INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München

LayLa – Eine Layout-Beschreibungssprache für robuste Layouts zur Unterstützung von mentalen Modellen

Jens Abraham

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Abgabe der Arbeit:	28. Februar 2001
Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Dr. Norbert Eisinger

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 Ziel

Ziel dieser Projektarbeit ist es, eine Grundlage für *ergonomische Struktur-Visualisierungs-Werkzeuge* zu schaffen. Sie besteht aus dem Entwurf einer Layout-Beschreibungssprache mit dem Namen `LayLa` und einer prototypischen Implementierung eines Algorithmus, der aus Layout-Beschreibungen in `LayLa` ein Layout berechnet.

1.2 Einführung in das Thema

1.2.1 Visualisierungs-Arten

1.2.1.1 Analoge Visualisierung Meist versteht man unter Visualisierung das Umwandeln von Zahlenkolonnen in übersichtliche Diagramme, wie etwa Tortendiagramme oder in Punkte in 2- bzw. 3-dimensionalen Koordinatensystemen.

Der Vorteil dieser Methoden liegt im wesentlichen darin, daß eine schwer überschaubare Menge an numerischer Einzelinformation in ein ganzheitlich aufnehmbares Bild aufbereitet wird. In diesem Bild können dem Menschen Zusammenhänge deutlich werden, die sonst im Verborgenen geblieben wären.

Man könnte dieses Verfahren als Kontrast zum Begriff Digitalisieren auch Analogisieren nennen. Das erzeugte Bild liegt natürlich auch digital auf einem Speichermedium vor. Insofern ist der Vergleich und damit die Bezeichnung nicht ganz zutreffend. Da das Bild im Gegensatz zu einer Abfolge von Zahlen vom Menschen zunächst ganzheitlich und erst in weiteren kognitiven Verarbeitungsprozessen symbolisch aufgenommen wird, ist der Begriff analog wiederum zutreffend.

Die Visualisierung hat in diesem Zusammenhang zwei Aufgaben. Zum einen dient sie der Illustration von bekannten Erkenntnissen. Zum anderen ist sie ein Medium, um die Fähigkeit des Menschen zur analogen bildlichen Informationsverarbeitung für das Finden neuer Erkenntisse nutzbar zu machen.

Ein Anwendungsbereich dieser Methoden ist beispielsweise das Datamining. Hier wird versucht, aus sehr umfangreichen und heterogenen Datenbanken verborgene Information zu gewinnen. Das Finden von geeigneten Methoden insbesondere für die letztgenannte Aufgabe ist Gegenstand aktueller Forschung (z.B. bei der Lehr- und Forschungs-Einheit Datenbanksysteme).

Derartiger Forschung liegt die Überzeugung zugrunde, daß die Fähigkeiten des Menschen, komplexe Zusammenhänge zu finden, denen des Computers in einigen Bereichen überlegen sind.

Dieser Art der Visualisierung liegt in der Regel das Erstellen eines Funktions-Graphen im mathematischen Sinne zugrunde. Ich nenne sie analoge Visualisierung.

1.2.1.2 Struktur-Visualisierung Seit der allgemeinen Verfügbarkeit von großformatiger Computergrafik bekommt eine weitere Form der Visualisierung zunehmend Bedeutung. Hierbei werden nicht-lineare sprachliche Strukturen visualisiert. Mit nicht-linearen sprachlichen Strukturen sind natürliche Sprachen genauso gemeint wie formale Sprachen.

Beispielsweise steht hinter *Hypertext* die Bestrebung, die nicht-linearen Sinn-Zusammenhänge eines längeren und zusammenhängenden Textes (Artikel, Buch, Lexikon, etc.) durch Verbindungen (Links) zwischen kleineren Teiltexten abzubilden. Seit dem ersten Aufkommen der Hypertext-Philosophie Anfang der 80-er Jahre und sehr verstärkt durch das WWW, das im Grunde ein weltweiter und unkontrolliert wachsender Hypertext ist, werden gut gestaltete Übersichts-Diagramme (Navigations-Diagramme) als Orientierungs- und Überblick-Medium nötig.

In der Pädagogik gibt es ebenfalls ein vergleichbares Bestreben, linearen Text in nicht-lineare Diagramme zu übersetzen. Zur Erarbeitung eines Textes wird als Hilfsmittel eine sogenannte *Konzept-Map* zum Beispiel seiner Argumentations-Struktur erstellt. Dieses Verfahren wird etwas vereinfacht als strukturierte Ergebnissicherung in der kollektiven Ideenfindung eingesetzt (u.a. mit der Bezeichnung Mindmap).

Auch bei Programmiersprachen und bei der Verwaltung großer Software-Systeme erlangen Übersichts-Diagramme zunehmend Stellenwert. Wie zum Beispiel bei *Entity-Relationship-Diagrammen*, die bei der Modellierung von Datenbanken genutzt werden. Die Software-Entwurfssprache *UML* entstand von Anfang an in einer diagrammatischen Notation. Seit längerer Zeit wird auch an der Erforschung *visueller Programmiersprachen* gearbeitet. Wesentliches Merkmal dieser Sprachen ist es, daß sie im Gegensatz zu herkömmlichen Sprachen keine lineare Abfolge von Zeichen sind, sondern aus einem Netzwerk bzw. einer Schachtelung von Grafik-Elementen mit meist metaphorischer Bedeutung bestehen.

Die Struktur, die all diesen Beispielen zugrunde liegt, wird in dieser Projektarbeit im Gegensatz zu der linearen Struktur von Texten eine *allgemeine Struktur* genannt.

Definition: Eine *allgemeine Struktur* ist ein gerichteter Graph im Sinne der Graphentheorie.

Definition: Eine *Struktur-Visualisierung* ist ein Bild einer allgemeinen Struktur, das für den Zweck der besseren Rezeption der Semantik einer Instanz einer allgemeinen Struktur im Kontext einer Anwendung durch den Menschen generiert wird.

Alle genannten Beispiele für Struktur-Visualisierungen können nicht durch konventionelle textuelle oder tabellenorientierte Layout-Algorithmen bzw. Werkzeuge erstellt werden (siehe Abschnitt: Visualisierungs-Werkzeuge).

Mit dieser Definition sind textuelle Daten ein Sonderfall einer allgemeinen Struktur, da sie aus einer Abfolge von Zeichen bestehen, so daß der zugehörige Graph die spezielle Gestalt einer Kette ohne Verzweigungen hat. Sonderfälle dieser Art werden im Rahmen dieser Arbeit nicht explizit betrachtet, da netzartige Strukturen im Vordergrund stehen.

1.2.2 Kognitionspsychologische Aspekte

Man erhofft sich von Visualisierungen ein effizienteres Arbeiten als mit textuellen Sprachen.

So faßt Wolfgang Schnotz in seinem Beitrag „Wissenserwerb mit Diagrammen und Texten“ [5]den heutigen Konsens der Kognitionspsychologie zusammen.

> Das Verstehen eines Texts oder einer bildhaften Darstellung heißt letztlich immer, eine mentale Repräsentation des jeweils dargestellten Sachverhalts zu konstruieren. ... sowohl beim Verstehen von Texten als auch beim Verstehen von Bildern [werden] multiple mentale Repräsentationen gebildet ....
>
> (Schnotz, Seite 89/90)

Diese mentale Repräsentationen sind zum einen *propositionale Repräsentationen* und zum anderen *mentale Modelle.*

Propositionale Repräsentationen sind komplexe mentale Symbole, die ähnlich wie Sätze einer natürlichen Sprache nach syntaktischen Regeln zusammengesetzt werden. Eine propositionale Repräsentation ist also eine Beschreibung des repräsentierten Gegenstandes in einer hypothetischen mentalen Sprache. Die propositionale Repräsentation kann somit weniger durch Visualisierung unterstützt werden.

Im Gegensatz dazu ist ein mentales Modell eine Vorstellung, die in einer Struktur- oder Funktions-Analogie zu dem dargestellten Gegenstand steht. Die Struktur-Visualisierung kann beim Leser das Entstehen eines struktur-analogen mentalen Modells unterstützen. Umgekehrt kann beim Erstellen einer Struktur-Visualisierung das mentale Modell des Autors als Vorlage dienen.

Schnotz hebt hervor, daß die Kombination von Text und Visualisierung einen wesentlichen Vorteil beim Wissenserwerb bringt. Dies führt er darauf zurück, daß durch die Kombination beide Klassen von mentaler Repräsentation unterstützt werden.

> Zwischen einer propositionalen Repräsentation und einem mentalen Modell besteht die gleiche Beziehung wie zwischen einer sprachlichen Beschreibung und der Darstellung eines Sachverhalts durch ein realistisches Bild oder durch ein Diagramm.
>
> (Schnotz, Seite 91)

In der Vorstellung von Schnotz werden die Visualisierungen dem Text beigeordnet. Dies ist möglicherweise darauf zurückzuführen, daß Schnotz den Begriff der Visualisierung sehr umfassend versteht. Mit den Termini dieser Projektarbeit ausgedrückt umfaßt der Begriff der Visualisierung bei Schnotz sowohl die analoge als auch die digitale Visualisierung.

Da sich diese Projektarbeit nur auf Struktur-Visualisierung beschränkt, kann die bloße Beiordnung auch als unnötige Einschränkung betrachtet werden. Ich vertrete die Meinung, daß insbesondere die computergestützte Struktur-Visualisierung geeignet ist, eine Integration von Text und Visualisierung zu erzielen. Z.B. indem verschiedene Text-Teile durch räumliche Relationen miteinander in Beziehung gesetzt werden. Durch die Integration wird eine Doppel-Repräsentation (Text+Visualisierung) vermieden. Dafür kommt es zu einer Teil-Ersetzung von textueller Repräsentation durch eine visuelle Repräsentation. Trotzdem werden beide mentale Repräsentationen unterstützt.

Für das Verstehen von Visualisierungen allgemein und Struktur-Visualisierungen insbesondere ist nach Schnotz ein Strukturabbildungs-Prozeß zentral:

> Verstehen von Bildern ist demnach ein Prozeß der Strukturabbildung eines Systems von externen räumlichen Relationen auf ein System von internen semantischen Relationen.
>
> (Schnotz, Seite 91)

Im Rahmen dieser Projektarbeit werden keine weiteren kognitionspsychologischen Aspekte betrachtet.

Ein Fazit dieses Abschitts ist, daß eine Entsprechung zwischen räumlichen Relationen einer Struktur-Visualisierung und semantischen Relationen des mentalen Modells besteht.

Die einzelnen räumlichen Relationen einer Struktur-Visualisierung bestimmen den visuellen Eindruck, der ganzheitlich vom Menschen wahrgenommen wird und in dieser ganzheitlichen Eigenschaft auch das mentale Modell prägen kann.

Definition: Ein Struktur-Visualisierungs-Werkzeug ist *ergonomisch*, wenn der visuelle Eindruck, den ein einmal generiertes Layout zur Unterstützung eines mentalen Modells vermittelt hat, im Verlauf einer Bearbeitung nicht verloren geht.

Das führt zu der Forderung, daß Auswirkungen von Änderungen (Ergänzungen, Löschungen) auf das Layout so gering wie möglich zu halten sind. Diese Forderung der *Robustheit gegenüber Änderungen* wird in der Forschungsgemeinschaft, die sich mit der automatischen Generierung von Graphlayout beschäftigt, *no surprise* genannt.

1.2.3 Visualisierungs-Werkzeuge

Ein großes Problem der Struktur-Visualisierung ist, daß es kaum Werkzeuge gibt, mit denen man derartige Diagramme editieren kann. Die Eigenschaften des Layout-Algorithmus eines solchen Werkzeugs unterscheiden sich grundsätzlich von den herkömmlichen Algorithmen, die z.B. von Texteditoren und Textbearbeitungssystemen benutzt werden. Graphlayout-Algorithmen erscheinen auf den ersten Blick am ähnlichsten.

Der Rückgriff auf die zahlreich vorhandenen Graphlayout-Werkzeuge ist allerdings problematisch, da zum einen die Graphlayout-Algorithmen nach Änderung eines dargestellten Graphen diesen meist völlig räumlich umstellen, sodaß der Anwender sich völlig neu orientieren muß. Dies stellt den Vorteil der Visualisierung wiederum in Frage und ist im Sinne dieser Arbeit nicht ergonomisch. Zum anderen fehlen geeignete Navigations- und Abstraktions-Mechanismen, die für das Bearbeiten großer Strukturen dringend nötig sind.

Vektor-Orientierte Zeichenprogramme finden ebenfalls Verwendung für das Erstellen und Bearbeiten von Struktur-Visualisierungen. Sie sind allerdings so unflexibel, daß man nur mit sehr hohem Zeitaufwand z.B. Ergänzungen an einem Diagramm vornehmen kann. Dazu müßten alle grafischen Objekte, die zur Seite gerückt werden müssen, um Platz zu schaffen, von Hand neu positioniert werden. Derartige Folgeänderungen sind in diesen Programmen nicht automatisiert, obwohl sie prinzipiell automatisierbar sind.

Ein für die Struktur-Visualisierung geeignetes Werkzeug muß ein stabiles Layout ohne Überraschungen sowie ein flüssiges Navigieren durch verschiedene Sichten der zugrundeliegenden Struktur ermöglichen. Idealerweise wäre es mit einem solchen Werkzeug möglich, Struktur-Visualisierungen so einfach zu bearbeiten wie man heute Texte bearbeitet.

1.3 Derzeit gebräuchliche Layout-Beschreibungssprachen

1.3.1 Layout-Beschreibungen für Bildverarbeitungs- und Zeichenprogramme

Im Anwendungsbereich der Bildverarbeitungsprogramme kann man kaum von Layout sprechen. Diese speichern die generierten Bilder auf der Grundlage von Bitmaps (rechteckiger 2-D-Pixel-Raum), die sich lediglich in Farb-Modell und Komprimierungs-Verfahren unterscheiden.

Bei vektor-orientierten Zeichenprogrammen kann man durchaus von einer Layout-Beschreibung sprechen. Ein Layout ist bei diesen eine Ansammlung von geometrischen Grundfiguren wie etwa Linien, Kreise, Ovale, Polygonzüge. Ihre Position wird allerdings fest in Pixel-Koordinaten beschrieben.

Zum letztgenannten Anwendungsbereich gehören im Grunde auch CAD-Systeme. Sie bilden allerdings eine eigene Klasse von Anwendungen. Um der Komplexität, die der Entwurfsprozeß von realen Maschinen, Gebäuden, etc. mit sich bringt, gerecht zu werden, sind sie mit eigenen Programmiersprachen ausgestattet, mit der nicht nur das Layout beeinflußt werden kann, sondern auch inhaltsverändernde Transformationen beschrieben werden. Obwohl eine nähere Betrachtung von CAD-Systemen möglicherweise gewinnbringend gewesen wäre, hätte sie den Rahmen dieser Arbeit gesprengt und mußte daher unterbleiben.

1.3.2 Layout-Beschreibungen für Graphlayout-Werkzeuge

Grundlage für jedes Graphlayout ist ein Graph im Sinne der Graphentheorie, also eine Menge oder Liste von Knoten und Kanten.

In Graphlayout-Werkzeugen wird diese Liste je nach Funktionsvielfalt mit einer Beschreibung der Parameter des verwendeten Layout-Algorithmus sowie grafischer Attribute (Farbe, Beschriftung, etc.) der Kanten und Knoten angereichert. Um der resultierenden Vielfalt und damit Inkompatibilität der Beschreibungen entgegenzuwirken, wurde ein Standard entwickelt, die Graph Modelling Language (GML).

1.3.3 Layout-Beschreibungen in grafischen Benutzerschnittstellen (GUIs)

Aufgrund der großen Interaktivität, die ein wesentliches Merkmal der GUI-Programmierung ist, könnte man sich fragen, ob es hier überhaupt Layout-Beschreibungen gibt. Es waren allerdings von Anfang an GUI-Werkzeuge verfügbar, mit denen man den nicht-interaktiven Teil eines GUIs bauen konnte. Die Dateien, die diese Werkzeuge als Ausgabe haben, kann man als GUI-Layout-Beschreibung begreifen.

Aus der Sicht der für diese Arbeit interessanten Aspekte lassen sich GUIs in zwei Klassen einteilen. In den Anfängen der GUI-Programmierung ist festspezifizierte Pixel-Positionierung die Regel. Es gibt zwar Schachtelungen mit relativem Koordinatensystem, doch in diesem wird mit absoluten Pixel-Koordinaten positioniert. Jede relative Positionierung mußte durch das Anwendungsprogramm dynamisch generiert werden.

Mit zunehmender Plattformunabhängigkeit und der breiten Streuung der zur Verfügung stehenden Bildschirmauflösungen wurde auch hier eine Abstraktion über die Ebene der Pixel-Positionierung auf Seite des GUI nötig. Die Layout-Container des Java AWT (abstract window toolkit) sind hierfür ein schönes Beispiel.

Zugrundeliegendes Modell ist dabei eine Tabelle, die aus aneinandergefügten rechteckigen Zellen zusammengesetzt ist. Die Größe der Zellen in Breite und Höhe wird abhängig von der verfügbaren Layout-Fläche beschrieben. Dabei können sich prozentuale und fixe Pixel-Angaben mit Restplatz-Angaben ergänzen. Zellen können auch zusammengefaßt werden.

Die Sprachmittel des Standard-AWT sind ausreichend für Layout-Beschreibungen von tabellenartigen Layouts, aber nicht für Layout-Beschreibungen von allgemeineren Strukturen.

Die AWT-Software-Architektur bildet allerdings einen günstigen Ausgangspunkt für die prototypische Implementierung.

1.3.4 Layout-Beschreibungen im WWW

Die im WWW eingesetzte Layout-Beschreibungssprache *HTML* (Hypertext Markup Language) ist ein durch Wildwuchs geformtes Sammelbecken verschiedenster Konzepte. Sie entstand, um verteilte Hypertexte zu beschreiben und wurde mit der Zeit um Multimedia-Elemente und mit einer Programmiersprache ergänzt. Sie trennt nicht zwischen Inhalt- und Layout-Beschreibung wie das etwa bei dem neueren Web-Standard XML (eXtended Markup Language) der Fall ist.

Das Konzept, Tabellen zu beschreiben, ist ähnlich zu dem weiter oben beschriebenen AWT-Konzept. HTML ist fließtext-orientiert und ist nicht in der Lage, das Layout von allgemeinen Strukturen zu beschreiben.

1.4 Anforderungen an die zu entwickelnde Layout-Beschreibungssprache `LayLa`

Eine Layout-Beschreibung in Form von Pixelkoordinaten ist für die geforderte Robustheit gegenüber Änderungen viel zu konkret. Eine Layout-Beschreibung in Form eines Graphen im Sinne der Graphentheorie ist dagegen zu abstrakt: sie erlaubt kaum Aussagen über das Erscheinungsbild und insbesondere keine räumlichen Angaben wie zum Beispiel die Angabe von Richtungen. Dies ist ein großer ergonomischer Nachteil, da das Erscheinungsbild das mentale Modell prägt und damit die Grundlage für das semantische Verständnis bildet (siehe Abschnitt: Kognitionspsychologische Aspekte).

Daher ist eine neue Form der Layout-Beschreibung zu finden, die in ihrem Abstraktions-Niveau zwischen pixel- und graph-artigen Beschreibungen liegt.

Die Layout-Beschreibungssprache soll für *ergonomische Struktur-Visualisierungs-Werkzeuge* anwendbar sein. Um dies zu erreichen, beschreibt sie die räumlichen Beziehungen zwischen 2-dimensionalen grafischen Objekten mit rechteckiger Ausdehnung.

Außerdem soll `LayLa` ermöglichen, daß ein Layout-Algorithmus eine pixel-orientierte Anordnung der grafischen Objekte ermitteln kann.

Das Layout ist möglichst unabhängig von der zur Verfügung stehenden Pixel-Fläche zu beschreiben. Bei Änderung der Pixel-Fläche soll nach Neuberechnung des Layouts ein ähnliches Bild entstehen.

`LayLa` soll auf einem geeigneten Abstraktionsniveau erlauben, verschiedene Visualisierungen zu beschreiben (z.B. tabellarisches Layout, zentrales Layout, Baum). Da auch Graphen darstellbar sein sollen, muß es eine Möglichkeit geben, Verbindungslinien zu beschreiben.

Ziel von `LayLa` ist es *nicht*, Änderungen in einem bereits vorliegenden Layout zu beschreiben. Änderungen im Layout finden durch Änderung der Beschreibung in `LayLa` und durch die Neuberechnung der Pixel-Koordinaten statt.

2 Entwurf der Layout-Beschreibungssprache `LayLa`

2.1 Leitlinien

2.1.1 Annahmen und Terminologie

Für das Layout steht eine rechteckige Fläche zur Verfügung. Diese wird im Folgenden *Rahmen* genannt. Die Größe des Rahmens ist beliebig, aber für die Berechnung eines Layouts fest. Veränderungen der Rahmengröße sind möglich. Jede Veränderung verursacht eine Neuberechnung des Layouts. Die Seiten des Rahmens werden als Achsen eines Koordinatensystems aufgefasst, dessen Ursprung in der linken oberen Ecke des Rahmens liegt. Die x-Werte steigen nach rechts an und die y-Werte nach unten.

Innerhalb des Rahmens können grafische Objekte angeordnet werden, die im Folgenden *Komponenten* genannt werden. Jede Komponente hat eine räumliche Ausdehnung, und zwar ein Rechteck, dessen Seiten parallel zu den Seiten des Rahmens sind. Die linke obere Ecke dieses Rechtecks dient als Bezugspunkt der Komponente.

`LayLa` umfaßt Sprachmittel zur Beschreibung von räumlichen Relationen zwischen Komponenten. Diese Sprachmittel werden *Relations-Beschreibung*en genannt. Da im Folgenden der Begriff Relation in keiner weiteren Bedeutung vorkommt, ist mit Relation ab sofort immer eine räumliche Relation gemeint.

Außer Komponenten können innerhalb des Rahmens auch *Verbindungslinien* zwischen Komponenten angeordnet werden. Eine Verbindungslinie verbindet zwei Komponenten. Einer Verbindungslinie sind keine weiteren Objekte zugeordnet, auch keine Beschriftungen.

2.1.2 Ergonomische Randbedingung

Der visuelle Eindruck muß laut Anforderung erhalten bleiben, wenn sich die Größe des Rahmens ändert. Auch kleinere Änderungen in der Layout-Beschreibung sollen den visuellen Eindruck erhalten.

Die wesentlichen Aspekte von visuellem Eindruck sind nach meiner Einschätzung:

- relative Position der Komponenten untereinander (Ausrichtung)
- relativer Abstand der Komponenten untereinander
- relative Größenverhältnisse der Komponenten untereinander
- Position der Komponenten relativ zum Rahmen (in Ecke, im Zentrum)
- gleichbleibender visueller Eindruck des Inhalts der Komponenten

Diese wesentlichen Aspekte – bis auf den Inhalt der Komponenten – sind durch die *Relations-Beschreibung*en von `LayLa` beschreibbar. Diese räumlichen Relationen müssen demnach nicht nur die relative Position der Komponenten untereinander beschreiben, sondern auch Positionen relativ zum Rahmen. Außerdem muß die Größe der Komponenten beschreibbar sein.

2.1.3 Syntax

Jede in `LayLa` formulierte Layout-Beschreibung ist in Zeichen kodiert. Es gibt verschiedene Standards, die einen Zeichenvorrat festlegen. Sinnvoll ist ein internationaler Zeichenvorrat, um keine unnötigen Einschränkungen zu machen. Hierfür bietet sich der Unicode-Standard an.

Jede in `LayLa` formulierte Layout-Beschreibung besteht aus einer Aneinanderreihung von *Komponenten-Beschreibung*en oder *Relations-Beschreibung*en in beliebiger Reihenfolge. Das Trennzeichen ist den üblichen Gepflogenheiten folgend das Semikolon.

Eine *Komponenten-Beschreibung* oder *Relations-Beschreibung* besteht aus Feldern, zwischen denen als Trenner nichtleere Folgen von Whitespace-Zeichen stehen.

Manche *Relations-Beschreibung*en haben benannte Attribute. Diese belegen jeweils zwei direkt aufeinander folgende Felder. Das erste Feld enthält den Namen und das zweite den Wert des Attributes. Die Reihenfolge der Attribut-Wert-Paare ist wegen der Bezeichnung der Attribute beliebig. Insofern ist die Syntax von `LayLa` flexibler als es in der Grammatik zum Ausdruck kommt.

Eine *Komponenten-Beschreibung* enthält unter anderem einen *Komponenten-Bezeichner*, der in *Relations-Beschreibung*en verwendet werden kann. Dafür gelten die üblichen Bedingungen: jeder irgendwo verwendete *Komponenten-Bezeichner* muß in einer *Komponenten-Beschreibung* vorkommen, und kein *Komponenten-Bezeichner* darf in mehr als einer *Komponenten-Beschreibung* vorkommen.

Man beachte im folgenden Syntax-Überblick von `LayLa` die unterschiedlichen Schriftarten für *Nichtterminal-Symbole* und `Terminal-Symbole`.

Layout-Beschreibung ::= ((*Komponenten-Beschreibung* | *Relations-Beschreibung*) `;`)$^+$

Komponenten-Beschreibung ::= `Komponente` *Komponenten-Bezeichner* *Komponenten-Typ* *Komponenten-Inhalt*
Komponenten-Bezeichner ::= *NatürlicheZahl*
Komponenten-Typ ::= `Text`
Komponenten-Inhalt ::= *Zeichenkette*

Relations-Beschreibung ::= *Basis-Relations-Beschreibung* | *Abgeleitete-Relations-Beschreibung*
Basis-Relations-Beschreibung ::= *Rel-VarAusdruck* | *Rel-Neben* | *Rel-Linie*
Abgeleitete-Relations-Beschreibung ::= *Zweistellige-Relations-Beschreibung* | *Einstellige-Relations-Beschreibung*
Zweistellige-Relations-Beschreibung ::= *Rel-Gleichwert* | *Rel-Gleichgross* | *Rel-RelGross*
Einstellige-Relations-Beschreibung ::= *Rel-Fix* | *Rel-NebenRand* | *Rel-Zentrum* | *Rel-Rahmen*
Rel-VarAusdruck ::= `VarAusdruck` [*Attr-Quelle*] *Attr-Ziel* *Attr-Gleichung*
Rel-Neben ::= `Neben` *Attr-Quelle* *Attr-Ziel* *Attr-Richtung* *Attr-Abstand*
Rel-Linie ::= `Linie` *Attr-Quelle* *Attr-Ziel*

Rel-Gleichwert ::= `Gleichwert` *Attr-Quelle* *Attr-Ziel* *Attr-Variable*
Rel-Gleichgross ::= `Gleichgross` *Attr-Quelle* *Attr-Ziel*
Rel-RelGross ::= `RelGross` *Attr-Quelle* *Attr-Ziel* *Attr-Wrel* *Attr-Hrel*

Rel-Fix ::= `Fix` *Attr-Ziel* *Attr-Variable* *Attr-Pixel*
Rel-NebenRand ::= `NebenRand` *Attr-Ziel* *Attr-Richtung* *Attr-Abstand*
Rel-Zentrum ::= `Zentrum` *Attr-Ziel*
Rel-Rahmen ::= `Rahmen` *Attr-Ziel* *Attr-Wrel* *Attr-Hrel*

Attr-Quelle	::=	`Quelle` *Komponenten-Bezeichner*
Attr-Ziel	::=	`Ziel` *Komponenten-Bezeichner*
Attr-Richtung	::=	`Richtung` (*Winkelrichtung* | *Himmelsrichtung*)
Attr-Abstand	::=	`Abstand` (*ArithmetischerAusdruck* | `Unspezifiziert`)
Attr-Variable	::=	`Variable` *Var*
Attr-Wrel	::=	`Wrel` *NatürlicheZahl*
Attr-Hrel	::=	`Hrel` *NatürlicheZahl*
Attr-Pixel	::=	`Pixel` *NatürlicheZahl*
Attr-Gleichung	::=	`Gleichung` *ZielVar=ArithmetischerAusdruck*
Winkelrichtung	::=	*NatürlicheZahl*
Himmelsrichtung	::=	`o` | `s` | `w` | `n` | `so` | `sw` | `nw` | `no`
AlleVar	::=	*QuellVar* | *ZielVar* | *RahmenVar*
QuellVar	::=	`Quelle.`*Var*
ZielVar	::=	`Ziel.`*Var*
RahmenVar	::=	`Rahmen.`*Var*
Var	::=	`x` | `y` | `w` | `h`
NatürlicheZahl	::=	⟨Standardnotation⟩
Zeichenkette	::=	⟨Standardnotation⟩
ArithmetischerAusdruck	::=	⟨Standardnotation mit Operatoren `+`, `-`, `*`, `/` und Operanden *AlleVar* sowie Zahlen⟩

2.2 Das LayLa-Konstrukt **Komponenten-Beschreibung**

Komponenten sind rechteckige Platzhalter. Der Inhalt ist abhängig von den Anwendungs-Programmen, die `LayLa` nutzen. Hier sind viele Situationen denkbar. Die *Graph Modelling Language* beispielsweise beschränkt sich auf Text mit Darstellungs-Attributen wie Farbe oder Zeichensatz-Größe und -Typ sowie verschiedene Umrahmungen. Zeitgemäßer wären m.E. URL-Verweise, deren Inhalt in der Rechteck-Fläche dargestellt wird. Aufgrund der vielen denkbaren Anwendungen sind in `LayLa` konkrete Beschreibungen des Inhalts exemplarisch nur auf Texte beschränkt. Daher ist als *Komponenten-Typ* auch nur `Text` vorgesehen.

2.2.1 Syntax

Mit der Produktion *Komponenten-Beschreibung* läßt sich jeweils eine Komponente beschreiben.

Der *Komponenten-Bezeichner*, der in *Relations-Beschreibung*en genutzt wird, um die Komponente eindeutig zu referenzieren, ist eine dezimale positive Integer-Zahl. Der *Komponenten-Inhalt* besteht aus einer Zeichenkette. Für die Kodierung der Zeichenkette wird die Standardnotation vorausgesetzt. Die Definitionen von *Komponenten-Bezeichner* als *NatürlicheZahl* und *Komponenten-Inhalt* als *Zeichenkette* sind bewußte Vereinfachungen der Grammatik im Rahmen dieser Projektarbeit und nicht wesentlich für `LayLa`. Entsprechendes gilt für die

Einschränkung auf einen einzigen *Komponenten-Typ*.

Beispiel:

```
Komponente 1125 Text "Hello World";
```

2.3 Die LayLa-Konstrukte **Basis-Relations-Beschreibung**

2.3.1 Relation **Rel-VarAusdruck**

Position und Größe einer Komponente lassen sich mit 4 Variablen beschreiben. Das erste Variablen-Paar (`x`, `y`) bestimmt die Koordinaten und damit die Position der linken oberen Ecke der Komponente. Das zweite Variablen-Paar bestimmt die Größe der Komponente durch Breite `w` und Höhe `h`. Räumliche Relationen zwischen Komponenten lassen sich ganz allgemein als Beziehungen zwischen diesen Variablen beschreiben. Die allgemeinste Relation, die explizite Ausdrücke über diese Variablen zuläßt, ist *Rel-VarAusdruck*, die im Folgenden beschrieben wird.

In den Attributen *Attr-Quelle* und *Attr-Ziel* sind die *Komponenten-Bezeichner* angegeben. Es wird die Relation zwischen einer Variablen der Ziel-Komponente, die durch Voranstellen von `Ziel.` gekennzeichnet ist, und einer beliebigen Auswahl von Variablen der Ziel- und Quell-Komponente beschrieben. Diese Relation wird in dem Attribut *Attr-Gleichung* durch eine Gleichung genau bestimmt. Die linke Seite der Gleichung enthält die Ziel-Variable, die in der Produktion *ZielVar* beschrieben ist. Die rechte Seite enthält einen arithmetischen Ausdruck in üblicher Notation.

In dem arithmetischen Ausdruck dürfen die Infix-Operatoren `+,-,*,/`, Klammerung, dezimale Konstanten und die Variablen der Produktion *AlleVar* vorkommen. Zu diesen Variablen gehören nicht nur die jeweils vier Variablen der Ziel- und Quell-Komponente, sondern auch die vier Variablen des Rahmens, die unabhängig von Ziel- und Quell-Komponente immer verwendet werden können. Sinnvoll verwendbar sind alllerdings nur die Variablen `Rahmen.w` und `Rahmen.h`, mit denen Bezug auf die Breite und Höhe des Rahmens genommen werden kann. Die Variablen `Rahmen.x` und `Rahmen.y` haben immer den Wert 0.

Laut Grammatik ist es zwar erlaubt, die Variable auf der linken Seite der Gleichung in der rechten Seite nochmals zu verwenden. Dies macht allerdings keinen Sinn und führt höchstens zu Problemen in einem Layout-Algorithmus.

Für den Fall, daß nur die Variablen von der Ziel-Komponente und eventuell des Rahmens benötigt werden, kann das Attribut *Attr-Quelle* auch weggelassen werden. Dies hat zur Folge, daß in dem arithmetischen Ausdruck die Quell-Variablen mit den entsprechenden Ziel-Variablen identisch sind.

Beispiele:

```
VarAusdruck Quelle 11 Ziel 12 Gleichung Ziel.x=Quelle.x+1.5*Quelle.w;
```

```
VarAusdruck Ziel 10 Gleichung Ziel.w=Ziel.h;
```

2.3.2 Relation Rel-Neben

2.3.2.1 Syntax Die Relation *Rel-Neben*, die eine direkte Nachbarschaftsbeziehung beschreibt, hat die Attribute *Attr-Quelle* und *Attr-Ziel*. Hier müssen die beiden *Komponenten-Bezeichner* der an der Nachbarschaftsbeziehung beteiligten Komponenten angegeben werden.

Die beiden folgenden Attribute *Attr-Richtung* und *Attr-Abstand* dienen dazu, die direkte Nachbarschaftsbeziehung zwischen Quell-Komponente und Ziel-Komponente genauer festzulegen. Dies geschieht erstens durch Angabe der Richtung in der die Ziel-Komponente relativ zur Quell-Komponente angeordnet ist und zweitens durch Angabe des Abstands zwischen den beiden Komponenten. Die Angabe der Richtung kann qualitativ und quantitativ erfolgen.

Beispiele:

```
Neben Quelle 42 Ziel 50 Richtung so Abstand Quelle.w ;
```

```
Neben Quelle 2 Ziel 3 Richtung n Abstand Rahmen.h/4 ;
```

```
Neben Quelle 2 Ziel 5 Richtung n Abstand Unspezifiziert ;
```

```
Neben Quelle 143 Ziel 42 Richtung 120 Abstand 100 ;
```

```
Neben Quelle 3 Ziel 4 Richtung 300 Abstand Quelle.h/2 ;
```

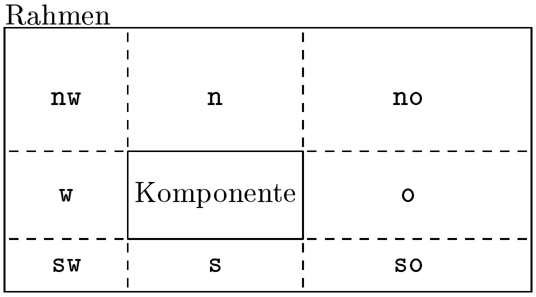
2.3.2.2 Attribut Attr-Richtung

2.3.2.2.1 Quantitative Richtung Im quantitativen Modus folgt auf das Symbol `Richtung` eine natürliche Zahl zwischen 0 und 360, die für einen Winkel gemäß der für GUIs üblichen Konvention steht. Im Gegensatz zu den mathematischen Gepflogenheiten werden bei GUIs die Winkelwerte *im* Uhrzeiger-Sinn abgetragen, da in GUIs die y-Achse entgegengesetzt zur mathematischen Konvention orientiert ist und somit das Koordinatensystem um die x-Achse gespiegelt ist. 0 bzw. 360 Grad entspricht der Waagerechten vom Zentrum nach Rechts.

2.3.2.2.2 Qualitative Richtung Im qualitativen Modus folgt auf das Symbol `Richtung` die Angabe einer Himmelsrichtung. Möglich sind hier die vier Himmelsrichtungen (Norden, Osten, etc.) mit den 4 Zwischen-Richtungen (Süd-Osten, etc.).

Diese Richtungs-Angabe ist *keine* Kurz-Schreibweise für Intervalle von quantitativ angebbaren Winkeln. Bei quantitativen Richtungsangaben ist die Form der Komponenten im Grunde beliebig. Es sollte nur ein Mittelpunkt als Ursprung für die Richtungsangabe vorhanden sein. Die Himmelsrichtungs-Darstellung beruht dagegen auf der rechteckigen Form der Komponenten. Verlängert man die vier Seiten eines Rechtecks zu Geraden bis zum Rahmen, dann entstehen 8 *Oktanten* um das Rechteck herum. Jedem dieser Oktanten ist die entsprechende (intuitiv einsichtige) Himmelsrichtung zugeordnet. In der Syntax von `LayLa` sind die

Himmelsrichtungen abgekürzt.



2.3.2.3 Attribut Attr-Abstand

Auf das Symbol `Abstand` folgt ein *ArithmetischerAusdruck*, der einen Pixel-Wert beschreibt, der den Abstand zwischen den benachbarten Komponenten bestimmt. Soll der Wert des Abstands unspezifiziert bleiben, kann anstatt des arithmetischen Ausdrucks auch das Schlüsselwort `Unspezifiziert` benutzt werden. Der *ArithmetischerAusdruck* ist der gleiche, der auch in der Relation *Rel-VarAusdruck* Verwendung findet. Er kann also auch alle 12 Variablen enthalten (Quell-, Ziel- und Rahmen-Variablen).

Wenn die Richtung quantitativ angegeben ist, bestimmt dieses Attribut den Abstand zwischen den Mittelpunkten der Komponenten. Auf diese Weise können Komponenten ohne Rücksicht auf Überlappung plaziert werden. Die quantitative Angabe von sowohl Richtung als auch Abstand entspricht also der Idee von Polarkoordinaten.

Ist die Richtung qualitativ angegeben, bestimmt dieses Attribut den Abstand zwischen den Rändern der Komponenten, da es ein Ziel der qualitativen Richtungsangabe ist, benachbarte Komponenten möglichst überlappungsfrei zu plazieren.

2.3.2.4 Bemerkungen zur Semantik

Stellt man die Anforderung, daß sich die Komponenten nicht überlappen dürfen, sondern nur nebeneinander plaziert werden dürfen, dann müßten gemachte Angaben über die Position gegebenenfalls korrigiert werden. Dies würde dem Sinn der quantitativen Darstellung widersprechen. Sie ist das Mittel, um jede beliebige Position der beiden Komponenten zueinander *genau* zu beschreiben. Daher wird bei dieser Richtungs-Angabe ohne Rücksicht auf eventuelle Überdeckungen oder Überschneidungen positioniert.

Anders verhält es sich mit der Himmelsrichtungs-Notation. Die Angaben sind qualitativ und lassen dem Layout-Algorithmus die Freiheit, die Komponenten möglicherweise überlappungsfrei zu positionieren.

2.3.3 Relation Rel-Linie

Diese Relation beschreibt eine Verbindungs-Linie zwischen den beiden angegebenen Komponenten. Es ist nicht notwendig, daß zwischen diesen Komponenten auch eine *Rel-Neben*-Relation spezifiziert ist. Besteht zwischen den Komponenten allerdings noch eine *Rel-Neben*-Relation, dann wird die Linie auf keinen Fall von anderen Komponenten überdeckt, was sonst passieren kann.

Beispiel:

```
Linie Quelle 42 Ziel 50 ;
```

2.4 Die LayLa-Konstrukte **Abgeleitete-Relations-Beschreibung**

Alle folgenden Relationen sind abgeleitete Relationen. Sie sind Sonderfälle der Relation *Rel-VarAusdruck* und lassen sich somit auch mit Hilfe der Relation *Rel-VarAusdruck* beschreiben. Sie sind im folgenden gegliedert in Relationen, bei denen Ziel- und Quell-Komponente verschieden (zweistellig) bzw. gleich (einstellig) sind.

2.4.1 Die LayLa-Konstrukte **Zweistellige-Relations-Beschreibung**

2.4.1.1 Relation **Rel-Gleichwert**

Die Relation

```
Gleichwert Quelle q Ziel z Variable v;
```

entspricht folgender *Rel-VarAusdruck*-Relation

```
VarAusdruck Quelle q Ziel z Gleichung Ziel.v=Quelle.v;
```

2.4.1.2 Relation **Rel-Gleichgross**

Die Relation

```
Gleichgross Quelle q Ziel z;
```

entspricht folgenden *Rel-VarAusdruck*-Relationen

```
VarAusdruck Quelle q Ziel z Gleichung Ziel.w=Quelle.w;
VarAusdruck Quelle q Ziel z Gleichung Ziel.h=Quelle.h;
```

2.4.1.3 Relation **Rel-RelGross**

Die Relation

```
RelGross Quelle q Ziel z Wrel wr Hrel hr;
```

entspricht folgenden *Rel-VarAusdruck*-Relationen

```
VarAusdruck Quelle q Ziel z Gleichung Ziel.w=Quelle.w*wr/1000;
VarAusdruck Quelle q Ziel z Gleichung Ziel.h=Quelle.h*hr/1000;
```

2.4.2 Die LayLa-Konstrukte **Einstellige-Relations-Beschreibung**

Die bisherigen Relationen haben zwei Komponenten räumlich miteinander in Beziehung gesetzt. Die einstelligen Relationen bestimmen die räumlichen Eigenschaften *einer* Komponente.

Dies kann durch die Angabe konstanter Werte oder durch eine implizite räumliche Beziehung zum Rahmen geschehen.

2.4.2.1 Relation **Rel-Fix**

Mit der Relation *Rel-Fix* kann für eine der vier Variablen `x|y|w|h`, die die Position und Größe der Komponente beschreiben, ein konstanter Wert vorgegeben werden.

Die Relation

```
Fix Ziel z Variable var Pixel wert ;
```

entspricht der folgenden *Rel-VarAusdruck*-Relation

```
VarAusdruck Ziel z Gleichung Ziel.var=wert ;
```

Es handelt sich um einen Pixel-Wert. Da man zum Zeitpunkt der Layout-Beschreibung allerdings nicht weiß wie groß die Layout-Fläche in Pixel ist, ist diese Relation nur begrenzt sinnvoll. Es kann allerdings sein, daß der Inhalt einer Komponente eine feste Pixel-Ausdehnung hat, wie zum Beispiel bei nicht skalierbaren Zeichensätzen oder Bildern.

Beispiele:

```
Fix Ziel 14 Variable w Pixel 300;
```

```
Fix Ziel 14 Variable h Pixel 200;
```

2.4.2.2 Relation **Rel-NebenRand**

Die Relation *Rel-NebenRand* erlaubt es, eine Relation zwischen einer Komponente und dem Rahmen zu beschreiben. Das Attribut *Attr-Richtung* bestimmt, an welchem Rand bzw. an welcher Ecke die Komponente ausgerichtet wird. Die Zwischen-Himmelsrichtungen beschreiben die Ecken. Zum Beispiel richtet `nw` von der linken oberen Ecke des Rahmens aus und `w` richtet vom linken Rand des Rahmens aus.

Das Attribut *Attr-Abstand* beschreibt den Abstand vom Bezugs-Rand bzw. von der Bezugs-Ecke. Der Intuition folgend wird beim Rand der Abstand horizontal bzw. vertikal gemessen und bei den Ecken diagonal. Der maximale Abstand (1000) entspricht der Länge des Rahmen-Randes minus der Länge des parallel liegenden Komponenten-Randes.

Der maximale Abstand (1000) bei der Eck-Ausrichtung entspricht der Positionierung der Komponente in der diagonal gegenüberliegenden Ecke, sodaß die Komponente in ihrer Ausdehnung noch ganz innerhalb des Rahmens liegt. Wenn z.B. im Attribut *Attr-Richtung* der Wert `nw` angegeben ist, dann positioniert der Abstand 1000 die Komponente in der rechten unteren Ecke des Rahmens. Die Punkte auf der Diagonalen zwischen der linken oberen Ecke des Rahmens und der linken oberen Ecke der mit Abstand 1000 positionierten Komponente bilden die Bezugspunkte für die Abstände zwischen 0 und 1000 in gleicher Richtung.

Bei der Eck-Ausrichtung ist die Position einer Komponente durch eine *Rel-NebenRand*-Relation eindeutig bestimmt. Bei der Rand-Ausrichtung wird nur die x- oder y-Position bestimmt, sodaß für eine eindeutige Position zwei *Rel-NebenRand*-Relationen mit orthogonalen

Richtungen nötig sind.

Die laut Grammatik mögliche Angabe einer *Winkelrichtung* ist bei dieser Relation nicht erlaubt.

Die Relation

```
NebenRand Ziel z Richtung w Abstand a ;
```

entspricht z.B. der folgenden *Rel-VarAusdruck*-Relation

```
VarAusdruck Ziel z Gleichung Ziel.x=(Rahmen.w-Ziel.w)*a/1000;
```

Die Relation

```
NebenRand Ziel z Richtung nw Abstand a ;
```

entspricht den folgenden beiden *Rel-NebenRand*-Relationen

```
NebenRand Ziel z Richtung n Abstand a ;
```

```
NebenRand Ziel z Richtung w Abstand a ;
```

Beispiele:

```
NebenRand Ziel 14 Richtung w Abstand 200;
```

```
NebenRand Ziel 14 Richtung s Abstand Rahmen.h/4;
```

2.4.2.3 Relation Rel-Zentrum Die Relation *Rel-Zentrum* positioniert den Mittelpunkt einer Komponente im Mittelpunkt des Rahmens.

Die Relation

```
Zentrum Ziel z ;
```

entspricht den folgenden *Rel-VarAusdruck*-Relationen

```
VarAusdruck Ziel z Gleichung Ziel.x=(Rahmen.w-Ziel.w)/2;
VarAusdruck Ziel z Gleichung Ziel.y=(Rahmen.h-Ziel.h)/2;
```

Beispiel:

```
Zentrum Ziel 15;
```

2.4.2.4 Relation Rel-Rahmen Die Relation *Rel-Rahmen* entspricht fast der Relation *Rel-RelGross*. Der Unterschied liegt darin, daß mit *Rel-Rahmen* nicht zwei beliebige Komponenten in Größen-Relation gesetzt werden können, sondern die Ziel-Komponente mit der Größe des Rahmens.

Die Relation

```
Rahmen Ziel z Wrel wr Hrel hr;
```

entspricht folgenden *Rel-VarAusdruck*-Relationen

```
VarAusdruck Ziel z Gleichung Ziel.w=Rahmen.w*wr/1000;
VarAusdruck Ziel z Gleichung Ziel.h=Rahmen.h*hr/1000;
```

Beispiel:

```
Rahmen Ziel 15 Wrel 200 Hrel 200;
```

3 LayLa-Anwendungsbeispiele

Die folgenden Beispiele sollen demonstrieren, wie man sehr verschiedene Layouts mit LayLa beschreiben kann.

3.1 Dialogbox

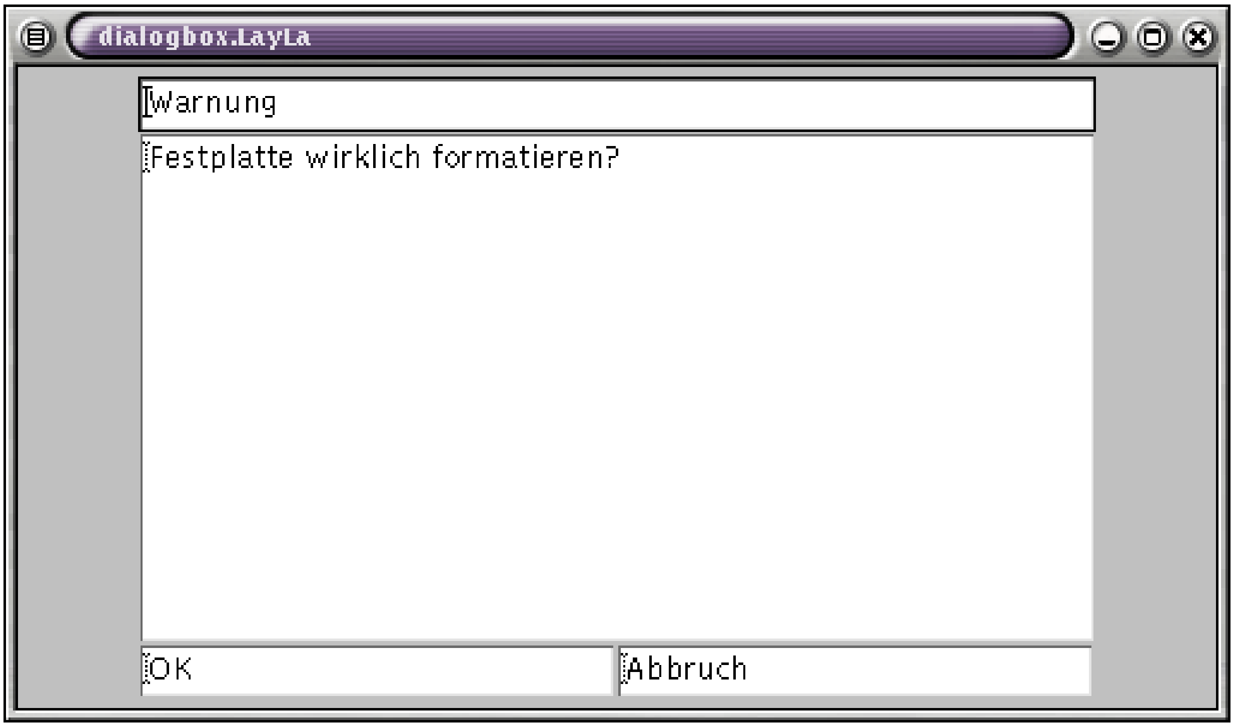
Das erste Beispiel ist eine Dialogbox, die aus 4 Komponenten besteht. Der Inhalt der Meldung des Dialogs ist in Komponente 1 enthalten, deren Ausdehnung 80% der Höhe und Breite des Rahmens beträgt.

Die Relation *Rel-Zentrum* plaziert sie im Zentrum der Dialogbox. Die Komponenten 3 und 4 sind jeweils die Aktions-Knöpfe für "OK" und "Abbruch". Wie in Dialogboxen üblich, sind diese im Süden der Meldung angeordnet. Da die Breite von Komponente 3 und 4 nicht festgelegt ist und beide Komponenten im Süden von Komponente 1 beschrieben sind, teilen sich beide die zur Verfügung stehende Breite von Komponente 1 im gleichen Verhältnis. Die Überschrift des Dialogs ist in Komponente 4 enthalten. Sie liegt im Norden der Komponente 1. Da auch für Komponente 4 keine Breite festgelegt ist und keine weitere Komponente um Platz konkurriert, sind die Breiten von Komponente 4 und 1 gleich.

Die Höhen der Komponenten 2, 3 und 4 sind durch die LayLa-Beschreibung ebenfalls nicht festgelegt. Diese Werte setzt der Layout-Algorithmus des Prototypen auf den konstanten Pixel-Wert 23. Dieser Wert ermöglicht die Darstellung einer Texte-Zeile mit dem Standard-Zeichensatz im inneren der Komponente.

```
Komponente 1 Text "Festplatte wirklich formatieren?" ;
Komponente 2 Text "OK" ;
Komponente 3 Text "Abbruch" ;
Komponente 4 Text "Warnung" ;
Zentrum Ziel 1 ;
Rahmen Ziel 1 Wrel 800 Hrel 800 ;
Neben Quelle 1 Ziel 4 Richtung n Abstand 0 ;
Neben Quelle 1 Ziel 2 Richtung s Abstand 0 ;
Neben Quelle 1 Ziel 3 Richtung s Abstand 0 ;
```

Ändert man die Größe des Rahmens der Dialogbox bei gleicher `LayLa`-Beschreibung zu einem Querformat, so bleibt der visuelle Eindruck erhalten:



3.2 Schema-Diagramm des Unix-Betriebsystems

Dieses Beispiel ist eng an ein Diagramm aus einem bekannten Fachbuch über Unix [1] auf Seite 18 angelehnt. Es stellt die Struktur wichtiger Teile, aus denen das Unix-Betriebssystem aufgebaut ist, und deren Zusammenhang schematisch dar (siehe Seite 25).

Dieser Zusammenhang wird zum einen mit Linien und zum anderen durch räumliche Beziehungen ausgedrückt. Bedeutung tragen die Abstände zwischen den Komponenten und deren Ausdehnung sowie die Anordnung nebeneinander und übereinander. Dies sind alles Eigenschaften, die sich gut mit `LayLa` ausdrücken lassen.

Das Diagramm ist vertikal ausgerichtet. Das Fundament bildet die Hardware, die durch die erste Komponente am unteren Rand des Rahmens repräsentiert ist. Mit Hilfe der Relation `NebenRand` ist die Komponente in nördlicher Richtung (aus der Sicht des Rahmens als Quelle) am Rahmen verankert (sie liegt also innerhalb des Rahmens am südlichen Rand). Der Abstand und die Komponentenhöhe ist mit 5% der Rahmenhöhe bemessen. Alle weiteren sichtbaren Komponenten, deren Höhe nicht indirekt spezifiert ist, werden explizit durch die 6 Gleichwert-Relationen auf die gleiche Höhe gesetzt. Komponente 1 und 2 sind gleichgroß und zwischen ihnen ist der gleiche Abstand wie zwischen dem Rahmen und Komponente 1.

Den Raum nördlich von Komponente 2 teilen sich Komponente 3 und 15 im Verhältnis 1/3 zu 2/3. Dabei ist nur die Breite von Komponente 3 angegeben (VarAusdruck). Durch die Unterspezifikation der Breite von Komponente 15 wird ihr automatisch der restliche zur Verfügung stehende Raum in der Breite zugeordnet. Die Reihenfolge der Neben-Relationen in `LayLa` bestimmt auch die Reihenfolge der Anordnung der Komponenten in der gleichen Richtung (von Westen nach Osten bzw. von Norden nach Süden). Daher liegt Komponente 15 östlich von Komponente 3. Da Komponente 15 die Höhe 0 hat, ist sie nicht sichtbar. Sie sorgt dafür, daß die sichbare Komponente 3 nicht automatisch zentriert nördlich von Komponente 2, sondern linksbündig plaziert wird. `LayLa` könnte um Konstrukte erweitert werden, die diese Ausrichtung ohne Hilfe von Phantom-Komponenten direkt beschreibbar machen.

Komponente 4 und 5 teilen sich den Raum nördlich von Komponente 3. Komponente 6 liegt wiederum nördlich von 5 und kann dort die gesamte Breite ausfüllen. Komponente 7 liegt wiederum nördlich von Komponente 6 aber auch nördlich von Komponente 4. Darum wird die unterspezifizierte Breite von Komponente 7 so bestimmt, daß Komponente 7 sowohl über Komponente 4 als auch Komponente 6 liegt. Da der y-Wert von Komponente 7 schon durch den Abstand in der Neben-Relation mit Komponente 6 genau bestimmt ist, ist der Abstand in der Neben-Relation zwischen Komponente 4 und 7 explizit unspezifiziert.

Nach dem gleichen Schema ist die Lage von Komponente 8 östlich von Komponente 6 und 7 beschrieben.

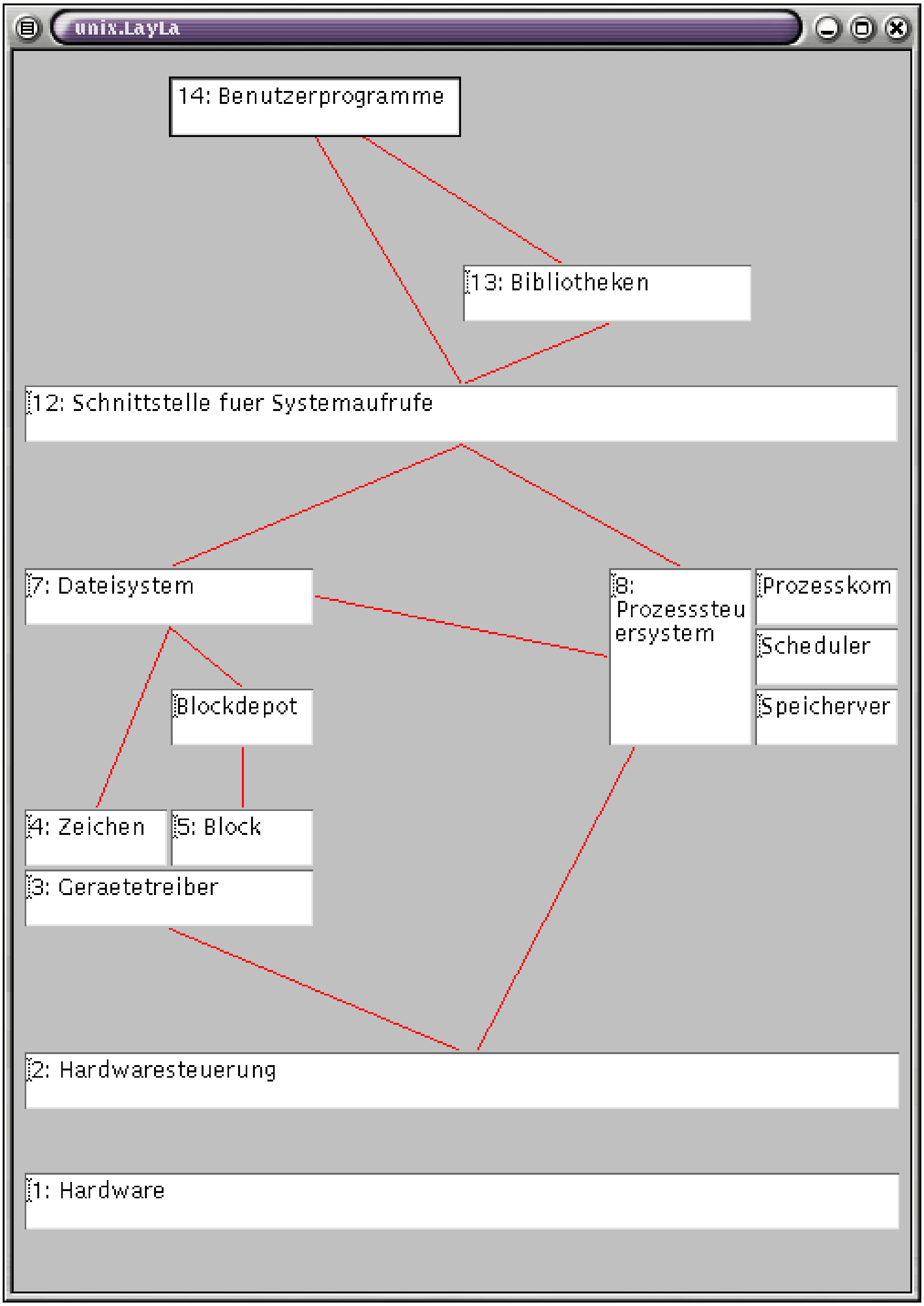
Den Raum östlich von Komponente 8 teilen sich die Komponenten 9, 10 und 11 zu gleichen Teilen, da die Höhen nicht spezifiziert sind.

Die Komponente 12 liegt nördlich von den Komponenten 7, 8 und 9. Analog zur Positionsbestimmung von Komponente 7 wird bei Komponente 12 der Abstand nur in einer der 3 Neben-Relationen spezifiziert.

Komponente 13 und 14 teilen sich zentriert den Platz nördlich von Komponente 12.

```
Komponente 1 Text "1: Hardware" ;
Komponente 2 Text "2: Hardwaresteuerung" ;
Komponente 3 Text "3: Geraetetreiber" ;
Komponente 4 Text "4: Zeichen" ;
Komponente 5 Text "5: Block" ;
Komponente 6 Text "6: Blockdepot" ;
Komponente 7 Text "7: Dateisystem" ;
Komponente 8 Text "8: Prozesssteuersystem" ;
Komponente 9 Text "9: Prozesskommunikation" ;
Komponente 10 Text "10: Scheduler" ;
Komponente 11 Text "11: Speicherverwaltung" ;
Komponente 12 Text "12: Schnittstelle fuer Systemaufrufe" ;
Komponente 13 Text "13: Bibliotheken" ;
Komponente 14 Text "14: Benutzerprogramme" ;
Komponente 15 Text "15: Unsichtbare Komponente zum Strecken" ;

NebenRand Ziel 1 Richtung n Abstand Rahmen.h/20 ;
Rahmen Ziel 1 Wrel 980 Hrel 50 ;
Neben Quelle 1 Ziel 2 Richtung n Abstand Rahmen.h/20 ;
Gleichgross Quelle 1 Ziel 2 ;
Gleichwert Quelle 1 Ziel 3 Variable h ;
Gleichwert Quelle 1 Ziel 4 Variable h ;
Gleichwert Quelle 1 Ziel 5 Variable h ;
Gleichwert Quelle 1 Ziel 6 Variable h ;
Gleichwert Quelle 1 Ziel 7 Variable h ;
Gleichwert Quelle 1 Ziel 12 Variable h ;

Neben Quelle 2 Ziel 3 Richtung n Abstand Rahmen.h/10 ;
Neben Quelle 2 Ziel 15 Richtung n Abstand Rahmen.h/10 ;
Fix Ziel 15 Variable h Pixel 0;
VarAusdruck Quelle 2 Ziel 3 Gleichung Ziel.w=Quelle.w/3;

Neben Quelle 3 Ziel 4 Richtung n Abstand 0  ;
Neben Quelle 3 Ziel 5 Richtung n Abstand 0  ;
Neben Quelle 5 Ziel 6 Richtung n Abstand Rahmen.h/20 ;
Neben Quelle 6 Ziel 7 Richtung n Abstand Rahmen.h/20 ;
Neben Quelle 4 Ziel 7 Richtung n Abstand Unspezifiziert ;

Neben Quelle 6 Ziel 8 Richtung o Abstand Unspezifiziert ;
Neben Quelle 7 Ziel 8 Richtung o Abstand Quelle.w ;

Gleichwert Quelle 6 Ziel 8 Variable w ;
Gleichwert Quelle 8 Ziel 9 Variable w ;
```

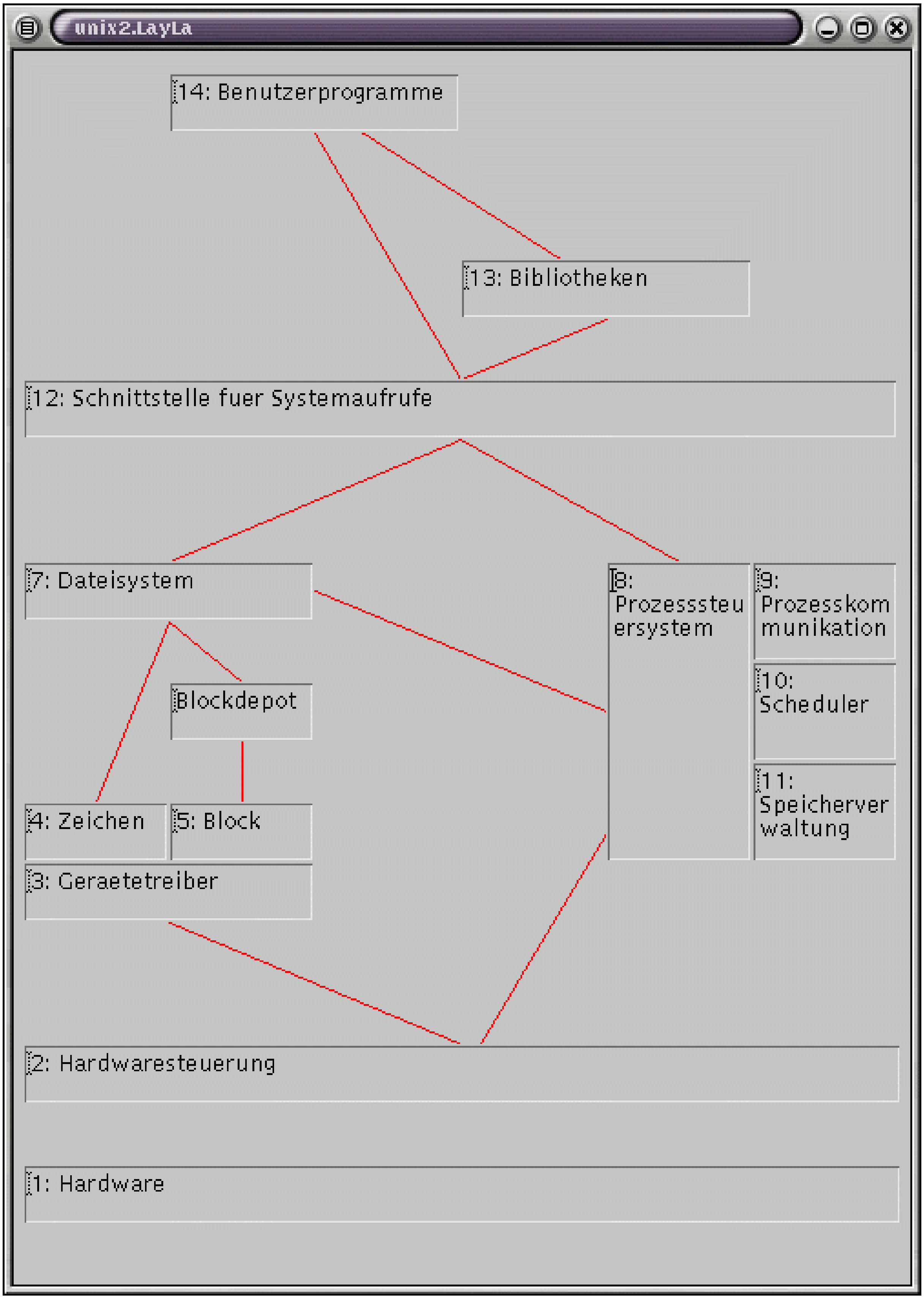
```
Neben Quelle 8 Ziel 9 Richtung o Abstand 0  ;
Neben Quelle 8 Ziel 10 Richtung o Abstand 0  ;
Neben Quelle 8 Ziel 11 Richtung o Abstand 0  ;

Neben Quelle 7 Ziel 12 Richtung n Abstand Rahmen.h/10 ;
Neben Quelle 8 Ziel 12 Richtung n Abstand Unspezifiziert  ;
Neben Quelle 9 Ziel 12 Richtung n Abstand Unspezifiziert ;

Gleichgross Quelle 7 Ziel 13 ;
Gleichgross Quelle 13 Ziel 14 ;
Neben Quelle 12 Ziel 14 Richtung n Abstand Rahmen.h/5 ;
Neben Quelle 12 Ziel 13 Richtung n Abstand Rahmen.h/20 ;

Linie Quelle 2 Ziel 3 ;
Linie Quelle 5 Ziel 6 ;
Linie Quelle 4 Ziel 7 ;
Linie Quelle 6 Ziel 7 ;
Linie Quelle 7 Ziel 12 ;
Linie Quelle 7 Ziel 8 ;
Linie Quelle 2 Ziel 8 ;
Linie Quelle 8 Ziel 12 ;
Linie Quelle 12 Ziel 13 ;
Linie Quelle 13 Ziel 14 ;
Linie Quelle 12 Ziel 14 ;
```


unix.LayLa
14: Benutzerprogramme
13: Bibliotheken
12: Schnittstelle fuer Systemaufrufe
7: Dateisystem
8: Prozesssteuersystem
Prozesskom
Scheduler
Speicherver
Blockdepot
4: Zeichen
5: Block
3: Geraetetreiber
2: Hardwaresteuerung
1: Hardware

Der visuelle Eindruck bleibt erhalten, wenn die Komponente 8 (Prozesssteuersystem) mit einer zusätzlichen `LayLa`-Zeile östlich von Komponente 5 (Block) spezifiziert wird:

```
Neben Quelle 5 Ziel 8 Richtung o Abstand Unspezifiziert ;
```

Das Hinzufügen der obigen Zeile in die vorhergehende `LayLa`-Beschreibung ergibt folgendes Layout, bei dem im Gegensatz zu allen anderen Bildern als Hintergrundfarbe der Komponenten nicht weiß, sondern das gleiche Grau wie für den Rahmen benutzt wird:



unix2.LayLa
14: Benutzerprogramme
13: Bibliotheken
12: Schnittstelle fuer Systemaufrufe
7: Dateisystem
8: Prozesssteuersystem
9: Prozesskommunikation
10: Scheduler
11: Speicherverwaltung
Blockdepot
4: Zeichen
5: Block
3: Geraetetreiber
2: Hardwaresteuerung
1: Hardware

3.3 Layout von Navigations-Diagrammen

3.3.1 Visualisierung von WWW-Navigation

Wie im Abschnitt zur Struktur-Visualisierung als Beispiel schon erwähnt, sind nicht erst seit der weitverbreiteten Nutzung von Hypertext und Hypermedia, die durch das WWW verursacht wurde, gut gestaltete Übersichts-Diagramme (Navigations-Diagramme) als Orientierungs- und Überblicks-Medium nötig.

In diesem Kapitel wird ein Vorschlag zur Gestaltung eines neuen Typs von Navigations-Diagramm mit dem Namen `NavLayout` vorgestellt. Er basiert auf der qualitativen Nachbarschafts-Relation mit Himmelsrichtungen, um ein ergonomisches Layout des Navigations-Diagrammes zu erreichen.

3.3.2 Terminologie

Eine ganze Forschungsgemeinschaft beschäftigt sich mit Hypermedia-Modellen. Hier wird nur ein kleiner begrifflicher Rahmen definiert, der das Verständnis des Anwendungsbeispiels ermöglichen soll. Ein umfassendes Modell findet sich etwa in [2].

Das für diese Arbeit verwendete Hypermedia-Modell besteht aus *Knoten* und *Hyperlinks* zwischen Knoten. Ein Knoten kann irgendein darstellbares Datenobjekt sein (z.B. eine HTML-Datei).

Üblicherweise wird zwar eine Verfeinerung dieses Modells benutzt, bei der Hyperlinks auch auf Stellen innerhalb eines Knotens, sogenannten *anchors*, verweisen können. Diese Verfeinerung ist für diese Arbeit nicht nötig. Das hier verwendete Hypermedia-Modell entspricht beispielsweise in HTML der Einschränkung, daß durch einen Hyperlink nur noch ganze Seiten und nicht Seiten-Fragmente adressiert werden dürfen. Die angegebene URL darf also kein Doppelkreuz (`#`) im Pfad-Teil enthalten.

Im Gegensatz zu dem Hypermedia-Modell von HTML erlaubt das hier verwendete Hypermedia-Modell, daß den Hyperlinks Typen zugeordnet werden können. Diese Typen bestimmen die Semantik des Hyperlinks näher (siehe Abschnitt semantische Relationen).

Der Knoten, der vom Benutzer fokussiert wird, heißt *aktueller Knoten*. Der *aktuelle Kontext* sind alle vom aktuellen Knoten über Hyperlinks direkt oder indirekt erreichbaren Knoten bis zu einer „sinnvollen“ Entfernung.

Navigation ist der Vorgang bei dem der Benutzer in eindeutiger zeitlicher Abfolge den aktuellen Knoten wechselt. Dies kann durch Verfolgen eines Hyperlinks geschehen – muß aber nicht. Der *Navigations-Pfad* ist die zeitlich geordnete Liste aller Knoten, die während der Navigation aktuell waren und des aktuellen Knoten.

3.3.3 `NavLayout`-Diagramme

Ein `NavLayout`-Diagramm soll eine Übersicht über den aktuellen Kontext und den vom Benutzer beschrittenen Navigations-Pfad vermitteln. Damit visualisiert ein `NavLayout`-Diagramm zwei sehr unterschiedliche Arten von semantischen Relationen zwischen den Knoten. Zum einen visualisiert es die allgemeine Struktur des aktuellen Kontextes, die durch die typisierten Hyperlinks zwischen den Knoten zustande kommt. Zum anderen visualisiert es den Navigations-Pfad, der durch die dynamische Navigation des Benutzers zustande kommt.

In einem `NavLayout`-Diagramm werden Knoten auf Komponenten abgebildet. Hyperlinks zwischen Knoten und der Navigations-Pfad werden auf räumliche Relationen *und* Linien zwischen den Komponenten abgebildet. Die Abbildungen müssen nicht total sein: Es ist meist nur ein Teil der Knoten, Hyperlinks und des Navigations-Pfades abgebildet.

In den Komponenten muß eine aussagekräftige visuelle Darstellung des Knoten-Inhalts erscheinen. Dies könnte z.B. eine Titel-Zeile, ein verkleinertes Bild oder ein Piktogramm sein. Der aktuelle Knoten ist als größte Komponente im Zentrum des Rahmens dargestellt.

Da die Navigation eine Interaktion zwischen Benutzer und Diagramm ist, ist der Aspekt der Interaktion auch wichtig. Er würde allerdings den Fokus und Rahmen dieser Arbeit sprengen und wird darum nicht näher betrachtet. Es reicht, wenn man sich vorstellt, daß man durch das Klicken auf eine Komponente den aktuellen Knoten wechselt. Dieser Wechsel bewirkt den Aufbau eines neuen Diagramms und eventuell auch den Wechsel des Inhalts in einem Browser- oder Editor-Fenster, der den eigentlichen Inhalt des durch die Komponente abgebildeten Knotens zugreifbar macht.

3.3.3.1 Abbildung semantischer Relationen auf räumliche Relationen Es sind viele Abbildungen von semantischen Relationen auf räumliche Relationen denkbar. Mir erscheint die Abbildung auf die Nachbarschafts-Relation am sinnvollsten. Dabei werden semantische Relationen auf Richtungen abgebildet. Die Anzahl der semantischen Relationen kann beliebig groß sein. Die Anzahl der in einem Diagramm sinnvoll unterscheidbaren Richtungen ist nach meiner Einschätzung auf die 8 Oktanten der qualitativen Richtungsbeschreibung begrenzt. Es ist daher möglich und wahrscheinlich, daß einige semantischen Relationen nicht auf diese Weise abgebildet werden können. Dieser „Rest“ kann auf andere Weise (z.B. Auflistung innerhalb der Komponente) oder gar nicht abgebildet werden.

Hyperlinks und die dadurch ausgedrückten semantischen Relationen sind im allgemeinen gerichtet. Für jede dieser semantischen Richtungen muß jeweils eine räumliche Richtung reserviert werden. Mit der qualitativen Richtungsbeschreibung lassen sich somit 4 semantische Relationen ausdrücken. Es ist sinnvoll die jeweils gegenüberliegenden Oktanten einer semantischen Relation zuzuordnen. Auf diese Weise erhält man 4 *semantische Achsen*.

Einer dieser semantischen Achsen wird die Folge-Beziehung des Navigations-Pfades zugeordnet, sie wird dadurch zur *Navigations-Achse*. Die verbleibenden 3 Achsen können den im Anwendungs-Kontext wichtigsten semantischen Relationen (=Hyperlink-Typen) zugeordnet werden (*Hyperlink-Achsen*).

Auf der *Navigations-Achse* sind nur Knoten enthalten, die nicht schon auf einer der *Hyperlink-Achsen* enthalten sind. Sollte die Navigation entlang einer der *Hyperlink-Achsen* stattfinden, so muß dies graphisch hervorgehoben werden (z.B. Verbindungslinie ändern).

Die Anzahl der darstellbaren Komponenten ist durch die Größen der Komponenten, deren Abstand und durch die Größe des Rahmens begrenzt.

Es könnte also erst durch eine gezielte Kommunikation eines `NavLayout`-Diagramm-Moduls mit dem Layout-Algorithmus entschieden werden, welche Knoten des aktuellen Kontextes als Komponenten im `NavLayout`-Diagramm erscheinen. Das dafür notwendige Protokoll ist nicht Thema dieser Arbeit und ist hier deshalb nur am Rande erwähnt.

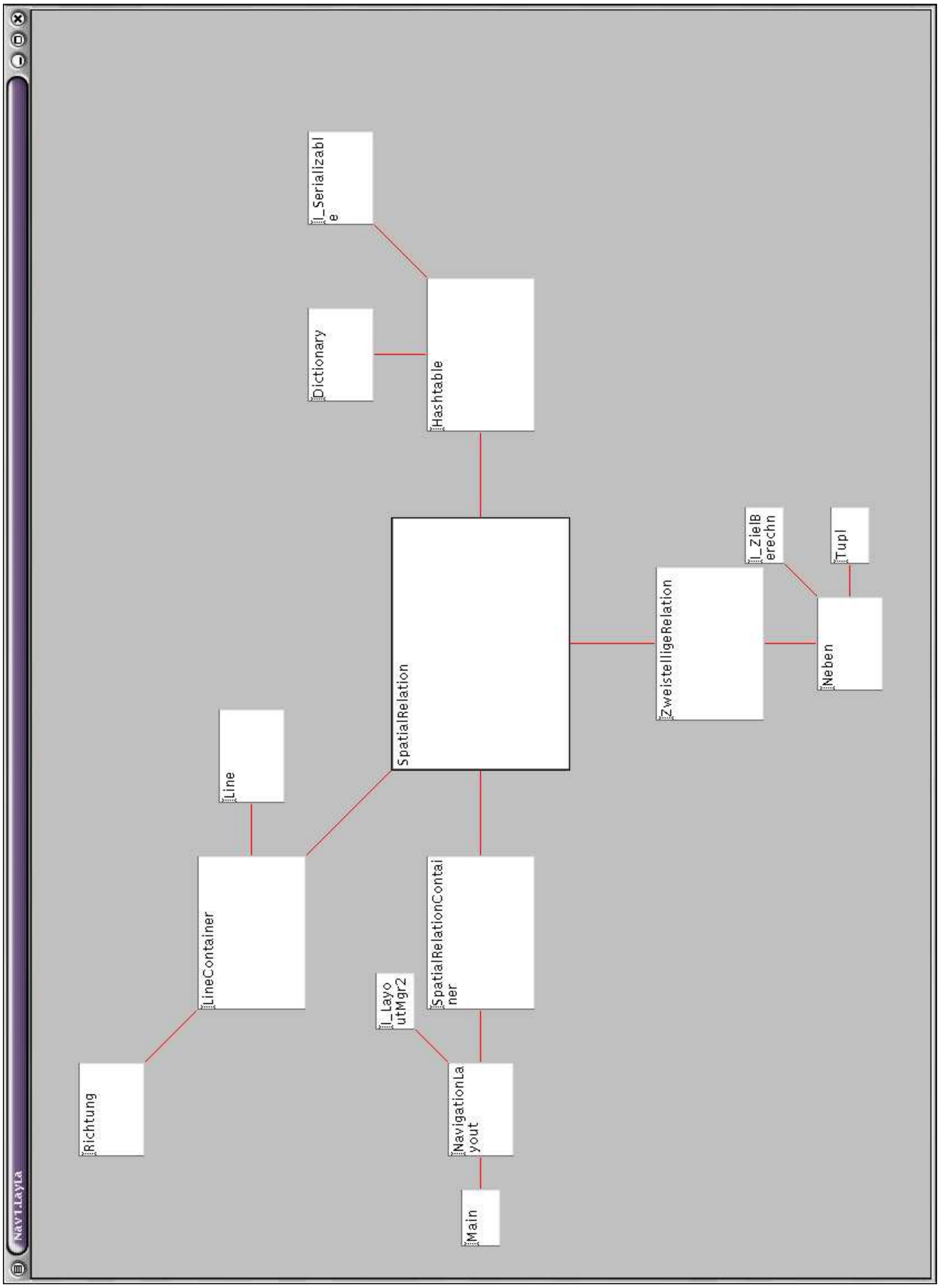
3.3.4 `LayLa`-Beschreibung von `NavLayout`-Diagrammen

Die Positionierung der Komponente des aktuellen Knotens mit der Relation *Rel-Zentrum* und der Aufbau der *semantischen Achsen* durch die Relation *Rel-Neben* mit qualitativer Richtungs-Angabe ist offensichtlich.

Schwieriger ist die Beschreibung der Größen der Komponenten und deren Abstände. Die Komponente im Zentrum soll am größten sein. Hier ist es sinnvoll, sie relativ zur Größe des Rahmens zu definieren. Dies ist mit der Relation *Rel-Rahmen* beschreibbar. Alle weiteren Komponenten sollten hinsichtlich ihrer Größe mit wachsendem Abstand (hier: Anzahl der semantischen Relationen, die dazwischen liegen) von der zentralen Komponente kleiner werden. Dies läßt sich mit der Relation *Rel-RelGross* direkt beschreiben. In den folgenden beispielhaften `NavLayout`-Diagrammen dient der *Goldene Schnitt* (gerundet auf 0.618) als Skalierungsfaktor, der in Breite wie Höhe angegeben wird (Attribute `Wrel 618 Hrel 618`). Die Abstände zwischen den benachbarten Komponenten nehmen mit der Breite der Quell-Komponente ab ($AbstandQuelle.w/3$).

Als Beispiel dient die Navigation durch einige Java-Klassen des für diese Arbeit erstellten Prototypen. In den Knoten bzw. Komponenten sind die Java-Klassen jeweils durch ihren Namen repräsentiert. Interfaces-Klassen sind durch ein dem Namen vorangestelltes `I_` zu erkennen. Die Navigationsachse geht von Nordwesten nach Südosten. Die dazu orthogonale Achse ist mit der Relation *implemeniert Interface* belegt. Die vertikale Achse ist mit der Unterklasse-Oberklasse-Relation (Generalisierung) belegt. Die horizontale Achse ist mit der *verwendet*-Relation belegt. Als „sinnvolle“ Maximalentfernung der zu visualisierenden Knoten wurde die semantische Entfernung 3 vom aktuellen Knoten gewählt.

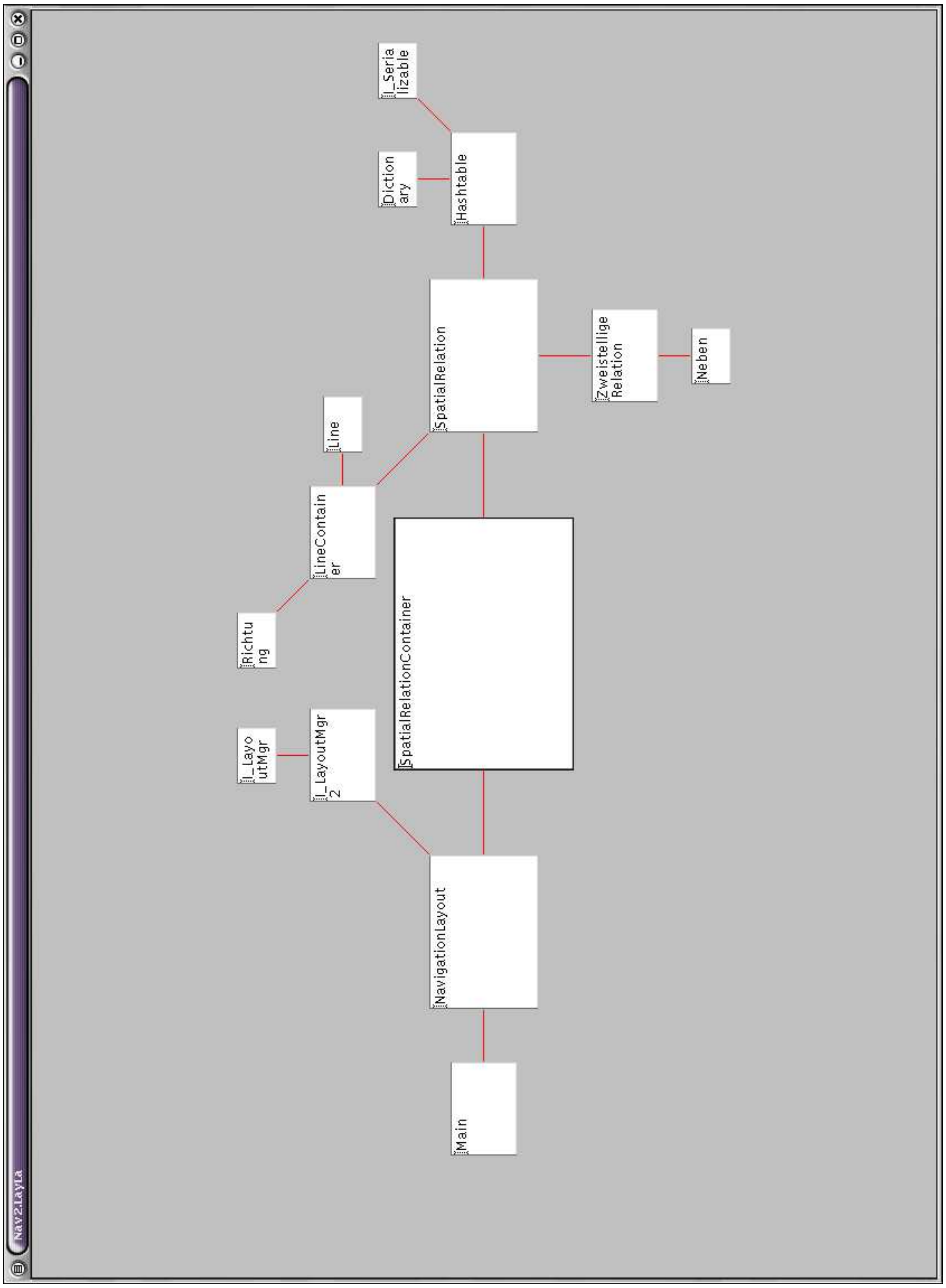
Die beiden folgenden Bildschirm-Abzüge zeigen die `NavLayout`-Diagramme und die dazugehörige `LayLa`-Beschreibung vor und nach einem Navigationsschritt vom Knoten „SpatialRelation“ zum Knoten „SpatialRelationContainer“.




```
Komponente 1 Text "SpatialRelation" ;
Komponente 2 Text "SpatialRelationContainer" ;
Komponente 3 Text "ZweistelligeRelation" ;
```

```
Komponente 4 Text "Neben" ;
Komponente 5 Text "I_ZielBerechne" ;
Komponente 6 Text "Tupl" ;
Komponente 7 Text "NavigationLayout" ;
Komponente 8 Text "I_LayoutMgr2" ;
Komponente 9 Text "Hashtable" ;
Komponente 10 Text "LineContainer" ;
Komponente 11 Text "Line" ;
Komponente 12 Text "Richtung" ;
Komponente 13 Text "Dictionary" ;
Komponente 14 Text "I_Serializable" ;
Komponente 15 Text "Main" ;
Zentrum Ziel 1 ;
Rahmen Ziel 1 Wrel 200 Hrel 200 ;
Neben Quelle 1 Ziel 2 Richtung w Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 2 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 2 ;
Neben Quelle 1 Ziel 9 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 9 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 9 ;
Neben Quelle 1 Ziel 3 Richtung s Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 3 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 3 ;
Neben Quelle 1 Ziel 10 Richtung nw Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 10 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 10 ;
Neben Quelle 10 Ziel 11 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 10 Ziel 11 Wrel 618 Hrel 618 ;
Linie Quelle 10 Ziel 11 ;
Neben Quelle 10 Ziel 12 Richtung nw Abstand Quelle.w/3 ;
RelGross Quelle 10 Ziel 12 Wrel 618 Hrel 618 ;
Linie Quelle 10 Ziel 12 ;
Neben Quelle 9 Ziel 13 Richtung n Abstand Quelle.w/3 ;
RelGross Quelle 9 Ziel 13 Wrel 618 Hrel 618 ;
Linie Quelle 9 Ziel 13 ;
Neben Quelle 9 Ziel 14 Richtung no Abstand Quelle.w/3 ;
RelGross Quelle 9 Ziel 14 Wrel 618 Hrel 618 ;
Linie Quelle 9 Ziel 14 ;
Neben Quelle 3 Ziel 4 Richtung s Abstand Quelle.w/3 ;
RelGross Quelle 3 Ziel 4 Wrel 618 Hrel 618 ;
Linie Quelle 3 Ziel 4 ;
Neben Quelle 4 Ziel 5 Richtung no Abstand Quelle.w/3 ;
RelGross Quelle 4 Ziel 5 Wrel 618 Hrel 618 ;
Linie Quelle 4 Ziel 5 ;
Neben Quelle 4 Ziel 6 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 4 Ziel 6 Wrel 618 Hrel 618 ;
Linie Quelle 4 Ziel 6 ;
```

```
Neben Quelle 2 Ziel 7 Richtung w Abstand Quelle.w/3 ;
RelGross Quelle 2 Ziel 7 Wrel 618 Hrel 618 ;
Linie Quelle 2 Ziel 7 ;
Neben Quelle 7 Ziel 8 Richtung no Abstand Quelle.w/3 ;
RelGross Quelle 7 Ziel 8 Wrel 618 Hrel 618 ;
Linie Quelle 7 Ziel 8 ;
Neben Quelle 7 Ziel 15 Richtung w Abstand Quelle.w/3 ;
RelGross Quelle 7 Ziel 15 Wrel 618 Hrel 618 ;
Linie Quelle 7 Ziel 15 ;
```




```
Komponente 1 Text "SpatialRelation" ;
Komponente 2 Text "SpatialRelationContainer" ;
Komponente 3 Text "ZweistelligeRelation" ;
```

```
Komponente 4 Text "Neben" ;
Komponente 7 Text "NavigationLayout" ;
Komponente 8 Text "I_LayoutMgr2" ;
Komponente 9 Text "Hashtable" ;
Komponente 10 Text "LineContainer" ;
Komponente 11 Text "Line" ;
Komponente 12 Text "Richtung" ;
Komponente 13 Text "Dictionary" ;
Komponente 14 Text "I_Serializable" ;
Komponente 15 Text "Main" ;
Komponente 16 Text "I_LayoutMgr" ;
Zentrum Ziel 2 ;
Rahmen Ziel 2 Wrel 200 Hrel 200 ;
Neben Quelle 2 Ziel 1 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 2 Ziel 1 Wrel 618 Hrel 618 ;
Linie Quelle 2 Ziel 1 ;
Neben Quelle 1 Ziel 9 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 9 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 9 ;
Neben Quelle 1 Ziel 3 Richtung s Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 3 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 3 ;
Neben Quelle 1 Ziel 10 Richtung nw Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 10 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 10 ;
Neben Quelle 10 Ziel 11 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 10 Ziel 11 Wrel 618 Hrel 618 ;
Linie Quelle 10 Ziel 11 ;
Neben Quelle 10 Ziel 12 Richtung nw Abstand Quelle.w/3 ;
RelGross Quelle 10 Ziel 12 Wrel 618 Hrel 618 ;
Linie Quelle 10 Ziel 12 ;
Neben Quelle 9 Ziel 13 Richtung n Abstand Quelle.w/3 ;
RelGross Quelle 9 Ziel 13 Wrel 618 Hrel 618 ;
Linie Quelle 9 Ziel 13 ;
Neben Quelle 9 Ziel 14 Richtung no Abstand Quelle.w/3 ;
RelGross Quelle 9 Ziel 14 Wrel 618 Hrel 618 ;
Linie Quelle 9 Ziel 14 ;
Neben Quelle 3 Ziel 4 Richtung s Abstand Quelle.w/3 ;
RelGross Quelle 3 Ziel 4 Wrel 618 Hrel 618 ;
Linie Quelle 3 Ziel 4 ;
Neben Quelle 2 Ziel 7 Richtung w Abstand Quelle.w/3 ;
RelGross Quelle 2 Ziel 7 Wrel 618 Hrel 618 ;
Linie Quelle 2 Ziel 7 ;
Neben Quelle 7 Ziel 8 Richtung no Abstand Quelle.w/3 ;
RelGross Quelle 7 Ziel 8 Wrel 618 Hrel 618 ;
Linie Quelle 7 Ziel 8 ;
Neben Quelle 7 Ziel 15 Richtung w Abstand Quelle.w/3 ;
```

```
RelGross Quelle 7 Ziel 15 Wrel 618 Hrel 618 ;
Linie Quelle 7 Ziel 15 ;
Neben Quelle 8 Ziel 16 Richtung n Abstand Quelle.w/3 ;
RelGross Quelle 8 Ziel 16 Wrel 618 Hrel 618 ;
Linie Quelle 8 Ziel 16 ;
```

Die beiden LayLa-Beschreibungen unterscheiden sich syntaktisch in vier Punkten.

Die erste Änderung

```
Zentrum Ziel 1 ;
Rahmen Ziel 1 Wrel 200 Hrel 200 ;
```

```
Zentrum Ziel 2 ;
Rahmen Ziel 2 Wrel 200 Hrel 200 ;
```

ergibt sich aus dem Wechsel des aktuellen Knotens, der als Komponente im Zentrum visualisiert werden soll.

Die zweite Änderung

```
Neben Quelle 1 Ziel 2 Richtung w Abstand Quelle.w/3 ;
RelGross Quelle 1 Ziel 2 Wrel 618 Hrel 618 ;
Linie Quelle 1 Ziel 2 ;
```

```
Neben Quelle 2 Ziel 1 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 2 Ziel 1 Wrel 618 Hrel 618 ;
Linie Quelle 2 Ziel 1 ;
```

kommt daher, daß die Nachbarschafts-Relation in dieser Anwendung immer ausgehend von der Komponente im Zentrum aufgebaut wird, der Navigationsschritt also zu einer Vertauschung der Komponenten-Bezeichner im Quell- und Ziel-Attribut führt. Dies ist nur eine Annahme über die NavLayout-Anwendung, um das Beispiel klarer zu machen.

Drittens entfielen

```
Komponente 5 Text "I_ZielBerechne" ;
Komponente 6 Text "Tupl" ;
Neben Quelle 4 Ziel 5 Richtung no Abstand Quelle.w/3 ;
RelGross Quelle 4 Ziel 5 Wrel 618 Hrel 618 ;
Linie Quelle 4 Ziel 5 ;
Neben Quelle 4 Ziel 6 Richtung o Abstand Quelle.w/3 ;
RelGross Quelle 4 Ziel 6 Wrel 618 Hrel 618 ;
Linie Quelle 4 Ziel 6 ;
```

das sind sämtliche Zeilen, die die Knoten `I_ZielBerechne` und `Tupl` betreffen. Diese Knoten haben nach dem Navigationsschritt einen semantischen Abstand vom aktuellen Knoten, der die gewählte Maximalentfernung 3 überschreitet, weshalb sie nicht mehr visualisiert werden.

Viertens kamen

```
Komponente 16 Text "I_LayoutMgr" ;
Neben Quelle 8 Ziel 16 Richtung n Abstand Quelle.w/3 ;
RelGross Quelle 8 Ziel 16 Wrel 618 Hrel 618 ;
Linie Quelle 8 Ziel 16 ;
```

neu hinzu, weil der Knoten `I_LayoutMgr` nach dem Navigationsschritt einen semantischen Abstand von 3 vom aktuellen Knoten hat und damit nicht mehr über der Maximalentfernung liegt.

Diese syntaktischen Unterschiede sind derart, dass ein Programm zur Bearbeitung eines Navigationsschritts aus der ersten `LayLa`-Beschreibung und der Angabe des alten und neuen aktuellen Knotens die zweite `LayLa`-Beschreibung berechnen kann.

Diese Berechnung steht hier aber nicht im Vordergrund des Interesses. Wichtiger ist, wie sich die beschriebenen Änderungen auf das Layout und den visuellen Eindruck auswirken.

Die weggefallenen und die hinzugekommenen Knoten haben jeweils die kleinste Ausdehnung, so dass ihr Wegfallen oder Hinzukommen den visuellen Eindruck nur wenig beeinflusst. Alle weiteren Knoten haben sich nur leicht verschoben. Da die Größenverhältnisse (bis auf die Vertauschung der Knoten 1 und 2) sowie die Anordnung durch die Himmelsrichtungen gleich bleiben, bleibt auch der visuelle Eindruck erhalten und das Layout ist im Sinne dieser Arbeit ergonomisch.

In dem Beispiel sind der alte und der neue aktuelle Knoten direkt benachbart. Für Navigationsschritte, bei denen die Entfernung zwischen diesen beiden Knoten größer ist, nehmen auch die Änderungen der `LayLa`-Beschreibung und damit des Layouts zu. Der visuelle Eindruck kann natürlich nicht bei beliebig großen Änderungen der `LayLa`-Beschreibung erhalten bleiben. Diese Forderung ergäbe auch keinen sinnvollen Ergonomiebegriff.

4 Überlegungen zum Layout-Algorithmus

4.1 Existenz und Eindeutigkeit eines Layouts

Ein Layout ist eine Funktion, die jeden *Komponenten-Bezeichner* auf ein 4-Tupel (x, y, w, h) in der Art abbildet, daß die ermittelte Pixel-Position x, y und -Größe w, h alle beschriebenen Relationen erfüllt und damit korrekt sind. Für jede Layout-Beschreibung in `LayLa` gibt es drei Fälle.

1. Es gibt kein korrektes Layout (widersprüchliche `LayLa`-Beschreibung).
2. Es gibt genau ein korrektes Layout (Eindeutigkeit).
3. Es gibt mehrere Möglichkeiten für ein korrektes Layout (Unterspezifikation).

4.1.1 Widersprüchlichkeit

Widersprüche, die in Layout-Beschreibungen in `LayLa` möglich sind, lassen sich in zwei Klassen einteilen.

Zum einen in aufhebbare Widersprüche und zum anderen in nicht-aufhebbare Widersprüche. Nicht-aufhebbare Widersprüche sind zumeist qualitativer Natur. Beispielsweise kann eine Komponente b nicht im Norden *und* im Süden einer Komponente a liegen. Diese Widersprüche lassen sich auch ohne Kenntnis der Größe des Rahmens finden.

Liegt allerdings nur ein quantitativer Abstands-Konflikt vor, ist es durchaus sinnvoll, den Widerspruch durch den Algorithmus aufhebbar zu machen. Befindet sich zum Beispiel die Komponente b mit Abstand 30 und mit Abstand 50 im Süden von a, dann könnte man den Widerspruch aufheben, indem man eine der Lösungen verwirft oder den Mittelwert 40 als Lösung nimmt.

4.1.2 Unterspezifikation

Es kann sein, daß die `LayLa`-Beschreibung z.B. die Position einer Komponente eindeutig bestimmt, aber nicht die Größe. In diesem Fall könnte der Algorithmus jeden beliebigen Wert für die Größe wählen, um ein korrektes Layout zu erhalten. Der Algorithmus sollte allerdings einen „sinnvollen“ Wert finden.

Ein sinnvolles Ziel für die Auswahl von Default-Werten ist das Verhindern von Überlappungen bei gleichzeitiger Maximierung der Größe der Komponente.

4.1.3 Überspezifikation

Der folgende an die Idee einer induktiven Definition angelehnte Versuch einer Präzisierung der Überspezifikation kann auf den ersten Blick etwas verwirren. Dieser außergewöhnlich weit gefaßte Begriff von Überspezifikation kann helfen, im Zusammenhang mit der Unterspezifikation die verschiedenen Fälle von sinnvollem Layout zu strukturieren. Es hätte allerdings den Rahmen und die Zielsetzung dieser Arbeit gesprengt, dies erschöpfend zu diskutieren, sodaß dies hier in erster Linie als Ansatzpunkt für weitergehende Arbeit zu verstehen ist.

Eine Überspezifikation liegt vor, wenn eine schon eindeutige `LayLa`-Spezifikation, die auch durch das Auflösen von Unterspezifikationen entstanden sein kann, um weitere Relationen ergänzt wird. Diese Ergänzung kann

1. zu aufhebbaren oder nicht-aufhebbaren Widersprüchen führen.
2. zu einer (teilweisen) Auflösung von Unterspezifikationen führen.
3. wiederum zu einer eindeutigen `LayLa`-Spezifikation führen. Hierbei gibt es drei Möglichkeiten:
   (a) Überspezifikation, die keine Änderung des resultierenden Layouts bewirkt, da die Ergänzungen redundant sind.
   (b) Überspezifikation, die zwar eine Änderung des resultierenden Layouts bewirkt, die aber nur aus hinzugefügten Komponenten besteht.
   (c) Überspezifikation mit neuer Information, die das Layout bisheriger Komponenten verändert.

Als Beispiel für die Fälle 3a, 3b und 3c soll die folgende Variante der `LayLa`-Dialogbox dienen.

```
Komponente 1 Text "Festplatte wirklich formatieren?" ;
Komponente 2 Text "OK" ;
Zentrum Ziel 1 ;
Rahmen Ziel 1 Wrel 800 Hrel 800 ;
Neben Quelle 1 Ziel 2 Richtung s Abstand 0 ;
```

Diese `LayLa`-Spezifikation ist nach Auflösung der Unterspezifikationen von Breite und Höhe der Komponente 2 eindeutig. Man betrachte die drei folgenden Varianten von Ergänzungen, die zu Überspezifikationen führen:

```
Neben Quelle 2 Ziel 1 Richtung n Abstand 0 ;
```

Die obige Ergänzung einer *Rel-Neben*-Relation ist zwar nicht syntaktisch gleich aber semantisch redundant zu der vorhandenen *Rel-Neben*-Relation und führt somit nicht zu einer Änderung des Layouts (Fall 3a).

Die folgende Ergänzungs-Variante entspricht dem Fall 3b. Hier verändern sich die bestehende Komponenten 1 und 2 nicht und werden nur durch die Komponente 3 ergänzt.

```
Komponente 3 Text "Abbruch" ;
Neben Quelle 1 Ziel 3 Richtung w Abstand 0 ;
```

Wenn man in der vorigen Ergänzung die Richtung ändert, dann handelt es sich um den Fall 3c.

```
Komponente 3 Text "Abbruch" ;
Neben Quelle 1 Ziel 3 Richtung s Abstand 0 ;
```

Die hinzugefügte Komponente 3, die wie Komponente 2 ebenfalls südlich von Komponente 1 liegt, legt ein zwar geändertes aber weiterhin eindeutiges Layout nahe: Komponente 2 und 3 müssen sich den Platz an der Südseite von Komponente 1 teilen. Da die Breiten von Komponente 2 und 3 beide unterspezifiziert sind und die Breite von 1 spezifiziert ist, liegt eine Aufteilung der Breite von 1 im gleichen Verhältnis auf 2 und 3 nahe. Die Ergänzung der Komponente 3 in Richtung Süden führt also unter Berücksichtigung der Unterspezifikation der Breiten zu einem eindeutigen Layout, in dem die Ausdehnung der bestehenden Komponente 2 verändert wird. Derartige Zusammenhänge zwischen Über- und Unterspezifikation sollte ein Algorithmus berücksichtigen.

4.2 Auswahl der Implementierungs-Sprache

Das Finden eines korrekten und „sinnvollen“ Layouts ist ein Suchproblem. Für eine Lösung sind besonders Prolog oder Constraint-Sprachen geeignet. Prolog und Constraint-Sprachen sind allerdings kaum geeignet, die grafische Ausgabe des Layouts zu realisieren. Für die grafische Ausgabe des Layouts eignet sich Java besonders. Für die Lösung des Suchproblems ist Java wiederum nicht so geeignet, da der Algorithmus für die Suche explizit implementiert werden muß. Der Programmieraufwand für die Grafik in Prolog oder Constraint-Sprachen erscheint mir höher als der Programmieraufwand für die Suche in Java, zumal die meisten `LayLa`-Relationen den Suchraum erheblich einschränken und das Finden einer optimalen Lösung nicht Ziel dieser Arbeit ist. Deshalb wurde Java als Implementierungssprache für den Prototyp gewählt.

4.3 Prinzip der Layout-Berechnung

Alle Relationen von `LayLa` können als Regeln aufgefaßt werden, um aus bekannten Pixel-Werten auf neue Pixel-Werte zu schließen. Die Erzeugung von Regeln aus Relationen muß unter Beachtung der schon beschriebenen Aspekte der *Unterspezifikation* und *Überspezifikation* erfolgen. Da Regeln unidirektional sind und Relationen eventuell bidirektional, muß bei der Erzeugung von Regeln aus Relationen auch der Aspekt der *Bidirektionalität* von Relationen betrachtet werden.

Das Ziel der Layout-Berechnung ist immer das vollständige 4-Tupel (x, y, w, h) für jede Komponente. Aus diesem Grund bietet es sich an, mit den erzeugten Regeln eine Sättigung durch Vorwärtsschließen zu betreiben.

Vor Beginn des Sättigungs-Prozesses ist die Größe des Rahmens (w, h) in Pixel bekannt. Immer anwendbar sind die Regeln, die aus Relationen abgeleitet werden können, die eine Komponente mit dem Rahmen in Beziehung setzen oder einer Variablen einen konstanten Wert zuordnen. Beispiele für solche Relationen sind Relation *Rel-Rahmen*, die die Ausdehnung einer Komponente linear von der Ausdehnung des Rahmens abhängig macht, oder die Relation *Rel-Fix*, die jeweils einen Pixel-Wert einer Komponente durch eine Konstante bestimmt.

Bei den genannten Relationen ist die Schlußrichtung eindeutig. Es kann nur von der Quelle (Rahmen oder Konstante) auf das Ziel geschlossen werden. Dabei wird durch jede Regelanwendung mindestens ein Wert der im Ziel-Attribut angegebenen Komponente bestimmt und auch eindeutig festgelegt.

Eine Umkehrung der Schlußrichtung vom Ziel zur Quelle hätte bei den Relationen zum Rahmen zur Folge, daß auf die Größe des Rahmens geschlossen würde. Dies ist aber nicht beabsichtigt, da die Größe des Rahmens als bekannt vorausgesetzt wird und nicht ermittelt werden soll. Relationen, die eine Umkehrung der Schlußrichtung erlauben, sind *bidirektional*. Alle anderen sind *unidirektional*. Insbesondere sind aus den gerade erläuterten Gründen alle abgeleiteten Relationen aus der Produktion *Einstellige-Relations-Beschreibung* unidirektional.

4.3.1 Bidirektionale räumliche Relationen

Bidirektional sind die Basis-Relationen *Rel-Neben* und *Rel-Linie* sowie die abgeleiteten Relationen aus der Produktion *Zweistellige-Relations-Beschreibung*.

Die dritte Basis-Relation *Rel-VarAusdruck* ist trotz der Symmetrie der Gleichheit nicht notwendig bidirektional, da hier eine Variable vor dem Gleichheitszeichen mit *mehreren* Variablen hinter dem Gleichheitszeichen in Relation gesetzt werden kann. Die Umkehrung der Schlußrichtung kann also mit einem Verlust der Eindeutigkeit der produzierten Regel verbunden sein. Es ist daher sinnvoll für diese Relation nur eine Regel zu erzeugen, die aus den bekannten Werten für alle Variablen hinter dem Gleichheitszeichen auf den Wert der Ziel-Variablen vor dem Gleichheitszeichen schließt.

Die Relation *Rel-VarAusdruck* ist für die Fälle bidirektional, in denen im arithmetischen Ausdruck hinter dem Gleichheitszeichen nur eine Variable enthalten ist. Dies ist bei allen abgeleiteten Relationen aus der Produktion *Zweistellige-Relations-Beschreibung* der Fall. Insbesondere bei den Relationen *Rel-Gleichwert* und *Rel-Gleichgross* ist die Umkehrung sehr einfach, da hier die Variablen ohne weitere Operatoren direkt in Gleichheitsrelation stehen. Bei der Relation *Rel-RelGross* muß bei der Umkehrung auch der Skalierungsfaktor umgestellt werden.

Für die Umkehrung der Relation *Rel-Neben* muß der Wert des Richtungs-Attributs durch Drehung um 180 Grad ebenfalls umgekehrt werden. Da Quell- und Ziel-Komponenten vertauscht werden, werden auch im arithmetischen Ausdruck des Abstands-Attributes Quell- und Ziel-Variablen vertauscht. Bei der Verwendung von Konstanten und Rahmen-Variablen hat die Vertauschung keinen Einfluß auf den Abstands-Wert.

Die Relation *Rel-Linie* ist bidirektional, da keine einseitigen Pfeile oder ähnliches vorhanden sind. Im Gegensatz zu den anderen Relationen ist die Betrachtung der Bidirektionalität von *Rel-Linie* allerdings nicht nötig, da *Rel-Linie* keinen Einfluß auf den Layout-Algorithmus hat. Linien werden erst nach Bestimmung aller Komponenten-Variablen gezogen und nicht zur Bestimmung der Werte dieser Variablen benötigt.

4.3.2 Der prototypische `LayLa`-Layout-Algorithmus `Laylator`

Der `Laylator` berücksichtigt die Aspekte der Bidirektionalität, Unterspezifikation und Überspezifikation nur am Beispiel der *Rel-Neben*-Relation mit qualitativer Richtungsangabe. Wie im Abschnitt zur Unter- und Überspezifikation schon erläutert, läßt die qualitative Richtungsangabe mit Himmelsrichtungen ganz natürlich einen gewissen Entscheidungsspielraum für die Positionierung an einer spezifizierten Seite, was bei einer quantitativen Richtungsangabe nicht der Fall ist. So bilden die Neben-Relationen mit qualitativer Richtungsangabe eine überschaubare kleine Klasse von Relationen, an denen man ebenfalls die Berücksichtigung der Bidirektionalität exemplarisch zeigen kann. Außerdem bedeutet die Beschränkung auf diese Klasse auch eine Einschränkung des Suchraumes und dient der Vereinfachung des `Laylator`. Es lassen sich trotzdem alle wesentlichen Aspekte der Layout-Generierung demonstrieren, sodaß diese Beschränkung sinnvoll gewählt ist.

Der `Laylator` läßt sich in vier Phasen gliedern, die aufeinander aufbauen. In den ersten drei Phasen werden die Regeln erzeugt. In der vierten Phase werden die erzeugten Regeln ausgewertet und damit das Layout bestimmt. Danach können Komponenten und Linien anhand der ermittelten Pixel-Werte positioniert werden.

4.3.2.1 Erzeugung der Regeln und Steuerung des Vorwärtsschließens Regeln sind im `Laylator` funktional. Aus Relationen werden im Verlauf der ersten drei Phasen des Algorithmus Funktionen erzeugt, die jeweils genau einen Variablen-Wert einer Ziel-Komponente errechnen können. Diese Funktionen benötigen abhängig von der Art der Relation entweder keinen, einen oder mehrere Parameter-Variablen-Werte von Komponenten oder des Rahmens.

Um eine geeignete Reihenfolge der Regelanwendung zu finden, sind die Abhängigkeiten *aller* im Verlauf des Layout-Prozesses erzeugten Funktionen untereinander in einer DAG-Datenstruktur (gerichteter und azyklischer Graph) repräsentiert. Jeder Knoten entspricht einer Funktion. Die Abhängigkeit zwischen Funktions-Wert einer Funktion f_1 und einer anderen Funktion f_2, die diesen Wert als Parameter hat, ist durch eine gerichtete Kante von dem f_1-Knoten zu dem f_2-Knoten repräsentiert. In Abhängigkeit von der Anzahl der Parameter von f_2 sind auf den entsprechenden Knoten keine, eine oder mehrere Kanten gerichtet. Die Kanten sind mit der entsprechenden Komponenten-Variablen markiert. Alle von einem Knoten weggerichteten Kanten haben die gleiche Markierung. Die Anzahl der weggerichteten Kanten hängt von der Anzahl der Funktionen (Knoten) ab, die den Funktions-Wert als Parameter nutzen.

Die „Wurzeln“ des Graphen („Quellen“ in der üblichen Terminologie der Graphentheorie) bilden die parameterlosen Funktionen, die die Werte der w- und h-Variablen des Rahmens liefern, sowie alle weiteren parameterlosen Funktionen, die zum Beispiel aus *Rel-Fix*-Relationen erzeugt werden.

Der zu erzeugende zyklusfreie Graph ist *vollständig*, wenn für jede Komponente vier Knoten und für den Rahmen zwei weitere Knoten vorhanden sind und auf jeden Knoten genausoviele Kanten gerichtet sind wie die repräsentierte Funktion Parameter hat. Jeder Parameter einer jeden Funktion ist dann durch den Funktionswert einer anderen Funktion bestimmbar.

Die ersten drei Phasen erzeugen einen vollständigen und zyklusfreien Graphen. Das Ergebnis der ersten Phase ist im Regelfall ein unvollständig aufgebauter DAG. Sollte der DAG bereits nach der ersten Phase vollständig sein, dann entfallen die zweite und dritte Phase. In der zweiten Phase wird das Ergebnis der ersten Phase in der Weise genutzt, daß geprüft wird, für welche Komponentenvariablen schon Funktionen (Regeln) erzeugt sind. Auf diese Weise lassen sich Unter- und Überspezifikationen finden. Im Regelfall sollte der DAG nach der zweiter Phase vollständig sein. Falls das doch nicht der Fall ist, wird in der dritten Phase der DAG „gewaltsam" mit konstanten Funktionen vervollständigt. In dem vollständig aufgebauten DAG können schließlich in der vierten Phase von den „Wurzeln" ausgehend alle Funktionen der Richtung der Kanten folgend in einer geeigneten Reihenfolge ausgewertet werden. Nach dieser Auswertung ist das Layout bestimmt.

4.3.2.2 Erste Phase In der ersten Phase berücksichtigt der prototypische `LayLa`-Layout-Algorithmus alle Relationen bis auf die qualitative *Rel-Neben*-Relation, die in der zweiten Phase auf das Ergebnis der ersten Phase aufbauend ausgewertet wird.

Der Algorithmus interpretiert die Relationen der ersten Phase ausschließlich *funktional* und *unabhängig* voneinander. Das bedeutet zum einen, daß aus einer Relation eine oder mehrere Funktionen erzeugt werden, ohne dabei die mögliche Bidirektionalität der Relation zu berücksichtigen. Zum anderen wird bei der Erzeugung der Funktionen aus einer Relation keine Rücksicht auf andere Relationen genommen. Überspezifikationen werden zwar während des Aufbaus der DAG-Datenstruktur erkannt, aber es wird nur in der Weise darauf reagiert, daß die erste erzeugte Funktion ausgewählt und weitere ignoriert werden. Durch diese Form der Behandlung von Überspezifikationen werden Widersprüche weder erkannt noch darauf reagiert.

4.3.2.3 Zweite Phase In der zweiten Phase von `Laylator` werden aus den in der ersten Phase ausgesparten *Rel-Neben*-Relationen mit qualitativer Richtung Funktionen erzeugt. Im Gegensatz zu der ersten Phase werden diese Relationen nicht unabhängig voneinander ausgewertet, sondern in einem bestimmten Kontext zu weiteren *Rel-Neben*-Relationen mit qualitativer Richtung, dem *Orbital-Kontext*, betrachtet. Die Betrachtung der Aspekte der Bidirektionalität, Unterspezifikation und Überspezifikation finden nur jeweils innerhalb eines Orbital-Kontextes statt. Der Orbital-Kontext bildet das zentrale Lokalitätskonzept von `Laylator`.

Die zweite Phase soll den in der ersten Phase berechneten unvollständigen DAG vervollständigen und benutzt dazu eine Hilfsdatenstruktur, die ebenfalls ein gerichteter Graph ist, der aber Zyklen enthalten kann. Diese Hilfsdatenstruktur heißt *Nachbarschafts-Verbund* und ist der maximal zusammenhängende Graph, in dem die Knoten durch Komponenten und die Kanten durch *Rel-Neben*-Relationen mit qualitativer Richtung zwischen den Komponenten gebildet werden. Die Kanten sind mit der jeweiligen Himmelsrichtung markiert und von der Quell-Komponente zur Ziel-Komponente gerichtet.

In einer `LayLa`-Beschreibung können mehrere Nachbarschafts-Verbünde enthalten sein.

Ausgehend von den bisher ermittelten Wurzeln des DAGs, die den konstanten x- oder y-Werten von Knoten im Nachbarschafts-Verbund entsprechen, durchläuft der Algorithmus einen Weg durch den Nachbarschafts-Verbund, auf dem möglichst alle Knoten besucht werden und für möglichst alle noch fehlenden Knoten im DAG Funktionen mit entsprechenden Parameter-Abhängigkeiten (Kanten im DAG) erzeugt werden.

Der Weg entlang der Kanten durch den Nachbarschafts-Verbund wird rekursiv gefunden. Er erfüllt eine wesentliche Eigenschaft: Parameter der Rekursion ist jeweils eine Komponente, deren x- und y-Wert zum Zeitpunkt des Rekursions-Aufrufes durch Auswertung des DAG eindeutig bestimmbar wäre. Diese Komponente heißt *Zentral-Komponente* und bezieht sich auf einen Rekursions-Aufruf. Alle der Zentral-Komponente direkt benachbarten Komponenten bilden den Orbital-Kontext.

Von der Zentral-Komponente ausgehend wird zunächst nacheinander jede Haupt-Himmelsrichtung (o,s,w,n) bearbeitet, indem durch die Kanten des Nachbarschaftsverbundes festgestellt wird, welche Nachbar-Komponenten des Orbital-Kontextes sich den Platz an einer Seite der Zentral-Komponente teilen müssen.

Die Auflösung von verschiedenen Arten der Über- und Unterspezifikation geschieht durch eine umfangreiche Fallunterscheidung. Ist die Ausdehnung der Zentral-Komponente in der aktuell bearbeiteten Richtung unterspezifiziert, müssen die Komponenten des Orbital-Kontextes in den beiden gegenüberliegenden Himmelsrichtungen die Ausdehnung bestimmen. Sind in den beiden Himmelsrichtungen Komponenten mit unterspezifizierter Ausdehnung enthalten, dann sind die Ausdehnungen nicht spezifizierbar und damit ist eine nicht-aufhebbare Unterspezifikation entdeckt.

4.3.2.4 Dritte Phase Zu diesem Zeitpunkt sind im DAG alle notwendigen Kanten enthalten, da jedesmal, wenn in den vorherigen Phasen ein Knoten erzeugt wurde, der eine Funktion mit Parametern repräsentiert, zu jedem Parameter auch die entsprechende Kante erzeugt wurde. In dem Fall, daß diese Kanten von Knoten ausgehen, die noch nicht existieren, wird der entsprechende Knoten erzeugt und als funktionslos markiert. In der dritten Phase werden allen diesen als funktionslos markierten Knoten Funktionen ohne Parameter, die den beliebig gewählten konstanten Pixel-Werte 23 zurückgeben, zugeordnet.

Um den DAG endgültig zu vervollständigen, werden alle noch fehlenden kantenlosen Knoten erzeugt. Diese repräsentieren ebenfalls Funktionen ohne Parameter, die den beliebig gewählten konstanten Pixel-Werte 23 zurückgeben.

Der Anwender wird außerdem durch die Ausgabe einer Warnmeldung vom `Laylator` darauf aufmerksam gemacht, daß nicht sinnvoll aufzulösende Unterspezifikationen vorliegen.

4.3.2.5 Vierte Phase In der vierten Phase wird das Layout eindeutig berechnet. Dazu wird die im vollständigen DAG enthaltene Information über die funktionalen Abhängigkeiten zwischen den Pixel-Variablen-Werten von den „Wurzeln“ ausgehend ausgewertet. Mit den so errechneten Pixel-Werten können die Komponenten und Linien gezeichnet werden.

5 Implementierung

5.1 Übersicht

Den Rahmen für die prototypische Implementierung bildet die plattformunabhängige Programmiersprache Java 1.1.8 mit den dazugehörigen Klassenbibliotheken. Zusätzlich wurden der Parser-Generator Antlr [4], die GNU-Implementierung von regulären Ausdrücken gnu-regexp-1.0.8 [6] und ein Auswerter für arithmetische Ausdrücke JEL [3] verwendet. Alle drei Pakete sind Open-Source-Implementierungen, die frei im Internet zugänglich sind.

Das System läßt sich in 3 Haupt-Komponenten gliedern, von denen die erste, die Steuerung und Visualisierung, am stärksten von den Java-Klassenbibliotheken Gebrauch macht und die dritte, die Implementierung des Layout-Algorithmus `Laylator`, am wenigsten.

5.2 Die Haupt-Komponenten der Implementierung

5.2.1 Steuerung und Visualisierung

Die Java-Klassenbibliothek enthält eine ereignisgesteuerte grafische Benutzer-Schnittstelle mit dem Namen AWT (Abstract Window Toolkit). Das AWT erlaubt es, die interaktiven grafischen Elemente, die z.B. in einem Fenster auf dem Bildschirm sichtbar sind, durch verschiedene Layout-Verfahren anzuordenen. Es ermöglicht auch, eigene Layout-Verfahren durch die Implementierung des Java-Interfaces `LayoutMgr2` in das AWT-System einzubetten. Die Klasse `NavigationLayout` implementiert dieses Interface und steuert auch den Aufruf des Parsers, um zum Zeitpunkt der Instanziierung eines Objektes dieser Klasse die entsprechende `LayLa`-Layout-Beschreibung zu laden. Der Prototyp bindet ein `NavigationLayout`-Objekt an ein AWT-Fenster, das bei Bedarf automatisch ein neues Layout anfordert. Dies passiert, wenn z.B. der Benutzer die Größe des Fensters geändert hat.

5.2.2 Der `LayLa`-Parser

Das Ergebnis des vom Antlr-Generator generierten Parsers ist ein Objekt der Klasse `SpatialRelationContainer`. In diesem Container sind alle vom Parser erkannten Relationen jeweils als Objekt enthalten. Jede räumliche Relation hat eine eigene Klasse, die von der Basis-Klasse `SpatialRelation` abgeleitet ist.

Eine Methode `layout` der Klasse `SpatialRelationContainer` steuert die Phasen des `Laylator` und wird von der Klasse `NavigationLayout` bei Bedarf aufgerufen.

5.2.3 Implementierung des `Laylator`

Den Aufbau und die Auswertung des `Laylator`-DAG steuert ein Objekt der Klasse `ZEOfactory`. Die Knoten und Kanten sind durch Objekte der Klasse `ZEO` implementiert.

Jedes Objekt einer quantitativen Relation ergänzt durch Aufruf der Methode `layout` mit Hilfe der `ZEOfactory` den DAG um die entsprechenden Funktionen und Abhängigkeiten.

Der Nachbarschaftsverbund ist durch Objekte der Klasse `Nachbar` realisiert. Er wird in der Methode `layout` der Klasse `SpatialRelationContainer` aus allen qualitativen Neben-Relationen zusammengesetzt. Die rekursive Methode `orbitalClustern` der Objekte der Klasse `Nachbar` durchläuft den Nachbarschaftsverbund und ergänzt mit Hilfe des `ZEOfactory`-Objekts nach und nach den DAG.

5.3 Übersetzung der Quellen des Prototyps

Die Quellen des `LayLa`-Prototypen sollten in ein eigenes Verzeichnis entpackt sein. Zum Übersetzen und Ausführen muß die Umgebungs-Variable CLASSPATH gesetzt werden. Dies geschieht unter der Bash-Shell von Unix z.B. durch:

```
export CLASSPATH="antlr.jar:jel.jar:gnu-regexp-1.0.8.jar"
```

Vor der Übersetzung der Java-Quellen muß die `LayLa`-Grammatik, die in der Datei `layla.g` beschrieben ist, mit Hilfe des Parser-Generators Antlr übersetzt werden:

```
java antlr.Tool layla.g
```

Das Ergebnis der Antlr-Übersetzung sind die Dateien `LayLaLexer.java`, `LayLaParser.java` und `LayLaParserTokenTypes.java`. Jetzt können alle Java-Quellen übersetzt werden:

```
rm *.class; javac *.java
```

5.4 Starten des Prototyps

Um den Prototyp auszuführen muß die Umgebungs-Variable CLASSPATH entsprechend gesetzt sein (siehe letzten Abschnitt). Das Haupt-Programm ist in der Klasse `LayLaMain` enthalten. Es benötigt zwei Parameter. Der erste ist eine dezimale ganze Zahl ≥ 0, sie gibt an, in welchem Umfang Debugging-Information ausgegeben wird, wobei 0 dem geringsten Umfang entspricht. Der zweite Parameter benennt die Datei, in der die zu layoutende `LayLa`-Spezifikation enthalten ist. Z.B. wird die Datei `unix.LayLa` mit dem folgenden Aufruf ohne Debugging-Information layoutet:

```
java LayLaMain 0 unix.LayLa
```

Literatur

[1] Maurice J. Bach. *UNIX - Wie funktioniert das Betriebssystem?* Hanser / Prentice-Hall Internat., München, Wien, London, 1991.

[2] M. Schwartz F. Halasz. The Dexter Hypertext Reference Model. In *Communications of the ACM, 37(2)*, pages 30–39, 1994.

[3] Konstantin Metlov (metlov@fzu.cz). *jel-0.9.4*, September 2000. Siehe http://galaxy.fzu.cz/JEL/.

[4] Terence Parr. *ANTLR Reference Manual*, Juli 2000.

[5] Wolfgang Schnotz. Wissenserwerb mit Diagrammen und Texten. In Ludwig J. Issing and Paul Klimsa, editors, *Information und Lernen mit Multimedia (2. überarbeitete Auflage)*, pages 83–105, Weinheim, 1997. BELTZ PsychologieVerlagsUnion.

[6] Wes Biggs (wes@cacas.org). *gnu.regexp-1.0.8*, März 1999. Siehe http://www.cacas.org/java/gnu/regexp/.