Daniel Neef

Automatisches Layout für Produktfunktionsdarstellung im CAD-Umfeld

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informatik
Mittweida, 2009

Erstprüfer: Prof. Dr.-Ing. Wilfried Schubert
Zweitprüfer: Dipl.-Math. Frieder Swoboda
Vorgelegte Arbeit wurde verteidigt am: 23. November 2009

Bibliographische Beschreibung:

Neef, Daniel:
Automatisches Layout für Produktfunktionsdarstellung im CAD-Umfeld. - 2009. - 40S.
Mittweida, Hochschule Mittweida, Fachbereich Informatik, Diplomarbeit,
2009

Referat:

Ziel der Diplomarbeit ist es, für den Einsatz in einer von CADsys entwickelten Anwendung, eine Bibliothek für ein optimiertes, automatisches Layout zur Darstellung von komplexen Zusammenhängen zwischen Entitäten (sogenannten Produktfunktionen) zu entwickeln.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

1.1. Motivation der Diplomarbeit

Die Firma CADsys entwickelt ein Programm, in dem Produktfunktionen dargestellt werden. Diese Produktfunktionen können als Diagramm dargestellt werden. Ein solches Diagramm besteht aus den Produktfunktionen und den Beziehungen zwischen diesen (Anhang A.1). Dabei ergibt sich ein Problem: Wie müssen diese Produktfunktionen angeordnet sein? Damit sich für den Betrachter ein möglichst übersichtliches Bild ergibt. Eine Antwort auf diese Frage zu finden ist Ziel dieser Diplomarbeit. Dabei kann auf Probleme und Lösungen der Graphentheorie zurückgegriffen werden. Im Konkreten auf das Problem des *Graph Drawing*, welches in den Grundzügen dem hier vorgestelltem Problem ähnelt.

1.2. Aufgabenstellung und Zielsetzung

Die CADsys Vertriebs- und Entwicklungs GmbH ist ein Systemhaus mit Schwerpunkten auf die Gestaltung und die Optimierung von Produkt- und Projektentwicklungsprozessen. Beginnend bei der Konzeption bis zur Umsetzung sowie von der Technik über die Organisationsgestaltung bis hin zur Personalentwicklung. Zu diesem Zweck entstand der Produktkonfigurator FOD. Der FOD-Modeller unterstützt alle an der Produktentwicklung Beteiligten beim Definieren und Recherchieren von Produktanforderungen, Produkteigenschaften, Produktfunktionen, Produktkomponenten und Produktstrukturen, beim Überwachen der Modellkonsistenz und der Zielkosteneinhaltung sowie der Variantenentstehung.

In dieser Diplomarbeit sollen existierende Algorithmen zur Darstellung von Entitäten untersucht und analysiert werden. Auf Basis dieser Analyse sollen Algorithmen entwickelt werden, welche auf die Darstellung von Funktionsstrukturen optimiert sind. Diese Algorithmen sollen anschließend in einer Softwarebibliothek umgesetzt werden.

Das Ziel dieser Diplomarbeit ist die Entwicklung von Algorithmen zur Anordnung von Entitäten (graphische Darstellung von Produktfunktionen). Diese Algorithmen sollen für die Bedürfnisse von Produktfunktionen optimiert sein. Als Grundlage soll dabei die Darstellung von Graphen dienen. Dabei wird folgende Vereinfachung verwendet: Produktfunktionen werden als Knoten und die Beziehung zwischen diesen als gerichtete Kanten betrachtet. Auf diese Grundlage entstandene Algorithmen werden durch Eigenschaften der Produktfunktionen erweitert und ergeben damit die fertigen Layout-Algorithmen für Funktionsstrukturen.

Ein weiteres Ziel ist die effektive Umsetzung dieser Algorithmen in eine Softwarebibliothek auf Basis des von Microsoft entwickeltem .NET Framework in der Version 3.5. Diese Bibliothek ist zur Nutzung in, von CADsys entwickelten, FOD vorgesehen. Ihr soll eine Liste von Produktfunktionen übergeben werden. Jede Produktfunktion besitzt Koordinaten, diese werden von den Algorithmen verändert. Die Darstellung der einzelnen Produktfunktionen ist nicht Aufgabe der Bibliothek. Die Algorithmen sollen in möglichst kurzer Zeit Funktionsstrukturen mit bis zu 200 Entitäten, für die Darstellung auf einem Bildschirm, anordnen können.

Im Kapitel 2 wird ein kurzer Überblick über die Graphentheorie sowie verschiedener Layout-Algorithmen gegeben. Das Kapitel 3 beschäftigt sich mit den Eigenschaften von Funktionsstrukturen und zeigt dafür optimierte Layout-Algorithmen. Die Umsetzung dieser Algorithmen, in einer Softwarebibliothek, wird im Kapitel 4 beschrieben.

2 Layout-Algorithmen für Graphen

In diesem Kapitel werden für das Verständnis benötigte Grundlagen und Anforderungen aufgezeigt und erläutert. Darüber hinaus werden verschiedene Algorithmen vorgestellt und miteinander verglichen. Die Grundlage bildet dabei ein Teilgebiet der Mathematik, die Graphentheorie. Im Folgenden wird sich auf die Darstellung von Graphen beschränkt.

2.1. Vorbemerkung und Anforderungen

Ein Graph (Abbildung 2.1) $G = (V, E)$ besteht aus einer Anzahl von Knoten (V) und Kanten (E). Die Knoten werden meist als Punkte und die Kanten als Linien, welche zwei Knoten miteinander verbinden, dargestellt. Verfahren um diese Punkte und Linien grafisch darzustellen, werden auch als Layout-Algorithmen bezeichnet. Im Rahmen dieser Diplomarbeit soll sich auf die Darstellung von zweidimensionalen, gerichteten Graphen ohne Mehrfachkanten beschränkt werden. Dabei sollen folgende ästhetische Kriterien berücksichtigt werden:

1. Gleichmäßige Verteilung der Knoten.
2. Wenige gekreuzte Kanten.
3. Stimmiges und übersichtliches Gesamtbild.

Nicht alle Algorithmen können diese Kriterien erfüllen, jedoch kommen die hier vorgestellten Algorithmen sehr nah an das Optimum heran.

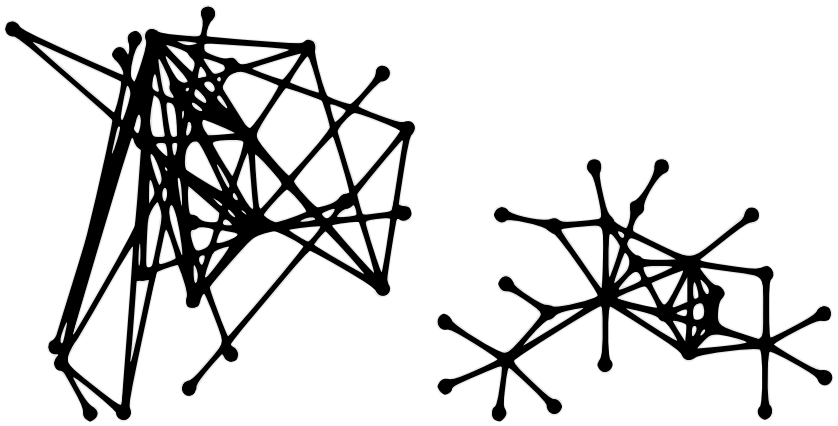

Abbildung 2.1.: ein ungeordneter und ein geordneter Graph

Diese Algorithmen lassen sich in hierarchische und kräftebasierte Algorithmen unterteilen. Hierarchische Algorithmen erzeugen ein strukturiertes Gesamtbild. Kräftebasierte Algorithmen erzeugen dagegen ein gleichmäßiges, verteiltes Gesamtbild.

Um diese Algorithmen miteinander vergleichen zu können, wird auf die Komplexitätstheorie zurückgegriffen. Sie ist ein Teilgebiet der theoretischen Informatik und beschäftigt sich mit der benötigten Rechenzeit und dem Speicherplatzbedarf von Algorithmen. Dabei wird das Landau-Symbol $\mathcal{O}$ verwendet. Es gibt, an wie viel Rechenzeit für einen Algorithmus benötigt wird. Für einen Algorithmus mit $\mathcal{O}(n)$ bedeutet dies einen linearen Anstieg der Rechenzeit. Dies ist abhängig von der Eingangsgröße n, verdoppelt sich diese verdoppelt sich somit auch die benötigte Rechenzeit.

Einige der hier vorgestellten Algorithmen verwenden Heuristiken. Heuristik bedeutet so viel wie Finden und Entdecken und wird in der Informatik wie folgt definiert: „Eine Methode, die mit der Hoffnung auf Erfolg, nicht mit der Garantie für Erfolg zur Lösung einer komplexen, nicht oder schlecht strukturierten Aufgabe eingesetzt wird.“ [GP08, S. 394] Das Ziel solcher Heuristiken ist ein Kompromiss zwischen dem Rechenaufwand und dem gewünschten Ergebnis. Um dies zu erreichen, werden Faustregeln, Annahmen und Schätzungen getroffen aber auch das Raten und der Vergleich mit Naturphänomenen kommen zum Einsatz. [DH02]

In den verschiedenen Algorithmen ist es immer wieder nötig, dass die Datenbestände nach verschiedenen Kriterien sortiert werden. Da das Sortieren von Daten ein sehr umfangreiches Themengebiet ist, kann an dieser Stelle nur kurz darauf eingegangen werden.

Es gibt vergleichsbasierte Sortierverfahren, dabei werden immer zwei Elemente miteinander verglichen und entschieden welches von beiden weiter vorn steht. Eine weitere Art von Sortierverfahren sind solche, die ohne Vergleiche auskommen, dies ist zum Beispiel Radixsort [CLRS07, S. 168ff]. Der Vorteil dieser Verfahren ist ihre Komplexität von $\mathcal{O}(n)$, jedoch funktionieren diese nur unter bestimmten Bedingungen. In dieser Diplomarbeit wurden nur vergleichsbasierte Sortierverfahren eingesetzt, da ihre Komplexität im Mittel bei $\mathcal{O}(n*\log(n))$ liegt und somit kaum schlechter ist. Der überwiegend eingesetzte Sortieralgorithmus ist Quicksort [CLRS07, S. 143ff]. Quicksort teilt die zu sortierenden Elemente, anhand eines Pivotelements, in zwei Listen. So, dass in der linken Liste alle Elemente kleiner und in der rechten Liste alle Elemente größer als das Pivotelement sind. Anschließend wird Quicksort rekursiv auf diese Listen angewandt, bis sich Listen ergeben die nur noch ein oder kein Element enthalten. Diese gelten als in sich sortiert. Das Wichtige bei Quicksort ist die Wahl des Pivotelements. Dieses sollte so gewählt werden, dass immer zwei gleich große Listen entstehen.

2.2. Hierarchische Layout-Algorithmen

Das Wort Hierarchie stammt aus dem Griechischen und bedeutet Herrschaft, Ordnung oder Prinzip. In Verbindung mit Layout-Algorithmen bezeichnet es das Ordnen von Knoten. Dabei werden einem Wurzelknoten alle anderen Knoten, entsprechend ihrer Beziehungen, untergeordnet. Bei diesen Algorithmen werden die Knoten auf Zeilen und Spalten verteilt. Im ersten Schritt wird dazu jeder Knoten einer Spalte zugeteilt. Im zweiten Schritt werden die Kantenkreuzungen reduziert, durch Umverteilung der Knoten in der jeweiligen Spalte. Als Letztes werden die neuen Koordinaten den Knoten zugewiesen.

2.2.1. Tree

Bei einem Baum(tree) wird zuerst die Wurzel (root) bestimmt. Als Wurzel wird ein Knoten bezeichnet, der keine Elternknoten besitzt. Von der Wurzel ausgehend werden alle Kinderknoten durchlaufen und ihrer Hierarchie nach den verschiedenen Schichten zugewiesen. Für das Durchlaufen des Baumes eignen sich sowohl die Tiefensuche als

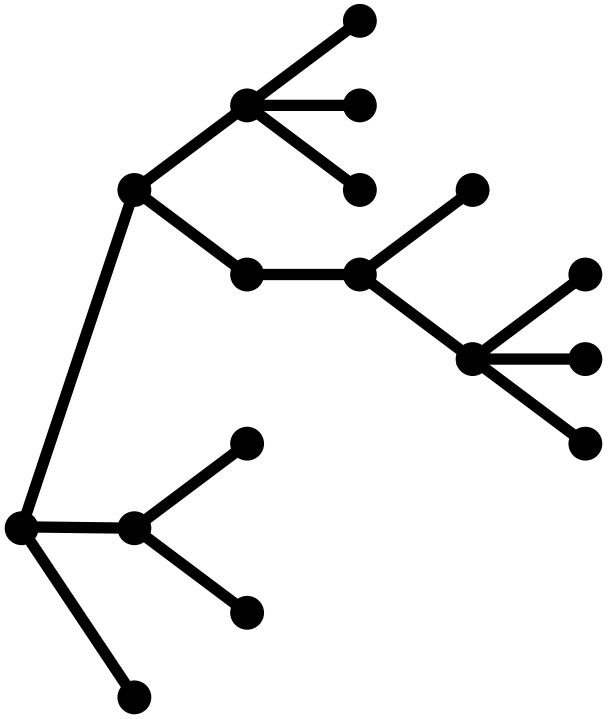

Abbildung 2.2.: Tree-Layout

auch die Breitensuche, da immer der komplette Baum durchlaufen wird, ergibt sich eine Laufzeit($\mathcal{O}$) von:

$$\mathcal{O} = (|V| + |E|) \tag{2.1}$$

dabei sei $|V|$ die Anzahl der Knoten und $|E|$ die Anzahl der Kanten. Zu welcher Schicht ein Knoten gehört, ist abhängig von seiner Entfernung zur Wurzel. Die Kinder der Wurzel gehören zur Schicht 1, die Kinder der Kinder gehören zur Schicht 2, usw. Nach dem alle Knoten einer Schicht zugewiesen wurden, werden sie innerhalb der Schicht überlagerungsfrei angeordnet. Damit kann der Tree-Algorithmus in zwei Phasen aufgeteilt werden:

1. Aufteilung der Knoten auf Schichten, entsprechend ihrer Entfernung zur Wurzel.
2. Zuweisen der Koordinaten.

Damit ergibt sich eine Gesamtlaufzeit des Algorithmus von:

$$\mathcal{O} = 2 * (|V| + |E|) \tag{2.2}$$

$$\Rightarrow \text{Gesamtaufwand von } \mathcal{O}(n) \tag{2.3}$$

2.2.2. Sugiyama

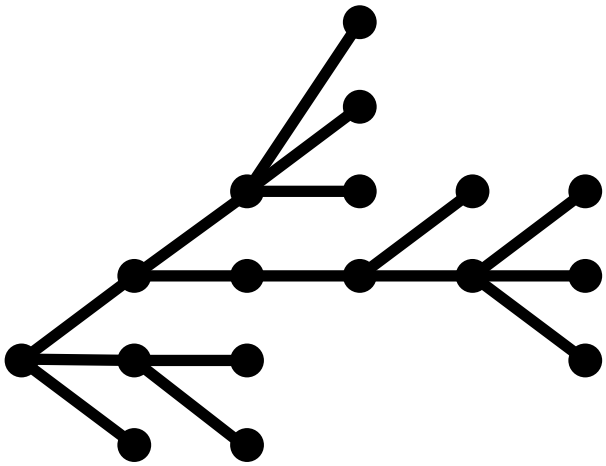

Abbildung 2.3.: Sugiyama-Layout

Der Sugiyama-Algorithmus ([KW01, S.87ff]) ist eine Weiterentwicklung des Tree-Algorithmus. Dabei wird die zusätzliche Phase der Kantenkreuzungsminimierung eingeschoben.

1. Aufteilung der Knoten auf Schichten, entsprechend ihrer Entfernung zur Wurzel.
2. Kantenkreuzungsminimierung.
3. Zuweisen der Koordinaten.

Für die Minimierung der Kantenkreuzungen werden immer zwei benachbarte Schichten betrachtet. Dabei entsteht ein bipartiter Graph, dessen Minimierung der Kantenkreuzungen NP-Schwer ist [PE94][PM02]. Aus diesem Grund werden verschiedene Heuristiken eingesetzt, zum Beispiel die Barycenter Heuristik. Dabei wird angenommen, dass eine Schicht fest ist. Für die andere Schicht wird für jeden Knoten der Barycenter aus der Position der Eltern errechnet. Sei s der aktuell betrachtete Knoten, a die Anzahl der Eltern und seien weiterhin t_i die Positionen der Eltern, dann gilt:

$$\text{barycenter}(s) = \sum_{i=0}^{a} t_i * \frac{1}{a} \tag{2.4}$$

Die Knoten einer Schicht werden nach ihrem Barycenter Wert sortiert und entsprechend angeordnet. Durch die Minimierung der Kantenkreuzung gibt sich ein Gesamtaufwand

von:

$$\mathcal{O}(n^2) \tag{2.5}$$

2.3. Kräftebasierte Layout-Algorithmen

Die Idee der kräftebasierten Algorithmen ist es, den Graphen als physisches System zu betrachten. Dabei werden die Knoten als Punkte dargestellt, die sich gegenseitig abstoßen. Die Kanten wirken entgegengesetzt. Sie ziehen die verbundenen Knoten an, ähnlich wie Sprungfedern. Realisiert wird dieses System durch entsprechende Kräftevektoren. Das Ziel des Algorithmus ist dabei eine Konstellation der Knoten zu finden, bei der die Summe der Kräftevektoren gleich null ist. Da dies nicht immer möglich ist oder in großen Graphen sehr viel Rechenzeit benötigt, wird meist eine Toleranz (x) verwendet.

2.3.1. Spring

Beim Spring-Algorithmus besitzt jeder Knoten dieselbe abstoßende Kraft, bestimmt durch eine Abstoßungskonstante. Die anziehende Kraft der Kanten wird dabei durch Federn realisiert. Der Spring-Algorithmus lässt sich in folgende zwei Phasen einteilen:

1. Für jeden Knoten alle auf ihn wirkenden Kräftevektoren aufsummieren.
2. Neue Position der Knoten bestimmen, durch entsprechenden Kräftevektor.

Wenn die Summer alle Kräftevektoren des Graphen nicht innerhalb der Toleranz x liegt wechselt der Spring-Algorithmus wieder in Phase eins. Aus der Berechnung der Kräftevektoren ergibt sich ein Gesamtaufwand von:

$$\mathcal{O}(n^2) \tag{2.6}$$

2.3.2. Fruchterman Reingold

Der Algorithmus von Fruchterman und Reingold [EMR91] ist eine Weiterentwicklung des Spring-Algorithmus. Er berücksichtigt dabei die Größe der Zeichenfläche ($Width, Hight$)

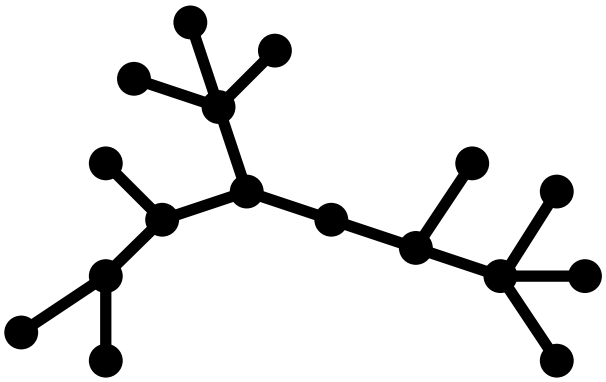

Abbildung 2.4.: Fruchterman-Reingold-Layout

und das verbundene Knoten nah beieinander sind. Die Kräfte werden dabei wie folgt berechnet:

$$f_{anziehend}(d) = \frac{d^2}{k} \tag{2.7}$$

$$f_{abstoßend}(d) = -\frac{k^2}{d} \tag{2.8}$$

Dabei sei d die Entfernung zweier Knoten (v, u) und k die optimale Entfernung zweier Knoten.

$$d = |v - u| \tag{2.9}$$

$$k = C * \sqrt{\frac{Width * Hight}{|V|}} \tag{2.10}$$

Die Konstante C muss dabei experimentell ermittelt werden. Die Phasen des Fruchtman-Reingold-Algorithmus sind dabei dieselben wie im Spring-Algorithmus. Somit beträgt der Gesamtaufwand des Algorithmus $\mathcal{O}(n^2)$.

2.3.3. Kamada-und-Kawais-Modell

Im Modell von Kamada und Kawais [KK89] wird nicht zwischen anziehenden und abstoßenden Kräften unterschieden. Jede Kante wird durch eine Feder ersetzt, diese stellt die anziehenden und abstoßenden Kräfte gleichzeitig dar. Die Grundlänge der Feder ist dabei der graphentheoretisch kürzeste Pfad. Ist die geometrische Entfernung größer als die graphentheoretische Entfernung, ziehen sich die Knoten an, andernfalls stoßen sie

sich ab. Es werden jedoch nicht alle Knoten gleichzeitig verschoben. In einem Iterationsschritt wird immer nur der Knoten mit der größten auf ihn wirkenden Kraft versetzt, sodass die auf ihn wirkenden Kräfte minimal werden. Der Kamada-Kawai-Algorithmus erfordert das Lösen des all-pairs-shortest-path-Problem. Dies ist zum Beispiel mit dem Floyd-Warshall-Algorithmus möglich [CLRS07, S. 629ff]. Daraus ergibt sich ein Gesamtaufwand von $\mathcal{O}(n^3)$.

2.4. Vergleich und Analyse

Die hier vorgestellten Algorithmen sind nur eine Auswahl der vielen existierenden Algorithmen und Verfahren um Graphen darzustellen. Nachfolgenden sollen diese miteinander verglichen und auf ihre Verwendbarkeit hin geprüft werden. Die Tabelle 2.1 stellt eine Übersicht der vorgestellten Algorithmen dar.

Algorithmus	**Komplexität**	**Optimierung**
Tree	$\mathcal{O}(n)$	keine
Sugiyama	$\mathcal{O}(n^2)$	Minimierung der Kantenkreuzungen
Spring	$\mathcal{O}(n^2)$	keine
Fruchterman-Reingold	$\mathcal{O}(n^2)$	Optimale Verteilung auf Zeichenfläche
Kamada-und-Kawais	$\mathcal{O}(n^3)$	keine

Tabelle 2.1.: Vergleich von Layout-Algorithmen

Der Tree-Algorithmus (Kapitel 2.2.1) kann stellvertretend für alle einfachen Algorithmen betrachtet werden. Die einfachen Algorithmen erfordern nur einen geringen Aufwand an Rechenzeit (Formel 2.3), können dafür aber nicht alle ästhetischen Kriterien erfüllen. Der Tree-Algorithmus besitzt zum Beispiel keine Minimierung der Kantenkreuzung. Weitere einfache Algorithmen sind Random, Circle oder Algorithmen die Knoten nach anderen geometrischen Formen anordnen. Alle diese Algorithmen eignen sich jedoch nur für bestimmte Graphen und sind für Funktionsstrukturen nicht geeignet. Ihre geringe Rechenzeit macht sie jedoch interessant zum Erzeugen von Vorbedingungen für andere Algorithmen.

Der Algorithmus von Sugiyama (Kapitel 2.2.2) erfüllt alle ästhetischen Kriterien und eignet sich besonders für hierarchische, gerichtete Graphen. Liegen die Knoten bereits

in Form eines sortierten Baumes vor, lassen sich auch größere Graphen problemlos darstellen. Ein Problem ist ihr jedoch die Kantenkreuzung, welche nur durch entsprechende Heuristiken in vertretbarer Zeit möglich ist. Jedoch bietet er eine gute Grundlage und wurde aus diesen Gründen für das hierarchische Layout verwendet (Kapitel 3.2.1).

Der Vorteil von kräftebasierten Algorithmen (Kapitel 2.3) liegt im übersichtlichen Gesamtbild des Graphen. So werden verbundene Knoten nah beieinander gezeichnet und Kantenkreuzungen entstehen meist erst gar nicht. Jedoch benötigen sie einige Vorbedingungen. So müssen für jeden Knoten Startkoordinaten vorhanden sein. Sind diese nicht vorhanden, müssen sie durch einfache Algorithmen erzeugt werden. Dies sind meist Zufalls Algorithmen aber auch der Tree-Algorithmus eignet sich dafür. Ein Nachteil ist, dass das Ergebnis der kräftebasierten Algorithmen stark von der Qualität dieser Startkoordinaten abhängig ist. Wie viele Iterationen nötig sind, hängt ebenfalls stark von der Position der Knoten, vor Anwendung des Algorithmus, ab. Dies führt zu dem zweiten Nachteil, dass die Laufzeit nur schwer abschätzbar ist. Schon kleine Veränderungen an den Startkoordinaten können die Laufzeit erheblich erhöhen aber auch stark verkürzen. Der einfachste Vertreter dieser Kategorie ist der Spring-Algorithmus (Kapitel 2.3.1).

Ein weiterer Vertreter dieser Kategorie ist der Fruchterman-Reingold-Algorithmus (Kapitel 2.3.2). Er besitzt noch den weiteren Vorteil, dass er die vorhandenen Knoten auf einer vorgegebenen Zeichenfläche gleichmäßig verteilt. Dies ist möglich durch die Einbeziehung der Höhe und Breite beim Berechnen der Kräfte.

Kamada-und-Kawais-Modell (Kapitel 2.3.3) zeichnet ebenfalls einen sehr guten Graphen. Es benötigt dabei aber erheblich mehr Rechenzeit und Speicherbedarf als die anderen Algorithmen.

Im Allgemeinen eignen sich kräftebasierte Algorithmen für alle Arten von Graphen. Bei besonders großen Graphen müssen jedoch Optimierungen in den Vorbedingungen (Startkoordinaten) getroffen werden.

3 Entwicklung von Layout-Algorithmen für Funktionsstrukturen

Funktionsstrukturen haben im Vergleich mit allgemeinen Graphen einiges Besonderheiten. Diese werden im folgenden kurz erläutert und im Anschluss werden verschiedene Algorithmen vorgestellt und miteinander verglichen.

3.1. Grundlagen und Anforderungen

Ein wesentlicher Bestandteil der von CADsys entwickelten Anwendung FOD erlaubt das Modellieren von Produktfunktionen. Eine Produktfunktion besteht aus einem Wirkungsprinzip und den damit verbundenen Parametern. Diese Parameter lassen sich in Eingangsparametern und Ausgangsparametern unterteilen. Ein Eingangsparameter wird dabei der Produktfunktion vorgegeben. Die Ausgangsparameter stellt die Produktfunktion zur Verfügung, diese können von anderen Produktfunktionen als Eingangsparameter verwendet werden. Die Produktfunktionen werden graphisch als Rechtecke und die Beziehungen zwischen Ausgangsparameter und Eingangsparameter werden als Pfeil dargestellt.

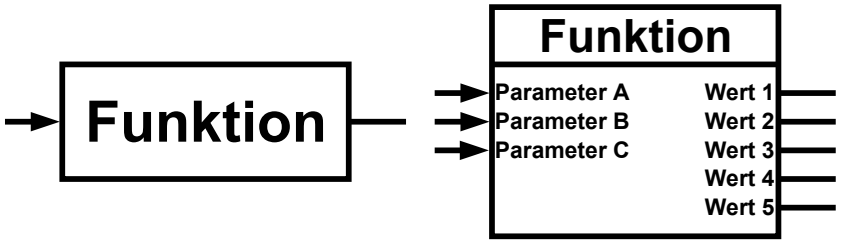

Abbildung 3.1.: Einfache Produktfunktion

In einem Produktfunktionsdiagramm gibt es eine Anfangsfunktion, diese besitzt keine Vorgängerfunktionen. Außerdem existiert eine Endfunktion, welche keine Nachfolgerfunktion besitzt. Jede andere Funktion kann n Vorgängerfunktionen und m Nachfolgerfunktionen besitzen. In der Standardansicht sind die Funktionen geschlossen, das Bedeutet einzelne Parameter sind nicht sichtbar. Die Parameter werden erst durch einen Klick auf den entsprechenden Button sichtbar. Dabei wird das Rechteck der Funktion nach unten und rechts erweitert. Dadurch nötige Verschiebungen der anderen Funktionen werden gesondert behandelt (3.2.3). Abbildung 3.1 zeigt eine Produktfunktion in Standardansicht und in Parameteransicht.

Ein Produktfunktionsdiagramm kann sich zu Beginn in einem von zwei Zuständen befinden. Der erste Zustand existiert, wenn den einzelnen Produktfunktionen noch keine Positionen zugewiesen wurden. In diesem Fall befinden sich alle Produktfunktionen übereinander und müssen gegebenenfalls erst verteilt werden. Wurden durch manuelle Positionierung oder anderen Algorithmen bereits Positionen vergeben, können diese als Anfangsposition verwendet werden. Besonders in größeren Diagrammen ist diese Anfangsposition für das Ergebnis der Layout-Algorithmen mitverantwortlich.

3.2. Vorstellung der Algorithmen

In Kapitel 2 wurden die Vorteile und Nachteile verschiedener Layout-Algorithmen aufgezeigt. Auf Grund der Eigenschaften und der Verwendung im Zielsystem wurden zwei verschiedene Ansätze gewählt. Zum einen das hierarchische Layout (Kapitel 3.2.1) und zum anderen das kräftebasierte Layout (Kapitel 3.2.2). Der dritte Algorithmus (Kapitel 3.2.3) wurde entwickelt, um effizient auf Veränderungen einzelner Produktfunktionen reagieren zu können.

Allen Algorithmen wird eine Aufzählung von Elementen übergeben. Jedes dieser Elemente beinhaltet eine Produktfunktion sowie einen Verweis auf alle direkten Nachfolgerfunktionen. Diese Elemente werden nachfolgend als *GraphEntity* bezeichnet.

Der allgemeine Ablauf ist bei beiden Ansätzen der selbe. Es muss zu erst einer der beiden Algorithmen ausgewählt werden. Danach können, je nach Algorithmus, verschiedene

Parameter gesetzt werden. Im Anschluss müssen dem Algorithmus die *GraphEntitys* bekannt gemacht werden. Als Abschluss weist der Algorithmus allen *GraphEntitys* ihre Position zu.

3.2.1. Hierarchisches Layout

Dieser Algorithmus baut auf den von Sugiyama (Kapitel 2.2.2) entwickeltem Verfahren auf. Es werden alle Elemente an einem Raster ausgerichtet und anschließend die Kantenkreuzungen reduziert.

Attribut	**Beschreibung**
X	X Koordinate
Y	Y Koordinate
Height	Höhe
Width	Breite
isRoot	TRUE wenn keine Vorgängerfunktion vorhanden
Level	Schicht der Funktion
Position	Position innerhalb der Schicht
Parents	Verweis auf Aufzählung der Vorgängerfunktion
Childs	Verweis auf Aufzählung der Nachfolgerfunktion
Barycenter	Wert für Minimierung der Kantenkreuzungen

Tabelle 3.1.: Datentyp *GraphEntity* für hierarchischen Algorithmus

Der hierarchische Algorithmus (Algorithmus 1) benötigt einen Parameter. *GraphEntitys* (Tabelle 3.1) ist eine Aufzählung von den anzuordnenden Produktfunktionen. Im ersten Schritt wird jeder *GraphEntity* eine Schicht zu gewiesen. Die Schicht, Spalte oder Level ist der horizontale Abstand zum Wurzelelement (Anfangsfunktion). Um diese zu ermitteln, wird die Funktion *SetLevel()* rekursiv, von dem Wurzelelement aus, aufgerufen (Zeilen 1 - 3). Im Regelfall besitzt eine Funktionsstruktur nur ein Wurzelelement, der Algorithmus verarbeitet aber auch mehrere. Als Vorbedingung muss aber mindestens ein *GraphEntity* als Wurzelelement (isRoot == true) markiert sein. Der Schritt zwei fügt virtuelle *GraphEntitys* ein, für Beziehungen, die über mehr als eine Schicht gehen. Dabei werden auch wieder alle *GraphEntitys*, beginnend beim Wurzelelement, rekursiv durchlaufen (Zeilen 4 - 6). Im Anschluss werden alle *GraphEntity* aufsteigend nach ihrer Schicht sortiert (Zeile 7). Beginnend mit der Schicht null werden im dritten

Schritt mithilfe der Barycenter Heuristik die Kantenkreuzungen reduziert. Dazu werden alle *GraphEntity* einzeln durchgegangen (Zeile 9). Sollte das aktuelle *GraphEntity* ein Virtuelles sein (Zeile 18) so bekommt es dieselbe Position wie sein einer Elternteil. Wenn es nicht virtuell ist, dann wird aus den Positionen der Eltern der *Barycenter-Wert* ermittelt (Zeilen 21 - 24). Wurden alle *GraphEntity* einer Schicht bearbeitet (Zeile 10), dann wird ihnen entsprechend ihrer *Barycenter-Werte* eine Position innerhalb der Schicht zugewiesen (Zeilen 12 - 16).

Algorithmus 1 Hierarchischer Algorithmus

Require: *GraphEntitys*

```
 1: for all GraphEntity of GraphEntitys where GraphEntity.isRoot do
 2:    SetLevel(GraphEntity, 0)
 3: end for
 4: for all GraphEntity of GraphEntitys where GraphEntity.isRoot do
 5:    SetVirtualEntity(GraphEntity, 0, null)
 6: end for
 7: GraphEntitys.SortByLevel()
 8: aktLevel := 0
 9: for all GraphEntity of GraphEntitys do
10:    if GraphEntity.Level != aktLevel then
11:       aktPosition := 0
12:       GraphEntitys.SortByBarycenter
13:       for all GraphEntity where Level == aktLevel do
14:          GraphEntity.Position := aktPosition
15:          aktPosition := aktPosition + 1
16:       end for
17:    end if
18:    if GraphEntity is virtual then
19:       GraphEntity.Position := GraphEntity.Parent.Position
20:    else
21:       for all parents of GraphEntity do
22:          GraphEntity.Barycenter+ = GraphEntity.Parent.Position
23:       end for
24:       GraphEntity.Barycenter/ = GraphEntity.ParentsCount
25:    end if
26: end for
```

Die rekursive Funktion *SetLevel()* (Algorithmus 2) weißt jeder *GraphEntity* eine Schicht zu. Dazu wird dem Wurzelelement die Schicht null zugewiesen. Von diesem Element aus werden alle nachfolgenden Elemente aufgerufen (Zeilen 4 - 6) und jeweils einer um eins erhöhten Schicht zugewiesen. Damit gibt die Schicht einer *GraphEntity* auch an, wie weit

es vom Wurzelelement entfernt ist.

Algorithmus 2 Funktion zur Einteilung der Schichten

Require: $GraphEntity, Level$

```
1: if GraphEntity.Level < Level then
2:    GraphEntity.Level := Level
3: end if
4: for all childs of GraphEntity do
5:    SetLevel(GraphEntity, Level + 1)
6: end for
```

Wenn eine Beziehung zwischen zwei *GraphEntitys* über mehr als eine Schicht geht, müssen in den Schichten dazwischen virtuelle *GraphEntitys* eingefügt werden. Diese virtuelle *GraphEntity* zählt dabei als Platzhalter, damit Beziehungen über mehr als eine Schicht nicht durch andere *GraphEntity* unterbrochen werden. Dies wird durch den rekursiven Aufruf von *SetVirtualEntity()* (Algorithmus 3) realisiert (Zeilen 5 - 7). Dabei werden alle *GraphEntitys* überprüft, beginnend beim Wurzelelement. Sollte eine Beziehung über mehr als zwei Schichten gehen (Zeile 1), werden entsprechend viele virtuelle Knoten dazwischen geschoben (Zeilen 2 - 3).

Algorithmus 3 Funktion zum Einfügen virtueller Knoten

Require: $GraphEntity, Level, GraphEntityParent$

```
1: if GraphEntity.Level! = Level then
2:    newVirtualGraphEntity()
3:    SetVirtualEntity(GraphEntity, Level + 1, VirtualGraphEntity)
4: else
5:    for all child of GraphEntity do
6:       SetVirtualEntity(child, Level + 1, GraphEntity)
7:    end for
8: end if
```

Nach dem die Knoten auf Schichten verteilt und ihre Reihenfolge in der jeweiligen Schicht festgelegt wurde, kann eine weitere Optimierung nötig sein (Abbildung 3.2). Hier für bietet sich die Knotenpositionierung mittels Prioritätswerten nach [Mut07, Kapitel 3.1] an. Dabei werden, wie bei der Kreuzungsminimierung, immer zwei benachbarte Schichten betrachtet. Eine Schicht wird dabei als nicht veränderbar angesehen. Dieses Verfahren wird zweimal angewandt, einmal mit der Annahme, dass die linke Schicht fest ist und anschließend das die rechte Schicht unveränderlich ist.

Algorithmus 4 wird für diese Optimierung verwendet. Er benötigt dafür die Aufzählung

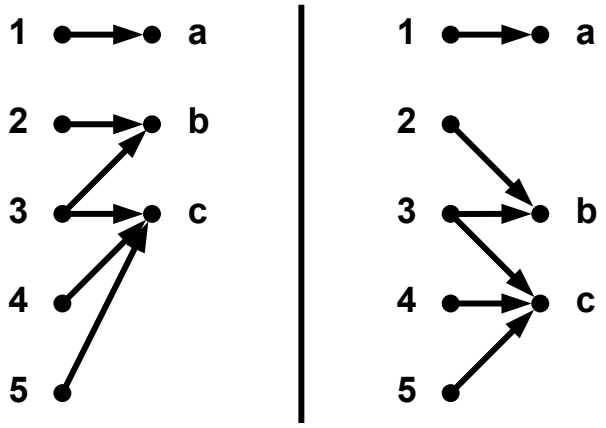

Abbildung 3.2.: Optimierung der Position

Algorithmus 4 TreeOptimizePosition

Require: *GraphEntitys*

1: *GraphEntitys.SortByLevel*()
2: **for all** *GraphEntity* of *GraphEntitys* **do**
3: *MoveToPosition*(*GraphEntity*, *GetFavoritePosition*(*GraphEntity*))
4: **end for**

der *GraphEntitys*, welchen vorher eine Schicht und Position zugewiesen wurde (Algorithmus 1). Unter der Annahme, dass die linke Schicht unveränderlich ist, werden die *GraphEntity* nach ihren Schichten aufsteigend sortiert (Zeile 1). Für die Reihenfolge der Abarbeitung innerhalb der Schicht ist der Prioritätswert verantwortlich. Dieser Prioritätswert richtet sich nach der Anzahl der Elternknoten. Danach ergibt sich zum Beispiel für die Abbildung 3.2 folgende Reihenfolge: c, b, a. Nach dieser Reihenfolge werden alle *GraphEntity* auf ihre optimale Position geschoben (Zeilen 2 - 4).

Algorithmus 5 Funktion um Position eines Knoten zu ändern

Require: *GraphEntity*, *GraphEntitys*, *NewPosition*

1: **if** *NewPosition* is free **then**
2: *GraphEntity.Position* = *NewPosition*
3: **end if**

Die Funktion *MoveToPosition* (Algorithmus 5) ist für die Verschiebung der *GraphEntitys* zuständig. Sie benötigt das zu verschiebende *GraphEntity*, eine Aufzählung aller auf der Schicht vorhanden *GraphEntitys* sowie die neue Position. Zuerst muss überprüft werden, ob die Verschiebung möglich ist (Zeile 1). Bei der Verschiebung darf sich die Reihenfolge nicht ändern und Knoten mit einer höheren Priorität dürfen nicht mehr verschoben

werden. Für das Beispiel aus Abbildung 3.2 bedeutet dies, dass a oberhalb von b und b oberhalb von c stehen muss. Außerdem darf, nachdem c auf Position 4 gesetzt wurde, c nicht mehr verschoben werden. Sind diese Bedingungen erfüllt, wird dem *GraphEntity* seine neue Position zugewiesen (Zeile 2).

Algorithmus 6 Funktion zum Ermitteln der besten Position

Require: *GraphEntity*
1: $position := 0$
2: $count := 0$
3: **for all** *Parent* of $GraphEntity.Parents$ **do**
4: $\quad position+ = Parent.Position$
5: $\quad count+ = 1$
6: **end for**
7: **return** $position/count$

Die optimale Position für einen Knoten wird durch die Funktion *GetFavoritePosition* (Algorithmus 6) bestimmt. Sie benötigt als Parameter das entsprechende *GraphEntity*. Von diesem *GraphEntity* werden die Positionen der Eltern aufsummiert (Zeilen 3 - 6) und mit der Anzahl der Eltern dividiert (Zeile 7). Da es keine gebrochenen Positionen gibt, wird das Ergebnis auf eine ganze Zahl gerundet und zurückgegeben.

Wurden die Position von allen *GraphEntitys* aktualisiert, wird der Algorithmus 4 erneut durchlaufen. Diesmal gilt jedoch die Annahme, dass die rechte Schicht unveränderlich ist. Daraus folgend wird die Aufzählung der *GraphEntitys* absteigend nach Schichten sortiert. Für die Berechnung der Prioritäten und optimalen Positionen innerhalb der Schichten gelten die gleichen Regeln, jedoch werden die Kinderknoten als Referenz verwendet.[MJ97]

3.2.2. Kräftebasiertes Layout

Der hier verwendete Algorithmus basiert auf dem von Reingold und Fruchterman (Kapitel 2.3.2) entwickelten Grundlagen. Es wird für jedes Element ein Kräftevektor ermittelt, anschließend werden die Elemente um diesen Vektor verschoben. Dies wird sooft wiederholt, bis ein stimmiges Gesamtbild entstanden ist.

Für das kräftebasierte Layout ist es wichtig das alle *GraphEntity* bereits Koordinaten besitzen und das nach Möglichkeit keine Überlappungen vorhanden sind. Darüber hinaus besitzt es die Parameter: *AreaHeight*, *AreaWidth*, *pushFactor*, *appealingFactor* und

Count. *AreaHeight* und *AreaWidth* geben dabei die Zeichenfläche an, auf der die *GraphEntitys* verteilt werden sollen. Wenn die beiden Parameter nicht angeben sind, wird so viel Platz wie nötig verwendet. Der *pushFactor* gibt an, wie stark sich die *GraphEntitys* voneinander abstoßen. Der *appealingFactor* ist für die Stärke der anziehenden Kräfte verantwortlich. Diese beiden Parameter müssen nur bei Bedarf entsprechend gesetzt werden.

Attribut	**Beschreibung**
X	X Koordinate
Y	Y Koordinate
Height	Höhe
Width	Breite
ForceVektor	Kräftevektor der auf *GraphEntity* wirkt
Child	Verweis auf Nachfolgerfunktion

Tabelle 3.2.: Datentyp *GraphEntity* für kräftebasierten Algorithmus

Der kräftebasierte Algorithmus (Algorithmus 7) benötigt zwei Parameter. *Count* ist eine Konstante, welche angibt, wie oft die Kräfte berechnet werden. Je höher sie ist, desto genau ist das Ergebnis. Jedoch erfordert jeder Durchlauf des Algorithmus einen Aufwand von $\mathcal{O}(n^2)$. Aus diesem Grund muss für die Konstante *count* ein entsprechender Kompromiss gefunden werden. Der zweite Parameter (*GraphEntitys*) ist eine Aufzählung von *GraphEntity* (Tabelle 3.2) Elementen, welche durch den Algorithmus platziert werden sollen.

Der Algorithmus wird sooft durchlaufen bis die Anzahl *count* erreicht wurde (Zeile 1). Es wird die Beziehung von jedem *GraphEntity* mit jedem anderen betrachtet (Zeilen 3 - 5). Dabei werden für jedes *GraphEntity* die abstoßenden Kräfte (Zeile 6) berechnet. Sollte zwischen den zwei betrachteten *GraphEntitys* eine Beziehung besteht, werden auch die anziehenden Kräfte berechnet (Zeilen 7 - 11). Daraus ergibt sich für jede *GraphEntity* ein Kräftevektor $forceSum$ (Zeile 12). Nach dem alle Kräftevektoren berechnet wurden, werden die *GraphEntitys* um diesen verschoben (Zeilen 16 - 18). Dabei wird auch sichergestellt, dass keine Überlappungen entstehen.

Zur Berechnung der abstoßenden Kräfte wird die Funktion *PushForce()* (Algorithmus 8) verwendet. Sie benötigt die beiden *GraphEntitys*, zwischen denen die abstoßende Kraft berechnet werden soll sowie die Konstante $pushFactor$. Diese Konstante gibt an, wie

Algorithmus 7 Kräftebasierender Algorithmus

Require: $count, GraphEntitys$

```
 1: for 0 to count do
 2:   forceSum := (0; 0)
 3:   for all GraphEntityA of GraphEntitys do
 4:     for all GraphEntityB of GraphEntitys do
 5:       if GraphEntityA! = GraphEntityB then
 6:         forceSum := PushForce(GraphEntityA, GraphEntityB)
 7:         if GraphEntityA a parent of GraphEntityB then
 8:           forceSum+ = AppealingForce(GraphEntityA, GraphEntityB)
 9:         else if GraphEntityB a parent of GraphEntityA then
10:           forceSum− = AppealingForce(GraphEntityB, GraphEntityA)
11:         end if
12:         GraphEntityA.ForceVector+ = forceSum
13:       end if
14:     end for
15:   end for
16:   for all GraphEntitys do
17:     GraphEntity+ = GraphEntity.ForceVector
18:   end for
19: end for
```

stark sich zwei *GraphEntitys* abstoßen. Der Rückgabewert dieser Funktion ist der Abstoßungsvektor für *GraphEntityA*. Als Bezugspunkt der Kräfte dient der Mittelpunkt des *GraphEntity* (Zeile 2). Wenn die zwei Parameter *AreaHeight* und *AreaWidth* gesetzt sind werden sie in Berechnung mit einbezogen (Zeilen 3 - 5). Sind sie nicht gesetzt, wird die Formel ohne Bezug auf die Zeichenfläche verwendet (Zeile 7).

Die Funktion *AppealingForce()* (Algorithmus 9) berechnet die anziehenden Kräfte zwischen zwei *GraphEntitys*. Sie benötigt, wie die *PushForce()* die *GraphEntitys* und eine Konstante (*appealingFactor*). Diese Konstante gibt an wie stark sich zwei verbundene *GraphEntitys* anziehen. Als Rückgabewert dient auch hier ein Kräftevektor für *GraphEntityA*. Der Bezugspunkt der Kräfte ist hier nicht der Mittelpunkt, sondern die Mitte der äußeren Seite. Beim *Eltern-GraphEntity* ist es die rechte Seite und bei dem *Kind-GraphEntity* die linke Seite. Sind Informationen über die Größe der Zeichenfläche vorhanden, werden sie mit einbezogen (Zeilen 3 - 5). Wenn die Informationen nicht vorhanden sind, erfolgt die Berechnung nach Zeile 7.

Algorithmus 8 Funktion für abstoßende Kräfte (PushForce)

Require: $GraphEntityA, GraphEntityB, pushFactor$

1: $force := (0; 0)$
2: $force := GraphEntityA\ -\ GraphEntityB$
3: **if** $AreaHeight$ isSet and $AreaWidth$ isSet **then**
4: $\quad k := pushFactor\ *\ \sqrt{(AreaHeight * AreaWidth)/number\ of\ GraphEntity}$
5: $\quad force := force^2\ /\ k$
6: **else**
7: $\quad force := force\ /\ |force|^2\ *\ pushFactor$
8: **end if**
9: **return** $force$

Algorithmus 9 Funktion für anziehende Kräfte (AppealingForce)

Require: $GraphEntityA, GraphEntityB, appealingFactor$

1: $force := (0; 0)$
2: $force := GraphEntityA\ -\ GraphEntityB$
3: **if** $AreaHeight$ isSet and $AreaWidth$ isSet **then**
4: $\quad k := appealingFactor\ *\ \sqrt{(AreaHeight * AreaWidth)/number\ of\ GraphEntity}$
5: $\quad force := k^2\ /\ force$
6: **else**
7: $\quad force := force\ *\ |force|\ /\ (|force|\ +\ appealingFactor)$
8: **end if**
9: **return** $force$

3.2.3. Abstands Kontroll Algorithmus

Ziel des Algorithmus ist die effiziente Prüfung der minimal Abstände(*InterspaceX, InterspaceY*) zwischen den *GraphEntitys*. Für diese Überprüfung und Verschiebung werden die *GraphEntitys* nach ihren Koordinaten Sortiert. Es wird Aufsteigend nach X-Koordinaten sortiert, sollten diese gleich sein entscheidet die kleinere Y-Koordinate. Nach dem Sortieren wird jede *GraphEntity* der Reihenfolge entsprechend überprüft. Sollte der Abstand zu gering sein wird die aktuelle *GraphEntity* entsprechend nach Unten oder nach Rechts verschoben.

Algorithmus 10 Abstands Kontroll Algorithmus

Require: $GraphEntity, InterspaceX, InterspaceY$

```
1:  maxX := 0, maxY := 0
2:  OffsetX := 0, OffsetY := 0
3:  for all GraphEntity of GraphEntitys do
4:    if GraphEntity.Y < maxY then
5:      maxY := 0, OffsetY := 0
6:      if (maxX + InterspaceX) > (GraphEntity.X + OffsetX) then
7:        OffsetX := (maxX + InterspaceX) − (GraphEntity.X + OffsetX)
8:      end if
9:    end if
10:   if maxY == 0 then
11:     maxY := GraphEntity.Height + GraphEntity.Y + InterspaceY
12:   else if (GraphEntity.Y + OffsetY) < maxY then
13:     OffsetY+ = maxY − GraphEntity.Y
14:   end if
15:   GraphEntity.X+ = OffsetX
16:   GraphEntity.Y+ = OffsetY
17:   if (GraphEntity.X + GraphEntity.Width) > maxX then
18:     maxX := GraphEntity.X + GraphEntity.Width
19:   end if
20: end for
```

Der Abstands Kontroll Algorithmus (Algorithmus 10) benötigt drei Parameter. Diese sind die Aufzählung der vorhanden *GraphEntitys* sowie der horizontalen Mindestabstand (*InterspaceX*) und der vertikalen Mindestabstand (*InterspaceY*). Weiterhin wird angenommen das die *GraphEntitys* vorher mit einem Layoutalgorithmus angeordnet wurden. Der Abstands Kontroll Algorithmus Prüft jede *GraphEntity*, sollte sie in der vertikalen Ausrichtung oberhalb der letzten liegen (Zeile 4) wird der vertikale Offset zurückgesetzt

und geprüft ob ein neuer horizontaler Offset benötigt wird (Zeile 6). Nach jeder dieser Zurücksetzungen wird $maxY$ neu gesetzt, sonst wird überprüft ob der vertikale Offset ausreichend ist oder erhöht werden muss (Zeilen 10 - 14). Im Anschluss wird der Offset zu den Koordinaten addiert (Zeilen 15 - 16) und $maxX$ wird, wenn nötig, neu gesetzt (Zeilen 17 - 19).

Diese Überprüfung und Verschiebung ist mit Linearem Aufwand ($\mathcal{O}(n)$) verbunden. Wurde zum Beispiel die Breite oder die Höhe einer *GraphEntity* verändert ist die Verwendung des Abstand Kontroll Algorithmus effizienter als eine kompletten Neuberechnung des Layouts. Ein genauer Vergleich der Algorithmen wird im Kapitel 3.3 vorgenommen.

3.3. Vergleich und Analyse der Algorithmen

Auf Grundlage der Implementierung (Kapitel 4) wurden Laufzeiten für verschiedene Fälle ermittelt. Um diese Zeiten zu ermitteln wurde eine Testanwendung in C# geschrieben. Die Messungen erfolgten mit den Standardeinstellungen der Algorithmen.

Anzahl GraphEntitys	**Hierarchisches Layout**	**Kräftebasiertes Layout**	**Abstands Kontroll Algorithmus**
2	10	21	10
10	29	44	11
20	93	125	11
50	263	355	43
100	461	1008	92
200	890	2645	170

Tabelle 3.3.: Laufzeitmessung bei verschiedener Anzahl an Funktionen

Bei der ersten Laufzeitmessung wurden sechs Funktionsdiagramme mit unterschiedlicher Anzahl an Funktionen verwendet. Die Anzahl der Beziehung ist dabei gleich der Anzahl der Funktionen. Die gewonnenen Werte sind in Tabelle 3.3 dargestellt. An der Abbildung 3.3 ist zu erkennen, dass die Laufzeit des kräftebasierten Algorithmus, mit zunehmender Anzahl von Funktionen, stark ansteigt. Der Anstieg des hierarchischen Algorithmus ist fast linear. Dies ist auf die geringe Anzahl an Beziehungen zurückzuführen.

Bei der zweiten Messung wurde ein Funktionsdiagramm mit zwanzig Funktionen verwendet. Dabei wurde die Anzahl der Beziehungen in sechs Schritten erhöht. Die Tabelle

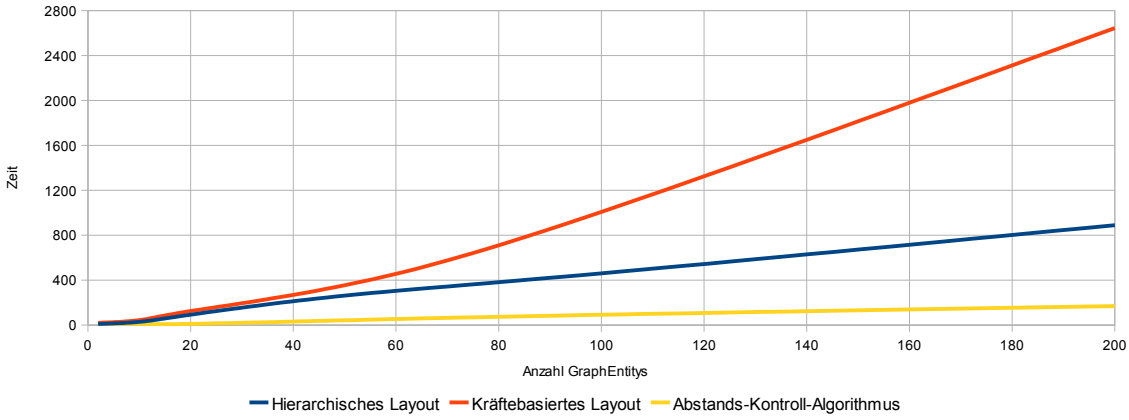

Abbildung 3.3.: Laufzeitdiagramm für Messung 1

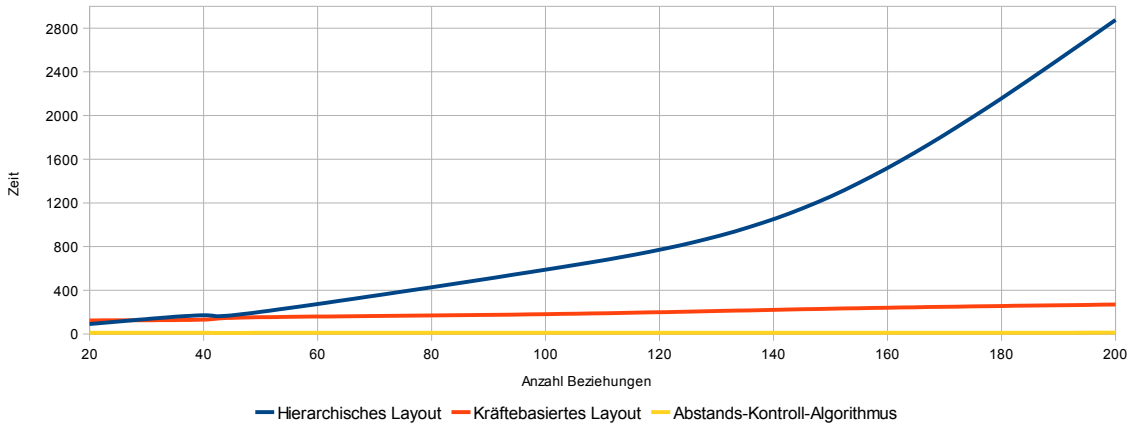

Abbildung 3.4.: Laufzeitdiagramm für Messung 2

Anzahl Beziehungen	**Hierarchisches Layout**	**Kräftebasiertes Layout**	**Abstands Kontroll Algorithmus**
20	93	125	11
40	172	132	11
50	203	154	11
100	589	182	12
150	1259	232	11
200	2875	270	13

Tabelle 3.4.: Laufzeitmessungen bei verschiedener Anzahl an Beziehungen

3.4 zeigt die Messwerte. Wie in Abbildung 3.4 zu erkennen ist, bleibt die Laufzeit des kräftebasierten Algorithmus relativ konstant. Der Aufwand für den hierarchischen Algorithmus steigt hingegen stark an. Dies liegt an der steigenden Anzahl an Kantenkreuzungen.

Es fällt auf, dass der Abstands-Kontroll-Algorithmus sehr geringe Laufzeiten hat. Dies liegt an seinem linearen Aufbau, da nur jeder Knoten einmal auf Überschneidungen geprüft und verschoben wird, benötigt dieser Algorithmus weniger Rechenzeit als die anderen.

Es kann auch festgestellt werden, dass die benötigte Rechenzeit des kräftebasierten Algorithmus stark von der Anzahl an Knoten, jedoch nicht von der Anzahl der Beziehungen abhängig ist. Der hierarchische Algorithmus hingegen benötigt bei steigender Anzahl an Knoten nur geringfügig mehr Rechenzeit. Steigt jedoch die Anzahl der Beziehungen, steigt auch die Rechenzeit stark an. Diese Gegenläufigkeit lässt sich mit dem unterschiedlichen Aufbau der Algorithmen erklären. Da beim hierarchischen Algorithmus die Knoten bei einem Durchlauf des Graphen platziert werden, erfordert dies nur geringe Rechenzeit. Das anschließende Optimieren, anhand der Beziehungen, erfordert jedoch erheblich mehr Rechenzeit. Bei dem kräftebasierten Algorithmus werden die Beziehungen hingegen als Vektoren aufgefasst und zu einem einzigen Vektor aufaddiert. Dies erfordert entsprechend wenig Rechenzeit im Vergleich zum Positionieren der Knoten. Da beim Finden der besten Position diese Vektoren auf die aktuelle Position der Knoten angewandt werden und dies so lange wiederholt wird, bis sich keine Verschiebung mehr ergibt. Da sich im praktischen Einsatz die Anzahl der Knoten und Beziehungen wahrscheinlich in der Mitte der hier getesteten Bedingungen befinden wird, kann die benötigte Rechenzeit fast vernachlässigt werden. Hier sollten keine Laufzeiten über zwei Sekunden auftreten. Es

bleibt jedoch anzumerken, dass die tatsächliche Laufzeit sehr stark von der verwendeten Hardware sowie des Betriebssystems abhängig ist.

Für die Laufzeit und das Ergebnis des kräftebasierten Algorithmus ist die Vorbedingung ein weiteres wichtiges Kriterium. Die Vorbedingung bezeichnet hierbei die Position der Knoten, bevor der Algorithmus auf diese angewandt wird. Für die Tests wurden die Knoten gleichmäßig, ohne Überschneidungen, auf der Zeichenfläche verteilt. Eine Alternative dazu ist die Knoten mithilfe des hierarchischen Algorithmus, ohne Optimierungen, zu platzieren.

Welcher Algorithmus sich für welches Diagramm am besten eignet, lässt sich nicht im allgemein sagen. Beide Algorithmen erzeugen immer ein gutes Gesamtbild. Welches das bessere ist, muss immer im Einzelfall vom Betrachter entschieden werden.

4 Konzept der Implementierung

4.1. Anforderungen

Der von CADsys entwickelte Produktkonfigurator FOD wurde auf Basis von Microsofts .Net 3.5 realisiert. Aus diesem Grund findet die Implementierung der Layout-Algorithmen in C# ebenfalls auf Basis des .Net 3.5 statt.

Die Layoutalgorithmen sollen in einer externen Bibliothek implementiert werden. Dazu wird eine Dynamic Link Library (DLL) erstellt. Die Kommunikation wird über zwei Schnittstellen (Abbildung 4.1) realisiert.

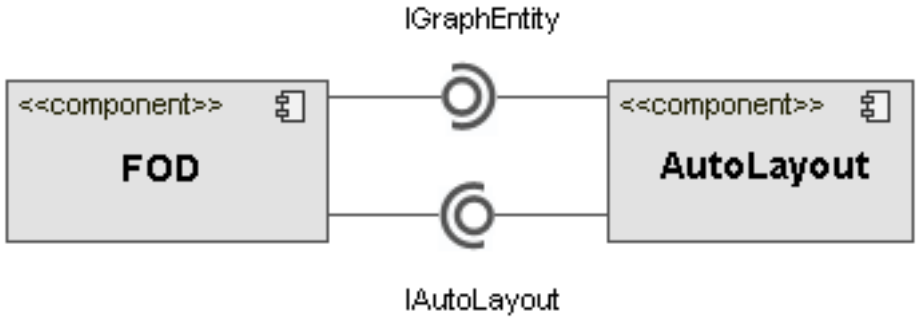

Abbildung 4.1.: Komponentendiagramm

Als Schnittstelle zur Übertragung der Daten wird in FOD das Interface IGraphEntity (Abbildung 4.2) implementiert. IGraphEntity stellt die für die Layout-Algorithmen benötigten Informationen über die Größe (getHeight() und getWidth()), Position (X und Y) und Beziehungen (getRelations()) zur Verfügung. Die Beziehung bezeichnet dabei eine Aufzählung vom Typ IEnumerable, welche Verweise auf alle direkten Nachfolger *GraphEntitys* enthält. Über das Property *IsRoot* werden die *GraphEntitys* gekennzeichnet, welche keine Vorgänger haben.

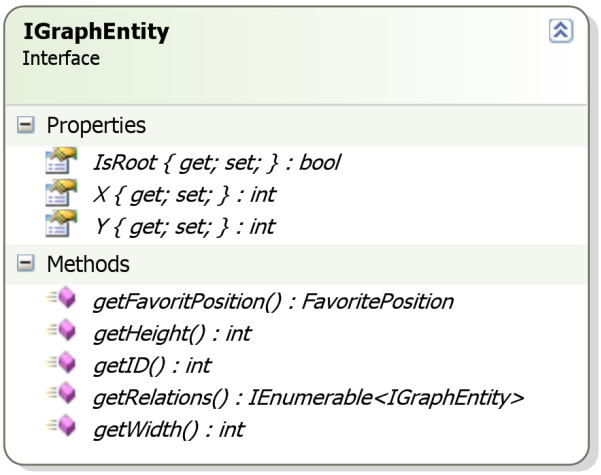

Abbildung 4.2.: Klassendiagramm von IGraphEntity

Mithilfe des Enum FavoritePosition (Abbildung 4.3) kann die zukünftige Position einzelner *GraphEntitys* beeinflusst werden. Diese Vorgabe wird, vom entsprechenden Algorithmus, nur soweit beachtet wie es innerhalb des Layouts sinnvoll ist.

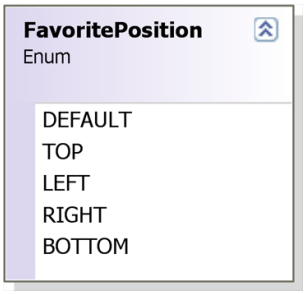

Abbildung 4.3.: Klassendiagramm von Enum FavoritePosition

Zur Steuerung der Layout-Algorithmen implementiert die AutoLayout.dll das Interface IAutoLayout (Abbildung 4.4). Für jeden Algorithmus existiert eine *create-Methode*, welche ein ILayout zurückgibt. Über das Interface ILayout wird der jeweilige Algorithmus gesteuert.

Diese offene Implementierung ermöglicht die einfache Erweiterung und Wartung der AutoLayout.dll. Fehler, die während der Verarbeitung auftreten werden als Ausnahmen über die *log4net-Bibliothek* (Kapitel 4.2.2) an FOD zurück gegeben.

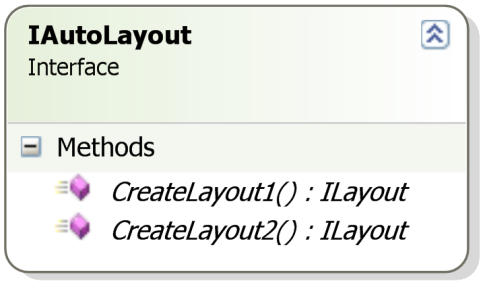

Abbildung 4.4.: Klassendiagramm von IAutoLayout

4.2. Entwurfsmuster und Klassendesign

Die zu entwickelnde AutoLayout.dll wird von außen nur über Interfaces angesprochen. Daraus ergibt sich eine sehr übersichtliche und einfache Bedienung der Algorithmen (Abbildung B.3).

Intern existiert für jeden Algorithmus eine Klasse sowie eine eigene Unterklasse von GraphEntity.

4.2.1. Verwendetete Entwurfsmuster

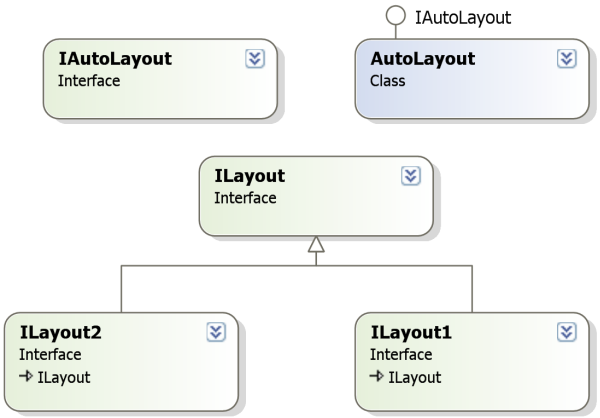

Abbildung 4.5.: Beziehung zwischen IAutoLayout und ILayout

Zum erzeugen von den Algorithmus-Objekten wird die *Factory Method* angewand [KE07, S. 24ff]. Dazu wurde die Schnittstelle *IAutoLayout* (Abbildung 4.4) definiert. Diese Schnittstelle wird von der Klasse *AutoLayout* implementiert. Durch die *create-Methoden*

entscheidet diese Klasse welche konkrete Layoutklasse instanziiert, konfiguriert und schließlich als *ILayout* zurückgegeben wird (Abbildung 4.5).

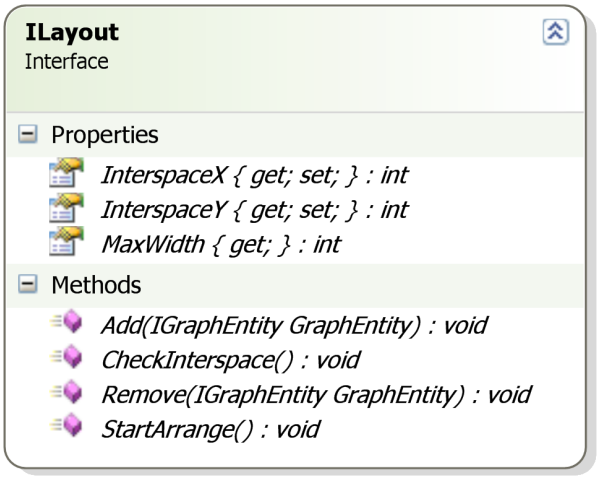

Abbildung 4.6.: Klassendiagramm von ILayout

Über die Schnittstelle *ILayout* können die verschiedenen Algorithmen angesprochen und gesteuert werden. So wird dem Algorithmus über die Properties *InterspaceX* und *InterspaceY* der minimale Abstand zwischen zwei *GraphEntitys* mitgeteilt. Mit den Methoden *Add()* und *Remove()* werden *GraphEntitys* hinzugefügt und wieder entfernt. Die *StartArrange()* führt dann den eigentlichen Layout-Algorithmus aus. Nach der Berechnung kann in dem Property *MaxWidth* die Breite des Diagramms abgerufen werden. Sollte sich nach der Berechnung die Größe einer *GraphEntity* ändern, können die Mindestabstände mithilfe der *CheckInterspace()-Methode* wieder hergestellt werden.

Von der Schnittstelle *ILayout* wird für jeden Algorithmus eine eigene Schnittstelle abgeleitet. Dies sind hier *ILayout1* für den kräftebasierenden und *ILayout2* für den hierarchischen Layout-Algorithmus. In diesen Schnittstellen werden spezielle Parameter und Funktionen, welche nur für einen speziellen Algorithmus benötigt werden, definiert. Über diese Parameter können die Algorithmen noch genauer an die zu lösenden Probleme angepasst werden. Das setzen der Parameter ist aber nicht zwingend erforderlich.

Der Vorteil dieser *Factory Method* ist das für den Benutzer der Bibliothek nur die Schnittstelle von Bedeutung ist. Dadurch können die Klassen der AutoLayout.dll beliebig ausgetauscht werden, ohne das eine Veränderung an FOD nötig wäre. Auch das Hinzufügen von neuen Layout-Algorithmen ist mit diesem Konzept sehr einfach zu realisieren. Für

jeden neuen Algorithmus muss nur eine neue *create-Methode* angelegt und wenn nötig eine eigene Ableitung von *ILayout* erstellt werden.

Um die einheitliche Steuerung der Algorithmen zu ermöglichen, ist neben der Schnittstelle *ILayout* auch ein einheitlicher Datentyp nötig. Dafür wurde die Schnittstelle *IGraphEntity* (Abbildung 4.2) definiert. Da jeder Algorithmus noch weitere zusätzliche Anforderungen an den Datentyp stellt, wurde für jeden eine eigene Implementation von *IGraphEntity* erstellt (Kapitel 4.2.3 und 4.2.4).

Ein weiteres Entwurfsmuster ist die *Strategy* nach Gang of Four (GoF). Jeder Algorithmus wurde in einer Klasse gekapselt. Damit diese Klassen möglichst von jedem Layout verwendet werden können, besitzen sie alle die *run-Methode*. Dadurch wird das Layout unabhängig vom eigentlichen Algorithmus und kann einfacher erweitert werden.

4.2.2. Ausnahmebehandlung

Exception	**Beschreibung**
NoStartArrangeBeforeException	Exception wird bei fehlendem StartArrange() Aufruf ausgelöst
NoRootElementFoundException	Exception wird ausgelöst wenn kein Element als IsRoot gekennzeichnet wurde
IncorrectRelationException	Exception wird bei fehlerhaften Beziehungen ausgelöst

Tabelle 4.1.: Exceptions von AutoLayout.dll

Treten zur Laufzeit eines Algorithmus Fehler auf, werden diese als Exception zurückgegeben. Tabelle 4.1 zeigt alle eigenen Exceptions der AutoLayout.dll.

Neben den Exception gibt es noch andere Daten, die zur Laufzeit für den Benutzer der AutoLayout.dll interessant sind, zum Beispiel bei der Fehlersuche. Diese Informationen können zum Beispiel Anzahl der *GraphEntitys*, Anzahl der benötigten Iterationen oder auch Informationen aus der *CheckInterspace*-Methode sein. Um diesen dem aufrufenden Programm mitzuteilen, wurde die *Log4Net* Bibliothek [Apa09] verwendet.

Log4Net ist eine Portierung des für Java entwickelten *Log4J* auf die .NET Plattform. Es besitzt 5 *logging Level* (Fatal, Error, Warn, Info, Debug) und die Möglichkeit die

Ausgabe auf verschiedene Medien zu lenken, zum Beispiel Ausgabe im Visual Studio, in Dateien oder Datenbanken. Gesteuert wird dies durch eine XML-Datei, somit ist auch ein logging im produktiven Einsatz möglich.

4.2.3. Hierarchisches Layout

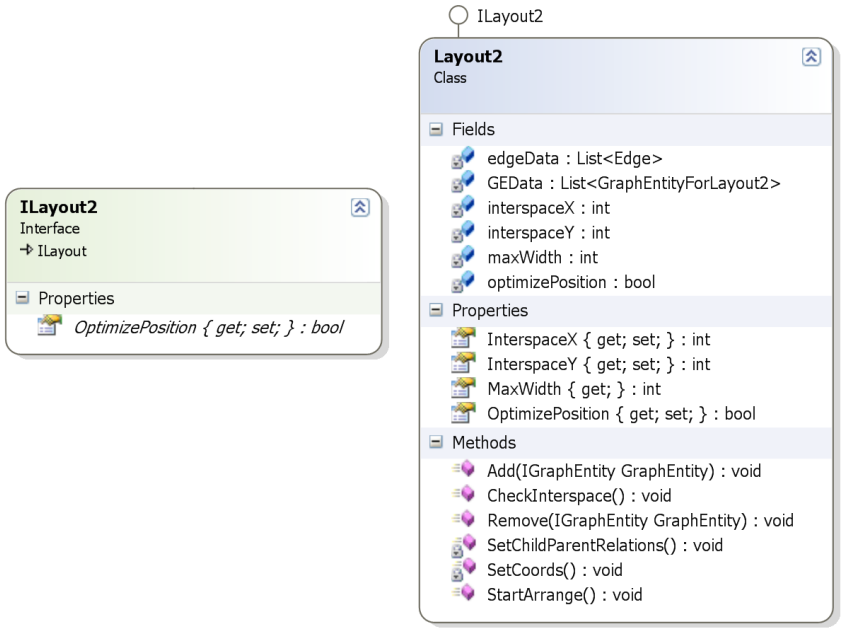

Abbildung 4.7.: Klassendiagramm von ILayout2 und Layout2

Alle öffentlichen Methoden werden durch die Schnittstelle *ILayout2* sowie *ILayout* vorgegeben. Die Klasse *Layout2* stellt die Implementierung dieser dar. Für die effiziente Abarbeitung des hierarchischen Layout-Algorithmus werden noch zwei weitere lokale Felder benötigt. Dies ist zum einen eine Aufzählung aller Beziehungen der *GraphEntitys* und zum anderen eine Aufzählung der *GraphEntitys* selbst. Für beide wird die generische Klasse List verwendet, welche vom .NET Framework aber Version 2.0 zur Verfügung gestellt wird.

Das Property *OptimizePosition* steuert, ob eine Optimierung der Position innerhalb einer Schicht vorgenommen werden soll.

Für die Beziehungen wird die Klasse *Edge* (Abbildung 4.8) verwendet. Für jede Beziehung zwischen zwei *GraphEntitys* wird eine Instanz von *Edge* erzeugt und zur Liste

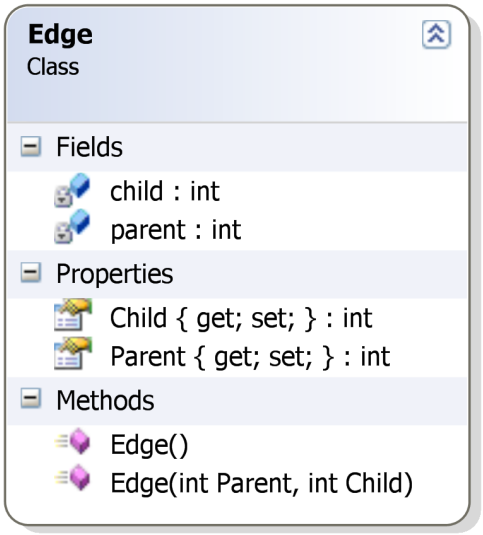

Abbildung 4.8.: Klassendiagramm von Edge

edgeData hinzugefügt. Neben den Beziehungen und den *GraphEntitys* benötigt der Algorithmus noch weitere Felder, in denen Informationen zur Berechnung abgelegt werden. Dafür wurde die Klasse *GraphEntityForLayout2* (Abbildung B.2) definiert.

GraphEntityForLayout2 beinhaltet ein Feld für das *IGraphEntity* Objekt und implementiert aus Kompatibilität auch die Schnittstelle *IGraphEntity*, welche auf das entsprechende Objekt verweist. Dies wird erweitert durch die Properties: *BarycenterValue* für die Kantenkreuzungsreduzierung, *Level* und *Position* für die Rastereinteilung sowie *VirtualEntity* zum Kennzeichnen von virtuellen Knoten.

Im Verlauf des Algorithmus ist es nötige, die Liste der *GraphEntitys* zu sortieren. Dies wird mit der Sortierfunktion der generischen Klasse List realisiert. Als Algorithmus verwendet das .NET Frameworke *QuickSort* (Aufwand $O(n \log n)$). Zur Realisierung wird die Schnittstelle *IComparable* implementiert sowie entsprechende *CompareBy-Methoden*.

Anhand der *edgeData* Liste werden jeder *GraphEntity* auch ihre Vorgänger zugewiesen. Dazu enthält *GraphEntityForLayout2* die Funktionen: *AddInversRelation()*, *GetInversRelation()* und *RemoveInversRelation()*. Diese umgekehrten Beziehungen sind für die Kantenkreuzungsreduzierung notwendig.

Nach dem Hinzufügen der *GraphEntitys*, über die *Add()* Methode von *ILayout2*, wird die Anordnung über die *StartArrange()-Methode* gestartet. Als Erstes werden von der

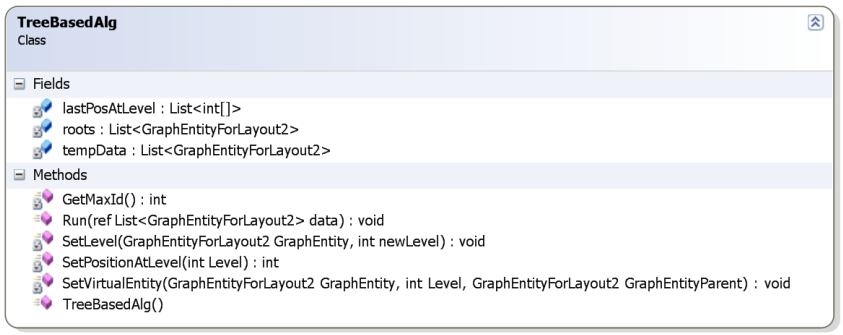

Abbildung 4.9.: Klassendiagramm von TreeBasedAlg

SetChildParentRelations()-Methode die umgekehrten Beziehungen gesetzt und anschließend die Klasse *TreeBasedAlg* instanziert. Ihrer *Run()-Methode* wird eine Referenz auf die Liste mit *GraphEntityForLayout2* Objekten übergeben. Im Anschluss werden, entsprechend des unter Kapitel 3.2.1 beschriebenen Algorithmus, die neuen Koordinaten bestimmt.

Die Implementierung der *CheckInterspace-Methode* ist identisch mit dem in Kapitel 3.2.3 vorgestellten Algorithmus. Da sich alle *GraphEntity* Elemente in einem Raster befinden, müssen nur die Abstände zum nächsten rechten und zum nächsten unteren überprüft werden. Ist der Abstand nach rechts zu gering, wird die gesamte Spalte entsprechend angepasst.

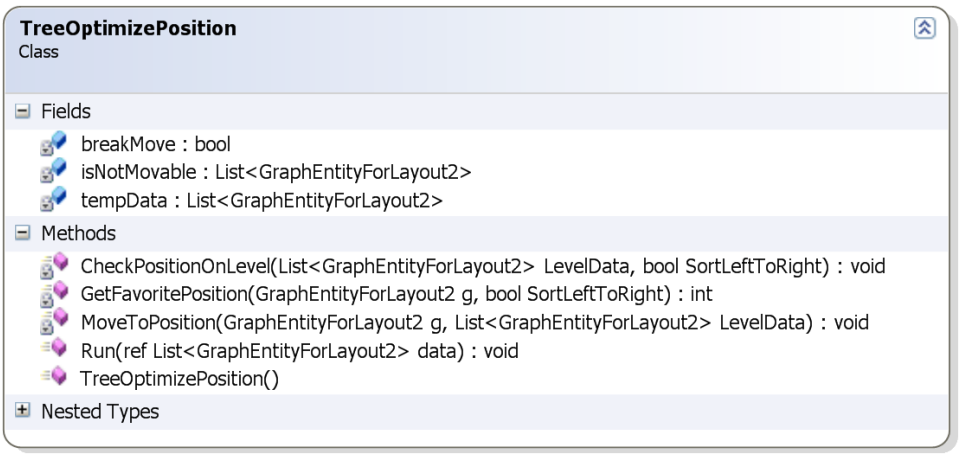

Abbildung 4.10.: Klassendiagramm von TreeOptimizePosition

Die Klasse *TreeOptimizePosition* (Abbildung 4.10) stellt die Implementierung der unter Kapitel 3.2.1 vorgestellten Optimierung der Position innerhalb einer Schicht dar. Die

zwei Listen (*isNotMoveable*, *tempData*) werden zusammen mit der Variablen *breakMove* für das Prüfen und Verschieben der *GraphEntitys* verwendet. Die Optimierung wird über die *Run()-Methode* gestartet, dieser wird eine Referenz auf die Liste mit *GraphEntityForLayout2* Objekten übergeben. Die *GetFavoritePosition()-Methode* ermittelt die beste Position, über den Parameter *SortLeftToRight* wird die Sortierrichtung angegeben. Durch den rekursiven Aufruf der *CheckPositionOnLevel()-Methode* wird überprüft, ob diese Wunschposition möglich ist. Die entsprechende Verschiebung der Knoten erledigt die *MoveToPosition() Methode*.

4.2.4. Kräftebasiertes Layout

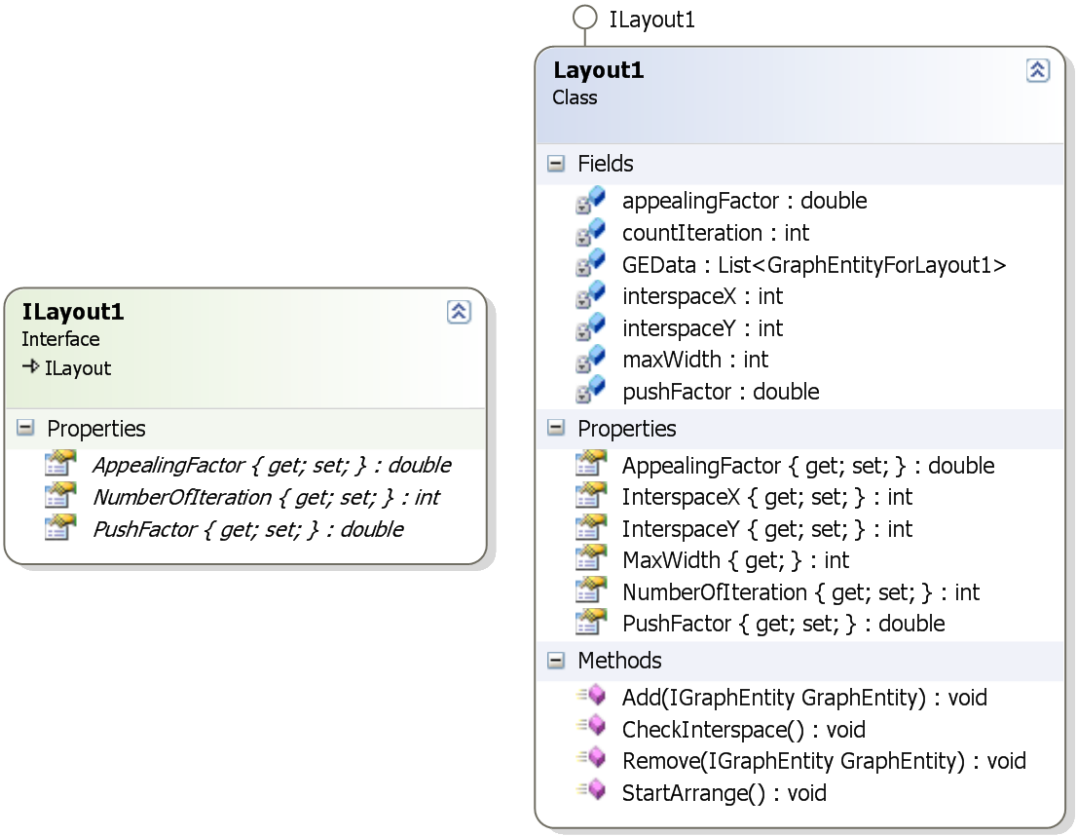

Abbildung 4.11.: Klassendiagramm von ILayout1 und Layout1

Bei dem kräftebasierten Layout werden die öffentlichen Methoden und Properties in *ILayout1* definiert. Da dies ein Unterinterface von *ILayout* ist müssen auch dessen Member implementiert werden. Die Klasse *Layout1* (Abbildung 4.11) besitzt eine generische

Liste vom Typ *GraphEntityForLayout1*. Diese Liste enthält die anzuordnenden *GraphEntity* Elemente. Durch *ILayout1* wird ein weiteres Property (*NumberOfIteration*) definiert. Dies ist vom Typ Integer und gibt die maximal mögliche Anzahl an Durchläufen des kräftebasierenden Algorithmus an.

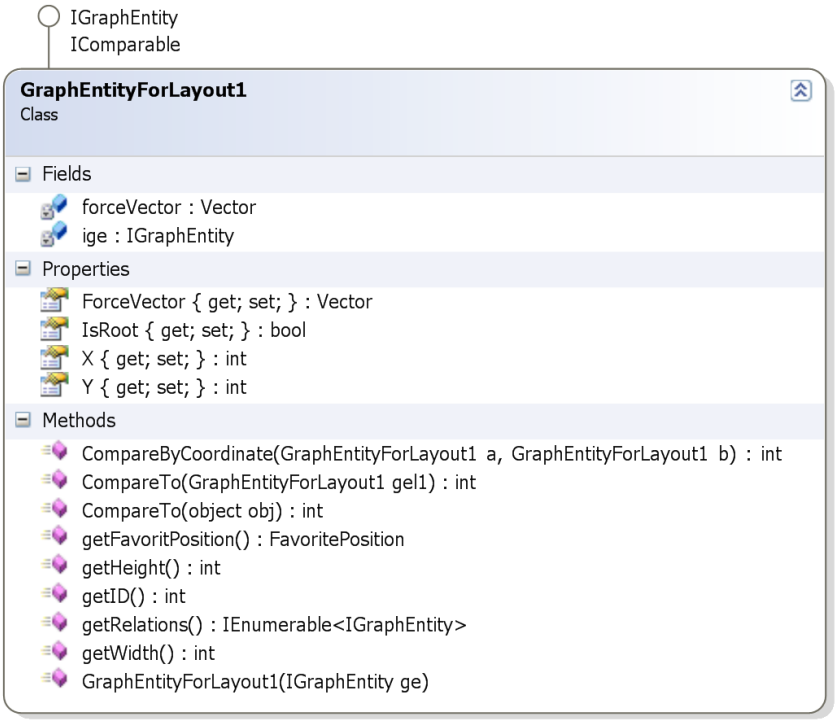

Abbildung 4.12.: Klassendiagramm von GraphEntityForLayout1

GraphEntityForLayout1 (Abbildung 4.12) besitzt ein Feld für das *IGraphEntity* Objekt und implementiert aus Kompatibilität auch die Schnittstelle *IGraphEntity*, welche auf das entsprechende Objekt verweist. Erweitert wird es durch das Property *ForceVector* vom Typ *System.Windows.Vector*. In diesem Vektor werden die auf das *GraphEntity* wirkenden Kräfte gespeichert. Nach jedem Durchlauf des Algorithmus wird das *GraphEntity* um diesen Vektor verschoben.

Für die *CheckInterspace()-Methode* ist eine Sortierung der *GraphEntitys* nötig. Zum Sortieren wird die *Sort-Methode* der generischen Liste verwendet. Dazu implementiert *GraphEntityForLayout1* die Schnittstelle *IComparable* und die Methode *CompareByCoordinate*.

Über die *StartArrange()-Methode* wird die Neuanordnung der *GraphEntity* Elemente angestoßen. Dabei wird die Klasse *ForceBasedAlg* (Abbildung 4.13) instanziert und der

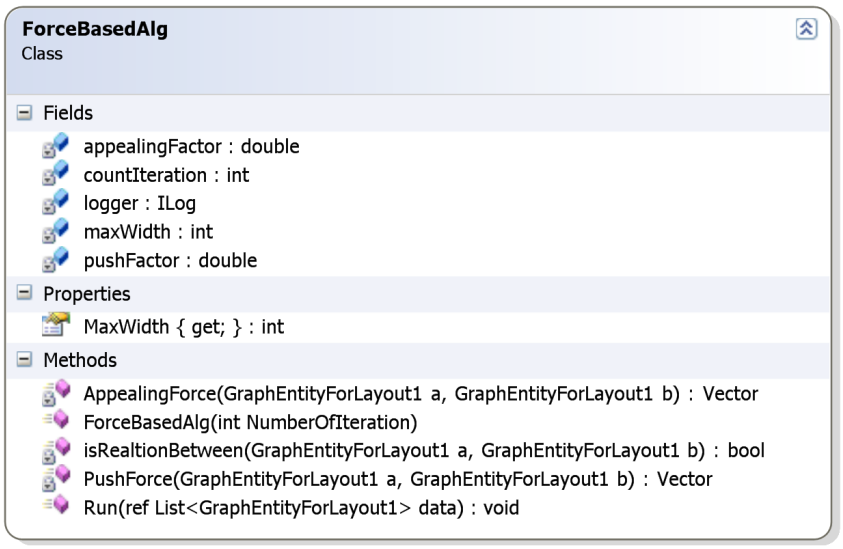

Abbildung 4.13.: Klassendiagramm von ForceBasedAlg

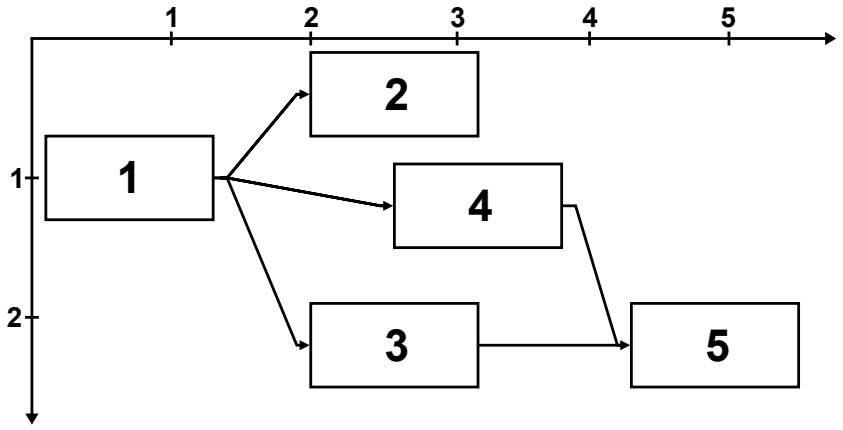

Abbildung 4.14.: Sortierung nach Koordinaten

Algorithmus durch Aufruf der *Run()* Methode gestartet. Der Ablauf des Algorithmus erfolgt nach dem in Kapitel 3.2.2 beschriebenem Verfahren. Für die anziehenden Kräfte ist die *AppealingForce()-Methode* und für die abstoßenden Kräfte die *PushForce()-Methode* zuständig. Die *isRealtionBetween()-Hilfsmethode* überprüft, ob zwischen zwei *GraphEntitys* eine Beziehung also anziehende Kräfte existieren. Über das Property *MaxWidth* kann ständig die maximale Breite des Diagramms ausgelesen werden.

Die Implementierung der *CheckInterspace-Methode* weicht leicht von dem in Kapitel 3.2.3 vorgestellten Algorithmus ab. Diese Abweichung ist notwendig, da es nicht möglich ist die *GraphEntitys* in ein Raster einzuteilen. Aus diesem Grund werden die *GraphEntitys* nach ihren Koordinaten sortiert (Abbildung 4.14). Der Koordinatenursprung liegt dabei in der linken, oberen Ecke. Die Sortierung findet aufsteigend von links nach rechts (X-Koordinate) und aufsteigend von oben nach unten (Y-Koordinate) statt. Beim Überprüfen der Mindestabstände wird jedes *GraphEntity* mit allen, in der Sortierung, folgenden *GraphEntitys* geprüft. Dies erfordert einen Aufwand von $O(n * \frac{n-1}{2})$.

5 Zusammenfassung

5.1. Darstellung der wichtigsten Ergebnisse

Im Verlauf der Diplomarbeit wurden fünf Graphen-Layout-Algorithmen untersucht. Dabei konnte festgestellt werden, dass es nicht den perfekten Algorithmus gibt. Jeder der untersuchten Algorithmen hatte sowohl Vorteile als auch Nachteile (Kapitel 2.4). Daraus ergibt sich, dass es auch für Funktionsstrukturen nicht den einen perfekten Algorithmus gibt. Hierarchische Anordnungsverfahren lassen sich relativ einfach auf Funktionsstrukturen anwenden. Dies liegt an den Grundeigenschaften, dass es immer eine Anfangsfunktion und eine Endfunktion gibt und das alle Beziehungen zwischen den einzelnen Funktionen gerichtet sind. Ein auf kräftebasierendes Verfahren eignet sich hingegen besonders bei größeren Strukturen. Dies liegt daran, dass hier zusammengehörige Funktionen nah beieinander sind und Funktionen, ohne Beziehungen zueinander, weit entfernt. Aus diesen Gründen wurde für beide Verfahren je ein Algorithmus entwickelt und implementiert. (Kapitel 3.3)

Die Implementierung wurde mit der Hilfe von Entwurfsmustern und den Werkzeugen des Frameworks .NET erfolgreich realisiert. Dabei entstand die AutoLayout.dll, welche sich problemlos in FOD integrieren lässt. Die korrekte Funktionsweise der Algorithmen wurde mit einem speziell dafür entwickelten Testprogramm überprüft. Die verwendete offene Implementierung über Interfaces ermöglicht eine einfache Wartung und Erweiterung der Algorithmen und stellt somit eine gute Lösung der Aufgabenstellung dar.

5.2. Ausblick

Die in dieser Diplomarbeit vorgestellten Algorithmen wurden erfolgreich umgesetzt. Sie könnten aber noch auf verschiedene Weise optimiert werden. Zum einen ist es möglich, die Implementierung der Algorithmen, in Bezug auf geringere Laufzeit und Speicherbedarf, zu optimieren. Darauf wurde hier jedoch verzichtet, um eine Weiterentwicklung der Algorithmen nicht zusätzlich zu erschweren. Eine weitere Optimierung wäre im Bereich der grafischen Ergebnisse wahrscheinlich möglich.

Eine Möglichkeit für die Verbesserung des kräftebasierenden Algorithmus wäre es den einzelnen, anziehenden Kräften verschiedene Stärken zuzuweißen. Dabei könnte die Anzahl da gemeinsamen Parameter die Stärke bestimmen. Somit wären zum Beispiel Funktionen die durch fünf Parameter miteinander verbunden sind näher beieinander als Funktionen, die nur mit einem Parameter verbunden sind.

Die hier vorgestellten Algorithmen weisen den Funktionen neue Koordinaten zu. Dabei werden die Positionen der Beziehungen (Kanten) nicht näher betrachtet. Sollte ein Abknicken der Kanten nötig sein, muss dies beim Zeichnen in FOD erfolgen. Eine Überlegung wäre es, auch die Position dieser Kanten durch die AutoLayout.dll bestimmen zu lassen. Dafür müssten aber Informationen über die Art der Darstellung in FOD den Algorithmen mitgeteilt werden. Dies würde aber dazu führen, dass die Bibliothek stärker an FOD gebunden und somit nicht mehr so leicht bei anderen Programmen oder Problemen eingesetzt werden kann.

Ähnliches gilt auch in Bezug auf die Rotation von ganzen Graphen und beim Auslassen einzelner Funktionen. Dies muss momentan außerhalb der Bibliothek erfolgen. Hier wäre es für die Zukunft auch eine Überlegung wert, ob diese Funktionen mit integriert werden sollen.

An einigen Stellen dieser Diplomarbeit wurden für Lösung von Problemen verschiedene heuristische Algorithmen verwendet. Verbesserungen bei diesen könnten auch eine Verbesserung bei den jeweiligen Layout-Algorithmen zur Folge haben.

A Screenshot

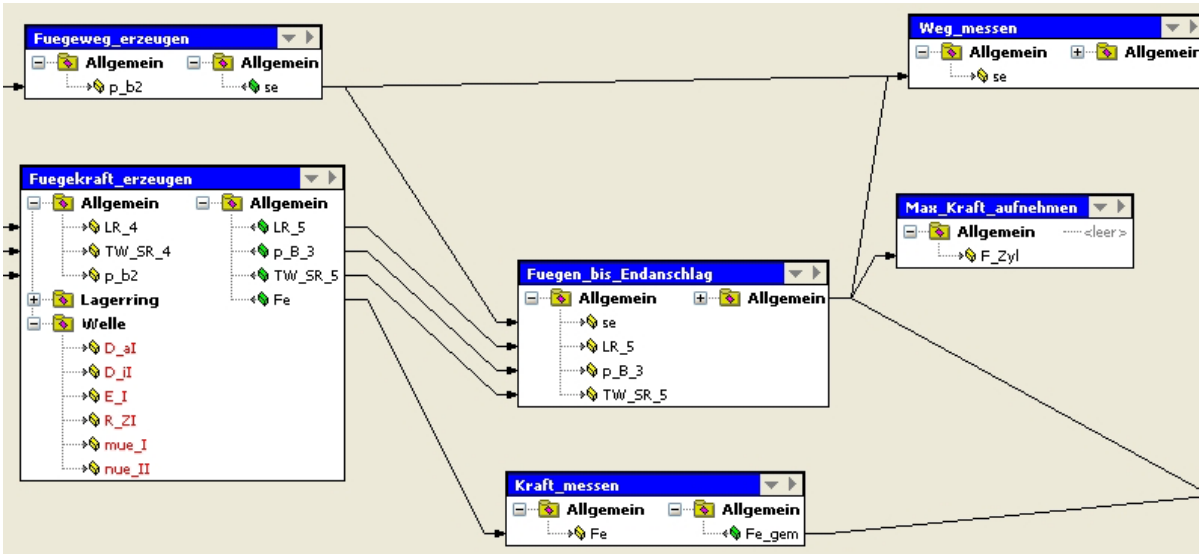

Abbildung A.1.: Screenshot eines Produktfunktiondiagramms

B Klassendiagramme

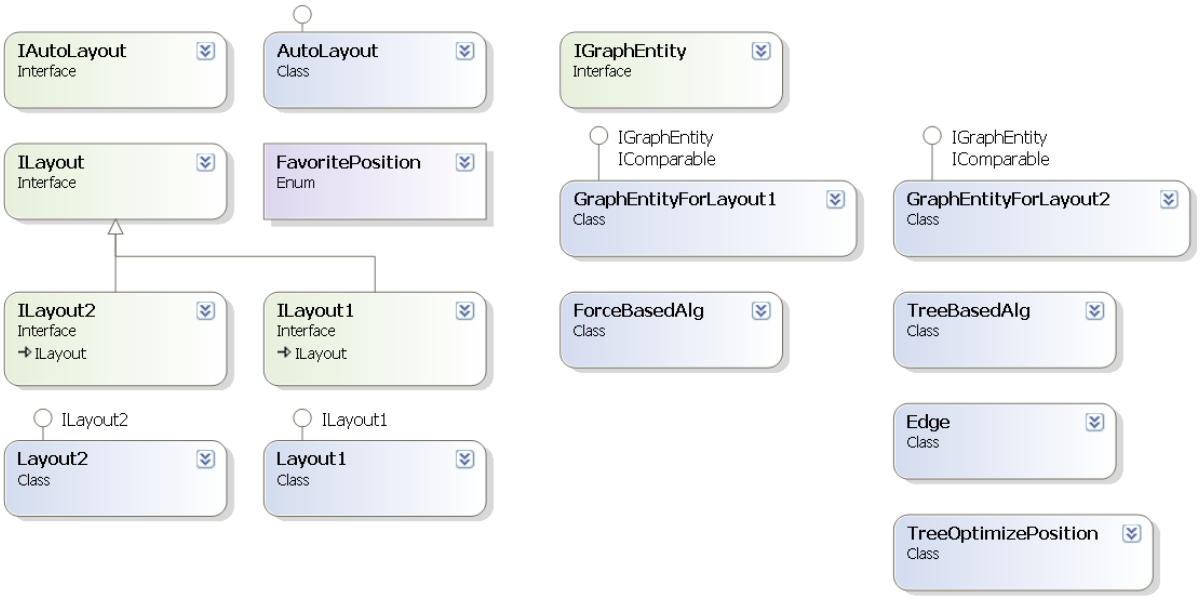

Abbildung B.1.: Klassenübersicht

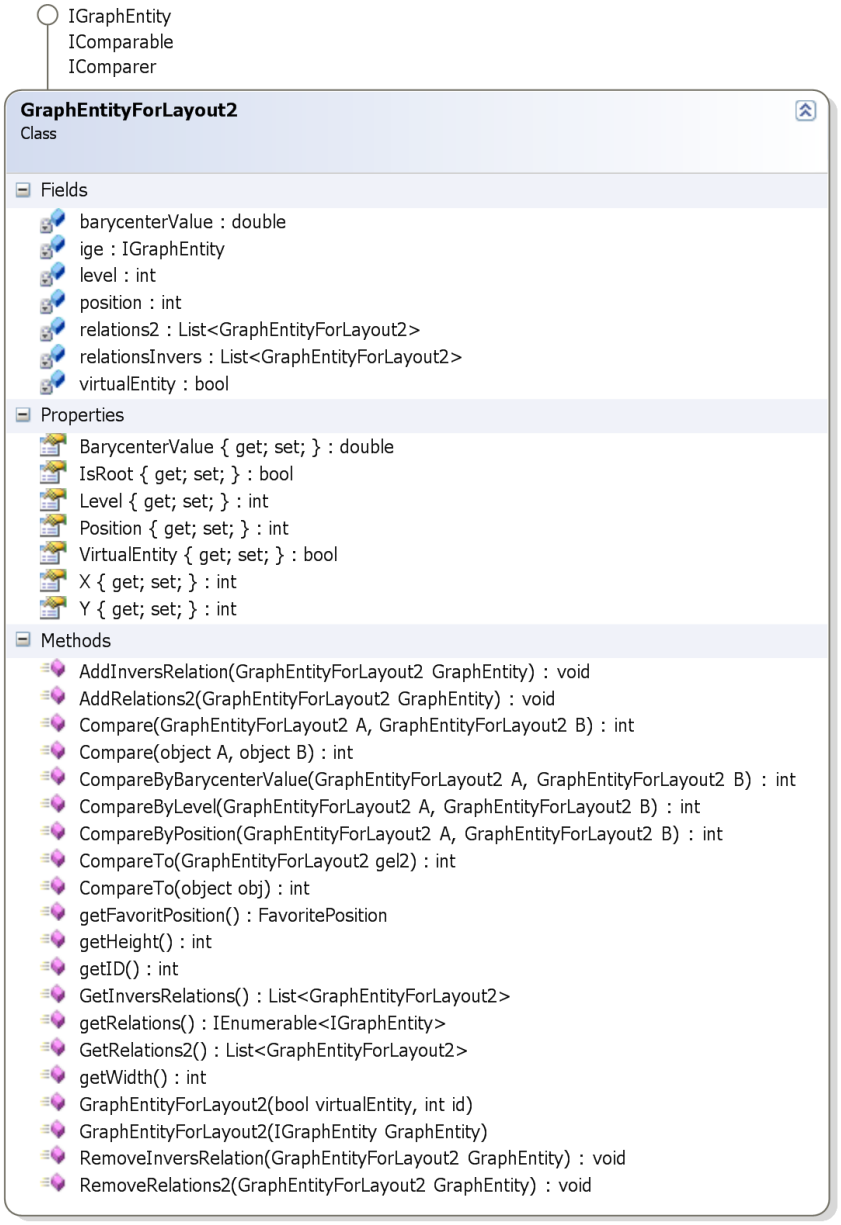

Abbildung B.2.: Klassendiagramm von GraphEntityForLayout2

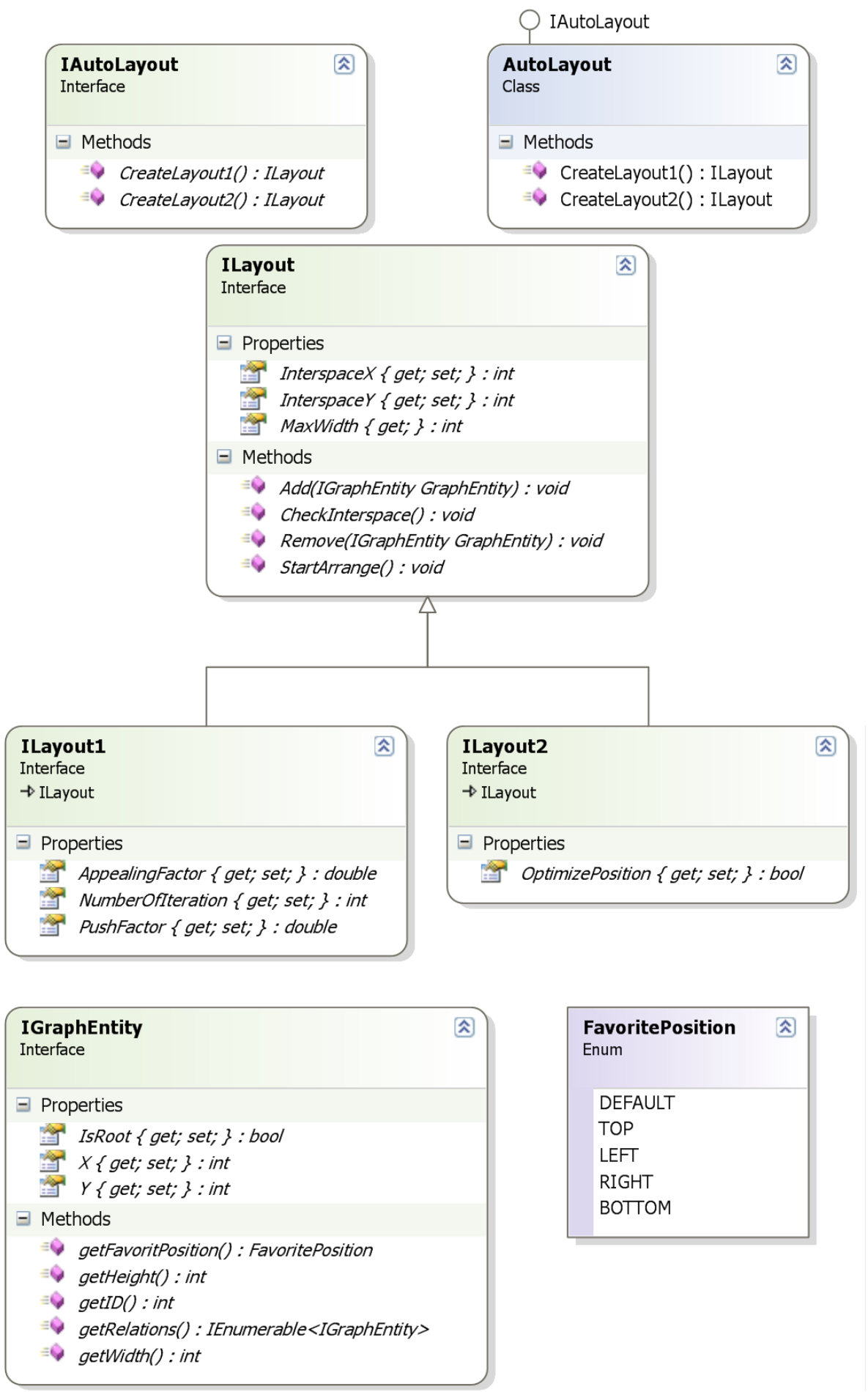

Abbildung B.3.: Externe Schnittstellen von AutoLayout.dll

Literaturverzeichnis

[Apa09] Apache Software Foundation: *Log4Net*. `http://logging.apache.org/log4net`, Oktober 2009.

[CLRS07] Cormen, Thomas, Charles Leiserson, Ronald Rivest und Clifford Stein: *Algorithmen - Eine Einführung*. Oldenbourg, 2. Auflage, 2007.

[DH02] David Harel, Yehuda Koren: *A Fast Multi-Scale Method for Drawing Large Graphs*. Journal of Graph Algorithms and Applications, 2002.

[EMR91] Edward M. Reingold, Thomas M. J. Fruchterman: *Graph Drawing by Force-directed Placement*. SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 21(1 1), 1129-1164, November 1991.

[GP08] Gustav Pomberger, Heinz Dobler: *Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung*. Pearson Education, 2008.

[KE07] Karl Eilebrecht, Gernot Starke: *Patterns kompakt*. Elsevier Spektrum Akademischer Verlag, 2007.

[KK89] Kamada, T. und S. Kawai: *An algorithm for drawing general undirected graphs*. Elsevier North-Holland, Inc., VOL. 31 issue 1, 7-15, 1989.

[KW01] Kaufmann, M. und D. Wagner: *Drawing Graphs - Methods and Models*. Springer, 2001.

[MJ97] Michael Jünger, Petra Mutzel: *2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms*. Journal of Graph Algorithms and Applications, 1997.

[Mut07] Mutzel, Professor Dr. Petra: *Zeichnen gerichteter Graphen.* Lehrstuhl für Algorithm Engineering Fachbereich Informatik LS11, Universität Dortmund, 2007.

[PE94] Peter Eades, Nicholas C. Wormald: *Edge crossings in drawings of bipartite graphs.* Algorithmica, 11:379-403, 1994.

[PM02] P Mutzel, W Barth, M Jünger: *Simple and efficient bilayer cross counting.* LNCS, 2528:130-141, 2002.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mittweida, 05.11.2009 Daniel Neef