zusammenfassen, einige Ansatzpunkte für mögliche Verbesserungen aufzeigen und zuletzt einen kurzen Blick auf zukünftige Aufgabenstellungen werfen, die auf der vorliegenden Arbeit aufbauen können.

### 8.1 Ergebnisse

In dieser Arbeit wurde ein Ansatz zur Darstellung und Bearbeitung großer Diagramme in DiaGen entwickelt, der es ermöglicht, mit Hilfe von Zoom-Transformationen den Detailgrad der Ansicht flexibel zu variieren. Mit dieser semantischen Fokus-und-Kontext-Darstellung kann der Bereich, der im Zentrum des Benutzerinteresses liegt, groß und detailliert dargestellt werden, ohne dass dabei der Bezug zu anderen Bereichen des Diagramms oder die Übersicht über das gesamte Diagramm verloren geht.

Gegenüber anderen existierenden Konzepten und Systemen zeichnet sich die vorgestellte Lösung dadurch aus, dass sie

- für ein breites Spektrum von Diagrammsprachen anwendbar ist,
- Diagrammteile selektiv unterschiedlich stark abstrahieren kann und so die vorhandene Bildschirmfläche optimal nutzt,
- strukturbasiert arbeitet und nicht nur durch optische Manipulation der Darstellung (und damit dem kognitiven Modell der Benutzerin besser entspricht), und schließlich
- an die jeweilige Diagrammsprache angepasst werden kann und so vorhandene Konstrukte zur Abstraktion geeignet verwenden kann.

Die Lösung harmoniert gut mit dem Konzept des DiaGen-Systems und kann vorhandene Funktionalität wie das mehrstufige Undo/Redo und die Animation von Diagrammtransformationen sinnvoll nutzen.

Die Implementierung des Konzepts in zwei Beispiel-Editoren beweist seine praktische Anwendbarkeit. Dabei wurde ein Editor für UML-Klassendiagramme entwickelt, der belegt, dass mit DiaGen auch Editoren für umfangreiche Diagrammsprachen entwickeln lassen; dieser Editor demonstriert auch die Vorteile des Bedienkonzepts von DiaGen, die sich durch die weitgehende Verwendung von direkter Manipulation ergeben. Zur Analyse von UML-Diagrammen wurde ein Weg gefunden, um auf einfache Weise einen abstrakter Syntaxgraph eines Diagramms zu generieren. Wir haben gezeigt, wie Objektmodelle, welche die "Semantik" von Diagrammen beschreiben, auf derartige abstrakte Syntaxgraphen abgebildet werden können und wir glauben, dass diese Technik zur Diagrammanalyse auch in anderen Diagrammsprachen Anwendung finden kann.

Schließlich haben wir einen Layout-Algorithmus für komplexe graph-artige Diagramme entwickelt, der die Erfordernisse einer semantischen Fokus-und-Kontext-Darstellung berücksichtigt. Dabei hat sich herausgestellt, dass sich constraint-basierte Layout-Techniken gut für die Verwendung in interaktiven Diagrammeditoren eignen, dass aber ein ausgesprochen primitiver Constraint-Propagation-Mechanismus für den praktischen Einsatz unter Umständen besser geeignet ist als existierende aufwendige Constraint-Solver. Auch das Layout durch Kraftsimulation, das als Graph-Layout-Algorithmus ohne Vorinformation mit vielen Problemen zu kämpfen hat, hat sich zur Layoutkorrektur in interaktiven Editoren als sehr flexibles und nützliches Hilfsmittel erwiesen.

## 8.2 Ansatzpunkte für Verbesserungen

Die erfolgreiche Implementierung des vorgestellten UML-Editors belegt die Leistungsfähigkeit und Verwendbarkeit sowohl des DiaGen-Systems selbst wie auch des vorgestellten Konzepts der Zoom-Transformationen. Trotzdem haben sich an einigen Stellen noch Defizite gezeigt (wie sie z. T. bereits im Text angesprochen wurden), die durch zukünftige Erweiterungen des Systems behoben werden könnten.

Wie bereits erwähnt, werden im Zusammenhang mit Zoom-Transformationen hohe Anforderungen an die automatische Layoutkorrektur gestellt. Insbesondere kommt es vor, dass sich der Kontext eines ausgeblendeten Diagrammteils beim Wiedereinblenden in seiner Anordnung verändert hat und der Layout-Algorithmus entsprechende Anpassungen vornehmen muss. Der häufigste derartige Fall besteht darin, dass die Abstraktionsdarstellung (der Anker) an eine andere Stelle verschoben wurde. Beim Wiedereinblenden erfolgt hier automatisch auch eine entsprechende Verschiebung der abhängigen Komponenten.

Probleme treten aber weiterhin auf, wenn die Abstraktionsdarstellung ihre Form völlig geändert hat und deshalb kein allgemeiner automatischer Korrekturmechanismus zur Verfügung steht. Als Beispiel kann der Fall dienen, dass mehrere gleichlaufende Assoziationspfeile ausgeblendet und durch einen "Ersatzpfeil" repräsentiert wurden (vgl. Abschnitt 3.6). Dessen Endpunkte können nun unabhängig voneinander beliebig verschoben werden. Wenn die Assoziationspfeile wieder angezeigt werden sollen, ist ihre Position evtl. völlig falsch; die Layoutkorrektur verfügt nur über die Information (durch die entsprechenden Relationen-Hyperkanten), zwischen welchen Klassen sie eigentlich verlaufen sollten, und sie werden nun einfach an den nächstgelegenen Punkt der Klasse (meistens eine Ecke) gehängt, was zu unschönen Effekten führt. Als Lösung könnte man sich vorstellen, zu einer ausgeblendeten Relationen-Hyperkante auch spezifische Layoutinformationen zu speichern, welche die spätere Wiederherstellung der geometrischen Relation erleichtern. Zum Beispiel könnte mit einer "enthalten-in"-Hyperkante die relative Position des enthaltenen Elements zum "Behälter" assoziiert werden. Dieses Vorgehen würde so auch das gegenwärtig realisierte Verschieben von einzublendenden Komponenten relativ zur neuen Position ihres Ankers einschließen.

Erweiterungen der Module zur Analyse und Graphtransformation könnten die Spezifikation von Diagrammsprachen erheblich vereinfachen; in Kapitel 5 wurde bereits mehrere dieser Punkte angesprochen:

- Die eingeschränkte Anwendbarkeit von Einbettungsproduktionen erschwert oder verhindert die Spezifikation mancher Aspekte von visuellen Sprachen. Allgemeinere Beschreibungskonzepte wären hier wünschenswert (vgl. Seite 61). Auch die in Abschnitt 5.9 beschriebene Einführung von mehreren Reduzierer-Schritten würde die Möglichkeiten zur Diagrammanalyse wesentlich erweitern.
- Die Konstrukte zur Beschreibung von Graphmustern weisen gegenwärtig viele "historisch bedingte" Einschränkungen und Inkonsistenzen auf. Wünschenswert wäre ein einheitliches Konzept zur Beschreibung von Kontext und Pfadausdrücken; Injektivitätsbedingungen beim Mustervergleich sowie die Aufnahme von Musterelementen in die Analysestrukturen (lex-Relation, vgl. Seite 52) sollten orthogonal dazu spezifizierbar sein.
- Die Kontrollstrukturen, welche zur Steuerung komplexer Diagrammtransformationen dienen, eignen sich nicht immer für eine intuitive und kompakte Beschreibung der gewünschten Modifikationen. Verbesserungsmöglichkeiten bieten sich hier einerseits durch die Erweiterung zu einer vollwertigen regelbasierten Programmiersprache mit Backtracking, wie sie in [31] beschrieben wurde. Alternativ würde eine bessere Integration von DiaGen-Transformationen mit der "Wirtssprache" Java die Möglichkeit bieten, auf in dieser Sprache definierte Kontroll- und Datenstrukturen zurückzugreifen.

- Hilfreich wäre schließlich auch eine Möglichkeit zur Spezifikation von Diagrammteilen, die zwar eigentlich nicht korrekt sind, bei denen aber eine Zusammengehörigkeit erkannt wurde, die auch beim Layout berücksichtigt werden soll. Beispielsweise sollten nicht korrekt verbundene Pfeile mit ihren Pfeilbeschriftungen als Einheit behandelt werden, obwohl sie nicht in die generierte Ableitungsstruktur aufgenommen werden können.

Ein wesentliches Problem liegt noch in der Tatsache, dass die Analysestrukturen im Verlauf einer zusammengesetzten Transformation nicht an Änderungen des Hypergraph-Modells angepasst werden und es so zu Inkonsistenzen im Gesamtmodell kommen kann (vgl. Seite 54). Hier ist ein tragfähiges Konzept notwendig, um die Komposition von Transformationen zu erleichtern und auftretende Seiteneffekte zu begrenzen.

Das System bietet inzwischen recht gute Möglichkeiten zur Fehlersuche beim Spezifizieren von Analyse- und Transformationsregeln. Trotzdem ist die Spezifikation einer umfangreichen Diagrammsprache noch ein fehleranfälliger Vorgang und erfordert bessere Unterstützung. Denkbar wäre z. B. eine Erweiterung der Spezifikation um Konsistenzbedingungen, wie globale Muster im HGM, die nicht auftreten dürfen, oder sich ausschließende Regeln, die nie beide auf ein Teilmuster angewandt werden dürfen. Eine Art verbesserte "Typprüfung" könnte sicherstellen, dass Knoten immer nur mit bestimmten Kombinationen von Hyperkanten über genau definierte Tentakel verbunden sein können, und so viele Flüchtigkeits- oder Tippfehler in der Spezifikation leichter aufdecken.

### 8.3 Ausblick

Obwohl also noch vielfältige Verbesserungen des DiaGen-Systems denkbar und wünschenswert sind, kann es in seiner gegenwärtigen Form wie gezeigt schon als Basis für umfangreiche Anwendungen dienen. Wir glauben, dass hier eines der wichtigsten Felder für weiterführende Arbeiten liegt; auf der Grundlage des vorgestellten UML-Editors lassen sich beispielsweise interessante neue Ideen zur Arbeit mit UML-Diagrammen realisieren: Die Kombination der UML mit den von DiaGen erzeugten Hypergraph-Modellen (HGM und ASG) bietet die Möglichkeit, Transformationen auf objektorientierten Strukturmodellen zu definieren und auszuführen. Mit solchen Transformationen könnte man z. B. Programmstrukturen halbautomatisch für den Einsatz in verteilten Objektsystemen vorbereiten.

Eine zusätzliche Grundlage für die Anwendung von Programmstruktur-Transformationen bieten anwendungsunabhängige Konstruktionsmuster für objektorientierte Programme, sogenannte "Design Patterns" [14]. Derartige Muster dienen dazu, bewährte Standard-Lösungen für häufig wiederkehrende Design-Probleme festzuhalten; sie beschreiben bestimmte Beziehungen innerhalb einer

Gruppe von Klassen (oder Objekten), wobei jede Klasse im Kontext des Konstruktionsmusters eine bestimmte "Rolle" ausfüllt. Die UML enthält eine standardisierte Notation, um das Auftreten solcher Patterns in Programmstrukturen anzuzeigen. (Eine Klasse kann dabei an mehreren Konstruktionsmustern in unterschiedlichen Rollen teilnehmen.)

In einem DiaGen-Editor könnte man nun Diagramm-Transformationen zum Anwenden (bzw. Einfügen) von Konstruktionsmustern in einem Diagramm definieren. Natürlich sollten solche Transformationen nicht direkt in das Spezifikations-Dokument kodiert werden, da man sonst zur Definition eines neuen Konstruktionsmusters den kompletten Editor neu generieren müsste. Benötigt wird also auch eine formalisierte abstrakte Beschreibung von Konstruktionsmustern. Diese könnte wieder in Form von separaten UML-Diagrammen bzw. den daraus generierten abstrakten Syntaxgraphen erfolgen, so dass die Benutzerin in der Lage wäre, Bibliotheken von Konstruktionsmustern zu erstellen, zu ergänzen und in den UML-Editor zu importieren. Der UML-Editor müsste (zur Laufzeit) aus diesen Diagrammen und Graphen dann Transformationen zum Einfügen der Konstruktionsmuster generieren können; man könnte auch Konsistenz-Prüfungen einführen, welche die in einem Diagramm verwendeten Konstruktionsmuster mit ihren Definitionen aus der Bibliothek vergleichen und so sicherstellen, dass alle Rollen korrekt besetzt sind.

Vielleicht wäre es auch möglich, durch Mustervergleich das Auftreten von Konstruktionsmustern in vorhandenen Diagrammen zu erkennen und so Re-Engineering zu betreiben. Außerdem erfordern viele Patterns auch "Hilfsklassen", die nur als Hilfskonstrukte zur programmtechnischen Realisierung dienen und für das Verständnis der Anwendungsstruktur nicht notwendig sind. Durch das Ausblenden solcher Klassen bietet sich hier eine weitere Anwendungsmöglichkeit für Diagrammabstraktion durch Zoom-Transformationen. Derartige Editorfunktionen würden das tool-gestützte Erstellen von objektorientierten Programm-Designs sicher deutlich vereinfachen; uns ist gegenwärtig noch kein kommerzielles Tool bekannt, das eine derartige Unterstützung bietet.

Um die angedeuteten Anwendungen zu realisieren, müsste allerdings, wie auf Seite x42 erwähnt, das "Unparsing-Problem" in DiaGen gelöst werden; d. h. es müsste möglich sein, aus den Graph-Strukturen wieder konkrete Diagramme mit einem korrekten Layout zu generieren, wobei das Layout "from scratch" generiert werden muss, da ja Diagrammstrukturen erst neu erzeugt und nicht nur modifiziert werden. Für den speziellen Fall von UML-Klassendiagrammen sollte sich dies aber unter Verwendung von etablierten Graph-Layout-Algorithmen möglich sein: Ein korrektes, aber unbefriedigendes Layout kann ja immer noch nachträglich durch die Benutzerin korrigiert werden.

Ein Aspekt des Bearbeitens von Diagrammen mit DiaGen-Editoren, der noch deutlich verbessert werden könnte, ist die Benutzungssteuerung. Eine einfach zu integrierende Erleichterung wäre hier die Möglichkeit, Textbeschriftungen direkt im Diagramm zu bearbeiten ("in-place-editing"); gegenwärtig muss zur Textein-

gabe erst eine Dialogbox geöffnet werden. Das Java2D-API bietet hier eine weitreichende Unterstützung.

Die unterschiedlichen Bearbeitungsmodi von DiaGen-Editoren können zwar die Bearbeitung beschleunigen, führen aber auch leicht zu Fehlbedienungen. Nützlich wären hier alternative Eingabemöglichkeiten, so dass alle Bearbeitungsfunktionen alternativ auch ohne Moduswechsel zur Verfügung stehen (auch wenn sie dann u. U. weniger einfach zu erreichen sind). Ausserdem wäre es wichtig, der Benutzerin den aktiven Modus deutlicher bewußt zu machen, und so Fehlbedienungen zu vermeiden.

Zusätzlich sollte die Editorsteuerung auch vereinheitlicht werden; aus Sicht der Benutzerin besteht z. B. kein wesentlicher Unterschied zwischen dem direkten Einfügen einer Komponente und der Ausführung einer Diagrammtransformation, die neue Komponenten erzeugt, daher sollten beide Funktionen auch gleich aufgerufen werden. Statt einfach einen Knopf ("Button") für jede mögliche Transformation zu verwenden, wäre es gut, wenn vielfältigere "Gesten" aus Mausbedienung, Selektion von Diagrammkomponenten und Benutzungsschnittstellen-Aufrufen (Knöpfe, Menüs) hierfür definiert werden könnten.<sup>1</sup>

Die natürlichste Benutzungsschnittstelle für die Bearbeitung von Diagrammen würde eine Stifteingabe mit einem Grafiktablett darstellen; noch günstiger wäre ein zur Stifteingabe geeigneter Flachbildschirm, der die Trennung zwischen Eingabe und Anzeige aufhebt. Die vergleichsweise aufwendige Bedienung von Diagrammeditoren führt nämlich immer noch oft dazu, dass Diagramme erst einmal auf Papier skizziert werden, bevor sie im Editor eingegeben werden. Durch die Kombination der DiaGen-Diagrammanalyse mit einem vorgeschalteten Mustererkennungsschritt ließe sich ein System realisieren, bei dem Diagramme mit der Hand digital skizziert werden und dann erkannt, "verschönert" und on-line weiterverarbeitet werden.

In einer solchen Benutzungsschnittstelle können auch vielfältige "Gesten" definiert werden, die das Bearbeiten des Diagramms ohne Moduswechsel erlauben. Da das DiaGen-System dafür ausgelegt ist, mit unvollständigen und nur teilweise korrekten Diagrammen zurecht zu kommen, bietet es sich für eine solche Umgebung besonders an. Zhao beschreibt in [23] ein Toolkit, das diesen Ansatz verfolgt, aber offensichtlich keine weitere Anwendung gefunden hat. Wie dort festgestellt wird, ist eine Rückkopplung von Diagrammanalyse zur Mustererkennung notwendig, da die skizzierten Strukturen oft mehrdeutig sind und erst durch weitere Informationen über korrekte Strukturen höherer Ordnung korrekt erkannt werden können. Wir glauben, dass die Kombination des DiaGen-Systems mit der Stifteingabe einen sehr interessanten Ausgangspunkt für zukünftige Arbeiten bieten würde.

---

<sup>1</sup>Ein Versuch, solche Gesten in einer früheren Version von DiaGen über ereignisgesteuerte Automaten zu definieren, scheiterte allerdings daran, dass derartige Spezifikationen zu aufwendig waren.

# Literaturverzeichnis

- [1] *The FISHEYE view: a new look at structured files*  
G.W.Furnas  
Bell Laboratories 1981, Technical Memorandum #81-11221-9
- [2] *Space-scale diagrams: understanding multiscale interfaces*  
G.W.Furnas und B.Bederson  
Proc. ACM CHI'95 Conference on Human Factors in Computing Systems  
1995, S. 234–241
- [3] *Context and Interaction in Zoomable User interfaces*  
S.Pook, E.Lecolinet, G.Vaysseix, E.Barillot  
Proc. AVI'2000 Advanced Visual Interfaces 2000, S. 227–231
- [4] *Making distortions comprehensible*  
M.Sheelagh, T.Carpendale, D.J.Coperthwaite und F.D.Fracchia  
Proc. VL'97 IEEE Symposium on Visual Languages 1997, S. 36–45
- [5] *Techniques for non-linear magnification transformations*  
T.A.Keahey und E.L.Robertson  
IEEE Visualisation 96, S. 38–45
- [6] *Graphical fisheye views of graphs*  
M.Sarkar und M.Brown  
Proc. ACM CHI'92 Conference on Human Factors in Computing Systems  
1992, S. 83–91
- [7] *Navigating hierarchically clustered networks through fisheye and full-zoom methods*  
Schaffer et al.  
ACM Transactions on CHI, 1996 Bd. 3(2), S. 162–188
- [8] *Pad: an alternative approach to the computer interface*  
K.Perlin und D.Fox  
ACM SIGGRAPH Computer Graphics 1993 Bd. 27, S. 57–64
- [9] *Pad++: a zooming graphical interface for exploring alternate interface physics*  
B.Bederson und J.D.Hollan

- Proc. UIST'94 Symposium on User Interface Software and Technology 1994,  
S. 17–26
- [10] *Implementing a zooming user interface: experience building Pad++*  
B.Bederson und J.Meyer  
Software – Practice and Experience, 1998 Bd. 28(10), S. 1101–1135
- [11] *Jazz: an extensible 2D+zooming graphics toolkit in Java*  
B.Bederson und B.McAlister  
University of Maryland HICL Technical Report 99–07
- [12] *Unified Modelling Language Specification*  
Object Management Group  
<http://www.omg.org/uml/>
- [13] *Meta Object Facility Specification*  
Object Management Group  
<http://www.omg.org/uml/>
- [14] *Design Patterns*  
E.Gamma, R.Helm, R.Johnson und J.Vlissides  
Addison Wesley 1995
- [15] *Integration of declarative and algorithmic approaches for layout creation*  
T.Lin und P.Eades  
Proc. GD'94 Graph Drawing, LNCS 894, S. 376–387
- [16] *Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm*  
M.Sannella et al.  
Dept. of CS&E, University of Washington, Technical Report #92-07-05a
- [17] *Ultraviolet: a constraint satisfaction algorithm for interactive graphics*  
A.Borning und B.Freeman-Benson  
Constraints: an International Journal, 1998 Bd. 3, S. 1–26
- [18] *An experimental comparison of force-directed and randomized graph drawing algorithms*  
F.Brandenburg, M.Himsolt und C.Rohrer  
Proc. GD'95 Graph Drawing, LNCS 1027, S. 76–87
- [19] *A simple and unified method for drawing graphs: magnetic spring algorithm*  
K.Sugiyama und K.Misue  
Proc. GD'94 Graph Drawing, LNCS 894, S. 364–375
- [20] *A fast adaptive layout algorithm for undirected graphs*  
A.Frick, A.Ludwig und H.Mehldau  
Proc. GD'94 Graph Drawing, LNCS 894, S. 388–403

- [21] *Layout adjustment and the mental map*  
K.Misue, P.Eades, W.Lai und K.Sugiyama  
JVLC'95 Journal on Visual Languages and Computing 1995 Bd. 6,  
S. 183–210
- [22] *Designing the user interface*  
B.Shneiderman  
Addison Wesley 1998, Dritte Ausgabe
- [23] *Incremental recognition in gesture-based and syntax-directed diagram editors*  
R.Zhao  
Proc. ACM InterCHI'93 Conference on Human Factors in Computing  
Systems 1993, S. 95–100
- [24] *Graph transformation and visual modeling techniques*  
R.Heckel und G.Engels  
Bulletin of the European Association for Theoretical Computer Science,  
2000 Bd. 71
- [25] *Applications of graph transformations to visual languages*  
R.Bardohl, G.Taentzer, M.Minas und A.Schürr  
Handbook of Graph Grammars and Computing by Graph Transformation,  
volume II: Applications, Languages and Tools, World scientific 1999, S. 1–77
- [26] *The PROGRES approach: language and environment*  
A.Schürr  
Handbook of Graph Grammars and Computing by Graph Transformation,  
volume II: Applications, Languages and Tools, World scientific 1999,  
S. 487–550
- [27] *The AGG approach: language and environment*  
C.Ermel, M.Rudolf und G.Taentzer  
Handbook of Graph Grammars and Computing by Graph Transformation,  
volume II: Applications, Languages and Tools, World scientific 1999,  
S. 551–603
- [28] *Positional grammars: a formalism for LR-like parsing of visual languages*  
G.Costagliola, A.DeLucia, S.Orefice und G.Tortora  
Visual Language Theory, Springer 1998, S. 172–191
- [29] *Creating semantic representations of diagrams*  
M.Minas  
Proc. AGITVE'99 International Workshop on Applications of Graph  
Transformation with Industrial Relevance, LNCS 1779, S. 209–224
- [30] *Generating diagram editors providing free-hand editing as well as syntax-directed editing*  
M.Minas und O.Köth

Proc. GRATRA'2000 Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, TU Berlin Technical Report #2000-2, Fachbereich 13 (Informatik), S. 32-39

- [31] *Towards typed generic rule-based visual programming*  
B.Hoffmann und M.Minas  
Proc. VL'2000 IEEE Symposium on Visual Languages 2000, S. 65-66
- [32] *Spezifikation und Generierung graphischer Diagrammeditoren*  
M.Minas  
Habilitationsschrift, Universität Erlangen, erscheint 2001

# Index

- Abhängigkeiten, UML-Elemente, 33
- Abstrakter Syntaxgraph, 39, 85
  - Erzeugung, 41
  - Konsistenzbedingungen, 41
- Abstraktionsbereiche, 51, 86
- Abstraktionsoperationen, *siehe* Zoom-Transformationen
- Algorithmische Layoutspezifikation, 68
- Anker-Hyperkanten, 49
- ASG, *siehe* Abstrakter Syntaxgraph
- Assoziationen, UML-Elemente, 32, 43
- Assoziationsklassen, UML-Elemente, 59
- Attributauswertung, 29, 85
- Attribute, UML-Elemente, 32
- Ausblenden von Diagrammteilen, 49, 86
  
- Baumdiagramme, 26
- Benutzungsschnittstelle, 2, 84, 95
  
- Constraint-Solving, 68
  - beim Layout von UML-Diagrammen, 74
- Cut & Paste, 86
  
- Degree of Interest, 13, 89
- Deklarative Layoutspezifikation, 68
- DiaGen
  - Bearbeiten von Diagrammen, 2
  - Generator-Konzept, 1
  - Programmmodule, 18
- Diagramm-Transformationen, Spezifikation, 47
- Direkte Manipulation, 2
  - in UML-Diagrammen, 34
- Drag & Drop, von Diagrammelementen, 75
  
- Einbettungsproduktionen, 28, 59
- Einblenden von Diagrammteilen, 49, 86
- Erinnerungs-Hyperkanten, 62
- Ersatzdarstellungen für UML-Pfeile, 34, 56
- Ersetzen, von HGM-Hyperkanten, 55
- Expansionsoperationen, *siehe* Zoom-Transformationen
  
- Fisheye-Darstellung, 8, 13
  - durch optische Verzerrung, 15
  - Probleme, 16
- Fokus-und-Kontext-Darstellung, 8, 13, 89
- Force-Directed Layout, *siehe* Kräftesimulation
  
- Generalisierungen, UML-Elemente, 33
- Grammatik-Produktionen, 28, 60
- Graph-Grammatiken, 28
- Graph-Layout, 72
  - bei hierarchischen Graphen, 80
  
- Hauptkomponente, eines Strukturbaum-Knotens, 84
- HGM, *siehe* Hypergraph-Modell
- Hypergraph-Modell, 26
  - für UML-Diagramme, 34
- Hypergraphen, 18
  
- Injektivitätsprüfungen, beim Mustervergleich, 54, 64
- Inkrementelles Layout, 72
  
- Jazz-Toolkit, 11
  
- Kantenkreuzungen, beim Graph-Layout, 17, 78
- Klassen, UML-Elemente, 32
- Komponenten-Hyperkanten, 26
- Konnektoren von Diagramm-Komponenten, 26
- Konsistenz-Bedingungen, im HGM, 26, 49, 94
  - Probleme, 58
- Konsistenz-Probleme zwischen HGM und Analysestrukturen, 54, 88
- Kontextfreie Grammatiken, 28
- Kontrollfluss, in Transformationen, 48, 64, 93
- Kopieroperator, für Relationen-Hyperkanten, 55
- Kräftesimulation, 77

- Iteration, 80, 81
  - zur Abstandskorrektur, 79
  - zur Korrektur nach Zoom-Transformationen, 81
- Layoutkorrektur, 13, 48, 67
  - im Gegensatz zu Layoutgenerierung, 78
  - Probleme mit ausgeblendeten Diagrammteilen, 92
  - restriktive, 71
- Level of Detail, 13, 89
- Markierungs-Hyperkanten, 56
- Mengenproduktionen, 60
- Meta-Kanten, 50, 52
- Meta-Object Facility, 38
- Metamodell, *siehe* UML-Metamodell
- MoF, *siehe* Meta-Object Facility
- Mustererkennung, zur Diagrammeingabe, 96
- Nichtterminal-Hyperkanten, 28
  - im ASG, 41
- Obertypen, fürHGM-Hyperkanten, 57
- Object Management Group, 31
- OMG, *siehe* Object Management Group
- Pad-Toolkit, 11
- Pakete, UML-Elemente, 33
- Parse-DAG, 28
- Patterns in der objektorientierten Modellierung, 94
- Petri-Netze, mögliche Abstraktionstransformationen, 90
- Pfadausdrücke, 51
- Pfeilbeschriftungen, Zuordnung, 61
- PROGRES, 64
- Reduzierer-Regeln, 26, 63
- Reduzierer-Schritte, mehrere, 44, 63
- Reduziertes Hypergraph-Modell, 26
- Relationen-Hyperkanten, 26
- Relevanz von Diagrammteilen, 13, 14, 88
- rHGM, *siehe* Reduziertes Hypergraph-Modell
- Semantische Fokus-und-Kontext-Darstellung, 9, 20, 89
- Semantisches Zoomen, 9, 10, 88
  - Probleme, 12
- sFK-Darstellung, *siehe* Semantische Fokus-und-Kontext-Darstellung
- Spezifikation einer Diagrammsprache, 1, 26
- Stifteingabe, 96
- Strukturbäume, 83
  - als Benutzungsschnittstellen-Komponenten, 84
  - Erzeugung, 85
- Tentakel von Hyperkanten, 18, 26
- Terminal-Hyperkanten, 28
- Transformationsregeln, 47
  - generische, 57
- Typen-Hierarchie, fürHGM-Hyperkanten, 57
- UML, *siehe* Unified Modeling Language
- UML-Klassendiagramme, 31
  - Syntax, 32
- UML-Metamodell, 37
- Unified Modeling Language, 31, 37
  - Kombination mit Graphtransformationen, 39, 94
- Unparsing-Problem, 42, 95
- Verfeinern, der Diagrammdarstellung, 87
- Virtuelles Hypergraph-Modell, 49
- Vollständiges Layout, 73
- XMI, Dateiformat, 38
- Zoom-Transformationen, 20, 47
  - Ausführung, 49, 58
  - Auswahl des Abstraktionsbereichs, 54
  - in Baumdiagrammen, 29
  - in UML-Diagrammen, 34, 56
  - Layoutkorrektur, 67
  - Markieren von Abstraktionsdarstellungen, 30, 35
  - zusammengesetzte, 87