Technische Universität Berlin

Evolutionäres Layout von Graphseqenzen unter Nutzung von anwendungsspezifischem Wissen

Diplomarbeit

zur Erlangung des Grades eines Diplom–Informatikers

am Institut für Softwaretechnik und Theoretische Informatik
Prof. Dr.–Ing. Stefan Jähnichen
Fakultät für Elektrotechnik und Informatik
Technische Universität Berlin

Dennis Graf
Matr.–Nr. 193360

Gutachter:

Prof. Dr.–Ing. Stefan Jähnichen
Dr. Gabriele Taentzer

Betreuerin:

Dipl.–Inform. Susanne Jucknath–John

Berlin, den 24. April 2006

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, die vorliegende Arbeit selbstständig und eigenhändig angefertigt zu haben.

Berlin, 24.04.2006

_____

(Dennis Graf)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Softwareentwicklungsprozess werden einige Modelle im UML–Standard als Graphen dargestellt, so auch das Klassendiagramm. Die Lesbarkeit dieser Graphen, und auch Graphen allgemein, hängt in großem Maße von ihrem Layout ab. Auf Klassendiagramme bezogen bedeutet dies einen Mehraufwand für Entwickler bei der Einarbeitung in ein Klassendiagramm mit schlechtem Layout.

Im Laufe eines Softwareprojekts wird meist nicht nur ein Klassenmodell entwickelt, sondern mehrere aufeinanderfolgende Versionen des Klassenmodells. In jedes dieser Modelle müssen sich die Entwickler neu einarbeiten. Das heißt, dass nicht nur der Aufwand für die Einarbeitung in einen Graphen, sondern in eine Sequenz von Graphen erforderlich ist. Die Entwickler müssen sich also mehrfach während eines Projekts in ein neues Modell einarbeiten. Dieser Aufwand kann verringert werden, wenn das Layout der gleich gebliebenen Teile des Modells möglichst erhalten bleibt. Es kann verwirrend wirken, wenn die Position von Klassen im Klassendiagramm nach der Änderung in großem Maße von der Position vor der Änderung abweicht. Eine solche Abweichung des Layouts kann eine Änderung der *Mental Map* [MELS95] (also des geistigen Bildes) des Entwicklers das Klassenmodell betreffend verursachen. Das vergrößert den Aufwand bei der Einarbeitung in ein geändertes Modell. Um die Einarbeitungszeit möglichst gering zu halten sollte die Änderung der Mental Map der Entwickler gering gehalten werden.

Aus dieser Problemstellung heraus wird in dieser Diplomarbeit ein Algorithmus zum Layout einer Sequenz von Graphen vorgestellt. Da dieser Algorithmus auch innerhalb eines Projekts genutzt werden soll, ist die Sequenz der Graphen nicht notwendigerweise vollständig. Konkret bedeutet dies, dass ein Entwickler zu Anfang eines Projekts nicht unbedingt wissen kann, wie das Klassenmodell zum Ende des Projekts aussehen wird. Da es sich hier um inhaltlich aufeinander folgende Graphen handelt, werden die Unterschiede zwischen zwei aufeinander folgenden Graphen durch Graphtransformationen realisiert. Daher bietet es sich an, den Layoutalgoritmus in das Graphtransformationstool AGG (siehe Abschnitt 3.3.1) zu integrieren. Dieses Layout wird hier als evolutionäres Layout bezeichnet, da auch das Layout von aufeinander folgenden Graphen aufeinander aufbaut. Dabei soll das Layout der einzelnen Graphen möglichst ähnlich sein. Dadurch soll der Wiedererkennungswert der aufeinander folgenden Graphen einer Sequenz hoch und das Ausmaß der notwendigen Änderung der Mental Map der Entwickler somit gering gehalten werden. Für die Berechnung des Layouts der Einzelgraphen kann Vorwissen über die Art und den Anwendungszweck der Graphen herangezogen werden. So können spezielle Anforderungen an das Layout von bestimmten Graphen realisiert werden. Ein Beispiel für eine solche Anforderung ist die Erwartung, dass in einem Klassenmodell in einer Vererbungsstruktur die Superklasse oberhalb ihrer Subklassen angeordnet ist.

Die hier bearbeitete Problemstellung lässt sich wie folgt zusammenfassen: Eine Se-

quenz von Klassenmodellen, die jeweils aufeinander aufbauen, soll so gelayoutet werden, dass die Einarbeitungszeit eines Entwicklers in diese Sequenz möglichst gering ist. Das gleiche gilt auch für Teile der Sequenz, wenn ein vorhergehendes Teilstück schon bekannt ist. Eine Einschränkung ist dadurch gegeben, dass die Sequenz der bisherigen Klassendiagramme während eines Projektverlaufs nicht als abgeschlossen gelten kann. Die Einarbeitungszeit kann am Begriff der Mental Map festgemacht werden. Die schnelle Bildung einer Mental Map ist eng mit der Qualität (Lesbarkeit) des Layouts eines Graphen verbunden. Die Erhaltung einer Mental Map hängt von der Differenz zwischen den Layouts zweier Graphen ab. Beides lässt sich anhand entsprechender Metriken messen (siehe Abschnitt 3.2). Die hierzu existierenden Vorgängerarbeiten (siehe Kapitel 3) behandeln weder das Layout von unvollständigen Graphsequenzen, noch betrachten sie spezielle Qualitätsmerkmale des Layouts von Klassenmodellgraphen. Dies wird im hier vorgestellten Lösungsansatz geleistet. Die Grundidee besteht darin, inhaltlich aufeinanderfolgende Graphen durch Graphtransformationen zu beschreiben. Dabei werden Regeln angewendet, die einen Graphen in seinen Nachfolger überführen. Diese Regeln können sich auf objektorientierte Inhalte beziehen, an die bestimmte Layouteigenschaften gekoppelt sind. Für das Layout kommt ein Spring Embedder Algorithmus (siehe Abschnitt 3.1) zum Einsatz. Dabei können Eigenschaften von Knoten (Klassen) und Kanten sowie die auf die objektorientierten Inhalte bezogenen Layouterwartungen die Berechnung des Layouts beeinflussen, so dass inhaltlich verwandte Klassen näher beieinander positioniert werden.

1.1 Gliederung

In diesem ersten Kapitel wird das Thema dieser Diplomarbeit kurz movitiert. Im Anschluss daran erfolgt in Kapitel 2 eine Definition des bearbeiteten Problems. Weiterhin werden in dieser Problembeschreibung Szenarien vorgestellt, in denen die Lösung des beschreibenen Problems hilfreich ist. In Kapitel 3 wird der Hintergrund dieser Arbeit erläutert. Dabei wird auf vorhandene Techniken, Lösungsansätze und bereits existierende Tools eingegangen.

Im darauf folgenden Kapitel 4 erfolgt eine ausführliche Beschreibung des eigenen Lösungsansatzes. Die praktische Umsetzung dieses Lösungsansatzes wird in Kapitel 5 erklärt. Das 6. Kapitel befasst sich mit Tests des Lösungsansatzes und seiner Implementierung an verschiedenen Beispielen.

In Kapitel 7 erfolgt eine Diskussion des eigenen Lösungsansatzes. Dabei wird das Erreichte kritisch beleuchtet und mit den Ergebnissen von existierenden Tools verglichen. Abschließend erfolgt in Kapitel 8 eine Zusammenfassung dieser Diplomarbeit. Im Anschluss daran gibt Kapitel 9 einen Ausblick auf zukünftige Problemstellungen, die auf den hier bearbeiteten Problem aufbauen. Dabei wird auch auf Erweiterungsmöglichkeiten des eigenen Lösungsansatzes und dessen Implementierung eingegangen.

2 Problembeschreibung

In dieser Arbeit wird das Problem des Layouts von aufeinander aufbauenden Klassenmodellen (Graphsequenzen) bearbeitet. Da die Reihe von Klassenmodellen im Verlauf eines Projekts nicht zwangsläufig abgeschlossen ist, werden hier nicht abgeschlossene Graphsequenzen betrachtet. Es können jederzeit weitere Graphen hinzugefügt werden. Schon bei einzelnen Graphen ist ein gutes Layout erforderlich um die Einarbeitungszeit in den Graphen zu minimieren. Bei einer Graphsequenz fällt diese Einarbeitungszeit bei jedem Graphen neu an. Daher ist es wichtig, dass der Wiedererkennungswert der Layouts zweier aufeinander folgender Graphen hoch ist. Das heißt, dass weiterhin bestehende Strukturen aufeinander folgender Graphen auch im Layout erhalten bleiben sollten. Dadurch fallen Änderungen eher auf und die Einarbeitung in den neuen Graphen wird erleichtert. In Abbildung 2.1 ist ein Graph in zwei verschiedenen Layouts (hierarchisch und radial) zu sehen. Es ist nicht auf den ersten Blick erkennbar, dass beide Darstellungen den selben Graphen zeigen. Eine mögliche Änderung des Graphen (z.B. das Hinzufügen eines Knotens) von einem Layout zum anderen wäre hier nicht leicht erkennbar. Dies verdeutlicht die Wichtigkeit eines wiedererkennbaren Layouts von aufeinanderfolgenden Graphen einer Sequenz für den Aufwand, der zur Einarbeitung in einen geänderten Graphen notwendig ist.

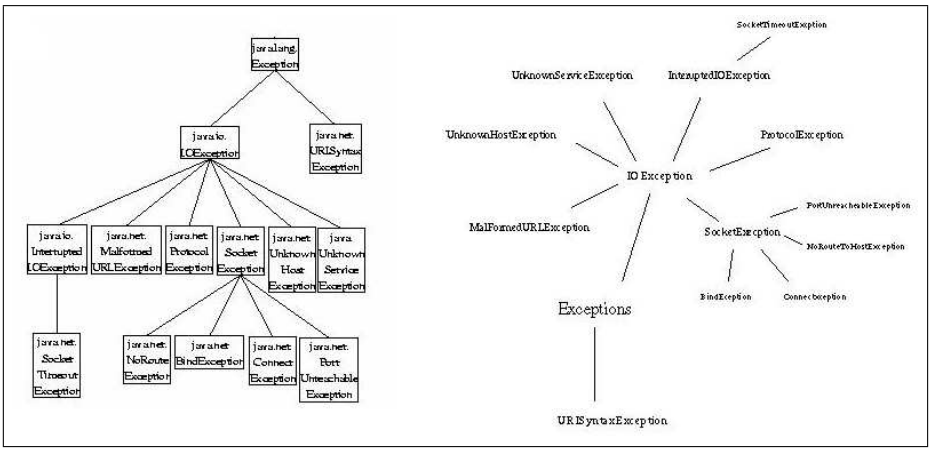

Abbildung 2.1: Vererbungsstruktur der EXCEPTION–Klassen des JAVA.NET–Pakets links in hierarchischer, rechts in radialer Form

In diesem Kapitel werden zwei Anwendungsfälle angeführt, in denen Sequenzen von Graphen erzeugt werden. Es wird erläutert, welche Anforderungen sich in diesen Fällen

an das Layout der einzelnen Graphen und der Sequenz als Ganzes ergeben. Weiterhin werden Probleme beschrieben, die speziell beim Layout von Graphsequenzen auftreten nicht aber beim Layout von Einzelgraphen.

2.1 Zugrunde liegende Probleme

Zuerst stellt sich das Problem, Klassenmodelle (und andere Graphen) automatisch so zu layouten, dass der Graph für einen Menschen gut lesbar ist. Dazu sind schon verschiedene Ansätze entwickelt worden, die in Abschnitt 3.1 erläutert werden. Die Lesbarkeit des Graphen wird dabei von mehreren Faktoren beeinflusst. Es sollten beispielsweise Knotenüberlappungen und Überschneidungen von Kanten und Knoten vermieden werden. Weiterhin ist die Einhaltung von Konventionen von Vorteil, die für das Layout von Klassendiagrammen und anderen speziellen Graphen existieren.

Darauf aufbauend ergibt sich das Problem des Layouts ganzer Sequenzen von aufeinander aufbauenden Graphen (bsp. Klassenmodellen). Dieses Problem ist als *Offline Graph Drawing Problem* bekannt. Dabei wird neben der Lesbarkeit der einzelnen Graphen besonderer Wert auf ein möglichst ähnliches Layout von aufeinander folgenden Graphen einer Sequenz gelegt. Dadurch wird bewirkt, dass der Benutzer sein einmal gebildetes geistiges Bild des Graphen (also seine Mental Map) im Verlauf der Seuqenz nur geringfügig ändern muss. Das soll den Betrachter dabei unterstützen, Änderungen des Graphen schneller zu erfassen. Das Offline Graph Drawing Problem ist allerdings auf vollständig bekannte Graphsequenzen beschränkt. Ein Lösungsansatz zum Offline Graph Drawing Problem ist das *Foresighted Graphlayout* von Stephan Diehl ([DGK00]), das in Abschnitt 3.1.1 erklärt wird.

2.2 Problemdefinition

In dieser Arbeit wird eine Erweiterung des Offline Graph Drawing Problem betrachtet. Die Erweiterung besteht darin, dass das Layout von unvollständigen aber inhaltlich aufeinander aufbauenden Graphen betrachtet wird.

Gegeben sei eine Sequenz von Klassenmodellen $K_1, ..., K_n$. Für jedes dieser Klassenmodelle soll ein Layout $L_1, ..., L_n$ erstellt werden, welches die folgenden Bedingungen erfüllt:

1. Jedes einzelne Layout L_i ist nach objektiven Layoutqualitätsmaßstäben möglichst optimal.

2. Die Differenz zwischen zwei aufeinander folgenden Layouts L_i und L_{i+1} soll möglichst gering sein. Dadurch kann die Mental Map des Entwicklers das Klassenmodell betreffend über die gesamte Sequenz hinweg größtenteils erhalten bleiben. Das heißt konkret, dass Klassen, die in mehreren Klassenmodellen K_i der Sequenz enthalten sind, nach Möglichkeit in jedem zugehörigen Layout L_i an der gleichen Position dargestellt werden sollten.

3. Zur Berechnung der optimalen Layoutsequenz muss die Sequenz der Klassendiagramme nicht vollständig vorliegen.

4. Jedes einzelne Layout sollte auf die Erwartungen eines Betrachters an das Aussehen eines Graphen mit spezieller Semantik (beispielsweise eines Softwareentwicklers in Bezug auf das Aussehen eines Klassenmodells) zugeschnitten sein. Es ist unter anderem üblich, eine Superklasse oberhalb ihrer Subklassen darzustellen. Dadurch wird das Erkennen bestimmter Strukturen im Klassenmodell erleichtert und eine Mental Map wird schneller aufgebaut.

Die Bedingung 1. und 2. sind auch Bestandteil des schon genannten Offline Graph Drawing Problem. Die dritte und vierte Bedingung wurden dabei bisher nicht betrachtet. Sie sind aber ein essentieller Bestandteil des Problems Graphtransformationssequenzen (oder auch unvollständige Graphsequenzen) zu layouten. Der oben erwähnte Lösungsansatz zum Offline Graph Drawing Problem ist auf die Erweiterung um die dritte und vierte Bedingung nicht anwendbar, da er eine vollständig bekannte Sequenz von Graphen voraussetzt. Daraus ergibt sich die Notwendigkeit eines neuen Lösungsansatzes.

Teilprobleme

Um das oben erläuterte Gesamtproblem zu lösen müssen folgende Teilprobleme betrachtet werden:

- Welche Layoutart eignet sich für Klassenmodelle aber auch für allgemeine Graphen?
- Wie kann die Qualität von Graphlayouts gemessen werden?
- Wie können die besonderen Qualitätsmerkmale des Layouts von Klassenmodellen gemessen werden?
- Wie kann die Differenz und damit der Wiedererkennungswert zwischen zwei Layouts gemessen werden?
- Integration des Layoutalgorithmus in das Tool AGG.

Für das Problem des Messens der Qualität und des Wiedererkennungswerts von Graphlayouts existieren bereits Lösungsansätze. Diese Anätze werden in Abschitt 3.2 erläutert.

2.3 Mögliche Szenarien

Es gibt verschiedene Anwendungen bei denen mit Sequenzen von Graphen gearbeitet wird. In dieser Arbeit wird exemplarisch die fortlaufende Änderung eines Klassendiagramms im Verlauf eines IT–Projekts betrachtet. Da der Lösungsansatz auf Graphtransformation beruht, wird zusätzlich die Erzeugung einer Sequenz von Graphen durch regelbasierte schrittweise Transformation eines allgemeinen Graphen vorgestellt.

2.3.1 Sequenzen von Klassendiagrammen

Im Verlauf eines IT–Projekts werden kontinuierlich Änderungen am Klassendiagramm vorgenommen. Dies ist erforderlich, wenn sich die Anforderungen an das zu enwickelde System ändern. Es ist auch möglich, dass die Entwicklung des Systems ausgehend von einem Prototypen mit kleinem Funktionsumfang durch schrittweises Hinzufügen von Funktionen erfolgt. Dabei ändert sich das Klassendiagramm im Projektverlauf mehrfach. Beim Projektstart ist die Entwicklung der Änderungen nicht vollkommen absehbar. Auch zu einem späteren Zeitpunkt steht die weitere Entwicklung nicht unbedingt fest. Die erzeugte Sequenz von Graphen ist also erst nach Projektabschluss vollständig bekannt.

Die Klassendiagramme werden meist mit Hilfe von sogenannten CASE–Tools (*Computer Aided Software Engineering*) bearbeitet und nach dem UML–Standard [BRJ99] als Graphen dargestellt. CASE–Tools sind Werkzeuge, die den Softwareentwicklungsprozess unterstützen. Solche Tools bieten oft einen automatischen Layoutalgorithmus, der das Diagramm nach objektiven Kriterien möglichst optimal layoutet. Dabei kann es vorkommen, dass ein Klassendiagramm nach einer Änderung völlig anders gelayoutet wird als vor der Änderung. Dadurch ist der Wiedererkennungswert zwischen zwei aufeinander folgenden Versionen des Klassendiagramms unter Umständen sehr gering. Dadurch erhöht sich der kognitive und zeitliche Aufwand, den ein Entwickler betreiben muss um sich in ein geändertes Klassendiagramm einzuarbeiten.

Es ist aber zu erwarten, dass im Verlauf des Projekts in einem Entwicklungsschritt nicht das ganze Klassendiagramm neu entworfen wird. Die am häufigsten auftretenden Änderungen werden das Hinzufügen von neuen Klassen und Assoziationen zwischen Klassen sein. Weitere zu erwartende Änderungen sind die Erweiterung von Vererbungsstrukturen, das Aufsplitten einer Klasse in mehrere Klassen, das Zusammenführen von Klassen sowie das Löschen von nicht mehr benötigten Klassen und Assoziationen. Das heißt, es werden von einer Version des Klassendiagramms zur nächsten große Teile des Graphen erhalten bleiben. Um einen hohen Wiedererkennungswert zwischen zwei aufeinanderfolgenden Klassendiagrammversionen zu erhalten sollten die gleich bleibenden Teile in beiden Versionen möglichst ähnlich dargestellt werden. Erhalten bleibende Klassen sollten daher ihre Position (zumindest relativ zueinander, wenn nicht absolut) im Layout behalten. Zudem wäre es wünschenswert, dass sich die oben genannten Änderungen, wie das Aufsplitten und Zusammenführen von Klassen, im Layout wiederfinden. Weiterhin sollten Erwartungen an das Layout bestimmter Strukturen im Klassendiagramm erfüllt werden. Es ist beispielsweise üblich, dass Subklassen unterhalb ihrer Superklassen dargestellt werden.

2.3.2 Allgemeine Graphsequenzen

Allgemeine Graphsequenzen können mit Hilfe von Graphtransformationen erzeugt werden. Dabei wird aus einem Ursprungsgraphen durch Anwendung einer Transformationsregel ein neuer Graph erstellt. Die grundlegenden Änderungen sind hier das Hinzufügen und das Löschen von einzelnen Knoten und Kanten. Diese elementaren Transformationen können beliebig komplex zu einer Regel zusammengefasst werden. Der Ergebnisgraph

einer Transformation dient dann als Ursprungsgraph für den nächsten Transformationsschritt. Dadurch entsteht eine Sequenz von Graphen. Da die Transformationsregeln nichtdeterministisch ausgewählt und auf passende Graphteile angewendet werden, ist die Graphsequenz hier nicht von vornherein bekannt. Erst wenn die Transformationen abgeschlossen sind, kennt man alle Graphen der Sequenz. Eine etwas ausführlichere Beschreibung von Graphtransformationen und dem Graphtransformationstool AGG erfolgt in Abschnitt 3.3. In dem Tool AGG können beliebige Transformationen an Graphen vorgenommen und der Ergebnisgraph nach jedem Transformationsschritt angezeigt werden. Dabei ist es wichtig, Änderungen am Graphen schnell erfassen zu können. Dazu sollten auch hier gleich bleibende Teile des Graphen vor und nach der Änderung möglichst ähnlich dargestellt werden. Außerdem sollten Layoutpattern, die in den Transformationsregeln enthalten sind, im Layout des erzeugten Graphen wiederzufinden sein. Ein solches Layout Pattern wäre zum Beispiel, dass ein neu hinzgefügter Knoten nicht oberhalb eines bereits vorhandenen Knotens dargestellt werden darf. Dabei sollten die Layouts der einzelnen Graphen der Sequenz möglichst optimal sein.

2.4 Konkrete Probleme des Layouts von Graphsequenzen anhand der Szenarien

Das fortlaufende Layout von nicht im voraus bekannten Graphsequenzen beinhaltet einige Probleme, die beim Layouten eines einzelnen Graphen nicht auftreten. Das schwerwiegendste dieser Probleme ist die Vergrößerung des Platzbedarfs des Graphen, die durch das Hinzufügen von Knoten verursacht wird. Da die zukünftigen Graphen bei der Berechnung des Layouts des aktuellen Graphen nicht bekannt sind, kann der zukünftige Platzbedarf nur geschätzt werden. Diese Schätzung könnte durch Vorwissen (speziell: der Vorstellung über die weitere Entwicklung des Graphen) der Entwickler bzw. des Anwenders des Graphtransformationssystems verbessert werden. Wenn aber nach einer Änderung des Graphen der vorhandene Platz nicht ausreicht, muss der Layoutalgorithmus dieses Problem angemessen lösen können. Das kann durch „Anbauen“ am Rand des Graphen sowie Hochskalieren des gesamten Graphen oder einzelner Teile in Verbindung mit dem Verschieben von Teilgraphen erfolgen. Dadurch ergeben sich aber wieder neue Probleme bei der Kantenführung. Weiterhin besteht die Gefahr des Verlustes des optischen Zusammenhangs von Knoten. Ein weiteres Problem besteht in der Darstellung der Veränderungen in der Graphsequenz um zum Beispiel neu hinzugekommene oder entfernte Knoten und Kanten leicht erkennbar zu machen. Diese Probleme werden durch die Anforderung der Ähnlichkeit des Layouts zweier aufeinanderfolgender Graphen der Sequenz noch verstärkt. Dadurch können sich unter Umständen Platzprobleme in Teilgraphen ergeben, obwohl für die Knotenzahl des Graphen grundsätzlich genug Platz vorhanden ist. Solche Platzprobleme treten auf, wenn zu einem Cluster von „alten“ Knoten viele neue Knoten hinzugefügt werden sollen. Aber auch die Kantenführung neuer Kanten kann durch die Positionstreue alter Knoten erschwert werden.

3 Hintergrund und State of the Art

Im vorangegangenen Kapitel 2 wurde das in dieser Arbeit betrachtete Problem als eine Erweiterung des Offline Graph Drawing Problems beschrieben. Dabei wurde betont, dass der vorhandene Lösungsansatz für dieses Problem nicht auf die hier erfolgte Erweiterung anwendbar ist. An dieser Stelle wird in Abschnitt 3.1 neben verschiedenen Arten von Layoutalgorithmen das Offline Graph Drawing Problem und sein Lösungsansatz näher erläutert. Im Anschluss daran werden in Abschnitt 3.2 einige Metriken zur Qualität von Graphlayouts vorgestellt, mit denen sich einige der in der Problemdefinition aufgezählten Teilprobleme lösen lassen. Dabei wird sowohl auf die Messung der objektiven Qualtität eines Layouts als auch auf die Messung der Differenz zwischen zwei Layouts eingegangen. In Abschnitt 3.3 wird das Thema Graphtransformation erläutert, das den Hintergrund des in der Einleitung (Kapitel 1) skizzierten Lösungsansatzes bildet. Dabei wird der Aufbau von Graphgrammatiken und der Ablauf von Transformationen erklärt. Weiterhin wird in diesem Abschnitt auf das Graphtransformationstool AGG eingegangen. Abschließend werden in Abschnitt 3.4 bestehende Tools, die Graphen layouten können, kurz vorgestellt.

An dieser Stelle erfolgt zunächst eine Definition des Begriffs *Graph*, wie er im Rahmen dieser Arbeit verstanden wird.

Definition 3.1 *Ein Graph $g = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten $E \subseteq V \times V$. Eine Graphsequenz G besteht aus einer Folge von Graphen $g_1, g_2, ..., g_n$.*

Diese allgemeine Definition lässt nur eine Kante zwischen zwei Knoten zu. Die Definition eines Graphen g als Vier–Tupel $g = (V, E, s, t)$ mit einer Menge von Knoten V, einer Menge von Kanten E sowie zwei Relationen $s, t : E \rightarrow V$ die auf die Quell– (Source) und Zielknoten (Target) von Kanten verweisen, erlaubt auch mehrere Kanten zwischen zwei Knoten. Solche Kanten können das Layout eines Graphen verkomplizieren. Daher wird in dieser Arbeit auf die erstgenannte Definition zurückgegriffen, was sich auf Layouts von Graphen mit mehreren Kanten zwischen zwei Knoten so auswirkt, dass diese Kanten alle übereinander wie eine Kante dargestellt werden.

3.1 Layoutalgorithmen

Es gibt verschiedene Arten, Graphen zu layouten. Für allgemeine Graphen eignen sich vor allem Ansätze, bei denen die Knoten frei in der Zeichenebene (oder im Raum) positioniert werden können. Die Kanten verbinden dabei meist die Knoten auf gerader Linie. Einen solchen Ansatz stellen die Spring Embedder Modelle dar. Auch orthogonale

Layoutansätze eignen sich für alle Arten von Graphen. Dabei werden die Knoten an einem Raster ausgerichtet. Die Kanten verlaufen nicht mehr unbedingt auf gerader Linie zwischen den Knoten, sondern können Knickpunkte haben. Die Kantenabschnitte dürfen nur in horizontaler oder vertikaler Richtung verlaufen. Ein orthogonaler Layoutalgorithmus wird unter anderem im CASE–Tool Omondo UML Studio verwendet.

Es gibt auch Layoutarten, die für spezielle Arten von Graphen entwickelt wurden. Die Ansätze des hierarchischen Layouts eignen sich besonders für baumartige Graphen. Hierbei wird die Baumstruktur im Layout wiedergespiegelt. Der Wurzelknoten des Baums kann dabei beispielsweise ganz oben angeordnet werden. Die Knoten, die direkt mit dem Wurzelknoten verbunden sind, werden dann unterhalb des Wurzelknotens angeordnet (meist alle auf gleicher Höhe). Dies wird dann sukzessiv für alle Ebenen des Baums wiederholt. Dieser Ansatz kann sowohl mit einfacher Kantenführung als auch kombiniert mit einem orthogonalen Ansatz realisiert werden. Ein Spezialfall des hierarchischen Layouts ist das radiale Layout. Dabei werden die Knoten kreisförmig um den Wurzelknoten angeordnet.

Es gibt auch bereits Ansätze, die sich mit dem Layout von Graphsequenzen befassen. Ein Ansatz, der sich mit dem sogenannten *Offline Graph Drawing Problem*, also dem Layout von vollständig bekannten Graphen, beschäftigt, ist das „Foresighted Graphlayout“ von Stephan Diehl. Dieser Ansatz wird in Abschnitt 3.1.1 erklärt.

3.1.1 Foresighted Graphlayout

Das *Foresighted Graphlayout* von Stephan Diehl ist ein Lösungsansatz für das in Abschnitt 2.1 angeführte Offline Graph Drawing Problem. Dieses Problem betrachtet das Layout von Graphsequenzen. Eine Vorbedingung dieses Problems ist, dass die darzustellende Graphsequenz vollständig bekannt sein muss. Das Layout soll dann folgende Bedingungen erfüllen:

- Das Layout jedes Graphen der Sequenz sollte nach objektiven Qualitätsmerkmalen möglichst optimal sein.
- Die Mental Map des Betrachters sollte möglichst über die gesamte Graphsequenz erhalten bleiben. Das heißt, dass Knoten und Kanten, die in mehreren Graphen der Sequenz enthalten sind, nach Möglichkeit immer an der gleichen Position dargestellt werden sollten.

Diehl definiert den Begriff Graph ähnlich wie er in dieser Arbeit verwendet wird (siehe Def. 3.1). Eine Graphsequenz bezeichnet er allerdings als Graphanimation. Die Grundidee des Foresighted Graphlayout besteht darin, alle Graphen einer Sequenz zu einem *Supergraphen* zusammenfassen. Dieser Supergraph enthält alle Knoten und Kanten die in den einzelnen Graphen der Sequenz enthalten sind. Knoten und Kanten, die in mehreren Einzelgraphen enthalten sind, erscheinen nur einmal im Supergraphen. Daher ist es wichtig, dass die Bezeichnung der Knoten und Kanten über die gesamte Sequenz hinweg konsistent ist. Dieser Supergraph kann dann mit einem herkömmlichen Layoutalgorithmus für einzelne Graphen gelayoutet werden. Im Anschluss daran wird jeder Einzelgraph

mit den Layoutinformationen des Supergraphen dargestellt. Dadurch werden Knoten und Kanten in allen Einzelgraphen, in denen sie enthalten sind, gleich positioniert.

Dieser Supergraph kann sehr große Ausmaße annehmen. Das gilt besonders für Graphsequenzen mit vielen Änderungen. Dabei kann es vorkommen, dass die Layouts der Einzelgraphen große leere Flächen aufweisen. Dadurch kann das Layout schnell unübersichtlich werden. Der Grund dafür besteht darin, dass die Einzelgraphen (zum Teil sehr kleine) Teilgraphen des Supergraphen sind. Viele Knoten sind nur in einer kleinen Anzahl von Einzelgraphen enthalten. Die Menge der Einzelgraphen, in denen ein Knoten enthalten ist, wird als Lebenszeit bezeichnet. Um ein übersichtlicheres Layout mit weniger Leerflächen berechnen zu können muss der Supergraph verkleinert werden. Diehl führt hierzu die Graphpartitionierung an. Dabei können mehrere Knoten v durch einen neuen Knoten $\tilde{v}$ repräsentiert werden. Die Kanten zwischen zwei Knoten im Graphen werden zu Kanten zwischen den sie repräsentierenden Knoten in der Graphpartitionierung. Der Supergraph wird so partitioniert, dass nur Knoten v zu einem neuen Knoten $\tilde{v}$ zusammengefasst werden, deren Lebenszeit disjunkt ist. Diehl nennt den so zusammengefassten Supergraphen eine *Graph Animation Partitioning* (GAP). Dieser GAP kann noch weiter verkleinert werden, indem die Kanten des GAP nach dem gleichen Schema zusammengefasst werden wie die Knoten bei der Erzeugung des GAP. Das Ergebnis wird dann als *Reduced Graph Animation Partitioning* (RGAP) bezeichnet. Diehl weist nach, dass die Erzeugung eines minimalen GAP sowie die Erzeugung eines minimalen RGAP $\mathcal{NP}$–vollständige Probleme sind. Minimal bedeutet in diesem Zusammenhang ein GAP mit der niedrigst möglichen Anzahl von Knoten beziehungsweise ein RGAP mit der niedrigst möglichen Anzahl von Kanten. Es werden jedoch Algorithmen vorgestellt mit denen sich nicht minimale aber gute GAPs und RGAPs erzeugen lassen, die nur quadratischen Aufwand haben. Der so erzeugte GAP bzw. RGAP wird dann, wie vorher der gesamte Supergraph, mit einem Layoutalgorithmus nach Wahl gelayoutet. Auch hier ergibt sich die Positionierung der Knoten der Einzelgraphen aus den Layoutinformationen der sie repräsentierenden Knoten des GAP bzw. RGAP. Empirische Ergebnisse zeigten, dass die aus den nicht minimalen GAPs und RGAPs erzeugten Layouts wesentlich besser (nach Layoutqualitätsmerkmalen) ausfielen, als die aus dem einfachen Supergraphen erzeugten Layouts. Aus minimalen GAPs und RGAPs berechnete Layouts boten aber oft keine weitere Verbesserung.

In Abbildung 3.1 ist eine Graphsequenz zu sehen, die aus nur drei Graphen besteht. In diesem Beispiel bestehen die Änderungen nur aus dem Hinzufügen von Knoten und Kanten. In der oberen Reihe wurden die Graphen einzeln gelayoutet. Dabei fällt auf, dass die Knoten, die im ersten Graphen zu sehen sind, im zweiten und dritten Graphen jeweils andere Positionen haben. In der unteren Reihe wird das Ergebnis des Foresighted Graphlayout–Algorithmus gezeigt. Hier behalten die Knoten aus dem ersten Einzelgraphen ihre Positionen auch im zweiten und dritten Einzelgraphen. Die Entwicklung der Sequenz vom Ausgangsgraphen zum Ergebnisgraphen über einen Zwischenschritt ist gut nachvollziehbar. Allerdings ist das Layout des ersten und zweiten Einzelgraphen nicht optimal. Vor allem die Kantenführung ist irritierend. Weiterhin wird mehr Platz benötigt, als in der oberen Darstellung.

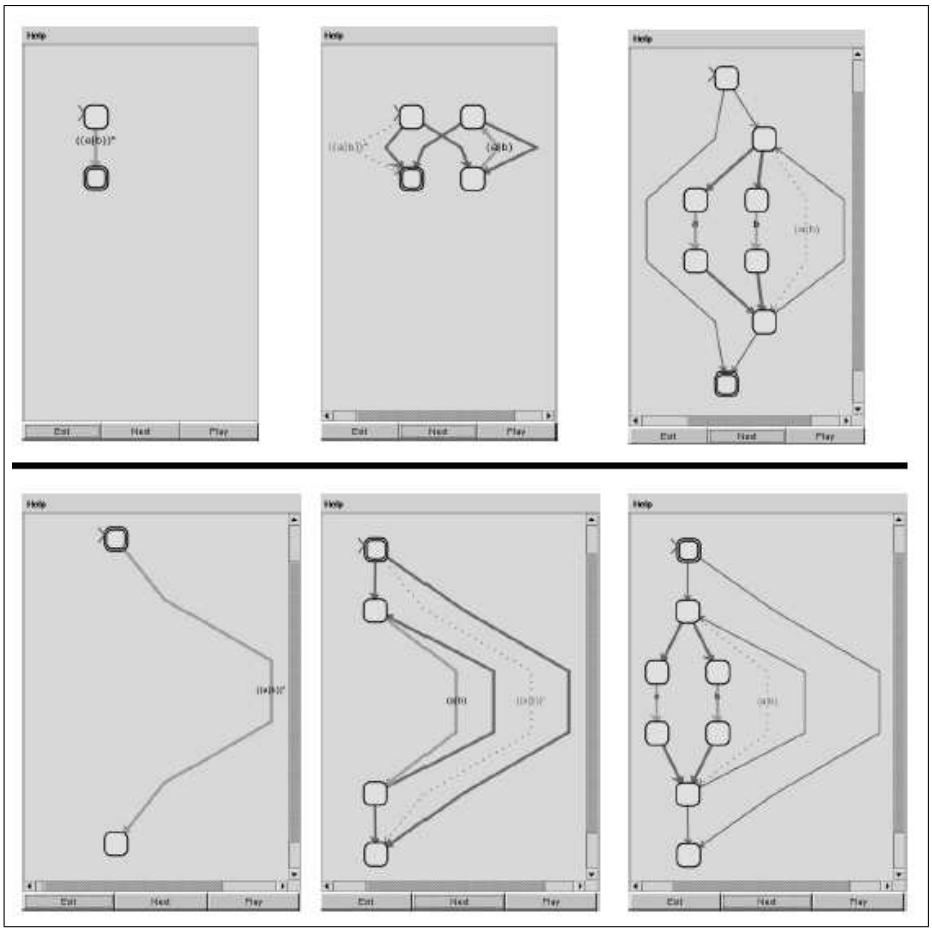

Abbildung 3.1: Einzellayout und Foresighted Graphlayout der Entwicklung eines finiten Zusandsautomaten [DGK00]

Eine formale Beschreibung der für das Forsighted Graphlayout entwickelten Algorithmen kann in [DGK00] nachgelesen werden. Dort finden sich auch Aufwandsabschätzungen und Angaben zur Implementation dieser Algorithmen sowie weitere Beispiele, in denen die Ergebnisse des Foresighted Graphlayout demonstriert werden.

3.1.2 Spring Embedder Layout Modelle

Die Spring Embedder Layout Modelle gehören zu den sogenannten *Force–Directed* (energiegetriebenen) Ansätzen zum Layout von Graphen. Force–Directed Algorithmen basieren auf einem Energiemodell. Dieses Energiemodell besteht aus anziehenden und abstoßenden Kräften, die zwischen den Knoten und Kanten eines Graphs wirken. Der Graph wird als vereinfachtes physikalisches System gesehen. Dabei wird das Layout vom Zusammenspiel der Kräfte bestimmt. Das Layout–Problem wird dabei zum Optimie-

rungsproblem. Das Layout ergibt sich aus der Minimierung der (physikalisch gesehen) potentiellen Energie in dem System Graph.

Der Begriff Spring Embedder Layout wurde von Eades in seiner Arbeit „ A Heuristic for Graph Drawing“ geprägt [Ead84]. Das Energiemodell von Eades stellt die Kanten des Graphen als Federn dar, die Anziehungskräfte auf ihre Endpunkte (Knoten) ausüben. Die Knoten selbst werden als positive elektrische Ladungen gesehen, die sich gegenseitig abstoßen. Die Abstraktion im Vergleich mit einem realen physikalischen System besteht darin, dass hier nur potentielle Energien betrachtet werden. Das Modell beinhaltet keine Faktoren wie Beschleunigung, Trägheit, Reibung oder sonstige Effekte, die in einem realen physikalischen System auftreten können. In Abbildung 3.2 ist das Energiemodell nach Eades illustriert. Die anziehenden Federkräfte f_a werden dabei durch die folgende logarithmische Funktion bestimmt, die abstoßenden Kräfte f_r (r: repulsive) zwischen den Knoten durch eine Wurzelfunktion (inverse quadratische Funktion):

$$f_a(d) = \frac{k}{\log d} \tag{3.1}$$

$$f_r(d) = \frac{k}{d^2} \tag{3.2}$$

Die Kräfte werden in Abhängigkeit von der Entfernung d der Knoten des jeweils betrachteten Knotenpaars berechnet. Der Faktor k beschreibt hierbei die Größe der Umgebung eines Knotens in der kein anderer Knoten liegen soll.

Als Ausgangspunkt für den Optimierungsalgorithmus in Eades Spring Embedder Modell dient eine zufällig generierte Positionierung der Knoten. Der Optimierungsalgorithmus selbst arbeitet eine festgelegte Anzahl von Iterationen ab. In jeder Iteration werden die an den Knoten auftretenden Kräfte berechnet. Dabei werden die wechselseitigen abstoßenden Kräfte zwischen den Knoten aufsummiert. Dazu kommen die durch eventuell vorhandene Kanten entstehenden Anziehungskräfte. Dann wird für jeden Knoten eine Positionsänderung so berechnet, dass die vorher ermittelte potentielle Energie verringert wird. Das Ziel dabei ist ein möglichst ausgeglichenes System, also möglichst wenig potenzielle Energie an den einzelnen Knoten. Der Zeitaufwand für die Berechnung der abstoßenden Kräfte ist quadratisch bezüglich der Knotenzahl, da sie für jedes Knotenpaar erfolgen muss. Da die Berechnung der anziehenden Kräfte nur für Knotenpaare erfolgt, die durch eine Kante verbunden sind, verursacht sie nur linearen Aufwand bezüglich der Kantenanzahl. Für eine Iteration des Algorithmus ergibt sich also bei einem Graphen $g = (V, E)$ ein Aufwand von $\Theta(\mid V \mid^2, \mid E \mid)$.

Da der Algorithmus verhältnismäßig komplexe logarithmische Berechnungen verwendet, ist er nur bedingt für das Layout von großen Graphen geeignet. Eades gibt an, dass der Algorithmus für Graphen mit bis zu fünfzig Knoten gedacht ist. Weiterhin stellte er fest, dass der Algorithmus bei stark verbundenen (Teil–) Graphen weniger gute Ergebnisse bietet als bei Graphen mit mäßiger Kantenanzahl.

Ein weiterer Ansatz zum Force Directed Layout wird von Fruchterman und Reingold in „Graph Drawing by Force–directed Placement“ vorgestellt [FR91]. Dieser Ansatz basiert auf der Arbeit von Eades. Der Optimierungsalgorithmus von Fruchterman und Reingold

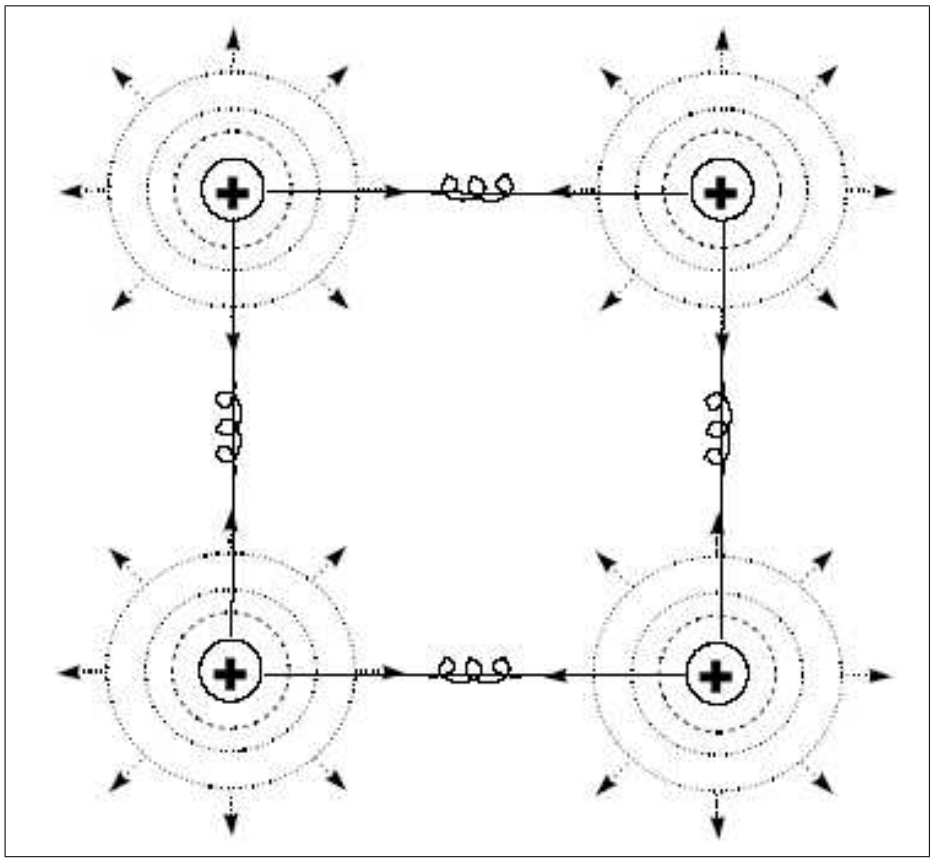

Abbildung 3.2: Energiemodell nach Eades [Ead84]

arbeitet nach dem gleichen Schema wie der von Eades. Auch hier werden in jeder Iteration die anziehenden und abstoßenden Kräfte für alle Knoten des Graphen berechnet und in Positionsänderungen umgesetzt. Die Positionsänderungen werden dabei von der sogenannten Temperatur begrenzt. Die Temperatur beziffert die Bewegungsfreiheit der Knoten nach dem Vorbild der Schwingungen beispielsweise von Atomen im Metallgitter. Je höher die (physikalische) Temperatur ist, desto stärker werden die Schwingungen der Atome. Die hier verwendete Temperatur wird nach jeder Iteration durch eine sogenannte *cooling*–Funktion veringert. Die Möglichkeit zur Positionsänderung der Knoten wird also mit jeder Iteration weiter eingeschränkt.

Der hauptsächliche Unterschied zum Spring Embedder Modell von Eades besteht in der Vereinfachung der Berechnung der zwischen den Knoten wirkenden Kräfte in dem ansonsten sehr ähnlichen Energiemodell. Auch Fruchterman und Reingold sehen abstoßende Kräfte zwischen allen Knoten und anziehende Kräfte nur zwischen Knoten, die durch eine Kante verbunden sind. Allerdings wurde die Formel zur Berechnung der anziehenden Kräfte f_a zu einer quadratischen Funktion vereinfacht. Die abstoßenden Kräfte werden ebenfalls durch eine quadratische Funktion berechnet:

$$f_a(d) = \frac{d^2}{k} \tag{3.3}$$

$$f_r(d) = \frac{-k^2}{d} \tag{3.4}$$

Der Aufwand für eine Iteration dieses Algorithmus liegt bei einem Graphen $g = V, E$ in der Klasse $\Theta(\mid V \mid^2, \mid E \mid)$. In dieser Aufwandsklasse liegt auch der Algorithmus von Eades. Durch die wesentlich einfacher zu berechnende Funktion der anziehenden Kräfte ergeben sich aber merkliche Laufzeitunterschiede. Der Algorithmus von Fruchterman und Reingold ist auch für das Layout größerer Graphen in akzeptabler Zeit geeignet.

Einen weiteren Ansatz zum Force-Directed Layout bieten Kamada und Kawai in „An Algorithm for Drawing General Undirected Graphs“ [KK89]. Anders als bei Eades werden hier die abstoßenden Kräfte nicht als elektrische Ladungen modelliert. Kamada und Kawai verbinden jedes Knotenpaar des Graphen durch eine Feder, durch die abstoßende und anziehende Kräfte gleichzeitig dargestellt werden. Den Federn wird eine Ausgangslänge (Ruhelänge) zugeordnet. Die Ausgangslänge einer Feder ist abhängig von der Länge des kürzesten Pfades zwischen ihren Endknoten im Graph. Ist der Abstand zweier Knoten größer als die Ausgangslänge der sie verbindenden Feder, so werden beide Knoten näher zueinander gezogen. Im umgekehrten Fall werden die Knoten auseinander geschoben. Der Optimierungsalgorithmus von Kamada und Kawai verändert pro Iteration nur die Position eines Knoten, im Gegensatz zu Eades, bei dem alle Knoten bewegt werden. Dabei wird jeweils der Knoten betrachtet, an dem die stärksten Federkräfte wirken. Der Knoten wird so verschoben, dass ein lokales Minimum der Kräfte erreicht wird.

Auch dieser Ansatz ist für große Graphen weniger geeignet, da er eine Berechnung der kürzesten Pfade zwischen allen Knotenpaaren erfordert. Diese Berechnung hat sowohl zeitlich ($\Theta(\mid V \mid^3)$) als auch den benötigten Platz betreffend ($\Theta(\mid V \mid^2)$) einen hohen Aufwand.

Es wurden noch weitere Energiemodelle zum Layout von Graphen entwickelt. Dazu zählen die Modelle von Frick [FLM94], Davidson und Harel [DH96] sowie das LinLog Modell von Lewerentz und Noack [LN03].

3.2 Qualität von Graphlayouts

Unter der Qualität des Layouts eines Graphen versteht man im Allgemeinen die Auswirkung des Layouts auf die Lesbarkeit des Graphen. Faktoren, die die Lesbarkeit eines Graphen beeinflussen, lassen sich messen und durch Metriken bewerten. In dieser Arbeit wird besonders das Layout von Sequenzen von Graphen im Hinblick auf den Wiedererkennungswert zwischen zwei aufeinander folgenden Graphen einer Sequenz betrachtet. Daher ist es erforderlich, neben der Lesbarkeit auch den Wiedererkennungswert messen zu können. Dies kann mit dem Konstrukt der mentalen Distanz erreicht werden, das ein Maß für den Wiedererkennungswert zweier Graphen darstellt. Es wurden bereits Metriken entwickelt, mit denen die mentale Distanz zwischen Graphlayouts gemessen werden kann. Im Folgenden werden Metriken zur allgemeinen Qualität von und zur mentalen Distanz zwischen Graphlayouts beschrieben.

3.2.1 Allgemeine Qualitätsmetriken

Es existieren verschiedene Qualitätsmaßstäbe für Graphlayouts. Diese Qualitätsmaßstäbe werden auch ästhetische Kriterien genannt. Zu den wichtigsten ästhetischen Kriterien gehören die Anzahl von Knotenüberlappungen, Überschneidungen von Knoten und Kanten sowie Kantenkreuzugen. Dabei gilt, je geringer die Anzahl desto höher die Layoutqualität. Weitere Krieterien für gutes Layout sind Symetrie, einheitliche Kantenlängen und die Ausnutzung des zur Verfügung stehenden Platzes. Bei orthogonalen Layouts kann auch die Anzahl der Knickpunkte in Kanten als ästhetisches Kriterium herangezogen werden. Davidson und Harel nutzen in [DH96] eine Reihe dieser Kriterien für eine Gütefunktion für Layouts. Sie optimieren diese Gütefunktion um ein Layout für einen Graphen zu berechnen. In [CP96] stellen Coleman und Parker den Layoutalgorithmus AGLO vor, der auf einem ähnlichen Ansatz basiert.

Purchase et al. haben die Auswirkungen von unterschiedlichen Layouts auf die Lesbarkeit von UML–Klassendiagrammen untersucht [PMCC01]. Dazu haben sie unter anderem die Auswirkungen von verschiedenen Ausprägungen einzelner ästhetischer Kriterien betrachtet. Zu den untersuchten Kriterien gehörten die Anzahl der Knickpunkte in Kanten, die Platzausnutzung, die Länge von Kanten sowie die Variation der Kantenlängen. Die Ergebnisse der durchgeführten Experimente waren nur in wenigen Fällen signifikant (z.B. bei der Anzahl der Knickpunkte). Zusätzlich widersprachen diese Ergebnisse den Erwartungen und waren schwer interpretierbar. Man kam zu dem Schluss, dass für die Lesbarkeit von Klassenmodellen die allgemeinen ästhetischen Kriterien weniger relevant sind. Purchase et al. stellen die Vermutung auf, dass die Semantik der Klassendiagramme und das Wiedererkennen von semantischen Strukturen im Layout eine grössere Bedeutung beim Verständnis von Klassendiagrammen haben.

3.2.2 Metriken zu mentaler Distanz

Das Konstrukt der mentalen Distanz beschreibt die Ähnlichkeit zweier Layouts eines Graphen. Eine geringe mentale Distanz bedeutet hierbei eine große Ähnlichkeit und damit einen hohen Wiedererkennungswert der Layouts.

Stina Bridgeman und Roberto Tamassia haben eine Suite von Metriken entwickelt, mit der man die mentale Distanz zwischen zwei Layouts messen kann [BT98]. Dabei wurden unter anderem folgende Faktoren untersucht:

- **Abstand:** Hier wird der Abstand zwischen den Positionen eines Knotens in beiden Layouts gemessen und für alle Knoten aufsummiert. Die Abstandsberechnung kann hierbei nach verschiedenen Verfahren, wie dem Euklidischen Abstand oder dem Hausdorff–Abstand, erfolgen. Je kleiner die Summe aller Abstände, desto kleiner ist die mentale Distanz zwischen den Layouts.

- **Clustererhaltung:** Hierbei wird die Erhaltung von Knotenclustern untersucht. Als Cluster wird dabei eine Ansammlung von Knoten verstanden, die nah beieinander angeordnet sind. Dabei können vordefinierte Cluster betrachtet werden. Cluster können auch zu jedem Knoten bestimmt werden. Dabei gelten alle Knoten die innerhalb eines Abstand *epsilon* zu dem Knoten liegen als ein Cluster. Dann werden für alle Knoten die zum jeweiligen Cluster gehörenden Knoten untersucht. Je mehr Übereinstimmungen in der Clusterzusammensetzung der beiden Layouts gefunden werden, desto kleiner ist die mentale Distanz.

- **Erscheinungsform der Kanten:** Diese Metrik betrachtet ausschließlich die Kanten des Graphen. Dabei werden die Anzahl der Knickpunkte einer Kante und die Reihenfolge der horizontalen und vertikalen Kantenabschnitte verglichen. Die Differenz der Anzahl der Knickpunkte wird aufsummiert. Je kleiner die Summe der Differenzen, desto kleiner ist auch die mentale Distanz. Haben die Kanten keine Knickpunkte kann auch die Richtung der Kanten betrachtet werden. Auch hier werden die Differenzen (z.B. der die Richtung angebenden Winkel) aufsummiert.

3.3 Graphtransformation

Eine Graphtransformation ist die Überführung eines Graphen in einen neuen Graphen anhand einer Transformationsregel. Eine Graphgrammatik besteht aus einem Graphen und einer Menge von Transformationsregeln. Weiterhin kann eine Graphgrammatik auch zusätzliche Bedingungen enthalten, an die eine Regelausführung gebunden werden kann. Ein Graph, bestehend aus einer Menge von Knoten und Kanten, stellt den Ausgangspunkt für die Transformationen dar.

Eine Transformationsregel schreibt vor, wie der Graph geändert werden soll. Mögliche Änderungen sind das Hinzufügen und das Entfernen von Knoten und Kanten sowie das Ändern von Attributen der Knoten. Eine Regel besteht im Allgemeinen aus zwei Teilen (links und rechts). Beide Regelteile werden als Graph dargestellt. Der erste Teil (links) stellt den Zustand des Teils des Graphen, der von der Änderung betroffen ist, vor der

Transformation dar. Der zweite Teil (rechts) der Regel stellt den entsprechenden Teilgraphen nach der Transformation dar. Im Extremfall kann ein Regelteil leer sein oder auch den ganzen aktuellen Graphen enthalten. Eine weitere Möglichkeit ist, dass beide Teile der Regel gleich sind. Eine solche Regel ist die sogenannte Identitätsregel. Eine Regel kann dabei beliebig komplex sein. Weiterhin kann eine Regel eine oder mehrere sogenannte Nichtanwendungsbedingungen enthalten. Auch diese werden als Graphen dargestellt. Dabei gilt, dass die Regel nicht angewendet werden soll, wenn im aktuell zu transformierenden Graphen ein Teilgraph existiert, der dem Graphen in der Nichtanwendungsbedingung entspricht. Desweiteren kann ein Mapping zwischen den Knoten der Graphen der linken und rechten Seite der Regel angegeben werden. Dieses Mapping gibt an, welche Knoten auf beiden Seiten gleich sind und erhalten bleiben sollen. Alle Knoten auf der linken Seite der Regel, für die kein Mapping existiert, werden bei der Transformation gelöscht. Analog dazu werden solche Knoten auf der rechten Seite des Graphen während der Transformation zum Graphen hinzugefügt. Eine einfache Graphgrammatik ist in Abbildung 3.4 in AGG–Darstellung gezeigt. Diese Grammatik beinhaltet neben einem Ausgangsgraphen mit nur einem Knoten nur eine Regel. Diese Regel bewirkt das Hinzufügen eines Knotens und einer Kante zwischen dem hinzugefügten und einem schon vorhandenen Knoten. Die Grammatik baut eine Baumstruktur auf. Weitere Beispiele für Graphgrammatiken werden in der Fallstudie in Kapitel 6 vorgestellt.

Eine Transformation ist dann die Anwendung einer Regel der Graphgrammatik. Der durch die Transformation entstandene Graph ist der Ausgangspunkt für die nächste Transformation. Bei der Anwendung einer Regel wird im aktuellen Graphen nach einem Teilgraphen gesucht, der dem linken Teil der Regel entspricht. Existiert kein solcher Teilgraph, kann die Regel nicht angewendet werden. Auch wenn ein Teilgraph gefunden werden kann, der der Nichtanwendungsbedingung entspricht, wird die Regel nicht ausgeführt. Wenn mehrere solcher Ansatzpunkte für eine Regel existieren, wird ein beliebiger dieser Teilgraphen ausgewählt. Dann wird der Graph so geändert, dass der ausgewählte Teilgraph dem Graphen auf der rechten Seite der Regel entspricht. Die Regeln können im Allgemeinen beliebig oft und in beliebiger Reihenfolge hintereinander ausgeführt werden. Da jede Transformation auf dem Ergebnisgraphen der vorherigen Transformation arbeitet, entsteht hier eine Sequenz von Graphen. Die Graphen dieser Sequenz sind nicht im vorhinein bekannt, da die Reihenfolge der Regelanwendungen und die Auswahl von passenden Teilgraphen beliebig ist, also nichtdeterministisch erfolgt. Um die nichtdeterministiche Auswahl der Regeln zu umgehen können die Regeln in Ebenen (Layer) eingeteilt werden. Wenn in einem Layer mehrere Regeln enthalten sind, erfolgt auch hier die Auswahl nichtdeterministisch. Dies geschieht solange, bis keine Regel des Layers mehr anwendbar ist. Dann werden die Regeln des nächsten Layers abgearbeitet. So lässt sich die Reihenfolge der Regelanwendungen beliebig genau festlegen.

Aus der nichtdeterministischen Arbeitsweise der Graphtransformationen entsteht eine Sequenz von Graphen, deren Entwicklung nicht genau vorhersehbar ist. Dies stellt eine Anforderung an das Layout von Graphtransformationen dar, die über das Offline Graph Drawing Problem hinaus geht. Diese Anforderung entspricht einer der Erweiterungen des Offline Graph Drawing Problems, die in Abschnitt 2.2 (Problemdefinition) angeführt wurden. Da mittels Graphtransformation beliebige Graphsequenzen mit be-

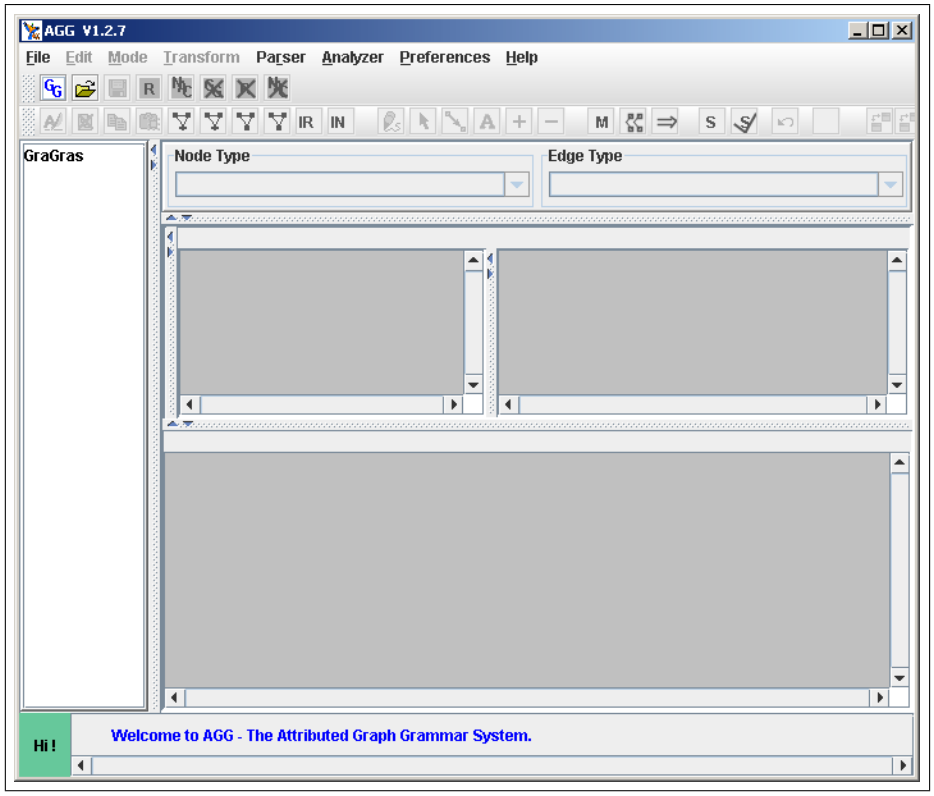

Abbildung 3.3: Benutzerschnittstelle des AGG

stimmter Semantik der enthaltenen Graphen erstellt werden können, kommt auch die Anforderung nach einem bestimmten Aussehen von bestimmten Graphen zum Tragen. Diese Anforderung ist eine weitere Erweiterung des Offline Graph Drawing Problems, die in der Problemdefinition angeführt ist und in dieser Arbeit betrachtet werden soll.

Weitergehende Informationen zum Thema Graphtransformation können in [EEPT06] und [EHK+97] nachgelesen werden.

3.3.1 Das Graphtransformationstool AGG

Mit dem Tool AGG (Attributed Graph Grammar System) können Graphgrammatiken erstellt und Transformationen durchgeführt werden. Über die folgende Beschreibung hinausgehende Informationen zu AGG finden sich in [ERT99] und [Tea06]. Dort kann das Tool auch heruntergeladen werden.

Das AGG hat eine graphische Oberfläche, die das Erstellen des Ausgangsgraphen und der Regeln ermöglicht. Die Oberfläche (siehe Abb. 3.3) ist in vier Teile geteilt. Im linken Teil befindet sich eine baumartige Übersicht über die geöffneten Graphgrammatiken inklusive der enthaltenen Regeln (Rule) und Bedingungen (Conditions, Constraints). Dort können unter anderem Regeln erstellt und ausgewählt werden. Im rechten unteren Teil

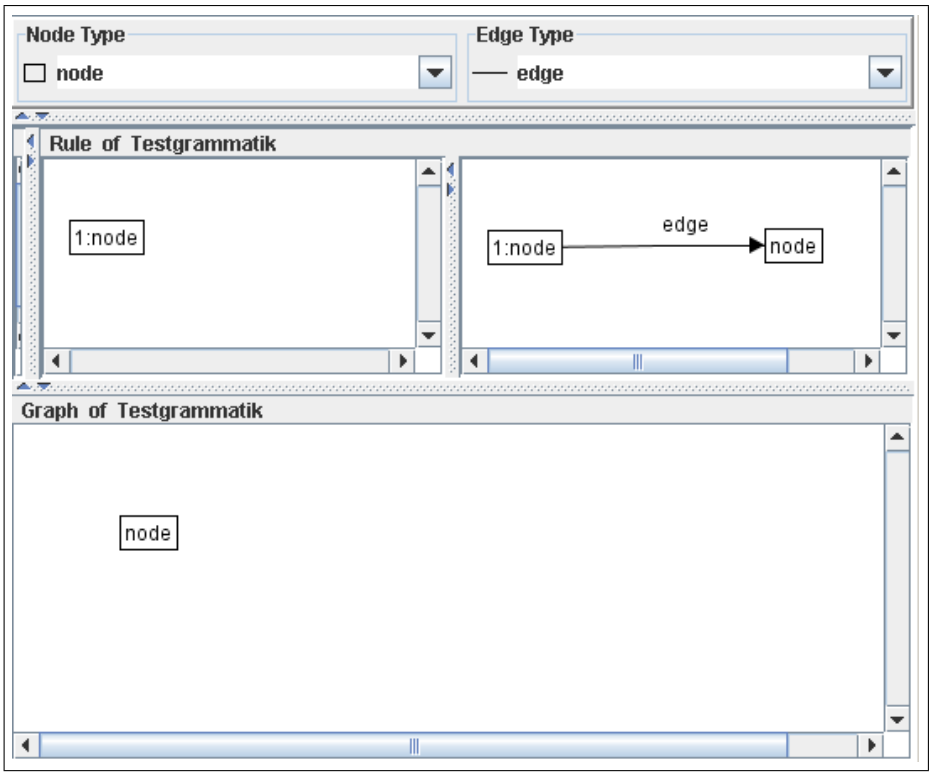

Abbildung 3.4: Ein einfacher Ausgangsgraph und eine Regel in AGG

befindet sich eine Zeichenfläche, in der der Ausgangsgraph erstellt und bearbeitet wird. Um hier Knoten und Kanten zu erzeugen müssen im rechten oberen Teil der Ansicht zunächst Knoten– und Kantentypen angelegt und ausgewählt werden. Der rechte mittlere Bereich der Ansicht enthält im Normalfall zwei Zeichenflächen, in denen die linke und rechte Seite der ausgewählten Regel bearbeitet werden können. Falls die Regel mindestens eine Nichtanwendungsbedingung (NAC) enthält, erscheint hier eine zusätzliche Zeichenfläche, in der diese Nichtanwendungsbedingungen bearbeitet werden können. In Abbildung 3.4 ist ein Ausgangsgraph, der nur aus einem Knoten vom Typ NODE besteht, sowie eine einfache Regel in AGG dargestellt. Die Regel bewirkt das Hinzufügen eines neuen Knotens vom Typ NODE und einer Kante vom Typ EDGE zwischen dem alten und dem neuen Knoten. Das 1: stellt das Matching zwischen den Knoten auf der linken und rechten Seite der Regel dar. Dadurch bleibt der vorhandene Knoten im Ergebnisgraphen erhalten. Abbildung 3.5 zeigt den Ergebnisgraphen einer Transformation, bei der diese Regel einmal auf den Graphen angewendet wurde.

Weiterhin gibt es eine Werkzeugleiste, die Buttons zum Erstellen von verschiedenen Teilen der Graphgrammatik enthält. Hier können auch die verschiedenen Editorfunktionen (unter anderem: Zeichenmodus, Bewegungsmodus) für die Zeichenflächen ausgewählt werden. Außerdem befinden sich hier die Buttons zum Ausführen von Transformationen.

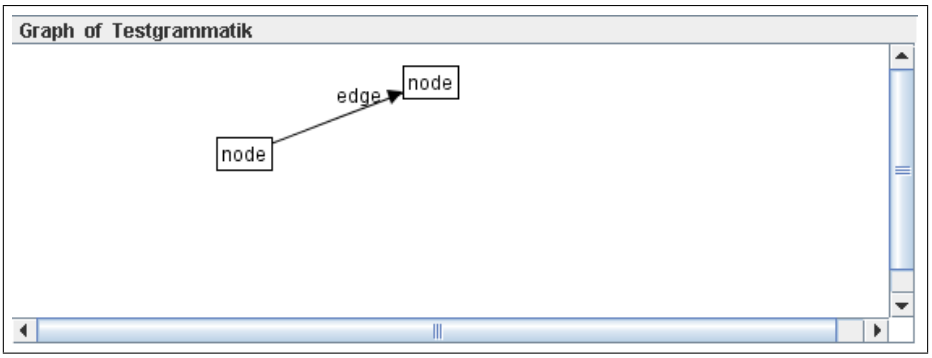

Abbildung 3.5: Graph nach der einmaligen Anwendung in Abb. 3.4 dargestellten Regel

Nachdem eine Graphgrammatik mit dem Ausgangsgraphen und mindestens einer Regel erstellt wurde, kann AGG Transformationen durchführen. Es gibt die Möglichkeit, die Transformation automatisch ablaufen zu lassen. Dabei wählt AGG nichtdeterministisch eine anwendbare Regel aus (sofern eine solche Regel existiert) und wendet diese Regel auf einen passenden Teil des aktuellen Graphen an, der ebenfalls nichtdeterministisch gewählt wird. Im Anschluss daran wird der Ergebnisgraph in der Graphzeichenfläche angezeigt und der nächste Transformationsschritt angestoßen. Dieser Ablauf wiederholt sich so lange, bis keine anwendbare Regel mehr gefunden werden kann oder der Benutzer die Transformation stoppt. Es ist aber auch möglich, die Transformation manuell einzeln anzustoßen und zu beeinflussen. Dazu muss der Benutzer eine Regel auswählen. Dann kann geprüft werden, ob diese Regel anwendbar ist. Ist das der Fall, kann der Benutzer auch ein Matching zwischen Knoten in der linken Seite der Regel und Knoten im aktuellen Graphen angeben und so den Teil des Graphen auswählen, der geändert werden soll. Dann führt AGG nur noch die Änderungen aus. Die ausgewählte Regel kann allerdings auch automatisch ausgeführt werden. Dabei erfolgt die Auswahl des zu ändernden Graphteils wie bei der vollautomatischen Transformation. Nach der Regelausführung wird der Ergebnisgraph angezeigt.

3.4 State of the Art

Es existieren verschiedene Tools die Graphen darstellen können. Dazu gehören auch verschiedene CASE–Tools, mit denen Klassendiagramme bearbeitet werden können. Ein CASE–Tool, das einen Layouter für Klassendiagramme beinhaltet, ist das Omondo Eclipse UML Studio [omo06]. Dieses Tool ist als Eclipse–Plugin [GB03] realisiert. Es kann Layouts für Klassendiagramme erstellen, die objektiven Qualitätsmaßstäben genügen und sich auch an den Erwartungen eines Entwicklers bezogen auf das Aussehen eines Klassendiagramms orientieren. Es kann trotzdem dazu kommen, dass die Diagramme schwer nutzbar sind. Bei Klassendiagrammen, die viele Klassen (mehr als 10) und/oder Klassen mit vielen Methoden(mehr als 10) enthalten, werden die Layouts so groß, dass sie auf einer Bildschirmseite nicht mehr lesbar angezeigt werden können. Weiterhin

erfolgt die Layoutberechnung jeweils nur für ein einzelnes Klassendiagramm. Für eine neue Version eines Klassendiagramms wird ein komplett neues Layout berechnet.

Das Graphtransformationstool AGG (siehe 3.3.1) kann Sequenzen von Graphen durch Tansformationen erstellen. Die durch die Transformationsschritte neu entstanden Graphen werden jeweils dargestellt. Dabei erfolgt die Layoutberechnung auf recht einfache Art und Weise. Die Knoten werden zeilenweise hintereinander in der Reihenfolge ihrer Erzeugung angeordnet. Die so enstehenden Layouts erfüllen die objektiven Qualitätsmaßstäbe oft nicht. Es kommt häufig zu Kantenkreuzungen und Überschneidungen von Knoten und Kanten sowie zu Überlappungen von neu hinzugekommenen Knoten mit älteren Knoten, die vom Benutzer positioniert wurden.

Ein weiteres Tool, das einen Layoutalgorithmus für unvollständige Sequenzen von Graphen beinhaltet, konnte während der Recherche zu dieser Arbeit nicht gefunden werden.

4 Eigener Lösungsansatz

Der im Folgenden vorgestellte Ansatz soll das in Kapitel 2 definierte erweiterte Offline Graph Drawing Problem lösen. Dabei sollen Sequenzen von Klassendiagrammen, die nicht vollständig bekannt sind, möglichst einheitlich gelayoutet werden. Die einzelnen Klassendiagramme einer Sequenz sind aufeinanderfolgende Versionen des Klassenmodells, die im Verlauf eines IT–Projekts entwickelt wurden. Sie haben also einen inhaltlichen Zusammenhang. In Abschnitt 4.1 wird erläutert, wie dieser inhaltliche Zusammenhang in einer Graphgrammatik dargestellt werden kann. Aus dieser Graphgrammatik kann dann mittels Graphtransformation die Sequenz von Klassendiagrammen rekonstruiert werden. Dadurch wird das Problem des Layouts einer Sequenz von Klassendiagrammen in das Layout einer durch Graphtransformation erstellten Sequenz von Graphen umgewandelt. In Abschnitt 4.3 wird ein Lösungsansatz für dieses Problem erläutert. Dabei wird zuerst ein Layoutalgorithmus für Einzelgraphen vorgestellt. Danach werden die für das Layout von Graphsequenzen notwendigen Erweiterungen dieses Algorithmus erklärt. Besonderer Wert wird dabei auf eine möglichst geringe Differenz zwischen den Layouts zweier aufeinanderfolgender Graphen gelegt. Weiterhin wird darauf eingegangen, wie Erwartungen an das Layout von Klassendiagrammen erfüllt werden können. Spezielle Layoutanforderungen anderer Graphen (z.B. Petri–Netze) können dann analog dazu realisiert werden. Abschließend erfolgt in Abschnitt 4.4 eine Komplexitätseinschätzung der in diesem Kapitel entwickelten Algorithmen.

Nähere Erläuterungen zur Implementation der in diesem Kapitel vorgestellten Algorithmen finden sich in Kapitel 5.

4.1 Datenaufbereitung

Der hier vorgestellte Lösungsansatz zum Layout von unvollständigen Sequenzen von Klassendiagrammen basiert darauf, diese Sequenzen mittels schrittweiser Graphtransformation aus einer Graphgrammatik (siehe 3.3) zu erzeugen. Die Umwandlung einer Sequenz von Klassendiagrammen in eine Graphgrammatik läuft wie folgt ab: Das erste Klassendiagramm wird zum Ausgangsgraphen der Graphgrammatik. Die Klassen werden durch Knoten des Graphen repräsentiert. Die Verbindungen (Assoziationen, Vererbungen etc.) zwischen den Klassen werden im Graphen durch Kanten zwischen den entsprechenden Knoten repräsentiert. Dabei werden alle Verbindungen zwischen zwei Klassen zu einer Kante zusammengefasst. Diese Kante wird typisiert, so dass man auch im Graphen erkennen kann, welche Art(en) von Beziehung(en) die Kante darstellt. Im Anschluss an die Generierung des Ausgangsgraphen werden alle Unterschiede zwischen zwei aufeinander folgenden Klassendiagrammen K_i und K_{i+1} der Sequenz ermittelt. Die grundlegenden Änderungen sind dabei das Hinzufügen oder Löschen von Klassen und

Beziehungen zwischen Klassen. Alle komplexeren Änderungen, wie das Aufspalten einer Klasse in mehrere, beziehungsweise das Zusammenlegen von mehreren Klassen in eine Klasse, lassen sich auf diese grundlegenden Änderungen zurückführen. Die ermittelten Änderungen werden so in eine Transformationsregel umgewandelt, dass die Transformation eines das Klassendiagramm K_i repräsentierenden Graphen nach dieser Regel einen Graphen erzeugt, der das Klassendiagramm K_{i+1} repräsentiert. Dieser Vorgang wird für alle Paare von aufeinander folgenden Klassendiagrammen der Sequenz durchgeführt. Konkret heißt das, dass die erste Regel das erste Klassendiagramm in das zweite Klassendiagramm überführt, die zweite Regel das Zweite in das Dritte, etc. Dabei muss sichergestellt werden, dass die Regeln in der richtigen Reihenfolge abgearbeitet werden. Dazu muss die nichtdeterministische Auswahl der Regeln während der Transformation durch sogenannte Layer umgangen werden. Jeder Regel wird also ein eigener Layer in der gewünschten Reihenfolge zugeordnet. Weiterhin darf jede Regel nur einmal ausgeführt werden. Dies wird erreicht, indem der Regel eine geeignete Nichtanwendungsbedingung hinzugefügt wird.

Mit der so erzeugten Graphgrammatik kann durch schrittweise Transformation jeder Graph der Sequenz erzeugt werden. Wird nun eine neue Version des betrachteten Klassendiagramms entwickelt und der Sequenz hinzugefügt, muss der Grammatik nur eine neue Regel hinzugefügt werden. Dieser Vorgang kann von Hand erfolgen, lässt sich aber auch automatisieren. Zur automatischen Umwandlung einer Sequenz von Klassendiagrammen in eine Graphgrammatik müssen die einzelnen Klassendiagramme in maschinenlesbarer Form vorliegen. Verschiedene CASE-Tools bieten einen Export von Klassendiagrammen in XML–Dateien an. Eine solche Darstellung lässt sich im Vergleich zu einer reinen Bilddarstellung (z.B. JPEG– oder Bitmap–Datei) einfach parsen. Da allerdings jedes Tool ein eigenes Dokumentformat nutzt, sind diese Dateien nicht untereinander kompatibel. In dieser Arbeit wurde das Omondo Eclipse UML Studio genutzt um Klassendiagramme zu erzeugen.

4.2 Content Pattern – Layout Pattern

Wie in den Abschnitten zur Problemdefinitionen (siehe 2.2) und zur Szenariobeschreibung (siehe 2.3.1) beispielhaft ausgeführt, haben Entwickler bestimmte Erwartungen an das Aussehen eines Klassendiagramms. Diese Erwartungen leiten sich aus objektorientierten Strukturen ab. Einfache Strukturen sind Vererbungen, komplexere Strukturen können durch sogenannte Design Pattern gegeben sein. Um diese Strukturen (kurz Content Pattern) im Graphen einfach wiederzuerkennen benötigt der Betrachter eine Hilfestellung im Layout des Graphen (kurz Layout Pattern). In diesem Abschnitt sollen sowohl einfache Layout Pattern (siehe 4.2.1), als auch spezifische Content Pattern (siehe 4.2.2) vorgestellt werden. Anschließend wird gezeigt, wie ein passendes Layout Pattern für ein spezifisches Content Pattern ausgewählt werden kann und inwieweit die Anforderungen des Betrachters durch diese Wahl erfüllt werden können.

4.2.1 Layout Pattern

Layout Pattern sind Regeln, die bestimmen, wie das Layout von Teilgraphen aussehen soll. Diese Teile können einzelne Knoten, durch bestimmte Kanten verbundene Knotenpaare oder bestimmte Teilgraphen sein. Im Folgenden werden vier Arten von Layout Pattern vorgestellt: Einzelpattern, Kantenpattern, Knotengruppenpattern und Gruppen–Gruppen–Pattern.

Die einfachste Art Pattern bestimmt die Positionierung einzelner Knoten. Diese Einzelpattern können den gewünschten Abstand eines Knotens zum Rand oder den Ecken des Darstellungsbereichs sowie zu bestimmten anderen Knoten(gruppen) angeben.

Die zweite Art Layout Pattern (Kantenpattern) bezieht sich auf Knotenpaare, die durch eine Kante verbunden sind. Es gibt zwei Arten von Kantenpattern. Die erste Art Kantenpattern regelt die Lage der Endknoten der betrachteten Kante zueinander. Sie geben für jede Koordinateachse der Darstellung an, ob der Zielknoten (Target) vor, hinter oder gleichauf mit dem Quellknoten (Source) positioniert werden soll. Im Fall einer zweidimensionalen Projektionsfläche ergeben sich also folgende Kantenpattern:

- x–Achse:
  - Target links der Source
  - Target rechts der Source
  - Target auf gleicher Höhe mit der Source
- y–Achse:
  - Target oberhalb der Source
  - Target unterhalb der Source
  - Target auf gleicher Höhe mit der Source

Jeder Kante darf dabei nur ein Pattern pro Koordinatenachse zugeordnet werden, da sich die Pattern gegenseitig ausschließen (ein Knoten kann nicht gleichzeitig links und rechts von einem anderen sein). Es sind aber Kombinationen von Pattern verschiedener Achsen denkbar (Targetknoten links und unterhalb des Sourceknotens). Dabei sollte jedoch die Kombination der Patterns „gleiche Höhe“ vermieden werden, da diese im Layout zwangsläufig zu einer Überlappung von Source- und Targetknoten führt. Die zweite Art Kantenpattern sind die Kantenlängenpattern. Sie legen fest, welche Länge eine Kante haben sollte. In dieser Arbeit werden Kantenpattern für alle Kanten eines Typs einheitlich definiert. Es ist aber auch denkbar, jeder Kante eigene Pattern zuzuordnen.

Die Knotengruppenpattern stellen die dritte Art von Layout Pattern dar. Sie legen fest, wie Gruppen von Knoten, die in einem semantischen Zusammenhang stehen, dargestellt werden sollen. Die Knotengruppenpattern können auf zwei verschiedene Arten definiert werden. Die einfache Art bestimmt analog zu den Einzelpattern die gewünschte Lage einer Knotengruppe im Layout. Die komplexere Art bestimmt das Layout der Knoten innerhalb der betrachteten Knotengruppe. Damit kann innerhalb der Knotengruppe ein anderer Layoutansatz bestimmt werden als im restlichen Graphen. So können

beispielsweise einzelne Knotengruppen in einem orthogonalen Layout radial dargestellt werden.

Die vierte Art Layoutpattern, die Gruppen–Gruppen–Pattern, regelt die Lage von zwei Knotengruppen (Teilgraphen) zueinander. Diese Art Pattern ist den Kantenpattern sehr ähnlich. Allerdings ist es hierbei nicht erforderlich, dass die Teilgraphen miteinander verbunden sind. Für zwei Teilgraphen T_1 und T_2 ergeben sich hier die einzelnen Pattern analog zu den Kantenpattern. Auch hier sind Kombinationen der Pattern mit den gleichen Einschränkungen möglich. Abbildung 4.1 zeigt das Layout eines Graphen, bei dem für die Teilgraphen T_1 und T_2 das Pattern „T_2 rechts unterhalb von T_1“ eingehalten wird.

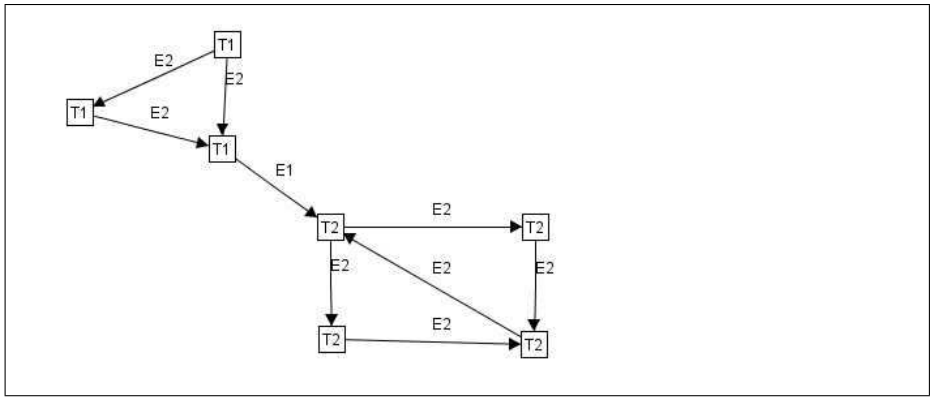

Abbildung 4.1: Graphlayout mit einem Gruppen–Gruppen–Pattern, das vorschreibt, dass der Teilgraph T_2 (Knoten $T2$) rechts unterhalb des Teilgraphs T_1 (Knoten $T1$) positioniert werden soll

Bei der Anwendung von mehreren Pattern im Layout eines Graphen kann es dazu kommen, dass sich einzelne Pattern widersprechen. Ein Beispiel für einen Widerspruch zwischen einem Einzelpattern und einem Kantenpattern kann folgendermaßen aussehen: Für einen Knoten K_1 existiert ein Einzelpattern, nach dem der Knoten direkt am linken Rand der Darstellungsfläche positioniert werden soll. K_1 ist aber auch der Sourceknoten einer Kante, für die ein Kantenpattern vorschreibt, dass der Targetknoten links vom Sourceknoten dargestellt werden soll. Beide Bedingungen können nicht gleichzeitig erüllt werden. Ein weiteres Beispiel kann an Abbildung 4.1 deutlich gemacht werden. In dem dargestellten Graphen wird zusätzlich zu dem Gruppen–Gruppen–Pattern „T_1 links oberhalb von T_2“ ein Kantenpattern für die Kante $E1$ so definiert wird, dass der Endknoten in T_1 rechts des Endknotens in T_2 dargestellt werden soll. Diese Pattern können nicht gleichzeitig erfüllt werden. Solche Widersprüche müssen bei der Definition von mehreren Layout Pattern für einen Graphen vermieden werden, da es sonst zu starken Qualititätseinbußen im Layout kommen kann.

4.2.2 Content Pattern

Content Pattern sind Muster, die in Klassendiagrammen während des Softwareentwicklungsprozesses auftreten. Es gibt einfache Content Pattern, die sich direkt aus dem objektorientierten Enwicklungsparadigma ergeben. Solche einfachen Content Pattern sind beispielsweise Vererbungsstrukturen sowie Assoziationen und Dependencies zwischen Klassen.

Komplexere Content Pattern können durch sogenannte Design Pattern gegeben sein. Design Pattern dienen zur Lösung von Problemen, die immer wieder in leicht abgewandelter Form auftreten. Sie beschreiben abstrakte Problemstellungen und dazu passende Lösungen, die als Leitpfad zur Lösung eines konkreten Problems dienen können. Das Prinzip der Design Pattern ist in vielen Ingenieurswissenschaften bekannt. Auch im Softwareentwicklungsprozess treten einige Problemstellungen immer wieder in leicht abgewandelter Form auf. Daher kann das Prinzip der Design Pattern auch im Software Engineering angewendet werden. Erich Gamma et al. stellt in „Design Patterns: Elements of Reusable Object–Oriented Software“ [GHJV94] Design Pattern für die objektorientierte Softwareentwicklung und deren Anwendung vor. Beispiele für Design Pattern nach Gamma sind das Facade Pattern, das Proxy Pattern und das Observer Pattern.

Das Facade Pattern behandelt das Problem des Zugriffs auf Funktionen eines Subsystems. Es bietet eine einheitliche Schnittstelle durch die auf die einzelnen Funktionen eines komplexen Subsystems zugegriffen werden kann. Klienten, die mehrere Funktionen verschiedener Klassen des Subsystems nutzen, müssen nur noch das Facade–Objekt kennen und nicht mehr Informationen zu allen genutzen Klassen vorhalten. Das vereinfacht die Arbeit mit Subsystemen. Weiterhin wird das Klassendiagramm zu dem Gesamtsystem übersichtlicher, da Subsysteme gekapselt dargestellt werden und weniger Verbindungen zwischen einzelnen Klassen vorhanden sind.

Das Proxy Pattern kann genutzt werden, wenn der Zugriff auf ein Objekt kontrolliert werden soll. Dazu wird ein Platzhalterobjekt (Proxy) verwendet, das die gleiche Schnittstelle wie das eigentliche Objekt bietet. Beim Zugriff auf ein Objekt über ein Proxyobjekt kann dann die Berechtigung des Zugreifenden geprüft werden, bevor der Zugriff zum eigentlichen Objekt weitergeleitet wird. Das Proxy Pattern kann auch verwendet werden um Daten, die auf entfernten Ressourcen liegen, transparent in das System einzubinden. Aufrufe können dann unabhänig davon, ob das Zielobjekt lokal oder entfernt vorgehalten wird, immer gleich erfolgen. Damit wird die eventuell erforderliche Kodierung (nach einem Kommunikationsprotokoll) eines Aufrufs (inklusive ihrer Parameter) und das Senden des kodierten Aufrufs an entfernte Ressourcen sowie das Empfangen und Dekodieren der Ergebnisse des Aufrufs in das Proxyobjekt verlagert. Das ist vor allem dann ein Vorteil, wenn Objekte verschiedener Klassen auf entfernt liegende Objekte einer Klasse zugreifen, da die Implementation der Kommunikationsvorgänge nur einmal erfolgen muss. Änderungen am Kommunikationsprotokoll können so leichter umgesetzt werden.

Das Observer Pattern kann genutzt werden, wenn viele Objekte von den Daten (dem Zustand) eines Objekts abhängen. Dazu wird eine $1 : n$ Abhängingkeit zwischen dem Datenobjekt und den abhängigen Observerobjekten definiert, durch die alle Observer

bei einer Zustandsänderung des Datenobjekts automatisch benachrichtigt werden. Ein Anwendungsbeispiel für das Observer Pattern ist die konsistente Darstellung von Daten eines Objekts in verschiedenen GUI–Elementen. Bei einer Änderung der Daten werden alle GUI–Elemente benachrichtigt und können ihre Ansicht aktualisieren.

4.2.3 Mapping: Layout Pattern – Content Pattern

An Strukturen, die durch solche Content Pattern in Klassendiagrammen entstehen, können bestimmte Erwartungen bezüglich des Layouts geknüpft sein. Das kann bei den einfachen Content Pattern genau so wie bei den komplexen Content Pattern der Fall sein. Das Ziel dieser Erwartungen ist neben einer übersichtlichen Darstellung ein leichtes Wiedererkennen der jeweiligen Struktur im Layout.

Das Paradebeispiel eines einfachen Content Pattern sind Vererbungsstrukturen. Die meisten Entwickler haben die Erwartung, dass Vererbungsstrukturen baumartig hierarchisch dargestellt werden. Dabei werden die Subklassen unterhalb ihrer jeweiligen Superklasse angeordnet. Das lässt sich mit einem Kantenpattern realisieren. Dazu wird ein Kantenpattern für Vererbungskanten erstellt, welches vorschreibt, dass die Subklassen (Sourceknoten) unterhalb der jeweiligen Superklasse (Targetknoten) positioniert werden sollen. Dies kann mit einem weiteren Kantenpattern kombiniert werden, welches den Vererbungskanten eine geringere Länge zuweist als den anderen Kanten. Damit kann der enge Zusammenhang von Klassen, die in einer Vererbungsbeziehung zueinander stehen, hervorgehoben werden. Analog dazu lassen sich auch andere Baumstrukturen in allgemeinen Graphen hierarchisch darstellen.

Während die Erwartungen bezüglich der Darstellung von einfachen Content Pattern wie Vererbungen bei den meisten Entwicklern sehr einheitlich sind, können sie bei komlexeren Strukturen durchaus variieren. Im Folgenden werden Darstellungmöglichkeiten für die oben erläuterten Observer–, Facade– und Proxy–Pattern sowie die Realisierung dieser Darstellungsmöglichkeiten durch Layoutpattern vorgestellt.

Das Observer–Pattern ist eine relativ einfache Struktur, in der eine Datenklasse mit mehreren Observerklassen verbunden ist. Um zu veranschaulichen, dass die Observer die Datenklasse überwachen, also auf sie herabsehen, kann die Darstellung wie in Abbildung 4.2 aussehen. Die Positionierung der Observerklassen oberhalb der Datenklasse kann dabei analog zu den Vererbungen mit einem Kantenpattern realisiert werden.

Die Struktur des Facade–Pattern kann in drei Gruppen von Klassen eingeteilt werden: Klientengruppe, Klassen des benutzten Subsystems und die Facade–Klasse. Um eine übersichtliche Darstellung zu erhalten sollten die Gruppe der Klientenklassen getrennt von den Klassen des Subsystems angeordnet werden. Die Facade–Klasse ist das Bindeglied zwischen diesen beiden Gruppen und sollte im Layout auch als solches zu erkennen sein. Daraus ergibt sich eine Darstellung wie in Abbildung 4.3. Die Klientenklassen sind links, die Klassen des Subsystems rechts angeordnet. Die Facadeklasse ist dabei zentral zwischen diesen Gruppen positioniert. Eine solche Darstellung lässt sich durch eine Kombination von zwei Gruppen–Gruppen–Pattern realisieren. Dabei wird die Facade–Klasse als Knotengruppe angesehen. Ein Gruppen–Gruppen–Pattern wird so definiert, dass die Gruppe der Klientenklassen links von der Facade–Klasse positioniert wird. Das zweite

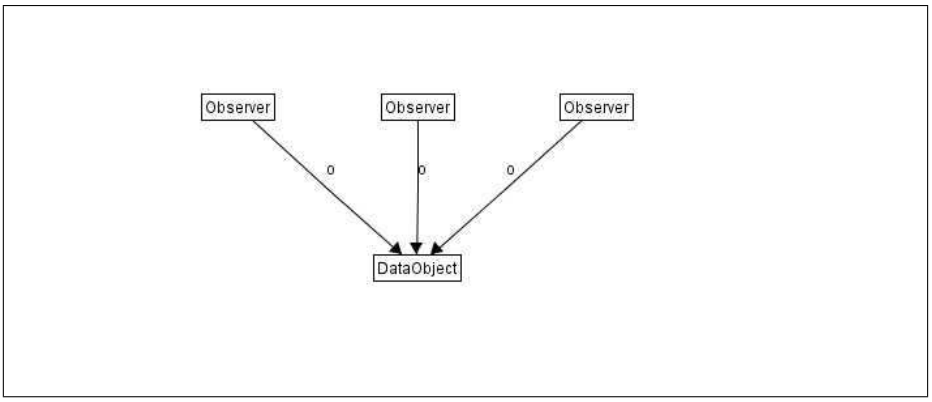

Abbildung 4.2: Mögliche Darstellung des Observer–Patterns in einem Klassendiagramm.

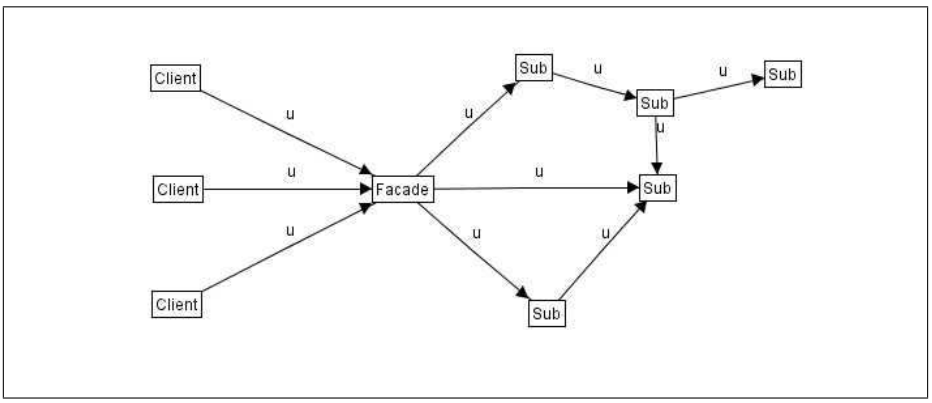

Abbildung 4.3: Darstellungmöglichkeit für das Facade–Pattern.

Gruppen–Gruppen–Pattern stellt dann sicher, dass die Klassen des Subsystems rechts von der Facade–Klasse dargestellt wird.

Das Proxy–Pattern bildet eine Struktur aus einer Gruppe von Klientenklassen und einem Paar einzelner Klassen (Proxy–Klasse und reale Klasse), die durch eine Kante verbunden sind. Die Proxy–Klasse steht hierbei vermittelnd zwischen Klienten und realer Klasse. Diese vermittelnde Funktion sollte analog zum Facade–Pattern auch optisch erkennbar sein. Die Proxy–Struktur kann auf verschiedene Arten dargestellt werden, von denen drei in Abbildung 4.4 zu sehen sind. Die Darstellung links im Bild ist eine hierarchische Anordnung, bei der die reale Klasse oben, die Klientenklassen unten und die Proxy–Klasse in der Mitte positioniert sind. Diese Anordnung ist übersichtlich und stellt die vermittelnde Funktion der Proxy–Klasse heraus, kann aber leicht mit der Darstellung von Vererbungsstrukturen verwechselt werden. Die Darstellung in der Bildmitte ordnet die Klientenklassen radial um die Proxy–Klasse herum an. Dabei wird die reale Klasse in das radiale Layout eingefasst. Dadurch wirkt die Darstellung etwas unübersichtlich. Das

Layout rechts im Bild stellt eine Abwandlung des Layouts für das Facade–Pattern dar. Die Klientenklassen werden hier radial links von der Proxy–Klasse angeordnet, während die reale Klasse rechts der Proxy–Klasse positioniert wird. Diese Darstellung ist ähnlich übersichtlich wie die hierarchische links im Bild. Es besteht zwar die Möglichkeit der Verwechslung mit dem Layout des Facade–Pattern, allerdings nur dann, wenn das Subsystem dabei nur aus einer Klasse besteht. In einem solchen Fall ist die Anwendung des Facade–Pattern aber nicht sinnvoll. Daher fiel die Wahl auf die halbradiale Darstellung rechts im Bild. Die Realisierung einer solchen Anordnung kann durch eine Kombination eines Gruppen–Gruppen–Pattern mit einem Kantenpattern erfolgen. Dabei stellt das Gruppen–Gruppen–Pattern sicher, dass die Gruppe der Klientenklassen links von dem Knotenpaar (Gruppe) der Proxy– und der realen Klasse positioniert wird. Das Kantenpattern wird dann für die Kante zwischen der Proxy–Klasse und der realen Klasse definiert, sodass die reale Klasse rechts von der Proxy–Klasse angeordnet wird. Es kann ein zusätzliches Kantenpattern erstellt werden, das bestimmt, dass die reale Klasse auf gleicher Höhe mit der Proxy–Klasse stehen soll. Wenn das Gesamtlayout orthogonal aufgebaut ist, muss zusätzlich ein Knotengruppenpattern erstellt werden, sodass in dem Teilgraphen, der durch alle Klassen des Proxy–Pattern gebildet wird, ein radiales Layout verwendet wird.

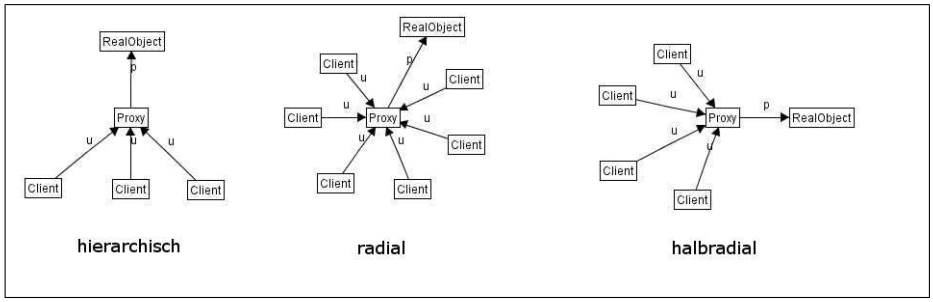

Abbildung 4.4: Darstellungsmöglichkeiten für das Proxy–Pattern.

4.3 Layoutalgorithmus

Im vorhergehenden Abschnitt wurde der Begriff Pattern sowohl für Layout Pattern als auch für Content Pattern eingeführt. Abschließend wurden Möglichkeiten vorgestellt Content Pattern mit Layout Pattern zu verknüpfen. Dieser Abschnitt beschäftigt sich mit dem eigentlichen Layoutalgorithmus, der sowohl die Vorgabe von Layout Pattern verarbeiten kann, aber auch ohne diese den Qualitätsmaßstäben (siehe 3.2) entsprechende Ergebnisse liefert.

Der hier entwickelte Layoutalgorithmus setzt die in der Grundidee (siehe 4.1) vorgestellte Umwandlung einer Sequenz von Klassendiagrammen in eine Graphgrammatik voraus. Dabei wird das initiale Klassenmodell einer Sequenz in einen Graphen umgewandelt. Aus den darauf aufbauenden Klassendiagrammen wird ein Regelsatz zur Graph-

transformation erstellt, der alle Änderungen des Klassenmodells im Laufe der Sequenz beinhaltet.

Der Layoutalgorithmus basiert auf einem Spring Embedder Modell (siehe Abschnitt 3.1.2), das auf dem von Fruchterman und Reingold [FR91] aufbaut. Allerdings wird hier mehr Wert auf einheitliche Kantenlängen gelegt. Ein Zusatz berücksichtigt, dass hier nicht nur ein Einzellayout sondern ein Layout für eine Sequenz von Graphen entwickelt werden soll. Jedem Knoten wird ein Alter zugewiesen, das anzeigt, in wie vielen Graphen der Sequenz er bis zu dem aktuell betrachteten schon vorhanden war. Je älter ein Knoten ist, desto größer wird sein Beharrungsvermögen. Das heißt, ein alter Knoten bewegt sich im Layout weniger als ein neu hinzugekommener Knoten. Dies ist ein Zugeständnis an die Mental Map des Betrachters und wird in der Diskussion (siehe Kapitel 7) noch weiter erläutert.

Ein weiterer Zusatz beinhaltet die Nutzung von Qualitätsmetriken (siehe Abschnitt 3.2) zur Berechnung der Qualität des erzeugten Layouts. Einige dieser Metriken beeinflussen den Ablauf des Layoutalgorithmus direkt, andere dienen nur zur Beurteilung der Ergebnisse der Layoutberechnung.

Im Folgenden werden der Spring Embedder Layoutalgorithmus und die ersten beiden Zusätze im einzelnen erklärt. Dies bezieht sich auf ein Layout ohne vorgegebene Pattern. Anschließend wird erläutert, wie die im vorangegangenen Abschnitt vorgestellten Layout Pattern in diesen Algorithmus integriert werden können. Weiterhin wird eine selbst entwickelte Metrik zur Einhaltung der Vorgaben von Layout Pattern vorgestellt. Mit dieser Metrik kann die Qualität des Layouts bezüglich der Anforderungen an das Layout einer Sequenz von Klassendiagrammen gemessen werden, wenn die entsprechenden Layoutpattern für objektorientierte Strukturen verwendet werden.

4.3.1 Spring Embedder Layout für Einzelgraphen

Der hier entwickelte Layoutalgorithmus basiert auf einem Spring Embedder Energiemodell ähnlich dem von Fruchterman und Reingold mit abstoßenden Kräften zwischen allen Knoten und anziehenden Kräften nur zwischen durch eine Kante verbundenen Knoten. Abstoßende Kräfte werden durch die magnetische Abstoßung gleicher Pole modelliert. Anziehende Kräfte werden als Federn zwischen den Knoten dargestellt. Zur Berechnung der abstoßenden Kräfte f_r zwischen zwei Knoten wird die folgende Funktion (von Fruchterman und Reingold addaptiert) benutzt:

$$f_r(d) = \frac{z^2}{d} \tag{4.1}$$

Das d steht dabei für die Entfernung der Knoten voneinander, während z den minimalen Abstand angibt, den ein Knoten von allen anderen haben sollte. Dieser minimale Abstand kann fest vorgegeben sein oder er lässt sich anhand der zur Verfügung stehenden Anzeigefläche und der Knotenanzahl des darzustellenden Graphen berechnen. Die anziehenden Kräfte f_a für die Kanten ergeben sich aus der aktuellen Kantenlänge und der bevorzugten Kantenlänge. Die bevorzugte Kantenlänge ist mit der Ausgangslänge

der Feder gleichzusetzen, die die Kante modelliert. Sie muss vorgegeben werden. Diese Kräfte bestimmen die Neupositionierung der Knoten. Bei der Berechnung der neuen Positionen der Knoten kommt eine Beschränkung der Verschiebung der Knoten durch die sogenannte Temperatur zum Tragen. Die initale Temperatur muss vorgegeben sein. Die Temperatur wird im Laufe der Berechnungen (in jeder Iteration) durch eine sogenannte cooling–Funktion immer weiter reduziert. Die hier verwendete cooling–Funktion in Abhängigkeit von der aktuellen Iterationszahl i lautet:

$$t(i) = t(i-1) - \frac{t(i-1)}{i+2} \tag{4.2}$$

Die Addition von 2 im Nenner war notwendig um Divisonen durch null und eins zu vermeiden. Bei einer Division durch eins an dieser Stelle wäre das Ergebnis der Funtion $t(i) = 0$. Dieses Ergebnis ist nicht sinnvoll, weil hierdurch Positionsänderungen von Knoten unmöglich werden.

Als Ausgangpunkt für die Berechnung eines neuen Layouts wird eine initiale Positionieung aller Knoten benötigt. Diese initiale Positionierung kann ein vorhandenes Layout sein, sie kann aber auch aus Zufallswerten erstellt werden. Die Möglichkeit, ein vorhandenes Layout als Ausgangspunkt nutzen zu können, ist später die Grundlage für das aufbauende Layout von Graphsequenzen.

Wenn die initiale Positionierung vorliegt, startet der Algorithmus mit der Hauptschleife. Dabei werden zuerst in n Iterationen die abstoßenden Kräfte zwischen allen Knoten berechnet. Diese werden anschließend durch die Temperatur beschränkt in Positionsänderungen umgerechnet. Diese Berechnungen erfolgen wie bei Fruchterman und Reingold in jeder der n Unteriterationen für alle Knoten. Im Anschluss daran werden die anziehenden Kräfte zwischen den durch eine Kante verbundenen Knoten bestimmt. Diese Berechnung erfolgt in j Iterationen. In jeder dieser j Unteriterationen wird analog zum Ansatz von Kamada (wo pro Iteration nur ein Knoten bewegt wird) nur eine Kante betrachtet. Dazu wird die Kante ermittelt, deren aktuelle Länge die größte Abweichung zur bevorzugten Kantenlänge aufweist. Diese Abweichung wird dann verringert, indem die Endknoten der Kante entsprechend verschoben werden. Das Verschieben der Knoten erfolgt in Richtung der Kante und wird ebenfalls von der Temperatur begrenzt. Am Ende jeder Hauptiteration wird die Temperatur entsprechend der cooling–Funktion (Gleichung 4.2) reduziert. Dadurch werden die Knoten in jeder Iteration etwas weniger verschoben. Dieser Algorithmus berechnet nur eine Positionierung für Knoten. Die Kanten werden dabei auf gerader Linie zwischen den jeweiligen Endknoten gezeichnet. Eine Kantenführung mit Knickpunkten ist nicht vorgesehen.

4.3.2 Aufeinander aufbauendes Layout von Graphsequenzen

Die hier folgende Erweiterung des oben vorgestellten Algorithmus ist ein Ansatz zur Lösung des Problems, eine nicht abgeschlossene Sequenz inhaltlich aufeinander aufbauender Graphen zu layouten. Ziel ist dabei, die Differenz zwischen zwei aufeinanderfolgenden Layouts möglichst gering zu halten. Durch eine geringe Differenz zwischen zwei

Layouts kann die Mental Map des Betrachters größtenteils erhalten bleiben. Die Differenz kann dabei mit Metriken zu mentaler Distanz von Layouts gemessen werden (siehe Abschnitt 3.2). Die Metrik zur Erscheinungsform von Kanten ist hier weniger geeignet, da die Kantenführung direkt ohne Knickpunkte erfolgt. Daher werden hier nur die Metriken zum Abstand und zur Clustererhaltung betrachtet.

Die Voraussetzung für gute Ergebnisse in diesen Metriken lässt sich folgendermaßen formulieren: Knoten, die in den Graphen G_i und G_{i+1} vorhanden sind, sollten in den dazugehörigen Layouts L_i und L_{i+1} möglichst an der gleichen Position dargestellt werden. Um die Mental Map des Betrachters über die gesamte Sequenz so gut wie möglich zu erhalten wird diese Forderung noch erweitert: Ein Knoten, der in allen Graphen G_i einer Teilsequenz enthalten ist, sollte in allen zugehöhrigen Layouts L_i möglichst an der gleichen Position dargestellt werden. Um das Beharren eines Knotens auf seiner Position zu erreichen wird jedem Knoten ein Alter zugeordnet. Die Knoten „altern“ innerhalb der Sequenz von einem Graphen zum nächsten. Das Alter der Knoten wirkt sich auf die Bewegungsfreiheit des Knotens bei der Layoutberechnung aus. Zur Berechnung der aufeinander folgenden Layouts wird der oben geschilderte Algorithmus um eine zusätzliche Beschränkung der Bewegungsfreiheit (Temperatur) der Knoten erweitert. Dabei wird die Bewegungsfreiheit mit zunehmendem Alter immer stärker verringert. Die Integration in den oben erläuterten Layoutalgorithmus erfolgt durch die Berechnung einer für jeden Knoten individuellen Temperatur. Dies erfolgt jeweils vor dem Verschieben eines Knotens in Folge der abstoßenden und anziehenden Kräfte. Zur Berechnung der individuellen Temperatur t_K wurde eine Funktion in Abhängigkeit vom Alter des Knotens a_K und dem Alter des Graphen a_G aus der in der jeweiligen Iteration allgemein geltenden Temperatur t_i entwickelt. Diese Funtion lautet:

$$t_K(i) = t(i) - \frac{t(i)}{a_G - a_K + 1} \tag{4.3}$$

Daraus ergibt sich, dass Knoten, die schon im initialen Graphen enthalten sind, keine Bewegungsfreiheit haben. Dadurch bleibt die Grundstruktur des Graphen optisch erhalten.

Allerdings kann diese Starre auch ein Problem erzeugen. Wenn eine Änderung des Graphen eine Kante zwischen zwei initialen Knoten hinzufügt, deren bevorzugte Länge von der Entfernung zwischen den Knoten abweicht, kann diese Abweichung nicht beseitigt werden. Hat diese Kante die größte Längenabweichung im aktuellen Graphen, so werden nach dem Algorithmus keine anderen Kanten betrachtet. Daher muss die Auswahl der zu betrachtenden Kante modifiziert werden, sodass nicht nur die Längenabweichung sondern auch die Anzahl der vorigen Neupositionierungsversuche berücksichtigt wird.

Ein weiteres Problem trat beim Löschen von alten Knoten auf. Durch die Positionstreue eines alten Knotens werden auch die mit ihm durch Kanten verbundenen jüngeren Knoten stabil in seiner Umgebung gehalten. Wird jetzt der alte Knoten gelöscht, verlieren die jüngeren Knoten ihren Anker. Dadurch verschwindet die Struktur um den gelöschten Knoten teilweise schlagartig. Um das zu verhindern, wurde der Algorithmus um ein sogenanntes „Blitzaltern“ von Knoten ergänzt. Dabei wird den jüngeren Knoten, die mit einem gelöschten Knoten verbunden waren, das Alter des gelöschten Knotens

gegeben. Diese Knoten behalten das höhere Alter so lange, bis ihr reales Alter das künstliche übersteigt. Dadurch ändert sich die optische Struktur um den gelöschten Knoten im weiteren Verlauf der Graphsequenz nur langsam.

Der so erweiterte Algorithmus nutzt als Ausgangpunkt für die Berechnung eines Layouts L_i die Positionen der Knoten im Layout L_{i-1}. Neu hinzugefügten Knoten werden zufällige Ausgangspositionen zugewiesen.

4.3.3 Verwendete Qualitätsmaßstäbe

Die Qualität eines Layouts kann auf verschiedene Arten gemessen werden (siehe Abschnitt 3.2). Um die Qualität des Layouts einer Sequenz von Graphen ermitteln zu können werden vorhandene Qualitätsmaßstäbe für Einzellayouts mit Differenzmetriken für aufeinander folgende Graphlayouts kombiniert werden.

Qualitätsmaßstäbe für Einzellayouts

In dieser Arbeit wird das Vorhandensein von Knotenüberlappungen als wichtigstes Merkmal eines unübersichtlichen Layouts angesehen, da übereinanderliegende Knoten den Betrachter sehr verwirren können. Der hier bisher entwickelte Algorithmus kann Knotenüberschneidungen im Layout nicht ausschließen. Aus diesem Grund wurde eine Möglichkeit zum Auflösen von Knotenüberlappungen in den Algorithmus integriert. Dazu wird nach Abschluss der Hauptschleife nach Knotenüberlappungen gesucht. Werden zwei Knoten gefunden, die einander überlappen, so werden diese Knoten so verschoben, dass sie nicht mehr übereinander liegen. Dadurch können allerdings auch neue Knotenüberlappungen entstehen. Daher wird dieser Vorgang so lange wiederholt, bis keine Überlappungen mehr gefunden werden.

In dieser Arbeit wird auch auf möglichst einheitliche Kantenlängen Wert gelegt, da dadurch die Übersichtlichkeit des Layouts gefördert werden kann. Dazu wird der Teil des Layoutalgorithmus modifiziert, der die Berechnung der anziehenden Kräfte (und daraus resultierende Positionsänderungen) vornimmt. Die Iterationsanzahl dieser Unterschleife wird nicht mehr nur statisch angegeben. Die Schleife sollte so lange durchlaufen werden, bis alle Kanten ihre gewünschte Länge (bis auf eine festgelegte Toleranzgrenze) erreicht haben. Eine feste Anzahl von Iterationen wird aber weiterhin als zusätzliches Abbruchkriterium benötigt, da Kanten existieren können, deren Endknoten sich nicht verschieben lassen. Ohne ein vorgegebenes Abbruchkriterium würde das zu einer Endlosschleife führen.

Zur Bewertung der Qualität des Einzellayouts werden zusätzlich zwei weitere Metriken herangezogen. So werden die Anzahl der Überschneidungen von Kanten und Knoten sowie die Anzahl der Kantenkreuzungen ermittelt. Dabei gilt: Je weniger Überschneidungen und Kreuzungen, desto besser ist das Layout. Die Vermeidung von Überschneidungen von Kanten und Knoten trägt dabei stärker zur Qualität des Layouts bei, als die Minimierung der Kantenkreuzungsanzahl. Diese Metriken werden aber nicht direkt zur Berechnung des Layouts herangezogen. Die Kantenkreuzungsmetrik wäre dazu auch nicht geeignet, da Kantenkreuzungen in vielen Graphen nicht vermieden werden können.

Es besteht aber die Möglichkeit ein neues Layout auf Basis der aktuellen Positionierung zu erzeugen, wenn die ermittelten Metrikwerte zu hoch erscheinen.

Metriken zur Differenz von Layouts aufeinanderfolgender Graphen

Zur Ermittlung der Differenz zwischen den Layouts von zwei aufeinander folgenden Graphen können die Metriken zur mentalen Distanz von Brigdeman und Tamassia [BT98] verwendet werden. Für diese Arbeit wurden daraus eine Abstandsmetrik und eine Clustermetrik ausgewählt. Die Abstandsmetrik ermittelt die durchschnittliche Positionsänderung aller Knoten der Graphen G_i und G_{i+1} von Layout L_i zu L_{i+1}. Die Clustermetrik misst, inwieweit Knotengruppen (Cluster) im Layout erhalten bleiben. Dazu wird in beiden Layouts für jeden Knoten ein *epsilon*–Cluster ermittelt. Als *epsilon*–Cluster eines Knotens K werden alle anderen Knoten bezeichnet, die in einem vorgegebenen Abstand *epsilon* von K positioniert sind. Die Cluster in beiden Layouts werden anschließend verglichen. Die Metrik ermittelt die durchschnittliche Anzahl zum Cluster hinzugefügter und aus dem Cluster entfernter Knoten. Auch bei diesen beiden Metriken weist ein kleiner Wert auf hohe Qualität hin.

4.3.4 Integration von Layout Pattern

Durch Layout Pattern können Regeln angegeben werden, die bestimmen, wie bestimmte Teilgraphen dargestellt werden sollen. Damit lassen sich bestimmte Erwartungen an das Layout von speziellen Graphen (unter anderem auch Klassendiagramme) erfüllen. Wodurch solche Erwartungen entstehen, wie sie aussehen und wie sie durch Layout Pattern realisiert werden können, wurde in Abschnitt 4.2 beschrieben. Die dort beschriebenen Layout Pattern lassen sich größtenteils in den hier entwickelten Layoutalgorithmus integrieren. Die auf die Kantenlänge bezogenen Pattern sind in Form der bevorzugten Kantenlänge schon integraler Bestandteil der Layoutberechnung.

Die komplexeren Knotengruppenpattern, die das Layout innerhalb einer Knotengruppe regeln, lassen sich in den hier entwickelten Layoutalgorithmus nicht integrieren. Der hier verwendete Layoutansatz sieht eine relativ freie Positionierung der Knoten nach einem Spring Embedder Energiemodell mit direkter Kantenführung vor. Damit ergibt sich ein eher radial orientiertes Layout. Daher lässt sich ein Umschalten auf ein orthogonales Layout für eine Knotengruppe nicht ohne weiteres einbinden.

Die Realisierung der restlichen hier vorgestellten Layout Pattern erfolgt grundsätzlich durch die Definition einer „Schranke“, die die Knoten bei einer Positionsänderung nicht überschreiten dürfen. Die Lage dieser Schranke wird bei Einzelpattern und Knotengruppenpattern allein von dem jeweiligen Pattern bestimmt. Bei Kantenpattern ist zusätzlich die Position des jeweiligen Partnerknotens, bei Gruppen–Gruppen–Pattern die Position aller Knoten aus der Partnergruppe ausschlaggebend. Im Rahmen dieser Arbeit wurden exemplarisch die Kantenpattern realisiert.

Bei jedem Versuch einen Knoten zu verschieben, wird überprüft, ob Layout Pattern für ihn existieren. Ist das der Fall, wird sichergestellt, dass der Knoten die von den Pattern vorgegebenen Schranken nicht überschreitet. Eine Ausnahme, bei der das Verschie-

ben von Knoten ohne Beachtung der Layout Pattern möglich ist, ist das Auflösen von Knotenüberlappungen. Wie schon in Abschnitt 4.3.3 beschrieben, erzeugen Knotenüberlappungen immer unübersichtliche Layouts. Ihre Beseitigung wird daher als wichtiger angesehen, als die Einhaltung der Layout Pattern.

Um den Grad der Einhaltung von Layout Pattern messen zu können wurde eine einfache Patternmetrik entwickelt. Diese Metrik zählt die Verstöße gegen die Pattern. Eine komplexere Patternmetrik könnte die durchschnittliche Überschreitung der durch die Layout Pattern bestimmten Schranken messen. In beiden Fällen kann das Ergebnis für die einzelnen Patternarten individuell ermittelt werden.

4.4 Komplexitätseinschätzung

In diesem Abschnitt erfolgt eine Komplexitätseinschätzung des vorgestellten Layoutalgorithmus. Zusätzlich werden auch die Umwandlung von Sequenzen von Klassendiagrammen in Graphgrammatiken sowie die Berechnung der in Abschnitt 4.3.3 vorgestellten Qualitätsmetriken auf ihre Komplexität hin untersucht. Die Aufwandsabschätzung erfolgt unter Annahme des schlimmst möglichen Falles.

Der Aufwand für die Umwandlung des initialen Klassendiagramms in den Ausgangsgraphen einer Graphgrammatik ist linear bezüglich der Anzahl der enthaltenen Klassen und Beziehungen. Bei der Regelerzeugung wird für alle Klassen und Beziehungen eines Klassendiagramms überprüft, ob sie im folgenden Klassendiagramm enthalten sind. Eine solche Prüfung hat quadratischen Aufwand. Wenn eine Sequenz von Klassendiagrammen i Diagramme beinhaltet, müssen $i-1$ Regeln erstellt werden. Das heißt, der Aufwand für dieUmwandlung einer Sequenz von Klassendiagrammen in eine Graphgrammatik berechnet sich wie folgt:

$$c_1(k+a) + \sum_{i-1} c_2(k^2+a^2) \tag{4.4}$$

Die Variablen k und a bezeichnen die maximale Anzahl der Klassen und Beziehung in einem Klassendiagramm. Dieser Aufwand lässt sich in der Aufwandsklasse $\Theta(k^2+a^2)$ zusammenfassen.

Der Aufwand des Layoutalgorithmus lässt sich in den Aufwand für die Berechnung der abstoßenden und anziehenden Kräfte unterteilen. Die Berechnung der abstoßenden Kräfte zwischen den Knoten erfolgt für jedes Knotenpaar, hat also quadratischen Aufwand. Die Berechnung der daraus resultierenden Positionsänderungen erfolgt für jeden Knoten, hat also linearen Aufwand. Ohne Betrachtung von Layout Pattern beträgt der Aufwand für die Berechnung der abstossenden Kräfte also $c_1 * \mid V \mid^2 + c_2 * \mid V \mid$. Wenn Layout Pattern genutzt werden, kann sich der Aufwand für die Berechnung erhöhen. Durch die Beachtung von Kantenpattern müssen bei der Berechnung von Positionsänderungen alle Kanten des jeweiligen Knotens auf vorhandene Pattern geprüft werden. Dadurch wird der Aufwand im schlimmsten Fall um den Faktor $\mid E \mid$ auf $c_1 * \mid V \mid^2 + c_2 * \mid V \mid * \mid E \mid$ erhöht.

Anziehende Kräfte treten nur zwischen Knoten auf, die durch eine Kante verbunden sind. Der Aufwand für die Berechnung dieser Kräfte und der daraus resultierenden Po-

sitionsänderungen ist also von der Kantenzahl abhängig. In jeder Iteration wird nur eine Kante betrachtet. Die Auswahl dieser Kante hat linearen Aufwand bezüglich der Kantenzahl, also $c_3 * \mid E \mid$. Der Aufwand des gesamten Layoutalgorithmus ergibt sich aus der Summe der Aufwände zur Berechnung der abstoßenden und anziehenden Kräfte. Ohne die Beachtung von Pattern liegt er also in der Aufwandsklasse $\Theta(\mid V \mid^2 + \mid E \mid)$. Werden Kantenpattern beachtet, erhöht sich der Aufwand zwar, bleibt aber weiterhin quadratisch.

Auch der Aufwand zur Berechnung der einzelnen Qualitätsmetriken hängt von der Anzahl der Knoten und Kanten des Graphen ab. Den höchsten Aufwand haben die Metriken zu Knotenüberlappungen ($\Theta(\mid V \mid^2)$), Kantenkreuzungen ($\Theta(\mid E \mid^2)$) und Clustererhaltung ($\Theta(\mid V \mid^2$). Der Aufwand zur Berechnung der Anzahl von Überschneidungen von Knoten und Kanten beträgt $\Theta(\mid V \mid * \mid E \mid)$. Die Berechnung der Metriken zur Längenabweichung und zur Patterneinhaltung haben jeweils linearen Aufwand bezüglich der Kantenanzahl ($\Theta(\mid E \mid)$). Der Aufwand zur Berechnung der Positionserhaltungsmetrik ist linear in Bezug auf die Knotenzahl ($\Theta(\mid V \mid)$).

5 Implementation des Algorithmus

Der im vorangegangenen Kapitel entwickelte Layoutalgorithmus wurde im Rahmen dieser Diplomarbeit in das Graphtransformationstool AGG (siehe Abschnitt 3.3.1) integriert. Um diese Integration zu ermöglichen waren zunächst einige Erweiterungen des AGG erforderlich. Diese Erweiterungen werden in Abschnitt 5.1 erläutert. Im Anschluss daran wird in Abschnitt 5.2 die Umsetzung der Grundidee, also die Umwandlung einer Sequenz von Klassendiagrammen in eine AGG–verständliche Graphgrammatik, vorgestellt. In Abschnitt 5.3 wird dann die Implementation des eigentlichen Layoutalgorithmus erklärt.

5.1 Vorbereitung

Die Integration des in dieser Arbeit entwickelten Layoutalgorithmus in das Tool AGG erfordert einige Erweiterungen dieses Tools. Vor allem musste die Datenstruktur erweitert werden, in der AGG Graphgrammatiken verwaltet. Diese Datenstruktur trennt inhaltliche (strukturelle) und für die Darstellung bestimmte Daten voneinander. Die strukturellen Daten über den Aufbau der Graphgrammatik inklusive des Ausgangsgraphen und der Transformationsregeln werden in den Klassen des Pakets AGG.XT_BASIS vorgehalten. Die Informationen bezüglich der Darstellung der Elemente einer Graphgrammatik sind in den Klassen des Pakets AGG.EDITOR.IMPL enthalten. Dort existieren zu den Basisklassen korrespondierende Klassen, welche die Darstellung der jeweiligen Elemente bestimmen. Für diese Arbeit sind davon hauptsächlich die Klassen EDGRAGRA, EDGRAPH, EDNODE und EDARC von Interesse. Die Position eines Knotens in der Darstellung ist in der Klasse EDNODE enthalten. Diese Klasse enthält eine Referenz auf ein Objekt der Klasse EDTYPE, das die Erscheinungsform von Knoten (rund, viereckig, etc.) bestimmt. Ähnliches leistet die Klasse EDARC für die Kanten des Graphen.

Um das Konzept der Trennung von Daten aus verschiedenen Ebenen beizubehalten wurden die für den entwickelten Layoutalgoritmus zusätzlich erforderlichen Informationen in neue Klassen gekapselt. Solche zusätzlichen Informationen sind unter anderem die bevorzugte und aktuelle Länge von Kanten sowie das Alter von Knoten und der bevorzugte Mindestabstand zu anderen Knoten. Dazu wurden im Paket AGG.LAYOUT die Klassen LAYOUTARC für Kantendaten und LAYOUTNODE für Knotendaten erzeugt. Diese Klassen werden durch eine Referenz in den Klassen EDARC beziehungsweise EDNODE in die vorhandene Datenstruktur eingebunden.

Zur Realisierung von Layout Pattern wurde die Klasse LAYOUTPATTERN erstellt. Damit werden vorerst nur Kantenpattern implementiert. Die Klasse EDGRAGRA wurde um einen VECTOR<LAYOUTPATTERN> ergänzt, in dem alle Layout Pattern für diese Grammatik vorgehalten werden. Weiterhin wird in dieser Klasse jetzt zusätzlich zum aktuellen

Graphen eine Kopie des direkten Vorgängergraphen vorgehalten. Der Vorgängergraph wird vor allem zur Berechnung der Differenzmetriken benötigt. Zu dem selben Zweck wurde die Klasse EdNode um eine eindeutige ID erweitert, mit der sich Knoten leicht unterscheiden lassen. Eine schematische Darstellung des Pakets agg.layout und seiner Einbindung in die AGG–Paketstruktur ist in Abbildung 5.1 dargestellt.

5.2 Datenaufbereitung

Um Sequenzen von Klassendiagrammen mit dem entwickelten Algorithmus layouten zu können muss eine solche Sequenz zuerst in eine Graphgrammatik umgewandelt werden. Wie diese Umwandlung erfolgen soll, ist in Abschnitt 4.1 theoretisch ausgeführt worden. Im Rahmen dieser Arbeit wird das Omondo Eclipse UML Studio als Datenquelle herangezogen. Dieses Tool speichert Klassendiagramme in einem XML-Dialekt in .ucd–Dateien. Eine Sequenz von so vorliegenden Klassendiagrammen muss also in eine Graphgrammatik umgewandelt werden, die im von AGG verwendeten .ggx–Format vorliegt. Die hierzu nötige Funktionalität ist in der Klasse Converter im Paket ucdConvert implementiert.

Ausgangspunkt der Umwandlung ist eine Konfigurationsdatei, in welcher die Pfade zu den einzelnen .ucd–Dateien mit den Klassendiagrammen in der richtigen Reihenfolge aufgeführt sind. Diese Dateien werden nacheinander mit Hilfe des JDOM–Frameworks [Hun06], [Har02] eingelesen. Dieses Framework überführt XML–Dateien [xml04], [Har98] nach dem Document Object Model (DOM) [dom98], [Mar02] in eine Java–Objektstruktur. Der Inhalt einer .ucd–Datei wird dabei in einem Document–Objekt gespeichert. Dieses Objekt bietet Methoden zum gezielten Zugriff auf Daten, die in der .ucd–Datei gespeichert sind. Dadurch wird das Parsen der .ucd–Dateien erleichtert.

Die Umwandlung wird in der Methode convert() angestoßen. Dabei wird zuerst ein GraGra–Objekt angelegt, das die Graphgrammatik repräsentiert. Für dieses Objekt wird ein TypeSet–Objekt erstellt, das die Typen der Knoten und Kanten verwaltet. In diesem TypeSet wird dann für jede Kantenart Klassendiagramm (Assoziation, Vererbung, Dependency und alle Kombinationen aus diesem Arten) ein Type–Objekt erstellt. Dann wird das erste Klassendiagramm mittels JDOM geparst. Aus dem so entstehenden Document–Object werden die Knoten und Kanten ausgefiltert. Für jeden Knoten werden der Name und seine Position in einem ClassNode–Objekt, für jede Kante ihr Typ und ihre Endknoten in einem Connection–Objekt zwischengespeichert. Die Methode createGraph() erstellt aus den zwischengespeicherten Informationen von Knoten und Kanten ein Graph–Objekt, welches das Klassendiagramm repräsentiert. Dabei wird für jeden Knoten ein eigenes Type–Objekt und damit ein Node–Objekt angelegt. Die Kanten werden durch Arc–Objekte repräsentiert. Das Graph–Objekt ist der Ausgangsgraph des GraGra–Objekts.

Im Anschluss daran werden in der Methode createRules() aus den weiteren Klassendiagrammen Regeln erstellt. Dazu werden in einer Schleife nacheinander alle aufeinander folgenden Paare von Klassendiagrammen eingelesen. Die Methode createRule()

vergleicht die Klassendiagramme eines solchen Pfades miteinander. Wenn Unterschiede festgestellt werden, wird ein RULE–Objekt erstellt, das diese Unterschiede repräsentiert. Dabei wird sichergestellt, dass jede so erstellte Regel nur genau einmal ausgeführt werden kann und die Ausführung in der richtigen Reihenfolge erfolgt. Die so erstellten Regeln werden dem GRAGRA–Objekt hinzugefügt.

Mit der so erstellten Grundstruktur der Graphgrammatik wird jetzt ein EDGRAGRA–Objekt erstellt. Das ist erforderlich um der Graphgrammatik Informationen zum Layout hinzuzufügen. Dabei wird für jeden Knoten die Position aus der .ucd–Datei übernommen. An dieser Stelle werden auch die bevorzugten Längen der Kanten gesetzt. Abschließend wird das EDGRAGRA–Objekt mit Hilfe der Klasse XMLHELPER aus dem Paket AGG.UTIL in eine .ggx–Datei geschrieben. Diese Datei kann dann mit dem AGG geöffnet werden.

5.3 Implementation des Layoutalgorithmus

In diesem Abschnitt erfolgt eine Bescheibung der Implementation des in Abschnitt 4.3 vorgestellten Layoutalgorithmus im Rahmen des Tools AGG. Da sowohl die Betrachtung von Graphsequenzen als auch die Berücksichtigung von Layout Pattern direkt in dem Algorithmus zum Layout von Einzelgraphen integriert ist, erfolgt die Implementation dieser Teilbereiche ebenfalls nahtlos. Die Details dazu werden in Abschnitt 5.3.2 erläutert. Die Berechnung der in Abschnitt 4.3.3 vorgestellten Metriken zur Qualitätsbestimmung erfolgt unabhängig vom Layoutalgorithmus. Da einige dieser Metriken während der Layoutberechnung verwendet werden, wird im Folgenden die Realisierung dieser Qualitätsmetriken beschrieben. Die Beschreibung der Algorithmusimplementierung erfolgt im Anschluss daran.

5.3.1 Qualitätsmetriken

Die Implementation der in Abschnitt 4.3.3 vorgestellten Qualitätsmetriken erfolgt in der Klasse LAYOUTMETRICS im Paket AGG.LAYOUT. Auch die in Abschnitt 4.3.4 neu hinzugekommene Metrik zur Einhaltung von Layoutpattern ist dort realisiert.

Die Berechnung der Anzahl der Knotenüberlappungen erfolgt in der Methode GETNODEINTERSECT(). Dort wird für jedes Knotenpaar geprüft, ob eine Überlappung der Knotendarstellung auftritt. Die Anzahl der Überlappungen wird über alle Knoten aufsummiert. Die Methode gibt die Summe als Integer–Wert zurück. Analog dazu werden in der Methode GETARCARCINTERSECT() die Anzahl der im Layout vorhandenen Kantenkreuzungen ermittelt. Die Anzahl der Überschneidungen von Knoten und Kanten wird in der Methode GETARCNODEINTERSECT() ermittelt. Hier wird für jede Kante bestimmt, wie viele Knoten von ihr geschnitten werden. Diese Werte werden über alle Kanten aufsummiert. Auch diese Methode gibt die ermittelte Summe als Integer–Wert zurück. Die Kantenlängenmetrik wird in der Methode GETAVERAGEARCLENGTHDEVIATION() berechnet. Dort werden in einer Schleife über alle Kanten die auftretenden Längenabweichungen von den bevorzugten Kantenlängen ermittelt. Als Ergebnis wird der Durchschnitt aller Abweichungen zurückgegeben.

Weiterhin wurden die Differenzmetriken zur Positionstreue und Clustererhaltung implementiert. Für diese Metriken werden zwei Graphen benötigt. Zusätzlich zu dem aktuellen Graphen wird der Graph vor dem letzten Transformationsschritt betrachtet. Hierfür wird die oben genannte Kopie des letzten Graphen benötigt. Die Methode getAverageNodeMove() errechnet in einer Schleife über alle Knoten den Abstand der Positionen jedes Knoten, der in beiden Graphen enthalten ist. Die Abstandsberechnung erfolgt hierbei mittels der sogenannten Manhatten–Distanz, also als Summe der Abstände in Richtung der Koordinatenachsen. Der Durchschnitt aller Abstände dient als Maß für die Positionstreue der Knoten. Ein Maß für die Clustererhaltung wird in der Methode calcClusterDiffs() ermittelt. Dazu wird zunächst für jeden Knoten in beiden Graphen je ein *epsilon*–Cluster gebildet. Die Clusterbildung erfolgt dabei nach dem euklidischen Abstand. Dann werden in einer Schleife über alle Knoten die jeweiligen Cluster in beiden Graphen miteinander verglichen. Dabei wird der Durchschnitt der Anzahl der in einem Cluster hinzugekommenen und entfernten Knoten ermittelt. Dieses Wertepaar stellt das Ergebnis der Clustererhaltungsmetrik dar.

Auch die in Abschnitt 4.3.4 vorgestellte Patternmetrik wurde in der Klasse LayoutMetrics implementiert. Dabei werden nur die Kantenpattern betrachtet. In der Methode getPatternMistakes() wird in einer Schleife über alle Kanten die Einhaltung vorhandener Kantenpattern geprüft. Die Summe aller Verstöße bilden das Ergebnis dieser Metrik.

5.3.2 Layoutalgorithmus

Nachdem die von AGG genutzte Datenstruktur (wie in Abschnitt 5.1 beschrieben) erweitert wurde, erfolgte die Implementierung des Layoutalgorithmus (siehe Abschnitt 4.3). Die Layoutfunktionalität ist in der Klasse Layouter im Paket agg.layout realisiert. Abbildung 5.1 zeigt eine schamatische Darstellung dieses Pakets und der Beziehungen der enthaltenen Klassen zu anderen Klassen der AGG–Implementation.

Die Layoutberechnung wird mit der Methode layoutGraph() angestoßen, die einen Graphen in Form eines EdGraph–Objekts als Parameter übergeben bekommt. Diese Methode beinhaltet die Hauptschleife des Algorithmus.

In jeder Iteration der Hauptschleife werden zunächst die abstoßenden Kräfte und die daraus resultierenden Positionsänderungen berechnet. Dazu werden in der Methode calcNodeRepulse() in einer Schleife über alle Knoten die gewünschten Positionsänderungen als Summe der abstoßenden Kräfte nach der Formel 4.1 ermittelt. Die Methode calcDistToPos() berechnet daraus die tatsächlichen Positionsänderungen. Dazu wird zuächst für jeden Knoten seine individuelle Bewegungsfreiheit (Temperatur siehe Abschnitt 4.3) in Abhängingkeit von der Iteration der Hauptschleife und dem Alter des Knotens mittels der Formeln 4.2 und 4.3 berechnet. Dies geschieht in den Methoden cool() und reduceTempByAge(). Damit wird die Positionsänderung beschränkt. An dieser Stelle wird auch nach relevanten Kantenpattern gesucht, welche die gewünschte Positionsänderung unter Umständen weiter einschränken. Zuletzt wird sichergestellt, dass die Knoten den verfügbaren Darstellungsbereich nicht verlassen.

Der zweite Schritt innerhalb der Hauptschleife besteht in der Angleichung der Kan-

tenlängen an die bevorzugten Werte. Dieser Schritt stellt die Berechnung der anziehenden Kräfte zwischen Knoten dar, die durch eine Kante verbunden sind. Die Berechnung erfolgt in der Methode LAYOUTBYARCLENGTH() in einer weiteren Schleife. Dort wird in jeder Iteration zunächst die Kante mit der größten Längenabweichung ermittelt. Die einzelnen Längenabweichungen werden dabei dadurch relativiert, wie oft eine Kante schon bearbeitet wurde. Dadurch wird verhindert, dass Kanten mit starren Endknoten eine Endlosschleife verursachen. Wenn eine Kante ausgewählt wurde, werden die Positionen ihrer Endknoten so geändert, dass die Längenabweichung minimiert wird. Diese Positionsänderungen unterliegen den gleichen Beschränkungen bezüglich der Temperatur, geltenden Layout Pattern und dem verfügbaren Darstellungsbereich, wie die aus den abstoßenden Kräften resultierenden Positionsänderungen (siehe oben). Die Schleife wird beendet, wenn alle Kantenlängenabweichungen innerhalb einer manuell festgelegten Toleranzgrenze (beispielsweise 50 Pixel) liegen. Durch diese Methode wird das Layout auf die Kantenlängenmetrik hin optimiert.

Nach Abschluss der Hauptschleife wird im erstellten Layout mittels der Methode GETNODEINTERSECT() nach Knotenüberlappungen gesucht. Gefundene Knotenüberlappungen werden durch Verschieben der jeweiligen Knoten gegeneinander aufgelöst. Die einzige Beschränkung dieser Verschiebung ist durch die Grenzen des verfügbaren Darstellungsbereichs gegeben. Dies optimiert das Layout bezüglich der Knotenüberlappungsmetrik.

Die Layoutberechnung wird nach jedem Transformationsschritt automatisch ausgeführt. Dazu wird in der Klasse GRAGRAEDITOR im Paket AGG.GUI ein Objekt der Klasse LAYOUTER instanziert. In der Methode TRASFOMEVENTOCCURED() wird nach jedem Transformationsschritt dann die Methode LAYOUTGRAPH() mit dem aktuellen Graphen ausgeführt. Vorher werden auch alle Vorbereitungen für das Layout durchgeführt. Dazu gehöhrt das Vorhalten des letzten Graphen, die „natürliche“ Alterung des Graphen und seiner Knoten sowie die „künstliche“ Alterung bestimmter Knoten (Blitzalterung siehe Abschnitt 4.3). Im Anschluss an die Layoutberechnung erfolgt auch die Berechnung der vorgestellten Metriken und die Ausgabe der Ergebnisse in einer Log–Datei.

Weiterhin kann die Berechnung eines Layouts für den aktuellen Graphen manuell angestoßen werden. Dazu wurde der Benutzeroberfäche von AGG ein Button hinzugefügt. Dieser Button löst ein ACTIONEVENT aus, das in der Methode ACTIONPERFORMED() bearbeitet wird. Dort wird die Methode LAYOUTGRAPH() mit dem aktuellen Graphen ausgeführt. Durch einen weiteren Button kann die Berechnung der Qualitätsmetriken für den aktuellen Graphen ausgelöst werden. Die Ergebnisse werden in einem Dialog angezeigt.

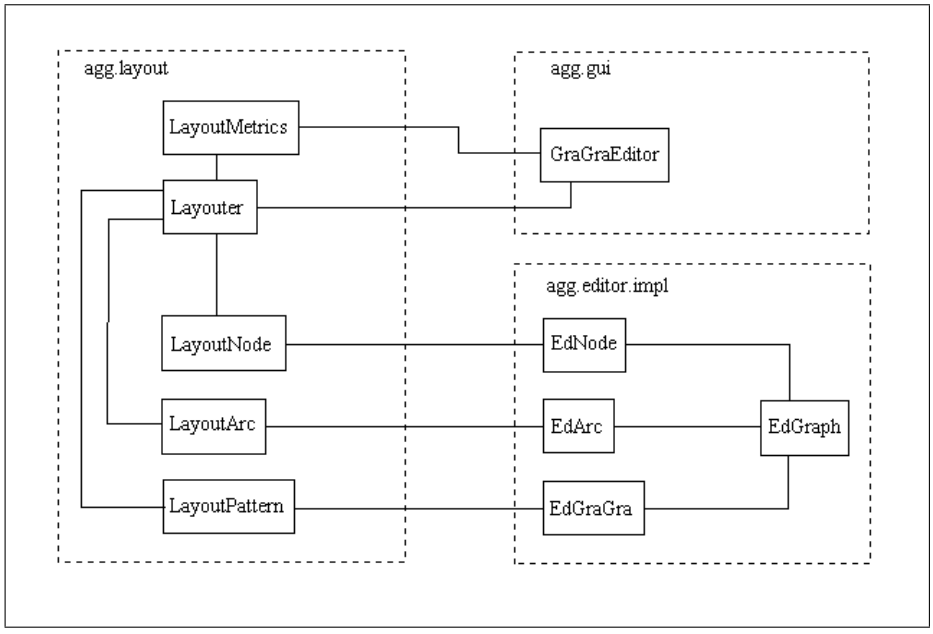

Abbildung 5.1: Schematische Darstellung des Pakets agg.layout und seiner Integration in das AGG

6 Fallstudien

Nachdem ein Lösungsansatz für das in Kapitel 2 motivierte Problem vorgestellt wurde, soll dieser Lösungsansatz (Kapitel 4) und seine Implementierung (Kapitel 5) in diesem Kapitel praktisch getestet werden. Da die Motivation aus zwei verschiedenen Feldern geschöpft wurde, soll auch die Überprüfung der Ergebnisse in diesen zwei Gebieten stattfinden: Zum einen werden im Bereich der Softwaretechnik Layouts von unvollständigen Sequenzen von Klassendiagrammen betrachtet. Zum anderen wird der Layoutalgorithmus anhand von Graphgrammatiken getestet, die Sequenzen von Graphen mit verschiedener Semantik erzeugen.

6.1 Sequenzen von Klassendiagrammen

Um den hier entwickelten Layoutalgorithmus auf unvollständige Sequenzen von Klassendiagrammen anwenden zu können musste ein geeignetes Projekt gefunden werden. Die wichtigste Anforderung an das Beispielprojekt bestand darin, dass Klassendiagramme für die verschiedenen Entwicklungsversionen zur Verfügung stehen. Das stellte sich als ein Problem heraus. Modell–Informationen (und auch der Quellcode) von kommerziellen Software–Projekten sind im allgemeinen nicht frei verfügbar. Daher wurden einige Projekte der Open Source–Plattform Sourceforge.net [Gro05] in Betracht gezogen. Auch bei diesen Projekten sind Modell–Informationen nur sehr selten vorhanden. Da aber der Sourcecode in allen Versionen über den Projektverlauf hinweg frei verfügbar ist, können Klassendiagramme mit einem CASE–Tool erstellt werden. Diese Klassendiagramme müssen in eine Graphgrammatik umgewandelt werden können, die dem AGG–Format (.ggx) entspricht. Im Rahmen dieser Arbeit wurde eine solche Umwandlung nur für Klassendiagramme im .ucd–Format des CASE–Tools Omondo Eclipse UML Studio realisiert. Die Klassendiagramme müssen also in dieser Form vorliegen oder mit Omondo aus dem Sourcecode des Projekts erstellbar sein. Konkret ergeben sich die folgenden Anforderungen:

- Die Implementationssprache ist Java.
- Klassendiagramme liegen im .ucd–Format (Omondo) vor oder lassen sich aus dem Sourcecode erstellen.
- Das Projekt sollte eine gewisse Größe haben, das heißt, es sollten mehrere Pakete mit unterschiedlicher Klassenanzahl vorhanden sein um verschieden große Beispieldiagramme zu erhalten. Es soll also kein sogenanntes Sandkastenbeispiel sein.
- Es muss eine gewisse Anzahl an Versionen des Projekts existieren. Das Projekt muss also schon über einen längeren Zeitraum entwickelt werden.

Nach einiger Recherche im großen Angebot der Enwicklerplattform Sourceforge.net fiel die Wahl auf das Projekt pmd [ea05, Cop05]. Dieses Projekt entwickelt ein Tool zur Analyse von Java–Sourcecode. Der Sourcecode von pmd ist frei verfügbar und es lassen sich mit Omondo Klassendiagramme erstellen. Das Projekt liegt momentan in Version 3.4 vor. Diese Version stellt das 38. Release dar. Ein Release ist eine im CVS–Repository [FB03] manuell erstellte Zusammenfassung einzelner Quelldateiversionen (Revisionen). Ein Release stellt einen Meilenstein des Projekts dar. Diese Releases können einzeln aus dem CVS–Repository bezogen werden, mit dem der Sourcecode des Projekts verwaltet wird. Die Begriffe Release und Version sind im Kontext dieser Arbeit gleichbedeutend. Von hier an wird aber der Begriff Release verwendet, da er die Herkunft des Sourcecode (und damit auch der Klassendiagramme) aus einem CVS–Repository verdeutlicht. Es existiert also eine ausreichende Anzahl von Releases. Das Projekt beinhaltet in dem aktuellen Release mehr als 600 Klassen in 53 Paketen.

Beispiel: Paket PMD.CPD.CPPAST

In diesem Beispiel wird die Entwicklung des Pakets PMD.CPD.CPPAST von Release 2.0 bis 3.4 (ingesamt 9 Releases) betrachtet. Dieses Paket wurde als erstes Beispiel ausgewählt, da es eine überschaubare Anzahl von Klassen (11) und Beziehungen zwischen Klassen (nach der Zusammenfassung 16 Kanten) aufweist. Weiterhin bleibt auch die Entwicklung von Release 2.0 bis 3.4 überschaubar. Es werden weder Klassen hinzugefügt noch entfernt. Änderungen finden nur bei den Beziehungen (Kanten) zwischen den Klassen statt. Aus den Klassendiagrammversionen dieses Pakets wurde eine Graphgrammatik erzeut. Für die so entstehende Graphsequenz wurde ein Layout berechnet, bei dem die Layout Pattern (siehe Abschnitt 4.2.1) beachtet werden und ein weiteres Layout, bei dem solche Pattern keine Beachtung fanden. Anhand der überschaubaren Entwicklung sollen Vor– und Nachteile eines Layouts mit Beachtung von Pattern im Vergleich zu einem Layout ohne Beachtung von Pattern diskutiert werden. Für das Beispiel mit Patternbeachtung wurde ein Layout Pattern für Vererbungskanten erstellt, sodass die Superklasse oberhalb der Subklasse dargestellt wird. In beiden Fällen wurde auch das Layout des Ausgangsgraphen neu berechnet. Der Versuch, die Layoutinformationen aus Omondo zu übernehmen, erwies sich als unpraktikabel, da dort die Knoten sehr weit voneinander entfernt dargestellt sind (siehe Abbbildung 7.1).

Die Abbildungen 6.1, 6.2 und 6.3 zeigen die Layouts des Klassendiagramms des Pakets PMD.CPD.CPPAST in den Releases 2.0, 2.1 und 2.2. Diese Layouts wurden unter Beachtung von Pattern erstellt. Die Knotenpositionen bleiben dabei sehr stabil. Nur beim Wechsel von Release 2.0 auf Release 2.1 erfolgt eine Positionsänderung der Klasse CPPPARSER. Diese Klasse wird nach unten verschoben. Die Ursache hierfür ist in der hinzugefügten Vererbungskante von der Klasse CPPPARSER zur Klasse TOKEN zu suchen. Es wird versucht, das Kantenpattern dieser Vererbungskante zu erfüllen. Dies gelingt hier allerdings nicht in einem Schritt. In Release 2.2 wurde die Vererbungskante wieder gelöscht. Die Klasse CPPPARSER bleibt an der Position aus Release 2.1, da das Kantenpattern nun nicht mehr gültig ist. Durch das Verschieben der Klasse fällt die Änderung von Release 2.0 auf 2.1 sofort ins Auge.

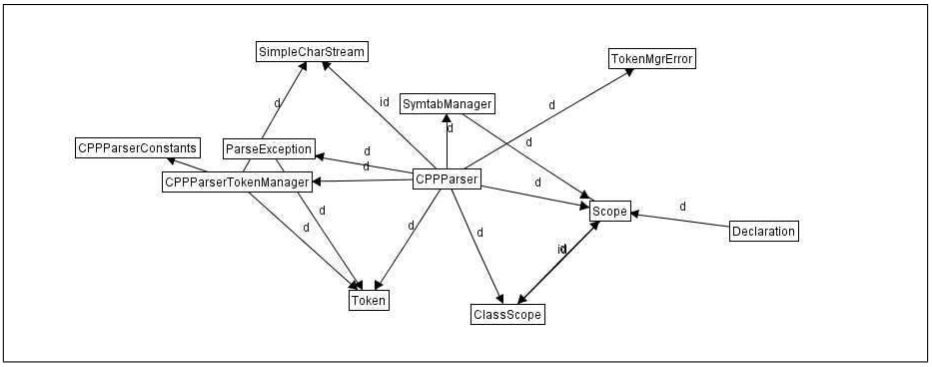

Abbildung 6.1: Klassendiagramm des Pakets pmd.cpd.cppast in Release 2.0

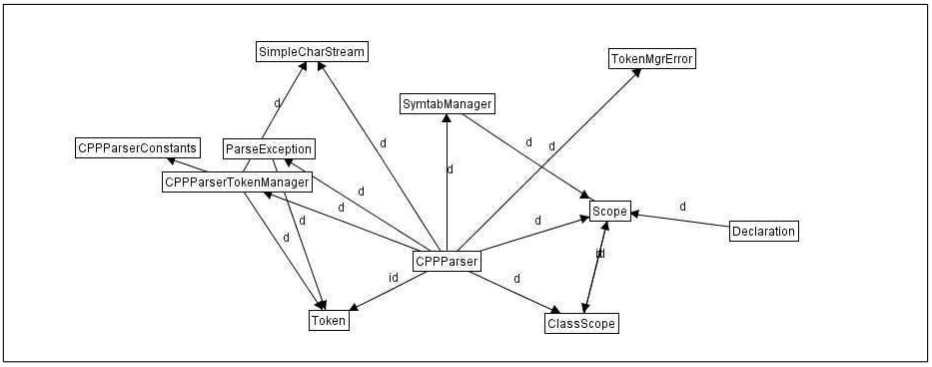

Abbildung 6.2: Klassendiagramm des Pakets pmd.cpd.cppast in Release 2.1

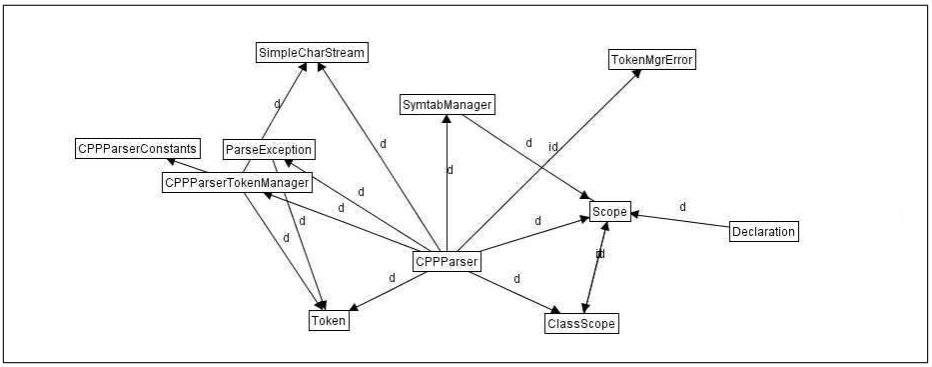

Abbildung 6.3: Klassendiagramm des Pakets pmd.cpd.cppast in Release 2.2

Wenn der Betrachter aber die Darstellung des Release 2.1 des Klassendiagramms überspringt, finden sich in Release 2.2 keine Gründe für die Positionsänderung. Dies kann zu Verwirrungen führen. In allen späteren Releases des Klassendiagramms bleiben die Positionen aller Klassen im Layout stabil. Durch das Hinzufügen von neuen Kanten wird die Qualität der einzelnen Layouts allerdings etwas verringert, da es hierbei zu neuen Kantenkreuzungen und Überschneidungen von Knoten und Kanten kommt. Dies spiegeln auch die Werte der entsprechenden Metriken wieder. Die Ergebnisse der Differenzmetriken bleiben dabei konstant gut.

Die ohne Beachtung von Pattern erstellten Layouts der Klassendiagramme in den Releases 2.0 und 2.1 sind in den Abbildungen 6.4 und 6.5 dargestellt. Durch das Ignorieren der Pattern bei der Berechnung der Layouts stimmen diese mit den oben besprochenen Layouts mit Patternbeachtung nicht überein. Die Positionen aller Klassen bleiben in diesem Fall über alle Releases hinweg stabil. Dadurch fällt allerdings das Hinzufügen einer Vererbungskante zwischen den Klassen CPPPARSER und TOKEN in Release 2.1 nicht sofort ins Auge. Auch die Änderungen in den nachfolgenden Releases sind nicht sofort ersichtlich. Erst die in Release 3.0 hinzugefügte Vererbungsbeziehung zwischen den Klassen CPPPARSER und DECLARATION ist auffälliger, da zwischen diesen Klassen bisher keine Kante vorhanden war (siehe Abbildung 6.6). Auch in diesem Fall verschlechtert sich die Qualität der Einzellayouts durch neu hinzugefügte Kanten leicht, da neue Kantenkreuzungen hinzukommen. Die Differenzmetriken liefern konstant gute Werte. Das gilt sowohl für die Positionsänderung als auch für die Clustererhaltung.

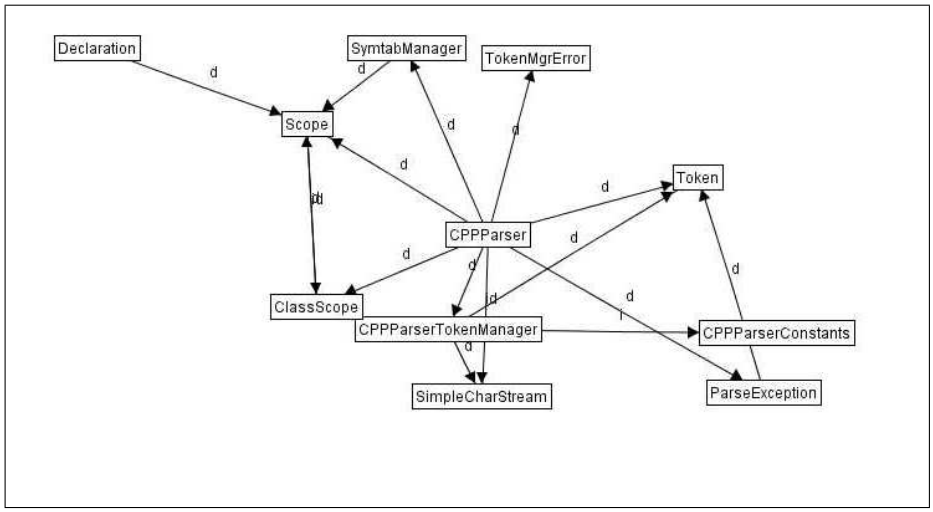

Abbildung 6.4: Klassendiagramm des Pakets pmd.cpd.cppast in Release 2.0 ohne Berücksichtigung von Layout Pattern

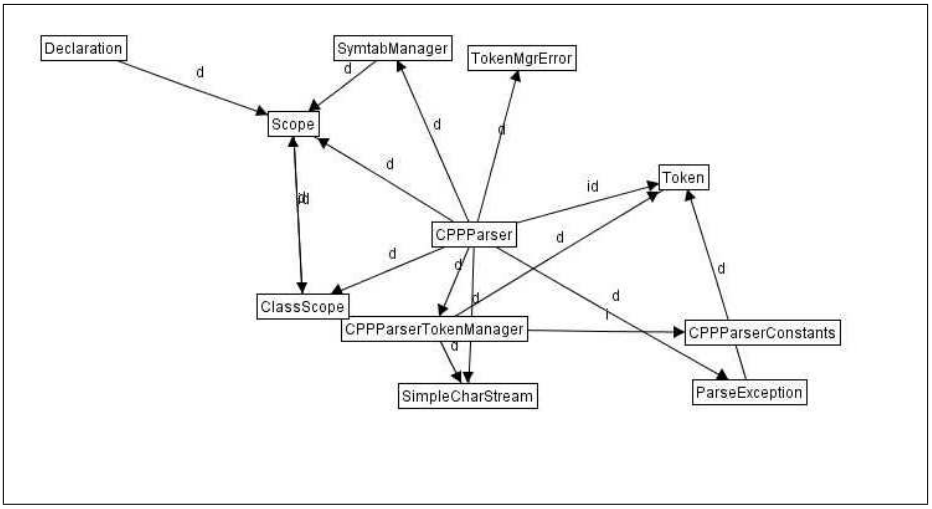

Abbildung 6.5: Klassendiagramm des Pakets pmd.cpd.cppast in Release 2.1 ohne Berücksichtigung von Layout Pattern

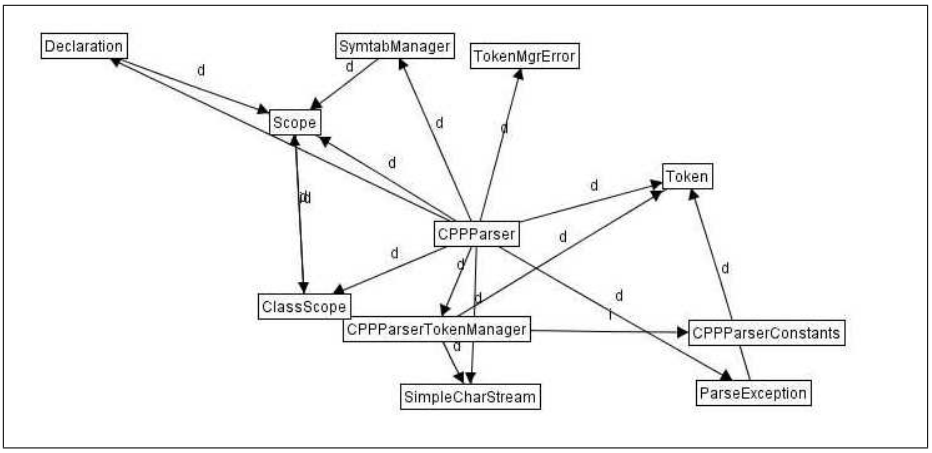

Abbildung 6.6: Klassendiagramm des Pakets pmd.cpd.cppast in Release 3.0 ohne Berücksichtigung von Layout Pattern

Beispiel PMD

Als Beispiel für ein großes, eher unübersichtliches Klassendiagramm wurde das Paket PMD in Release 0.1 herangezogen. Die Entwicklung dieses Pakets ist über alle Releases hinweg sehr unüberschaubar. Es werden von einem Release zum nächsten eine Vielzahl von Klassen und Beziehungen zwischen Klassen entfernt und hinzugefügt. Dabei ist ohne weiterführende Analyse nicht feststellbar, ob die Klassen in jedem Fall tatsächlich gelöscht und neue Klassen hinzugefügt oder ob einige Klassen nur umbenannt werden. Bei der Umwandlung der Klassendiagramme in eine Graphgrammatik entstanden dementsprechend umfangreiche Regeln. Diese Regeln führten das AGG an seine Grenzen. Es war nicht möglich, Transformationen nach diesen Regeln in akzeptabler Zeit durchzuführen. Die Ausführung der Regel zur Überführung des Klassendiagramms von Release 0.1 zu Release 0.2 war beispielsweise auf aktueller Hardware (Intel Pentium4 2,4GHz, 512 MB RAM) auch nach 12 Stunden noch nicht abgeschlossen. Die Aufteilung einer solchen Regel in mehrere kleinere Regeln ist zwar möglich, würde aber das Ergebnis des Layoutallgorithmus ändern, da dieser nach jeder Regelanwendung ausgeführt wird. Es würde also eine größere Anzahl von Layouts erzeugt werden, die kleinere Diagrammänderungen aufweisen.

Aus diesem Grund werden an dieser Stelle nur einzeln erzeugte Layouts von Klassendiagrammversionen des Pakets PMD betrachtet. Bei der Layoutberechnung wurde ein Layoutpattern für Vererbungskanten verwendet. Abbildung 6.7 zeigt ein Layout des Klassendiagramms des Pakets PMD in Release 0.1. Das Layout ist nicht sehr übersichtlich. Dies spiegelt sich auch in den Ergebnissen der Qualitätsmetriken wieder. Laut diesen Ergebnissen existieren mehr als 3000 Kantenkreuzungen und mehr als 300 Überschneidungen von Knoten und Kanten. Auch 3 Knotenüberlappungen konnten nicht vermieden werden. Der Grund hierfür ist neben der Knotenanzahl (42) in dem sehr hohen Verzweigungsgrad (gesamte Kantenzahl: 197) von einigen der Klassen zu finden. Die Klassen PMDTask, JavaParser und JavaParserVisitor sind Endpunkt sehr vieler Kanten. Dadurch kommt es zu Platzproblemen bei der Positionierung der Klassen, welche die jeweiligen zweiten Endpunkte dieser Kanten darstellen. Hierbei kommt es auch zu vielen Uberschneidungen von Kanten und Knoten.

In den Releases 0.2 und 0.3 weisen die Klassendiagramme weniger Klassen und Beziehnungen auf als in Release 0.1. Das Layout dieser Klassendiagramme (Abbildungen 6.8 und 6.9) ist schon dadurch übersichtlicher. Die Zahl der Knotenüberlappungen, Überschneidungen von Kanten und Knoten sowie der Kantenkreuzungen ist wesentlich geringer. Ein Grund dafür ist der geringere Verzweigungsgrad an den Knoten.

Allgemein kann festgestellt werden, dass die Qualität des Layouts eines Klassendiagramms mit steigender Klassenzahl und vor allem mit steigendem Verzweigungsgrad (Kantenanzahl pro Knoten) abnimmt.

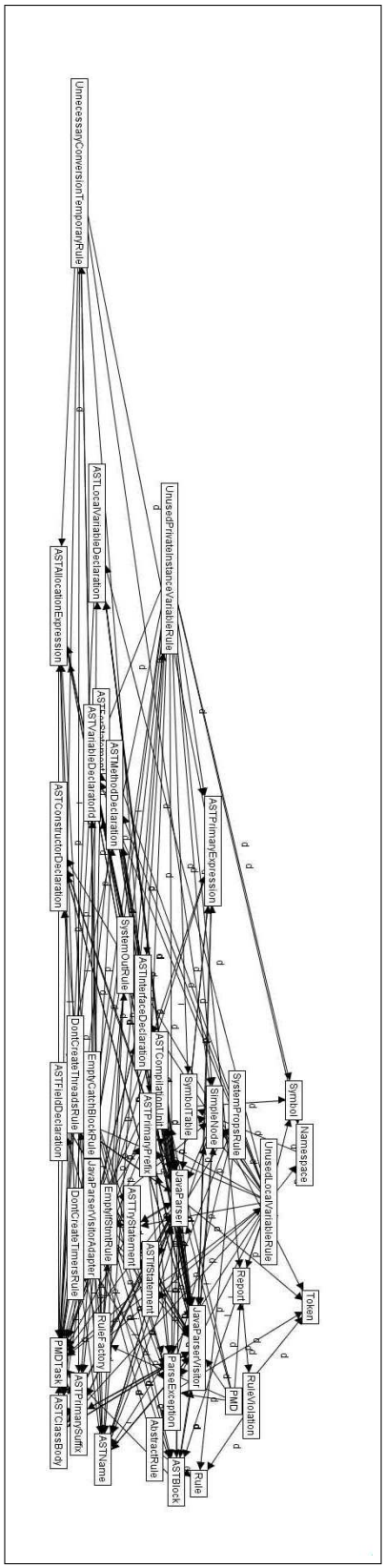

Abbildung 6.7: Klassendiagramm des Pakets pmd in Release 0.1

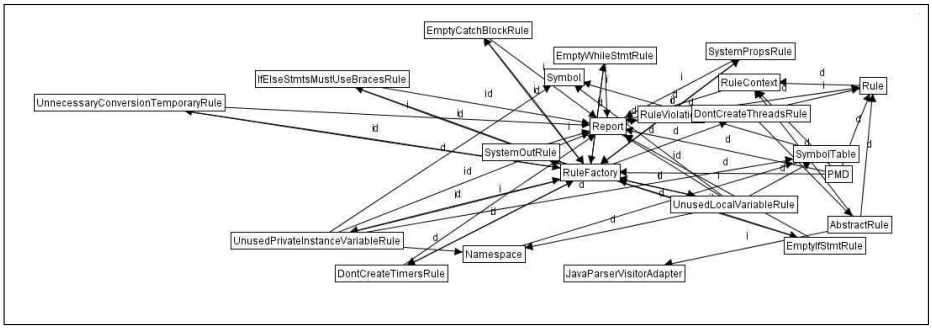

Abbildung 6.8: Klassendiagramm des Pakets pmd in Release 0.2

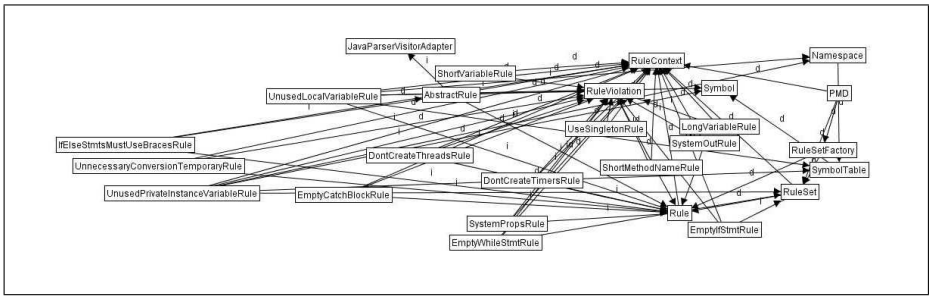

Abbildung 6.9: Klassendiagramm des Pakets pmd in Release 0.3

6.2 Sequenzen von allgemeinen Graphen

Nun soll die Eignung des in dieser Arbeit entwickelten Layoutalgorithmus für Graphsequenzen überprüft werden, die durch Graphtransformation aus einer Graphgrammatik erstellt wurden. Dazu wurden drei Beispielgrammatiken ausgewählt. Als einfachstes Beispiel wurde eine Grammatik erstellt, die Schritt für Schritt eine einfach verkettete Listenstruktur aufbaut. Die zweite Beispielgrammatik baut einen Graphen mit einer Baumstruktur auf. Diese beiden Grammatiken löschen keine Knoten aus dem aktuellen Graphen, sie fügen nur neue Knoten und Kanten hinzu. Die dritte Beispielgrammatik (CD2BD) wandelt ein einfaches Klassendiagramm in eine Datenbankdarstellung um. Dabei werden auch löschende Operationen durchgeführt.

Beispiel: Einfach verkettete Liste

Für dieses Beispiel wurde eine einfache Graphgrammatik erstellt. Diese Gramatik besteht aus einem Ausgangsgraphen mit nur einem Knoten und einer Regel. Die Regel fügt einen neuen Knoten hinzu und verbindet ihn durch eine Kante mit einem vorhandenen Knoten. Dabei wird durch ein Attribut sichergestellt, dass der neue Knoten mit dem zuletzt hinzugefügten Knoten verbunden (also hinten an die Liste „angehängt“) wird.

Die Abbildungen 6.10 bis 6.14 zeigen einige Layouts einer so erstellten Graphsequenz. Die Qualität der einzelnen Layouts ist mit den implementierten Qualitätsmetriken gemessen im Allgemeinen hoch. Die Differenz zwischen zwei aufeinander folgenden Layouts ist in den meisten Fällen gering (siehe Abbildungen 6.10 und 6.11 oder 6.13 und 6.14). Dies wird von den Ergebnissen der Metriken zur Positionsänderung (jeweils kleiner als 20) und Clustererhaltung (jeweils keine aus dem Cluster entfernte und nur ein bis zwei hinzugefügte Knoten) bestätigt. Die optische Struktur wird aber in einigen Fällen umgebrochen. Dies kommt vor allem dann vor, wenn ein Knoten in der Nähe eines Randes der Darstellungsfläche hinzugefügt wird (siehe Abbildungen 6.11 und 6.12). Dann „knickt“ die Darstellung ein. Auch dies ist aus den Ergebissen der Metrik zur Positionsänderung von Knoten erkennbar. Das Ergebnis für den Wechsel von Generation vier zu Generation fünf liegt mit 116 deutlich höher als die oben genannten 20 beim Wechsel von Generation drei zu vier bzw. neun zu zehn.

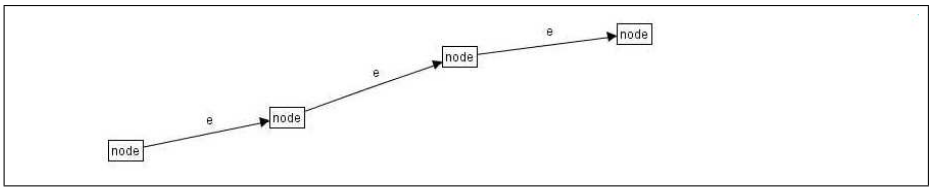

Abbildung 6.10: Liste in Generation 3

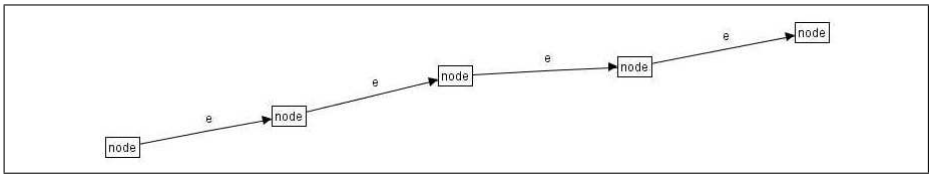

Abbildung 6.11: Liste in Generation 4

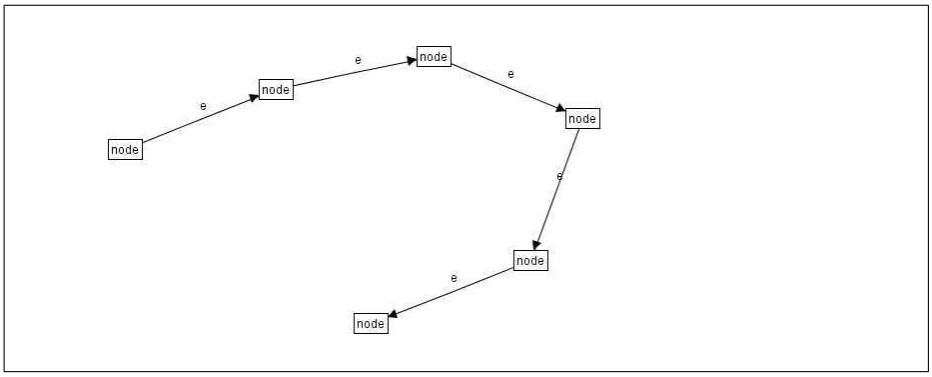

Abbildung 6.12: Liste in Generation 5

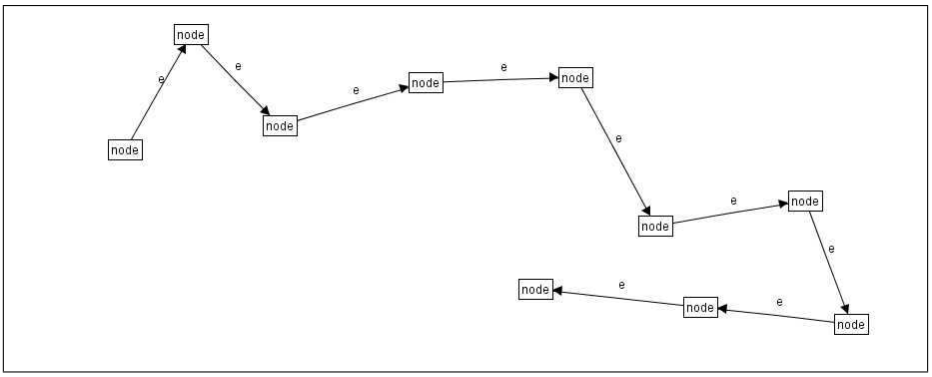

Abbildung 6.13: Liste in Generation 9

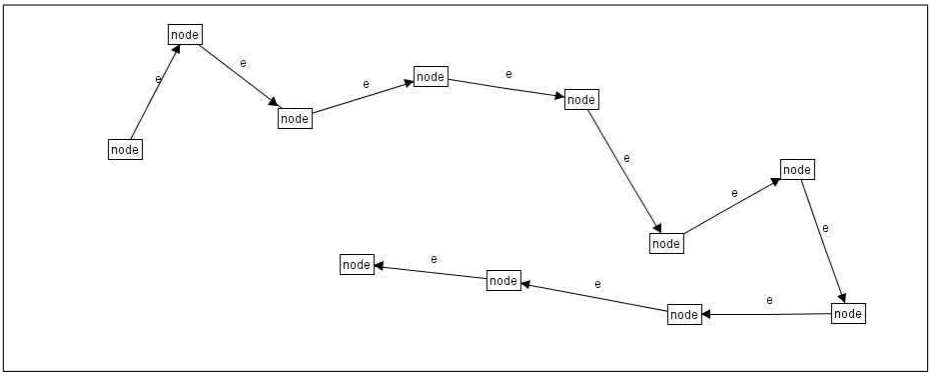

Abbildung 6.14: Liste in Generation 10

Beispiel: Baumstruktur

Als zweites Beispiel wurde eine Graphgrammatik erstellt, die eine Baumstruktur aufbaut. Auch diese Grammatik besteht aus einem Ausgangsgraphen mit nur einem Knoten und nur einer Regel. Diese Regel fügt dem aktuellen Graphen einen Knoten hinzu und erzeugt eine Kante zwischen diesem Knoten und einem schon vorhandenen Knoten. Anders als im vorherigen Beispiel wird der vorhandene Knoten, an den ein neuer Knoten „angehängt“ werden soll, zufällig ausgewählt. Da keine einzelnen Kanten zwischen vorhandenen Knoten hinzugefügt werden, entsteht so eine Baumstruktur. Die erzeugte Baumstruktur ist zusammenhängend, weil keine Regeln zum Löschen von Knoten oder Kanten existieren.

Für diese Grammatik wurde ein Layout Pattern definiert. Dieses Layout Pattern bewirkt, dass alle Kindknoten unterhalb ihres jeweiligen Wurzelknotens dargestellt werden. Auf Basis dieser Grammatik wurden zwei Baumstrukturen inklusive Layouts erstellt. Im ersten Fall wurde das Layout Pattern nicht beachtet, im zweiten Fall wurde das Layout nach dem Pattern ausgerichtet.

Da im ersten Fall keine Layout Pattern beachtet wurden, entstand ein radial orientiertes Layout. Die Abbildungen 6.15 und 6.16 zeigen die Layouts von zwei aufeinander folgenden Graphgenerationen. Dabei ist zu erkennen, dass beim Hinzufügen eines Knotens an einen Zweig des Baums die optische Struktur des anderen Zweigs erhalten bleibt. Dies spiegeln auch die verwendeten Differenzmetriken mit niedrigen Werten wieder. Wenn der Baum weiter vergrößert wird, tritt eine schrittweise Verschiebung der Knoten auf. Abbildung 6.17 zeigt das Layout des Baums in der 22. Generation. Während der Teilbaum im unteren rechten Bereich noch gut dargestellt wird, kommt es oben links zu mehreren Kantenkreuzungen. Dieses Problem tritt bei der Erweiterung des oben links dargestellten Teilbaums auf, da der Darstellungsbereich in diese Richtung nicht erweiterbar ist. Eine Vergrößerung des Darstellungsbereichs erfolgt immer nach rechts und nach unten. Daher kann genügend Platz für den unteren Teilbaum geschaffen werden, sodass hier ein gutes Layout weiterhin möglich ist. Aber auch mit dieser Einschränkung bleibt die Qualität der Layouts hoch, da laut der Qualitätsmetriken nur 3 Kantenkreuzungen aber keine Knotenüberlappungen und Überschneidungen von Knoten und Kanten auftreten.

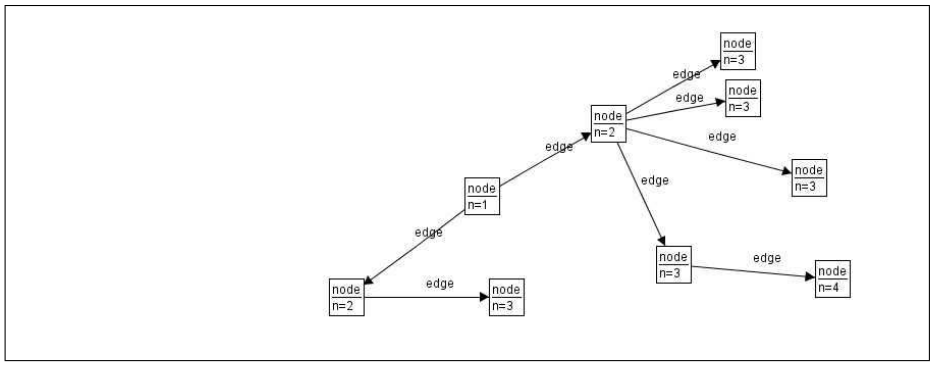

Abbildung 6.15: Baumstruktur in Generation 8 ohne Beachtung von Pattern

Im zweiten Fall wurde das oben definierte Layout Pattern beachtet. Durch das Layout Pattern erhält der Baum eine hierarchische Darstellung, die aber weiterhin radial orientiert ist (halbradial). Die Qualität der einzelnen Layouts ist hoch. Auch bei großen Bäumen treten Kantenkreuzungen und Überschneidungen von Knoten und Kanten seltener auf als ohne Beachtung des Pattern. Die Differenz zwischen aufeinander folgenden Layouts ist auch hier gering. Dies ist in den Abbildungen 6.18 und 6.20 zweier aufeinander folgender Layouts zu erkennen. Dort wird das Layout von Teilbäumen, die nicht geändert werden, nur leicht angepasst. In Abbildung 6.19 ist das Layout des Baums in der 22. Generation zu sehen. Das im ersten Fall (ohne Beachtung des Pattern) auftretende Platzproblem ist hier nicht festzustellen. Dies liegt daran, dass die Teilbäume durch das Pattern nicht mehr nach oben ausgerichtet werden können. Der Baum breitet sich also vermehrt nach unten aus, wo neuer Platz geschaffen werden kann. Die Darstellung unter Beachtung des Layout Pattern bietet einen weiteren großen Vorteil: Der Baum ist auf den ersten Blick als solcher zu erkennen, was in dem Beispiel ohne Layout Pattern nicht immer der Fall ist.

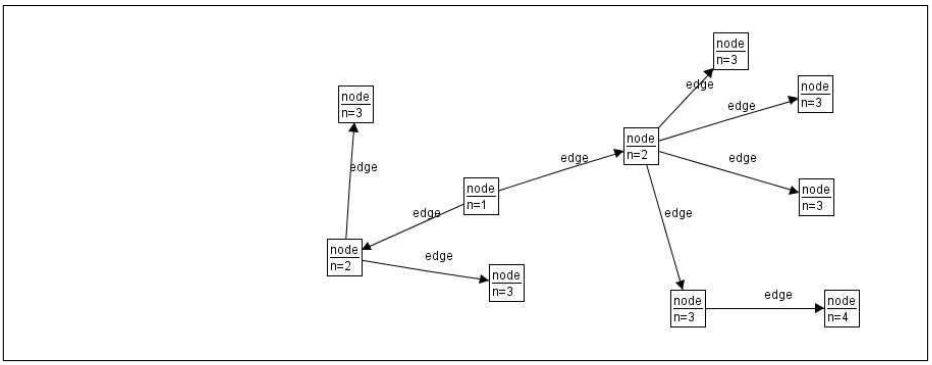

Abbildung 6.16: Baumstruktur in Generation 9 ohne Beachtung von Pattern

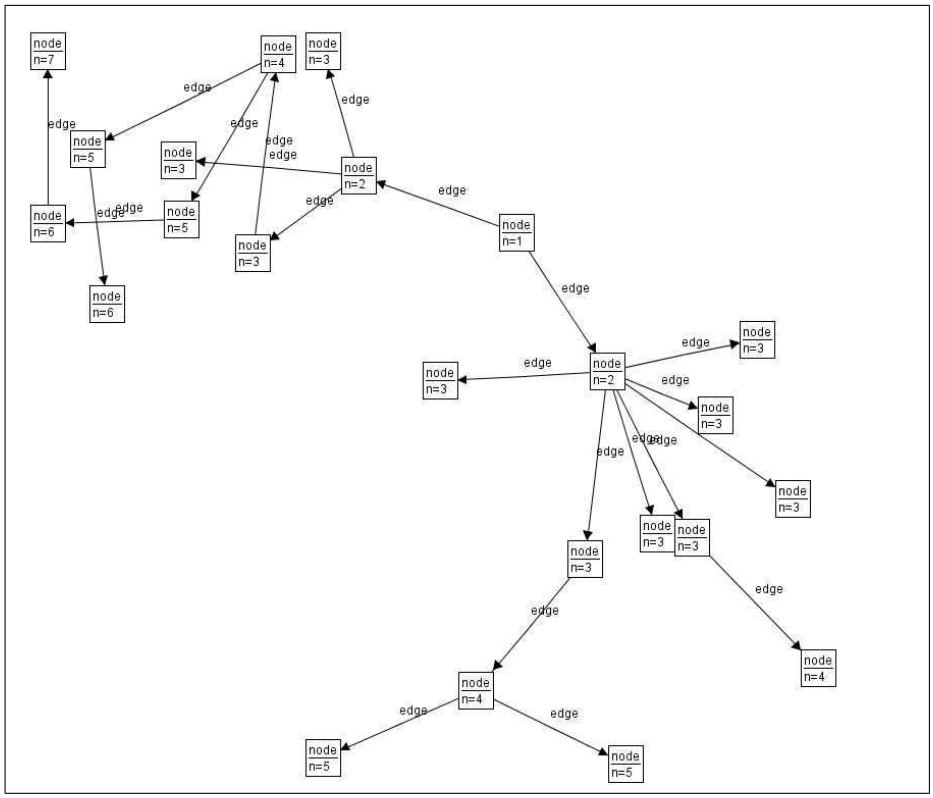

Abbildung 6.17: Baumstruktur in Generation 22 ohne Beachtung von Pattern

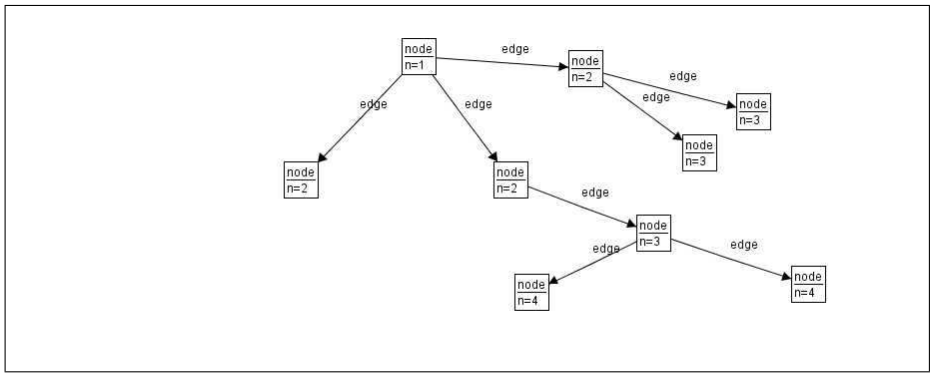

Abbildung 6.18: Baumstruktur in Generation 8 unter Beachtung von Pattern

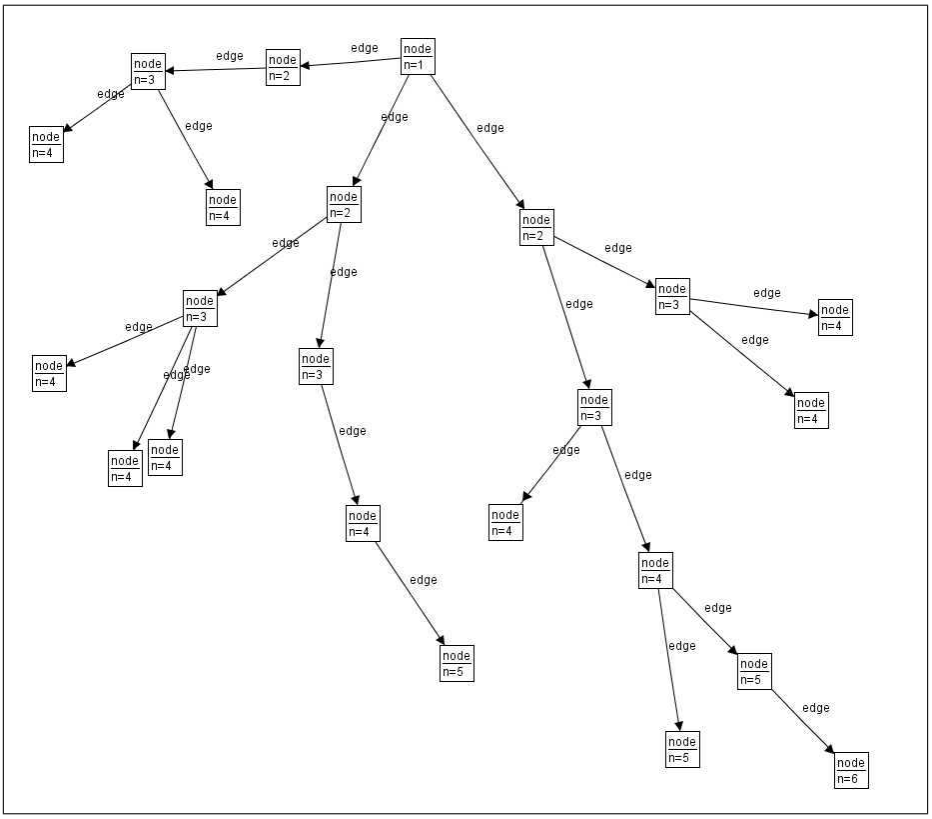

Abbildung 6.19: Baumstruktur in Generation 22 unter Beachtung von Pattern

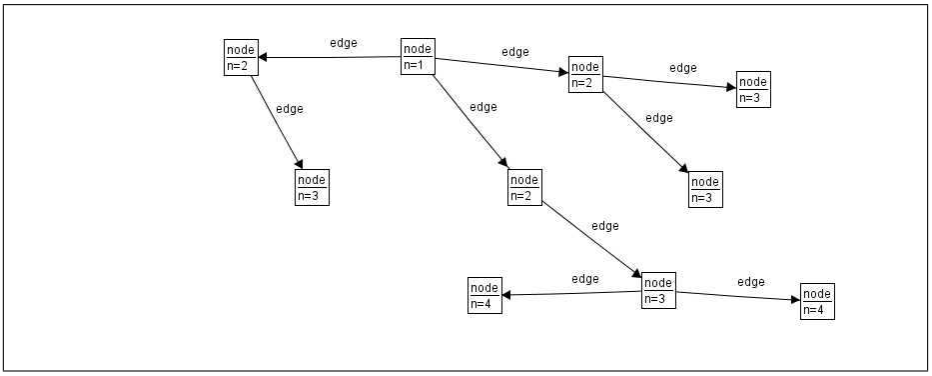

Abbildung 6.20: Baumstruktur in Generation 9 unter Beachtung von Pattern

Beispiel: CD2DB

Als drittes Beispiel wurde die etwas komplexere Grammatik CD2DB [mti05] gewählt. Diese Grammatik wandelt eine Klassendiagrammansicht in eine Datenbankstruktur um. Das Klassendiagramm stellt eine Datenstruktur dar. Soll diese Datenstruktur in einer Datenbank gespeichert werden, so muss festgelegt werden, wie die dafür nötigen Tabellen aufgebaut sein müssen. Die CD2DB–Grammatik erzeugt eine solche Tabellenstruktur mit den entsprechenden Spalten und Relationen zwischen den Tabellen.

Abbildung 6.21 zeigt ein Layout der Klassendiagrammrepräsentation. In den Abbildungen 6.22 und 6.23 sind die ersten Umwandlungsschritte in eine Datenbankstruktur illustriert. Dabei wird einzelnen Klassen jeweils eine Tabelle zugeordnet. In den Layouts sind keine Knotenüberlappungen zu finden. Auch Kantenkreuzungen und Überschneidungen von Knoten und Kanten kommen nicht vor. Die Qualität der Layouts (nach diesen Metriken gemessen) ist also hoch. Auch die Differenz zwischen den einzelnen Layouts ist sehr gering. Dies zeigt sich darin, dass die Positionen der Knoten, die in allen Graphen enthalten sind, nicht geändert werden. Die Änderungen der *epsilon*–Cluster sind auf die neu hinzugekommenen Knoten (Table und C) zurückzuführen.

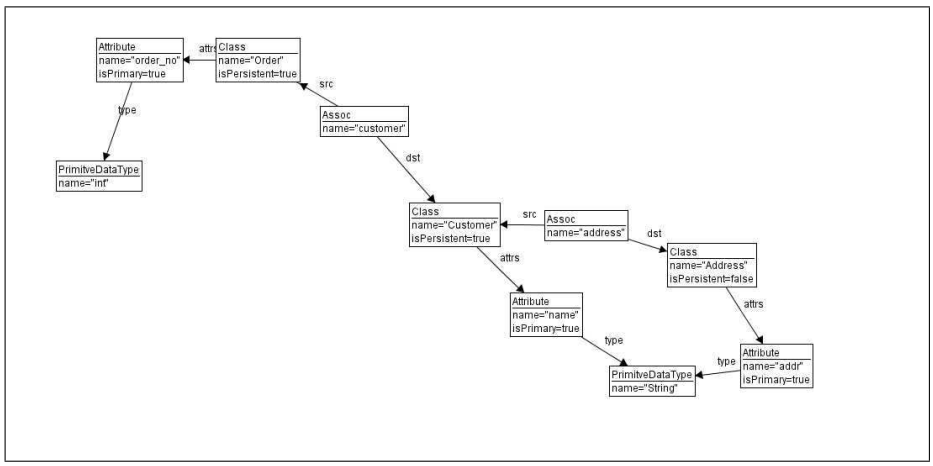

Abbildung 6.21: Layout des Initialgraphen der Grammatik CD2DB

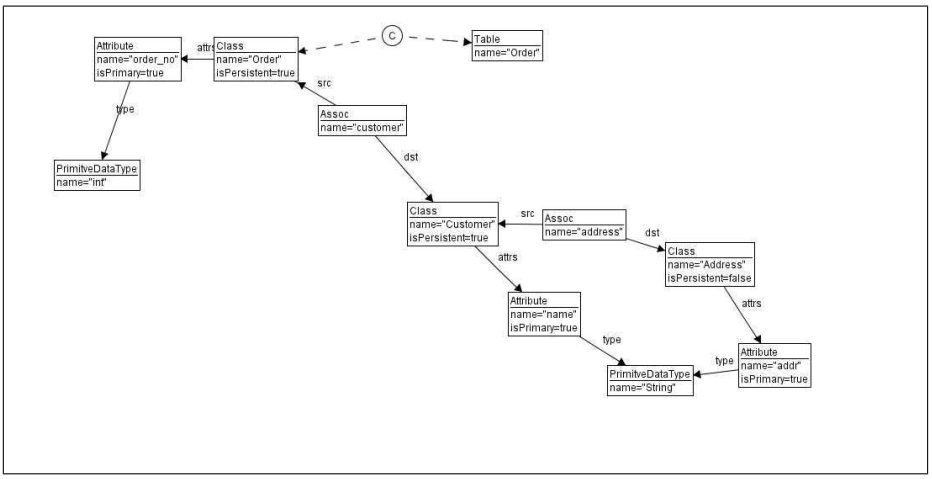

Abbildung 6.22: Layout des Graphen in Generation 1 der CD2DB–Grammatik

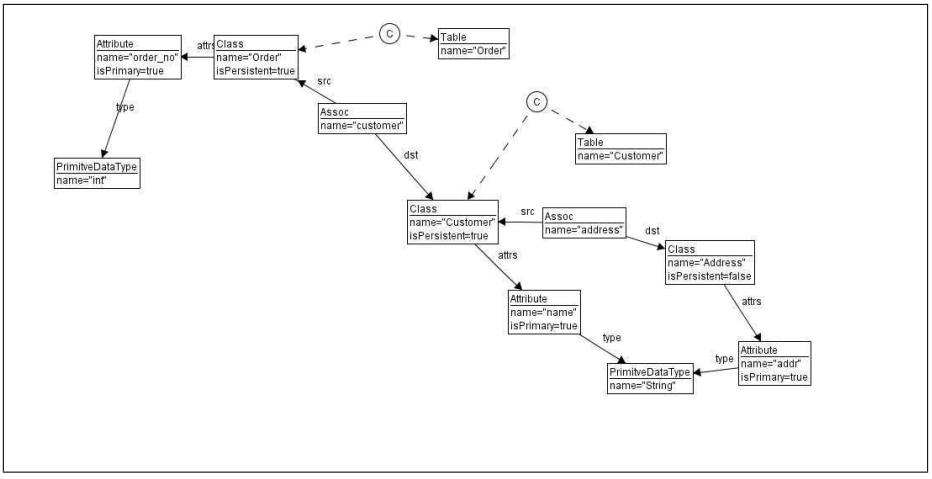

Abbildung 6.23: Layout des Graphen in Generation 2 der CD2DB–Grammatik

Die letzen zwei Graphen der von dieser Grammatik erzeugten Sequenz sind in den Abbildungen 6.24 und 6.25 dargestellt. Der letzte Transformationsschritt entfernt den letzen zum Klassendiagramm gehörenden Knoten (ein ATTRIBUTE). Auch in diesen Layouts treten weder Knotenüberlappungen noch Überschneidungen von Knoten und Kanten auf. Allerdings ist eine Kantenkreuzung feststellbar. Dadurch wird die Qualität der Layouts beeinträchtigt, kann aber weiterhin als gut bezeichnet werden. Die Differenz zwischen diesen Layouts ist ebenfalls gering. Die Positionen der einzelnen Knoten ändern sich zwar nicht unerheblich, aber relativ zu den anderen Knoten bleiben die Positionen eher stabil. Dies wird durch die geringe Änderung der Cluster gezeigt. Die einzelnen Positionsänderungen resultieren vor allem daraus, dass der Graph als ganzes verschoben wird. Der Grund hierfür ist das Löschen des ATTRIBUTE–Knoten, der durch seine sehr stabile Position und die abstoßenden Kräfte den anderen Knoten gegenüber eine Art Schranke darstellte, die der Graph nicht überschreiten konnte.

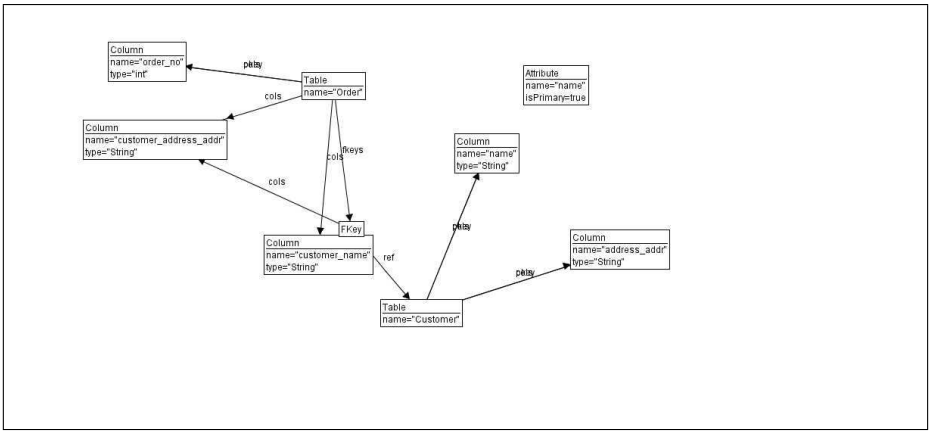

Abbildung 6.24: Layout des Graphen in Generation 25 der CD2DB–Grammatik

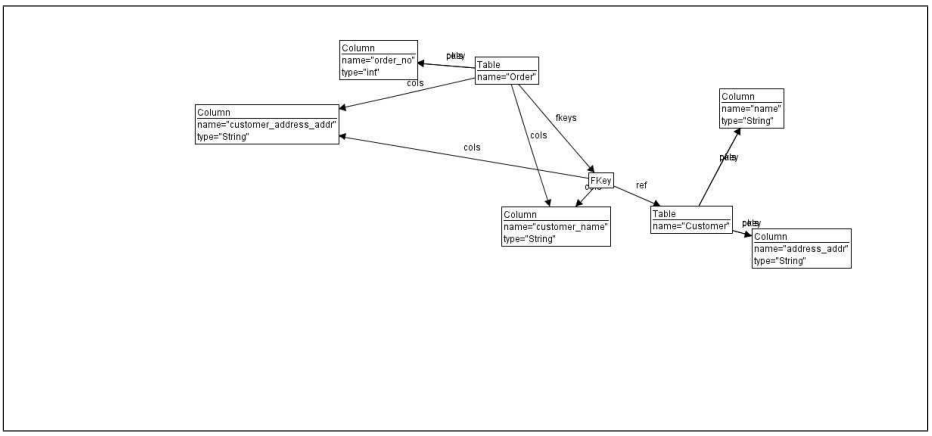

Abbildung 6.25: Layout des abschließenden Graphen (26) der CD2DB–Grammatik

7 Diskussion

Im vorhergehenden Kapitel 6 wurden die praktischen Ergebnisse des hier entwickelten Lösungsansatzes sowohl in der Softwaretechnik als auch in der theoretischen Informatik im Bereich der Graphtransformation vorgestellt. In diesem Kapitel soll diskutiert werden, inwieweit die Ergebnisse in Bezug auf die Anforderungen aus Kapitel 2 zufriedenstellend sind. Weiterhin wird erläutert, ob die Ergebnisse eine Verbesserung zu bestehenden Techniken darstellen. Dies sind hinsichtlich der Softwaretechnik das Layout unvollständiger Sequenzen von Klassendiagrammen (am Beispiel Omondo). Im Bereich der Graphtransformation werden die hier erstellten Layouts an dem vorhandenen Layout von AGG gemessen.

7.1 Erfüllung der grundlegenden Anforderungen

In der Fallstudie wurde bereits an verschiedenen Beispielen aus dem Bereich der Softwaretechnik (Layout von Klassendiagrammsequenzen) und dem Bereich der theoretischen Informatik (Layout von Graphtransformationen) gezeigt, dass der in dieser Arbeit entwickelte Layoutalgorithmus die in Kapitel 2 definierten Anforderungen erfüllt. Diese Anforderungen waren:

1. Jedes einzelne Layout L_i ist nach objektiven Layoutqualitätsmaßstäben möglichst optimal.

2. Die Differenz zwischen zwei aufeinander folgenden Layouts L_i und L_{i+1} soll möglichst gering sein. Dadurch kann die Mental Map des Entwicklers das Klassenmodell betreffend über die gesamte Sequenz hinweg größtenteils erhalten bleiben. Konkret heißt das, dass Klassen, die in mehreren Klassenmodellen K_i der Sequenz enthalten sind, nach Möglichkeit in jedem zugehörigen Layout L_i an der gleichen Position dargestellt werden sollten.

3. Zur Berechnung einer optimalen Layoutsequenz muss die Sequenz der Graphen (beispielsweise Klassendiagramme) nicht vollständig vorliegen.

4. Jedes einzelne Layout sollte auf die Erwartungen des Betrachters in Bezug auf das Aussehen eines Graphen mit bestimmter Semantik zugeschnitten sein (konkret auf die Erwartungen, die an das Layout eines Klassendiagramms gestellt werden). Dadurch wird das Erkennen bestimmter Strukturen im Graphen erleichtert und eine Mental Map wird schneller aufgebaut.

Beispiele für die Qualität der einzelnen Layouts sind unter anderem in den Abbildungen 6.16, 6.20 oder 6.19 dargestellt. Die Qualität der einzelnen Layouts wurde dabei

mit den Qualitätsmetriken gemessen, die in Abschnitt 4.3.3 vorgestellt wurden. Auch die zweite Anforderung konnte erfüllt werden, wie beispielsweise die Abbildungen 6.18 und 6.20 veranschaulichen, welche die Layouts zweier aufeinander foldender Graphen einer Sequenz zeigen. Dies wird durch die Ergebnisse der Differenzmetriken zur Positionsänderung (Abstandsmetrik) und zur Clustererhaltung gestützt. Die Erfüllung der dritten Anforderung ergibt sich direkt aus der Nutzung von Graphtransformationen zur Erzeugung von Graphsequenzen, da die Auswahl und Reihenfolge der jeweils genutzten Transformationsregel im allgemeinen nichtdeterministisch erfolgt (siehe Abschnitt 3.3). Die in der vierten Anforderung genannten Erwartungen des Betrachters an das Aussehen bestimmter Graphen kann durch eine geeignete Kombination der in dieser Arbeit vorgestellten Layout Pattern erreicht werden. So kann für die in Klassendiagrammen auftretenden Vererbungsstrukturen durch ein Kantenpattern eine hierarchische Darstellung erreicht werden. Auch Baumstrukturen in Graphen mit anderer Semantik können so hierarchisch dargestellt werden (siehe Abbildung 6.19). Die Einhaltung solcher Layout Pattern kann durch die in Abschnitt 4.3.4 entwicklete Patternmetrik überprüft werden. Mit weiteren Layoutpattern lassen sich auch komplexere Teilstrukturen in besonderer Weise layouten.

7.2 Layout von Klassendiagrammsequenzen

Datengrundlage für das evolutionäre Layout waren von Omondo durch Reverse–Engineering erstellte Klassenmodelle. Omondo ist in der Lage diese Klassenmodelle zu layouten. Abbildung 7.1 zeigt eine von Omondo erstellte Überblicksansicht des Pakets PMD.CPD.CPPAST (in Release 2.0), wobei der grau unterlegte Bereich die Größe des Darstellungsbereichs der Detailansicht in der höchsten Zoomstufe anzeigt. In der Überblicksansicht sind weder die Klassennamen noch die Beschriftung der Beziehungen (Kanten) lesbar. In der Detailansicht erweist sich die Rückverfolgung von Assoziationen als schwierig, da die Kanten aufgrund der weit voneinander entfernt positionierten Klassen sehr lang ausfielen. Das mit dem hier entwickelten Algorithmus erzeugte Layout des selben Pakets in Abbildung 6.1 bietet mehr Übersicht. Dabei gehen allerdings einige Detailinformationen wie Methodennamen und Kantenbeschriftungen verloren. Dadurch dass das Layout mit lesbaren Klassennamen auf eine Bildschirmseite passt, erhält man aber einen besseren Überblick über die Struktur des Pakets als mit den Omondo–Ansichten.

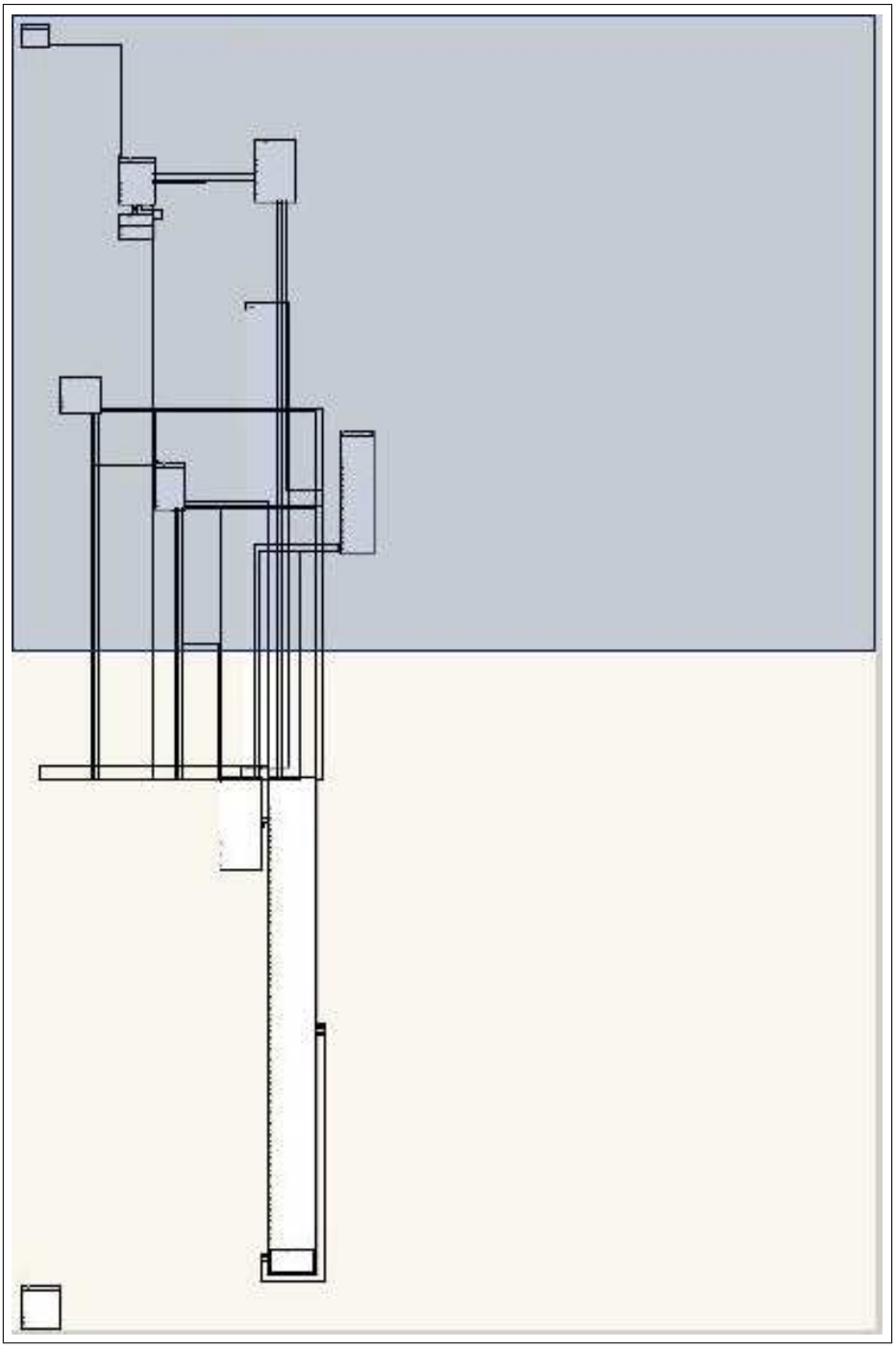

Abbildung 7.1: Überblick–Ansicht des Klassendiagramm des Pakets PMD.CPD.CPPAST in Release 2.0 in Omondo (vgl. Abbildung 6.1)

Wird nun die Entwicklung des Layouts des Klassendiagramms des Pakets PMD.CPD.CPPAST beim Übergang von Release 2.0 zu 2.1 und 2.2 betrachtet, so kann festegestellt werden, dass die Omondo–Überblicksansichten starke Veränderungen aufweisen (Release 2.1 in Abbildung 7.2 und Release 2.2 in Abbildung 7.3). Die tatsächlich im Klassendiagramm vorgenommenen Änderungen sind aber weit weniger umfangreich, als die Layoutänderung vermuten lässt. Es erfolgten nur Änderungen bezüglich der Beziehungen zwischen den Klassen, es wurden keine Klassen gelöscht oder neu hinzugefügt. Das evolutionäre Layout dieser drei Releases des Pakets bleibt wesentlich stabiler. Beim Sprung von Release 2.0 (Abbildung 6.1) zu Release 2.1 (Abbildung 6.2) ändert sich die Position von nur einer Klasse. Dies resultiert aus dem Versuch, ein Layout Pattern zu erfüllen, das an eine neu hinzugefügte Vererbungsbeziehung geknüpft war. Bei dem Übergang von Release 2.1 zu 2.2 bleiben die Positionen aller Klassen stabil (siehe Abbildung 6.3). Dies ist auch in allen weiteren Releases bis hin zum aktuellen Release 3.4 der Fall. Die aufeinander folgendenden Layouts haben also einen hohen Wiedererkennungswert. Dadurch kann die Mental Map der Entwickler bezüglich des Klassendiagramms weitgehend erhalten bleiben.

Als Beispiel für ein großes Klassendiagramm wurde in der Fallstudie das Paket PMD herangezogen. Wie schon erwähnt, konnte für dieses Paket keine Graphsequenz erstellt werden, da die Änderungen an diesem Paket und die daraus resultierenden Transformationsregeln so umfangreich waren, dass sie von AGG nicht ausgeführt werden konnten. Aus diesem Grund findet nur ein Vergleich der Einzellayouts des hier entwickelten Layoutalgorithmus mit denen von Omondo statt. In Abbildung 7.5 ist die Überblicksansicht des Klassendiagramms des Pakets PMD in Release 0.1 in Omondo dargestellt. Der grau unterlegte Bereich stellt die Größe und Position des maximalen Darstellungsbereichs in der Detailansicht dar. Dieser Bereich ist in Abbildung 7.4 zu sehen. Beide Ansichten sind unübersichtlich. Es sind weder Klassennamen, Methodennamen noch die Beschriftung der Kanten lesbar. Um eine lesbare Darstellung zu erhalten muss die Ansicht vergrößert werden. Wie auch schon im obigen Beispiel ist die Rückverfolgung der Kanten sehr schwer. Das in dieser Arbeit erstellte Layout dieses Klassendiagramms ist aufgrund seiner Größe ebenfalls nicht sehr übersichtlich (siehe Abbildung 6.7). Allerdings sind die Klassennamen lesbar und die Darstellung passt auf eine Bildschirmseite. Das Layout vermittelt einen Überblick über das Klassendiagramm. Es ist erkennbar, welche Klassen enthalten sind und welche Klassen zueinander in Beziehung stehen. In dieser Darstellung sind zwar keine Methodeninformationen enthalten, allerdings sind diese in der Überblicksansicht sowie in der verkleinerten Detailansicht in Omondo ebenfalls nicht erkennbar.

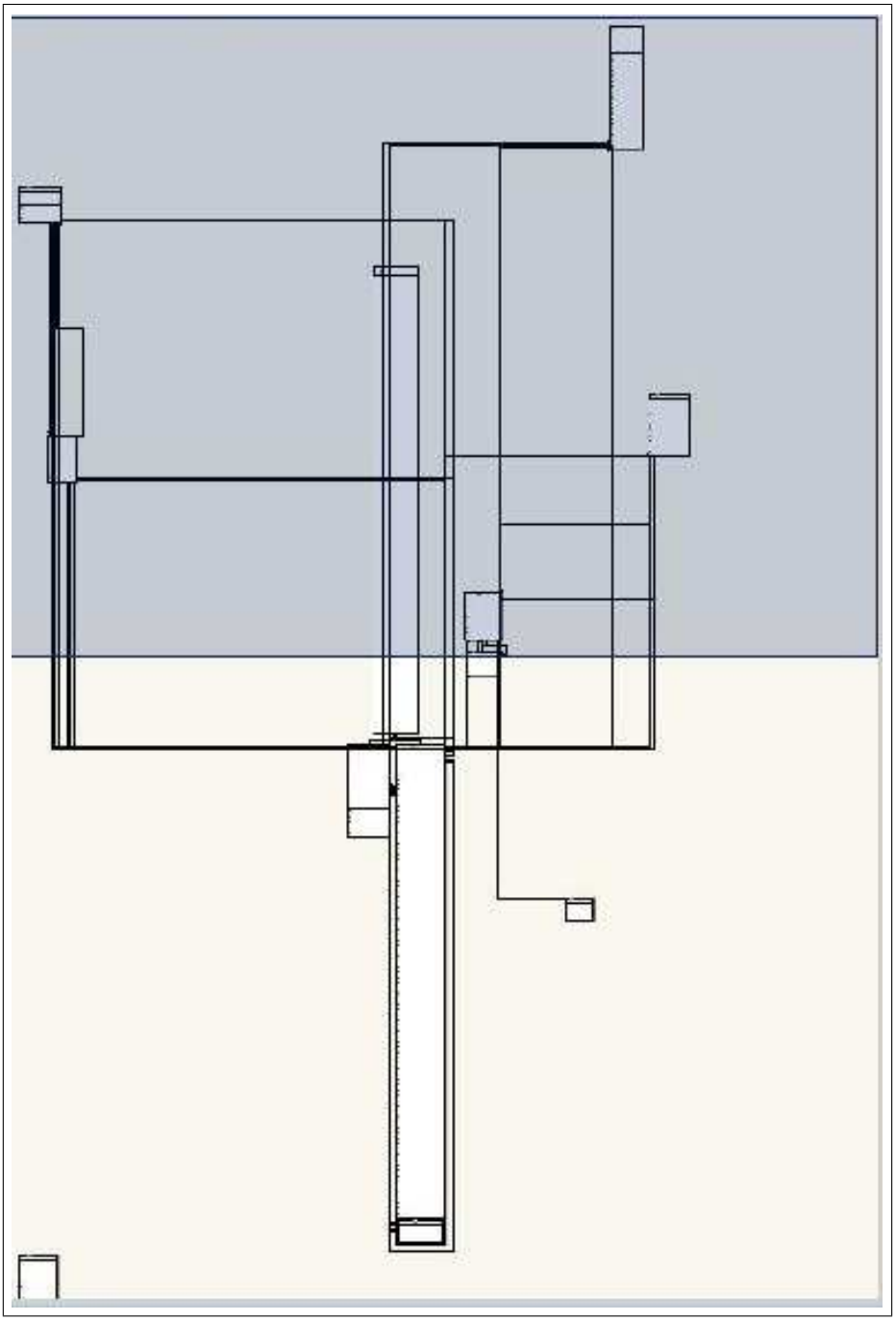

Abbildung 7.2: Überblick–Ansicht des Klassendiagramm des Pakets PMD.CPD.CPPAST in Release 2.1 in Omondo (vgl. Abbildung 6.2)

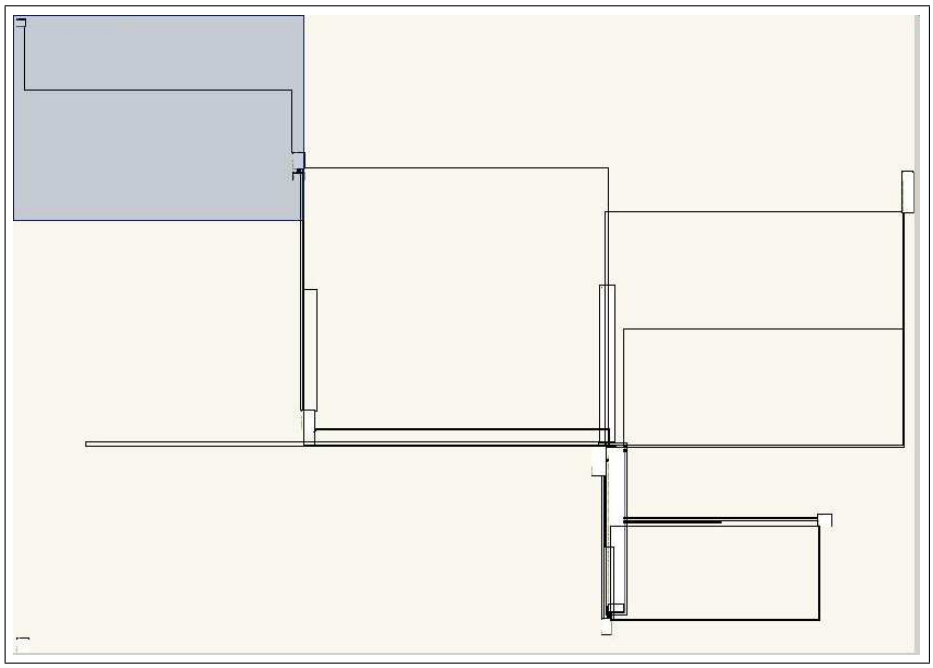

Abbildung 7.3: Überblick–Ansicht des Klassendiagramm des Pakets PMD.CPD.CPPAST in Release 2.2 in Omondo (vgl. Abbildung 6.3)

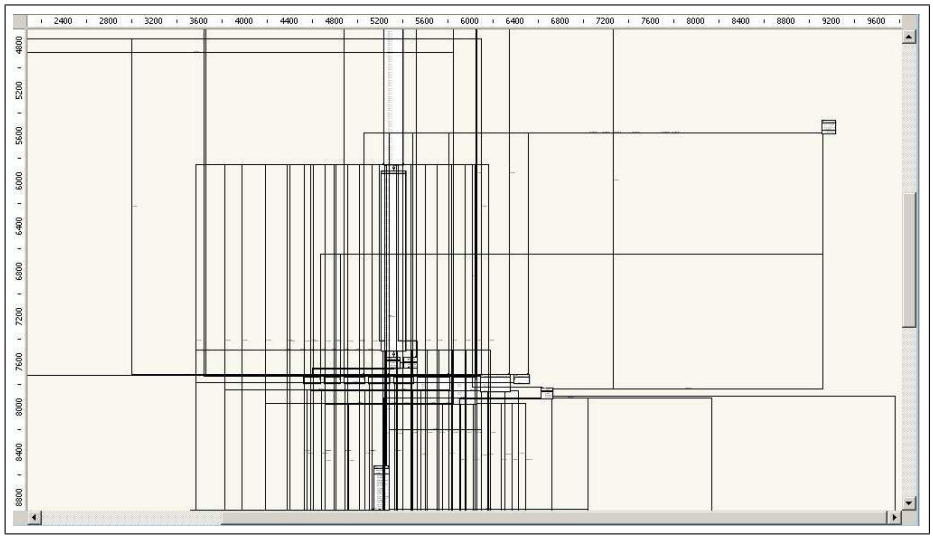

Abbildung 7.4: Detail–Ansicht des Klassendiagramm des Pakets PMD in Release 0.1 in Omondo in der höchsten Zoomstufe

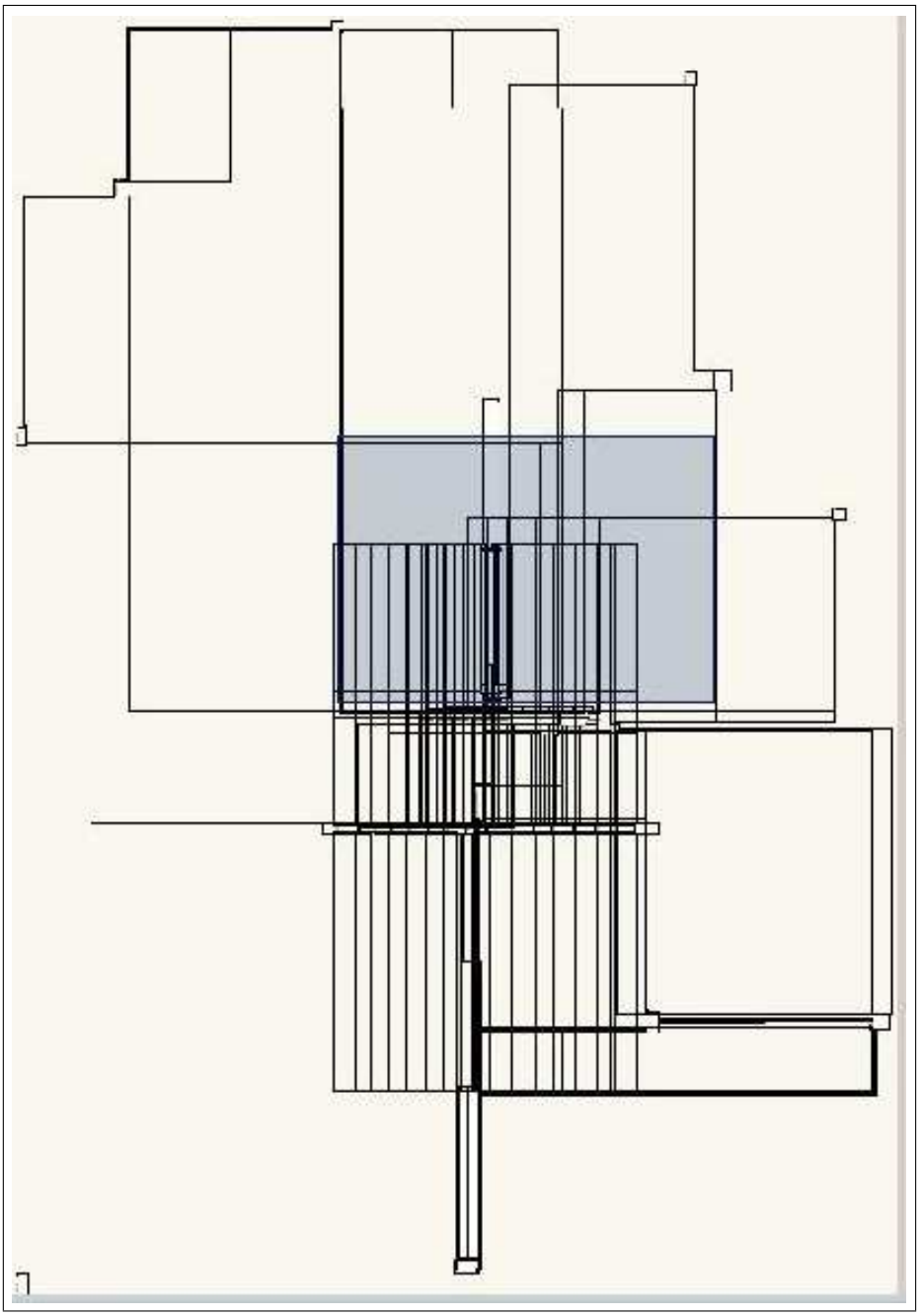

Abbildung 7.5: Überblick–Ansicht des Klassendiagramms des Pakets PMD in Release 0.1 in Omondo (vgl. Abbildung 6.7)

7.3 Layout von allgemeinen Graphsequenzen

In der Fallstudie (Kapitel 6) wurden auch die Ergebnisse des in dieser Arbeit entwickelten evolutionären Layoutalgorithmus beim Layout von Graphtransformationen erläutert. Zum Vergleich dieser Ergebnisse mit den Layouts, die AGG standardmäßig erzeugt, werden zwei der schon in der Fallstudie verwendeten Beispiele herangezogen.

Am Beispiel der Baumstruktur können die ungenügenden Ergebnisse des Standard–AGG–Layouts besonders gut gezeigt werden. Abbildung 7.6 zeigt das Layout einer Baumstruktur in der achten Generation in AGG. Es treten zwar noch keine Knotenüberlappungen auf, aber einige Knoten sind sehr dicht nebeneinander angeordnet. Dadurch kommt es zu vielen Überschneidungen von Knoten und Kanten. Einige Kanten sind vollkommen von Knoten überdeckt. Das Layout hat also eine schlechte Qualität. Auch optisch ist die Baumstruktur nicht erkennbar. Das in Abbildung 6.18 gezeigte Layout, das mit dem hier entwickelten Algorithmus unter Beachtung eines Layout Pattern erstellt wurde, weist dagegen eine hohe gemessene Qualität auf und der Graph ist dort als Baumstruktur zu erkennen.

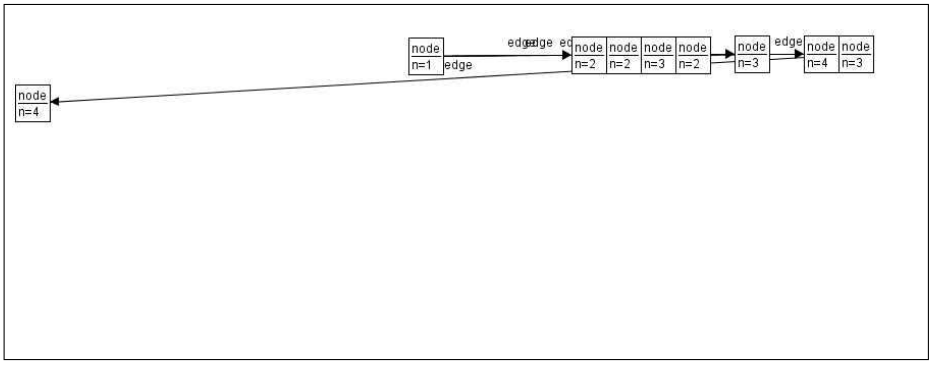

Abbildung 7.6: Baumstruktur in Generation 8 in Standard–AGG–Layout (vgl. Abbildung 6.18)

Bei größeren Graphen wird das Layout von AGG immer unübersichtlicher. In Abbildung 7.7 ist die 22. Generation einer Baumstruktur abgebildet. Es kommt dabei zu mehreren Knotenüberlappungen. Mehrere Knoten sind vollkommen überdeckt und daher nicht mehr sichtbar. Weiterhin treten vermehrt Überschneidungen von Knoten und Kanten sowie komplett überdeckte Kanten auf. Die Baumstruktur ist auch in diesem Layout nicht als solche erkennbar. Der hier entwickelte Layoutalgorithmus ist dagegen auch in der 22. Generation in der Lage, ein Layout zu erzeugen, das eine hohe gemessene Qualität aufweist (siehe Abbildung 6.19). Dabei ist auch die Baumstruktur sofort als solche zu erkennen.

Als zweites Beispiel wurde die CD2DB–Grammatik zum Vergleich herangezogen. Abbildung 7.8 zeigt ein manuell erzeugtes Layout der Klassendiagrammrepräsentation, die den Ausgangsgraphen dieser Grammatik darstellt. In den Abbildungen 7.9 und 7.10 sind die Layouts der Graphen nach dem ersten und dem zweiten Transformationsschritt

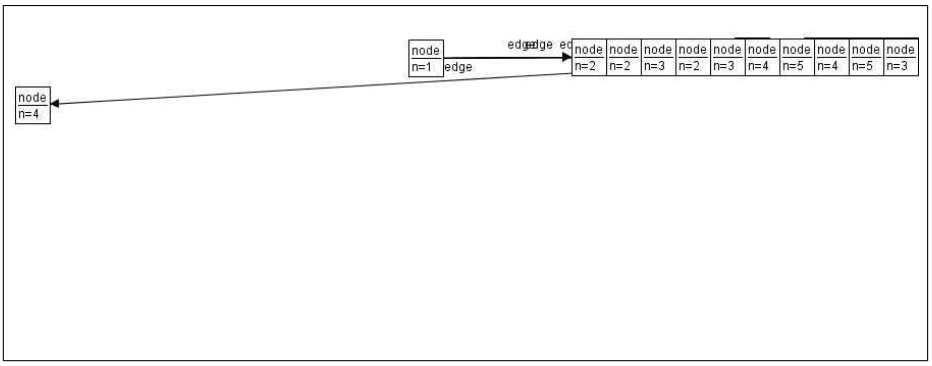

Abbildung 7.7: Baumstruktur in Generation 22 in Standard–AGG–Layout (vgl. Abbildung 6.19)

der Grammatik zu sehen. Die entsprechenden evolutionär erstellten Layouts sind in den Abbildungen 6.21, 6.22 und 6.23 dargestellt. Bei beiden Layoutvarianten bleiben die Differenzen zwischen den aufeinander folgenden Layouts gering. Die Positionen vorhandener Knoten werden nicht geändert, Clusterveränderungen entstehen nur durch hinzugefügte Knoten. Diese geringe Differenz in der AGG Layoutvariante ist dadurch zu erklären, dass Knoten bei ihrer Erzeugung eine Position zugewiesen bekommen, die dann nicht mehr geändert wird. Die Positionen der neu hinzugekommenen Knoten wird aber so berechnet, dass es zu Knotenüberlappungen und auch zu Überschneidungen von Knoten und Kanten kommt. Daraus resultiert eine schlechte Layoutqualität. Im Gegensatz dazu treten in den evolutionär erzeugten Layouts keine Verletzungen dieser Qualitätsmetriken auf.

Auch das von AGG erstellte Layout des abschließenden Graphen der Transformationssequenz weist mehrere Verletzungen der Qualitätsmetriken auf (siehe Abbildung 7.11). Es treten mehrere Überschneidungen von Knoten und Kanten sowie Kantenkreuzungen auf. Das entsprechende Layout des evolutionären Layouters weist hingegen nur eine Kantenkreuzung auf. Es ist also an den hier verwendeten Metriken gemessen qualitativ besser als das AGG–Layout.

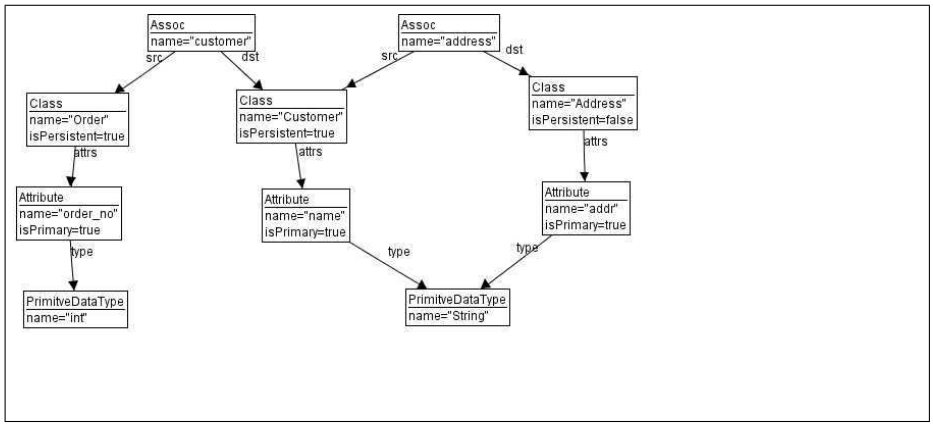

Abbildung 7.8: Initialgraph der CD2DB–Grammatik mit manuellem Layout (vgl. Abbildung 6.21)

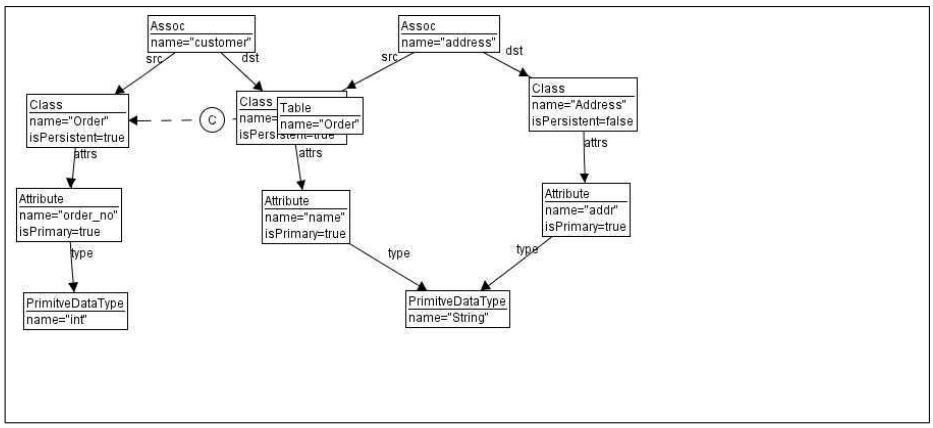

Abbildung 7.9: Generation 1 der CD2DB–Grammatik mit Standard–AGG–Layout (vgl. Abbildung 6.22)

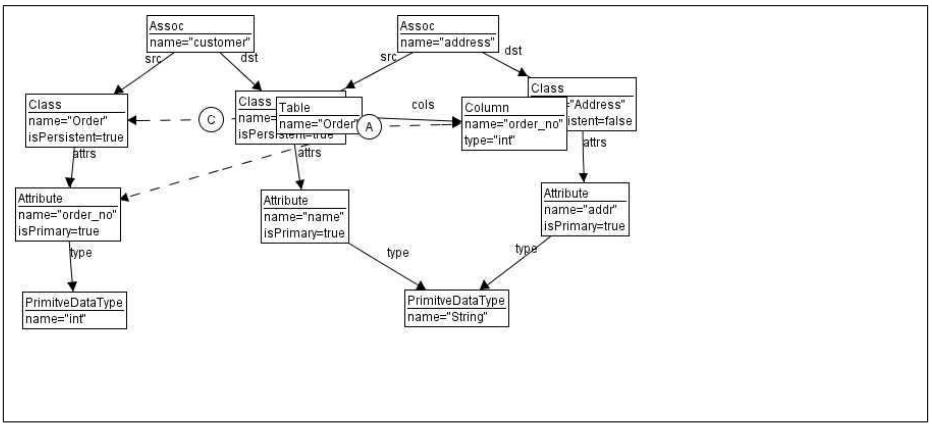

Abbildung 7.10: Generation 2 der CD2DB–Grammatik mit Standard–AGG–Layout (vgl. Abbildung 6.23)

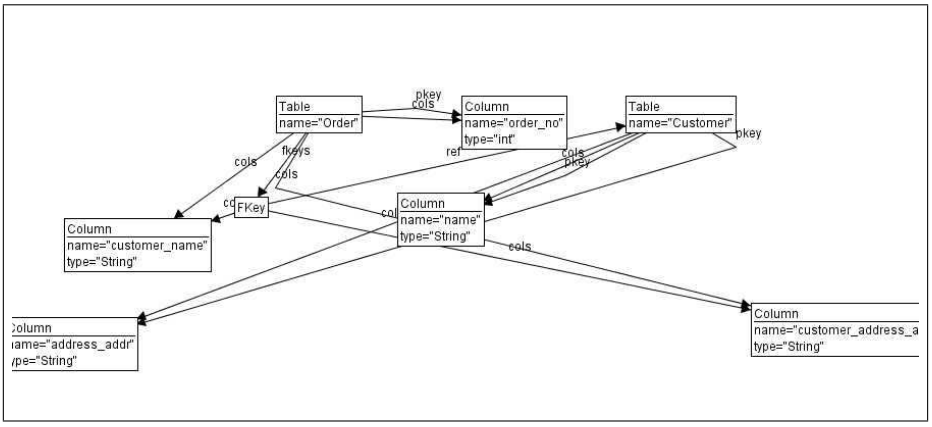

Abbildung 7.11: Generation 26 der CD2DB–Grammatik mit Standard–AGG–Layout (vgl. Abbildung 6.25)

8 Zusammenfassung

Das in dieser Diplomarbeit bearbeitete Problem des Layouts unvollständiger Graphsequenzen ist sowohl für die Softwaretechnik als auch für die theoretische Informatik relevant. Daher flossen Anforderungen aus beiden Gebieten in diese Arbeit ein. Der entwickelte Lösungsansatz — ein evolutionäres Layout — führt in beiden Gebieten zu zufriedenstellenden Ergebnissen, welche die bisherigen Ergebnisse übertreffen.

Im Bereich der Softwaretechnik wurden beispielhaft die Layoutergebnisse des Tools Omondo Eclipse UML Studio für einzelne Klassendiagramme herangezogen. Eine Sequenz von Klassendiagrammen kann von Omondo nur als eine Sequenz von unabhängigen Einzellayouts dargestellt werden, was zu Problemen führt, die in Abschnitt 7.2 gezeigt werden. Die bisherigen Ergebnisse im Bereich der theoretischen Informatik wurden durch das Layout von Graphtransformationen durch das Tool AGG repräsentiert.

In beiden Fällen ist eine Sequenz von Graphen zu layouten, wobei die zukünftigen Erweiterungen der Sequenz nicht bekannt sind. Das Layout muss die in der Problemdefinition (Abschnitt 2.2) beschriebenen Anforderungen des erweiterten Offline Graph Drawing Problems erfüllen. Diese Anforderungen beinhalten die Anforderungen des Offline Graph Drawing Problems, also sowohl möglichst optimale Einzellayouts als auch eine möglichst geringe Differenz zwischen zwei aufeinander folgenden Layouts. Die Erweiterungen bestehen aus zwei hinzugekommenen Anforderungen: Erstens ist die Graphsequenz nicht vollständig bekannt. Zweitens sollen die Erwartungen erfüllt werden, die ein Betrachter an das Aussehen eines Graphen mit spezieller Semantik (beispielsweise an ein Klassendiagramm) haben kann.

Die Anforderungen des Offline Graph Drawing Problems können durch schon bestehende Layoutalgorithmen gelöst werden. Diese Algorithmen sind aber auf die Erweiterungen des Problems nicht mehr anwendbar, da nicht alle Informationen über die zukünftige Entwicklung der Graphsequenz vorliegen. Der hier entwickelte Algorithmus (evolutionäres Layout) begreift eine Sequenz von Graphen als Graphtransformation, wobei eine Graphsequenz mittels eines Regelsatzes aus einem Initialgraphen erstellt wird. Die weitere Entwicklung der Graphsequenz ist dabei im Allgemeinen nicht bekannt. Für jeden so erzeugten Graph wird mit einem adaptierten Spring Embedder Algorithmus ein Layout erzeugt. Dazu wird auf das Layout seines Vorgängers zurückgegriffen. Dadurch wird die Differenz zwischen zwei aufeinanderfolgenden Layouts klein gehalten. Um die Erwartungen des Betrachters an das Layout von speziellen Graphen (beispielsweise Klassendiagrammen) erfüllen zu können wurden Layout Pattern entwickelt. Diese Layout Pattern legen fest, wie einzelne Graphteile dargestellt werden sollen.

Dieses evolutionäre Layout wurde in das Graphtransformationstool AGG integriert und nach der Implementierung praktisch getestet. Dazu wurden zum einen Sequenzen von Klassendiagrammen in Graphgrammatiken überführt und mit dem evolutionären

Layout dargestellt. Zum anderen wurden sowohl einfache als auch komplexere Graphtransformationen in AGG mit diesem Layoutalgorithmus visualisiert. Anschließend wurden die erzeugten Layouts mit den bisherigen Ergebnissen verglichen, die von Omondo und AGG geliefert werden.

Zusammenfassend kann man sagen, dass das Ziel dieser Diplomarbeit mit der Entwicklung des evolutionären Graph Layout Algorithmus erreicht wurde. Dieser Algorithmus schafft einen Kompromiss zwischen guten Einzellayouts (bis zu einer gewissen Knoten– und Kantenzahl im Graphen) und geringer Differenz zwischen aufeinander folgenden Layouts. Die optischen Strukturen im Layout von Klassendiagrammsequenzen bleiben wesentlich stabiler als in den von Omondo erzeugten Sequenzen von Einzellayouts. Die einzelnen Klassendiagramme werden oft auch übersichtlicher dargestellt. Dies kann das Verständnis der Klassenmodelle und des dahinter stehenden Sourcecodes sowie die Kommunikation zwischen Entwicklern erheblich erleichtern. Auch bei anderen Arten von Graphen erreicht der Algorithmus gute Ergebnisse und stellt eine Verbesserung zu den bisherigen Lösungen zum Layout von Graphtransformationen dar.

9 Ausblick

Im Rahmen dieser Diplomarbeit wurde ein Algorithmus zum Layout von unvollständigen Graphsequenzen entwickelt. Dieser Algorithmus wurde in einer exemplarischen Implementation in das Tool AGG integriert. In diesem Kapitel werden sowohl Erweiterungsmöglichkeiten des Algorithmus als auch während der Entwicklung neu aufgetretene Problemstellungen aufgezeigt, deren Lösung den Rahmen einer Diplomarbeit überstiegen hätte. Weiterhin werden hier einige Punkte beschrieben, die in der Implementation nicht enthalten sind, aber die Nutzerfreundlichkeit und Konfigurierbarkeit des Layoutalgorithmus verbessern.

9.1 Neu hinzugekommene Problemstellungen

Während der Tests des Layoutalgorithmus kam es in einigen Fällen im Verlauf einer Graphsequenz zu größeren Positionsänderungen einzelner Knoten oder ganzer Gruppen von Knoten. Ein solcher Fall tritt im Layout des letzten Transformationsschritts der CD2DB–Grammatik auf (siehe Abbildungen 6.24 und 6.25). In diesem Beispiel kann der gelöschte Knoten (und der damit verbundene Wegfall von abstossenden Kräften) als Ursache für das Verschieben des gesamten Graphen identifiziert werden. Solche Positionsänderungen sind aber unter Umständen schwer nachvollziehbar, da viele Faktoren dafür verantwortlich sein können. Zu diesen Faktoren gehören die in Abschnitt 4.2.1 entwickelten Layout Pattern. Aber auch das Hinzufügen oder Löschen von Knoten und Kanten kann zu umfangreichen Layoutänderungen führen. Oft kann die tatsächliche Ursache nicht sofort eindeutig identifiziert werden. Es bleibt zu untersuchen, wie die Ursachen einzelner Positionsänderungen von Knoten in aufeinander folgenden Layouts eindeutig ermittelt werden können.

Ein weiteres Problem, das im Rahmen dieser Diplomarbeit nicht gelöst werden konnte, besteht in der Erkennung von Content Pattern. Dieses Problem trat infolge der Entwicklung der Layout Pattern auf. Layout Pattern sollen spezielle inhaltliche Strukturen von Graphen im Layout ersichtlich machen. Für solche Strukturen wurde der Begriff Content Pattern eingeführt. Beispiele für Content Pattern lassen sich in vielen Arten von Graphen finden. Eine weit verbreitete Variante sind Baumstrukturen in einem Graphen (z.B. Vererbung in Klassendiagrammen). Spezielle Content Pattern für Klassendiagramme sind durch die Design Pattern (siehe Abschnitt 4.2.2) gegeben. Bevor Layout Pattern für solche Content Pattern erstellt werden können, müssen diese zuerst in dem vorliegenden Graphen identifiziert werden. Es stellt sich die Frage, ob das Erkennen von Content Pattern automatisch erfolgen kann und welche zusätzlichen Informationen dazu benötigt werden.

9.2 Konzeptionelle Erweiterungen

Der in dieser Arbeit entwickelte evolutionäre Layoutalgorithmus kann an verschiedenen Stellen erweitert oder modifiziert werden. Eine mögliche Erweiterung besteht in der Integration einer verbesserten Randbehandlung, also einem Verfahren, das die Positionsänderungen von Knoten am Rand der Anzeigefläche beeinflusst. Die Randbehandlung erfolgt bisher auf eine sehr einfache Art. Das Verschieben eines Knotens über den Rand des Anzeigebereichs hinaus wird durch eine zusätzliche Beschränkung der Bewegungsfreiheit verhindert. Der Knoten wird dann nur noch am Rand entlang verschoben. Eine Möglichkeit einer anderen Randbehandlung besteht darin, die Knoten an den Rändern abprallen zu lassen. Ein solches Verfahren wird schon von Fruchterman und Reingold [FR91] vorgestellt.

Zusätzlich zu einer erweiterten Randbehandlung könnte ein Mechanismus zur Vermeidung von Platzproblemen am linken und am oberen Rand des Darstellungsbereichs (siehe Abschnit 6.2 und Abbildung 6.17) in den Algorithmus integriert werden. Ein Ansatz hierzu besteht in dem Versuch, den gesamten Graphen auf dem Anzeigebereich zu zentrieren. Dazu muss der verfügbare Anzeigebereich unter Umständen vergößert werden.

Eine komplexere Erweiterung stellt der Versuch dar, die Klassendiagramme einzelner Pakete eines Projekts zu einem großen Diagramm zu kombinieren und ein Layout für dieses Diagramm zu erstellen. Eine Möglichkeit dazu ist ein Paketabhängigkeitsgraph, wie er in der Diplomarbeit von Jacek Bochnia vorgestellt wird [JJB06]. Ein Layout für einen solchen Paketabhängigkeitsgraphen kann mit dem hier entwickelten Algorithmus erstellt werden. Die Idee dabei ist, die einzelnen Pakete durch verkleinerte Darstellungen der Layouts ihrer Klassendiagramme zu visualisieren. Der Layoutalgorithmus muss dazu zuächst Layouts für die Klassendiagramme aller Pakete erzeugen, bevor ein Layout für den Paketabhängigkeitsgraphen berechnet wird.

Der in dieser Arbeit entwickelte Layoutalgorithmus basiert auf einem Spring Embedder Energiemodell, das zum Teil von dem Energiemodell von Fruchterman und Reingold adaptiert und zum Teil selbst entwickelt wurde. Es wäre auch denkbar, ein anderes Energiemodell zu verwenden. Solche Energiemodelle sind von Eades [Ead84] sowie von Kamada und Kawai [KK89] entwickelt worden. Ein Vergleich der hier erzielten Layouts mit solchen, die durch ein geändertes Energiemodell berechnet werden, könnte aufschlussreich sein. Ein solcher Vergleich übersteigt allerdings den dieser Diplomarbeit, da er eine Implementation von verschiedenen Energiemodellen voraussetzt.

9.3 Implementationserweiterungen

Die Implementation des in dieser Arbeit entwickelten Layoutalgorithmus erfolgte exemplarisch im Rahmen von AGG. Dabei wurde weniger Wert auf Nutzerfreundlichkeit und leichte Konfigurierbarkeit des Algorithmus gelegt. Im praktischen Einsatz sind diese Faktoren jedoch sehr wichtig. Um die Konfigurierbarkeit zu erhöhen sollten für alle wichtigen Optionen, welche das Layout beeinflussen, Konfigurationsdialoge implementiert werden.

Die wichtigsten Optionen, die der Nutzer individuell ändern können sollte, betreffen die Nutzung und Definition von Layout Pattern sowie die bevorzugte Länge von Kanten. Es sollte also eine Möglichkeit geschaffen werden, einzelnen Kanten (oder Kantentypen) eine bevorzugte Länge zuzuweisen. Dies kann bei der Definition eines Kantentyps oder dem Erstellen einer Kante erfolgen. Denkbar ist auch der Aufruf eines Dialogs über das Kontextmenü der Kanten um nachträglich Änderungen vornehmen zu können. Die Bearbeitung von eigenen Kantenpattern kann auf die gleiche Weise erfolgen.

Weiterhin sollte der Benutzer die Möglichkeit haben anzugeben, ob eine Ausgabe der einzelnen Graphen einer Transformationssequenz in Bilddateien sowie die Ergebnisse der Layoutmetriken in einer .log–Datei erfolgen soll. Diese Optionen sollten in das Optionsmenü des AGG integriert werden. Hier kann auch festgelegt werden, ob Layout Pattern genutzt werden sollen.

In einigen Fällen kann es auch sinnvoll sein, einzelne spezifische Parameter des Algoritmus anzupassen. Zu diesem Parametern gehören die Iterationszahl (bestimmt in wievielen Schritten die Layoutberechnung erfolgt) und die initiale Bewegungsfreiheit (Anfangstemperatur) der Knoten sowie die Größe des Bereichs um einen Knoten, in dem kein anderer Knoten positioniert werden soll. Auch hierfür sollten Einstellungsmöglichkeiten im Optionsmenü von AGG geschaffen werden.

Eine andere Erweiterungsmöglichkeit besteht bei der Umwandlung von Sequenzen von Klassendiagrammen in Graphgrammatiken. Diese Umwandlung erfolgt momentan mittels eines eigenständigen Kommandozeilenprogramms. Hier bietet sich die Integration als Importfilter in die AGG–Oberfläche an.

9.4 Weitere Veröffentlichungen des Algorithmus

Im Rahmen dieser Diplomarbeit entstanden in Zusammenarbeit mit meiner Betreuerin Susanne Jucknath–John (SWT) und Gabriele Taentzer (TFS) zwei Veröffentlichungen die zum Zeitpunkt der Abgabe meiner Diplomarbeit schon bei Konferenzen eingereicht wurden. Die Entscheidung bezüglich der Annahme steht aber noch aus.

Literaturverzeichnis

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modelling Language User's Guide*. Addison-Wesley, 1999.

[BT98] Stina Bridgeman and Roberto Tamassia. Difference Metrics for Interaktiv Orthogonal Graph Drawing Algorithms. *Lecture Notes in Computer Science*, 1547:57–71, 1998.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, 2005.

[CP96] Michael K. Coleman and D. Stott Parker. Aesthetics-based Graph Layout for Human Consumption. *SoftwarePractice and Experience*, 26(12):1415–1438, 1996.

[DGK00] Stefan Diehl, Carsten Goerg, and Andreas Kerren. Foresighted Graphlayout. 2000. `ftp://cs.uni-sb.de/pub/techreports/FB14/fb14-00-02.ps.gz` [01.10.2005].

[DH96] Ron Davidson and David Harel. Drawing Graphs nicely using Simulated Annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[dom98] Document Object Model (DOM). 1998. `http://www.w3.org/dom/` [15.04.2006].

[ea05] Tom Copeland et. al. Pmd. 2002-2005. `http://pmd.sourceforge.net/` [17.08.2005].

[Ead84] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS monographs. Springer, 2006.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Loewe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1997.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. The agg approach: Language and enviroment. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Application, Languages and Tools, 1999.

[FB03] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Paraglyph Press, third edition, 2003.

[FLM94] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In Roberto Tamassia and Ioannis G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 388–403, Berlin, Germany, 10–12 1994. Springer-Verlag.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience*, 21(11):1129–1164, 1991.

[GB03] E. Gamma and K. Beck. *Contributing to Eclispe*. Addison Wesley, 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison Wesley, 1994.

[Gro05] Open Source Technology Group. Sourceforge – the world's largest development and download repository of open source code and applications. 2005. `http://sourceforge.net/` [17.08.2005].

[Har98] Elliotte R. Harold. *XML Extensible Markup Language*. Hungry Minds, 1998.

[Har02] Elliotte R. Harold. *Processing XML with Java*. Addison-Wesley, 2002.

[Hun06] Jason Hunter. Jdom.org. 2006. `http://www.jdom.org/` [15.04.2006].

[JJB06] Susanne Jucknath-John and Jacek Bochnia. Code Dependencies meet Developer Dependencies. *Proceedings of IASTED International Conference on Software Engineering SE 2006*, 2006.

[KK89] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31:7–15, 1989.

[LN03] C. Lewerentz and A. Noack. Crococosmos – 3dvisualization of large object-oriented programs, 2003.

[Mar02] Joe Marini. *Document Object Model : Processing Structured Documents*. Osborne/McGraw-Hill, 2002.

[MELS95] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.

[mti05] *Model Transformation in Practice*, 2005. Satellite Workshop of MODELS 2005.

[omo06] Omondo. 2006. `http://www.omondo.com/` [15.04.2006].

[PMCC01] Helen C. Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph Drawing Aesthetics and the Comprehension of UML Class Diagrams: An empirical study. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*, Sydney, Australia, 2001. ACS.

[Tea06] The AGG Team. AGG. 1997-2006. `http://tfs.cs.tu-berlin.de/agg/` [20.03.2006].

[xml04] Extensible Markup Language (XML). 2004. `http://www.w3.org/XML/` [15.04.2006].