

---

---

# Aesthetics and Automatic Layout of UML Class Diagrams

---

---

Dissertation zur Erlangung des  
naturwissenschaftlichen Doktorgrades  
der Bayerischen Julius-Maximilians-Universität Würzburg



vorgelegt von

**Holger Eichelberger**

aus  
Dinkelsbühl

Würzburg, 2005

Eingereicht am: 22.03.2005

bei der Fakultät für Mathematik und Informatik

1. Gutachter: Prof. Dr. Jürgen Wolff von Gudenberg, Universität Würzburg
2. Gutachter: Prof. Dr. Franz Josef Brandenburg, Universität Passau

Tag der mündlichen Prüfung: 15.06.2005

*To Mum, Mone, Doris and  
Roswita.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Diagram Basics</b>	<b>7</b>
2.1	The Unified Modeling Language . . . . .	7
2.1.1	Diagrams of UML . . . . .	8
2.1.2	UML Class Diagrams . . . . .	10
2.1.3	Criticism on UML . . . . .	20
2.1.4	Alternative Visualizations . . . . .	24
2.2	Automatic Diagram Layout . . . . .	26
2.2.1	Why to Draw a Class Diagram Automatically? . . . . .	26
2.2.2	Graph Drawing – an Overview . . . . .	28
2.2.3	State-of-the-Art in Drawing UML Class Diagrams . . . . .	36
<b>3</b>	<b>Functional Specification</b>	<b>43</b>
3.1	Requirements for UML Class Diagram Layout . . . . .	43
3.2	Input and Output . . . . .	47
3.3	Aesthetics . . . . .	51
3.3.1	Terminology . . . . .	52
3.3.2	Traditional Graph Drawing Aesthetics . . . . .	54
3.3.3	Human Computer Interaction and Cognitive Psychology . . . . .	61
3.3.4	Software Engineering . . . . .	65
3.3.5	Software Visualization (SV) . . . . .	69
3.3.6	Semantic Aesthetic Principles for UML Class Diagrams . . . . .	71
3.3.7	Aesthetic Conclusions . . . . .	85
<b>4</b>	<b>The Layout Algorithm</b>	<b>93</b>
4.1	<i>SugiBib</i> – Just Another Hierarchical Algorithm? . . . . .	93
4.2	Structural Conventions for Graphs . . . . .	105
4.3	Basic Definitions . . . . .	112
4.3.1	Notational Conventions . . . . .	112
4.3.2	Primitives . . . . .	113
4.3.3	Graphs, Nodes, Edges and Operations . . . . .	114
4.3.4	The Node Naming Function . . . . .	118

4.4	Preprocessing Steps . . . . .	123
4.4.1	Adjust Semantical Issues . . . . .	123
4.4.2	Semantic Ordering . . . . .	124
4.4.3	Deduce Hierarchy . . . . .	126
4.4.4	Insert Nesting Relations as Edges . . . . .	127
4.4.5	Compress Hyperedge Connection Nodes . . . . .	127
4.4.6	Remove Direct Cycles . . . . .	128
4.4.7	Compress Association Classes . . . . .	129
4.4.8	Compress Comments . . . . .	130
4.4.9	Remove Disconnected Nodes . . . . .	132
4.4.10	Virtual Root and Leaf . . . . .	132
4.4.11	Breaking Cycles . . . . .	133
4.4.12	Conclusions . . . . .	134
4.5	Rank Assignment . . . . .	136
4.5.1	Previous Work . . . . .	136
4.5.2	Basic Definitions . . . . .	138
4.5.3	Validity Rules . . . . .	140
4.5.4	The Core Algorithm . . . . .	143
4.5.5	UML and Cluster Specific Adjustments . . . . .	145
4.5.6	Conclusions . . . . .	154
4.6	Edge Crossings . . . . .	156
4.6.1	Previous Work . . . . .	156
4.6.2	Basic Definitions . . . . .	160
4.6.3	Crossing Theory . . . . .	162
4.6.4	Cluster Handling . . . . .	190
4.6.5	Extended Crossing Algorithms . . . . .	197
4.6.6	Conclusions . . . . .	203
4.7	Intermediary Processing . . . . .	204
4.7.1	Expand Composite Nodes for Association Classes or Hyperedges . . . . .	204
4.7.2	Remove Nesting Edges . . . . .	205
4.7.3	Conclusions . . . . .	205
4.8	Coordinates Assignment . . . . .	206
4.8.1	Previous Work . . . . .	207
4.8.2	Basics . . . . .	210
4.8.3	The Coordinates Assignment Algorithm . . . . .	215
4.8.4	Coordinates Preprocessing . . . . .	217
4.8.5	Iterative Coordinates Assignment . . . . .	223
4.8.6	Postprocessing . . . . .	230
4.8.7	Conclusions . . . . .	234
4.9	Postprocessing . . . . .	236
4.9.1	Association Classes . . . . .	236
4.9.2	Hyperedges and Constraints . . . . .	237
4.9.3	Annotations . . . . .	238

4.9.4	Disconnected Nodes . . . . .	239
4.9.5	Alignment to a Specified Grid . . . . .	239
4.9.6	Create the Result Graph . . . . .	241
4.9.7	Conclusions . . . . .	241
4.10	Summary . . . . .	242
<b>5</b>	<b>Measurements</b>	<b>245</b>
5.1	Measuring Aesthetics . . . . .	245
5.2	Layout Comparison . . . . .	262
5.3	Runtime Measurements . . . . .	269
<b>6</b>	<b>Implementation</b>	<b>281</b>
6.1	Architecture of <i>SugiBib</i> . . . . .	281
6.2	Layout Algorithm . . . . .	288
6.2.1	Preprocessing . . . . .	288
6.2.2	Rank Assignment . . . . .	291
6.2.3	Edge Crossings . . . . .	292
6.2.4	Intermediary Processing . . . . .	294
6.2.5	Coordinates Assignment . . . . .	295
6.2.6	Postprocessing . . . . .	296
6.3	Applications for UML class diagrams . . . . .	298
6.3.1	Information Classes for UML Class Diagrams . . . . .	298
6.3.2	Application Library . . . . .	303
6.3.3	Layout Metrics . . . . .	305
6.4	Testing & Debugging <i>SugiBib</i> . . . . .	307
6.5	Runtime Optimizations . . . . .	312
6.6	Extensions & Modifications to <i>SugiBib</i> . . . . .	321
<b>7</b>	<b>Drawing Class Diagrams – an Ongoing History</b>	<b>327</b>
7.1	Future Work . . . . .	327
7.2	Conclusions . . . . .	330
<b>A1</b>	<b>Example Drawings by <i>SugiBib</i></b>	<b>333</b>
<b>A2</b>	<b>Lists and Hashtables</b>	<b>344</b>
<b>A3</b>	<b>Improved Algorithms</b>	<b>347</b>
A3.1	Cluster Validity . . . . .	347
A3.2	Edge Crossing Reduction . . . . .	349
<b>A4</b>	<b>Bibliography</b>	<b>350</b>
<b>A5</b>	<b>List of Figures</b>	<b>372</b>

<b>A6 List of Algorithms</b>	<b>376</b>
<b>A7 List of Tables</b>	<b>377</b>
<b>A8 Index</b>	<b>378</b>

# 1 Introduction

---

One fine day, in a company, a software engineer is told to participate in a process for specifying and implementing an object-oriented software system. The engineer knows that these days, specifying software is one of the most convenient tasks one could be assigned to do. Different kinds of tools are available to support that work to create a product of high quality. One of these tools is a unified, visual specification language introduced by an international standard. That visual language simplifies the communication with other software engineers presumably working in different companies located in several countries by providing highly intuitive diagrams which lead to a precise description of the software system. That visual language is furthermore supported by various software tools, all interacting with each other without any problem. Using these tools, all engineers involved could change all specification documents in parallel. Considering static and dynamic information from the model nearly perfect code is generated. When requirements change, modifications in the diagrams are registered by the tool and the layout of the diagrams is perfectly adjusted with minimum number of changes to maintain the mental map of the engineer. Even if the engineer had been assigned to a maintenance project for documenting and maintaining ancient code, which was produced neither with standardized specification nor documentation, this would have been also a convenient task. The same tools mentioned above are able to analyze the old sources, to automatically recognize design patterns according to a pattern catalog and to produce perfect diagrams and documentation. Software engineers have created for themselves the perfect world to work in.

Wouldn't it be nice if that was true? The present situation however, is far from that perfect situation. In the future, this software engineering fairy tale may become reality, the current situation, however, looks more like a horror story: The international standard language mentioned above, the Unified Modeling Language (UML) exists, but it is far away from reaching that state of precision. Most of the tools supporting UML are not compliant to the current version of the standard, and even ancient versions are not completely supported. Most of these Computer Aided Software Engineering (CASE) tools implement a distributed repository, so that every developer of a company can read and modify all specification documents according to certain access rules. A history is kept on all changes, source code for different programming languages as well as documentation in different formats can be generated, maintained and analyzed, etc. But tasks like generating code respecting the responsibilities of the elements, analyzing code and retrieving and



displaying design specific issues like design patterns are far away from perfection. Complex but required features like a distributed repository or various generators make such tools enormously expensive. At a first glance, automatic layout of diagrams does not seem to be a key feature for CASE tools. But taking unpredictable and unanticipated changes into account, like new or modified requirements or synchronization of design documents against code changed while implementation, manual modifications to the diagrams to ensure consistency and readability is a tedious, time-consuming task. Most tools provide such automatic layout facilities, and the result of applying these features usually lead to a crowded layout, and sometimes it seems that the elements were simply randomly shuffled. Furthermore, a tool usually presents different layout methods for a concrete diagram and each method provides a large set of options. Mostly, neither the names of the methods, nor the descriptions of the options, nor the concepts behind these topics are known by the engineer, because these technical terms relate to another completely different discipline of computer science, namely graph drawing. Do these problems with automatic layout occur due to bugs in the implementation of the CASE tools or is it extremely difficult to solve the problem of calculating the layout of software engineering diagrams?

Different disciplines are involved in the problem of automatic layout of software engineering diagrams, and it seems that basic common knowledge spread over these disciplines is present. Because of computational as well as a communicational problems no appropriate layout algorithms exist for most complex types of diagrams. From the viewpoint of a software engineer, UML provides a large degree of freedom to specify software on different levels of abstraction providing different modeling and presentational options. Most of the tool vendors select parts of the standard to be implemented in their tools, despite the fact that such selections are not allowed from the viewpoint of UML. Furthermore, after reading some basic literature about drawing general graphs, tool vendors try to implement automatic layout features which do not take the structural and semantical information of software engineering diagrams into account. Researchers working in the field of graph drawing then tend to tailor their layout algorithms for UML class diagrams based on their experiences in drawing general graphs, but oftentimes without taking the underlying semantics and software engineering philosophy into account. These algorithms then support abstract aesthetic features designed for general graphs but also without considering the application domain specific structural and semantical requirements. On the one hand, from the viewpoint of human perception and cognitive psychology, neither the software engineering standards nor the layout algorithms from graph drawing are sufficient to provide intuitive, readable and error-minimizing representations. But on the other hand, the designers of software specification languages tend to ignore the results from Human Computer Interaction (HCI) and cognitive psychology. Oftentimes, the researches from general graph drawing try to capture these fundamental issues on human perception by some non-application domain specific aspects. And finally, the results from user studies and experiences in Software Visualization are often not respected in other disciplines that are involved in the area of automatically drawing software engineering diagrams.

This work will present basic results as well as practical solutions for the problem of automatically calculating the layout of UML class diagrams. We will unify common results from the four disciplines graph drawing, software engineering, software visualization and HCI to derive aesthetic criteria as well as a layout algorithm supporting these criteria for UML class diagrams.

---

Furthermore, an implementation of the algorithm and an automatic testing approach for layout results will be given.

To introduce the kind of diagrams, the diagram elements and the relations to be drawn, at the beginning of the second chapter we will first describe UML in general and more specifically UML class diagrams. A further section will then highlight the necessity of drawing software engineering diagrams automatically and introduce graph drawing, the discipline responsible for theoretical and practical results on drawing general and application domain specific graphs. Thereby, basic approaches will be listed and work done for UML class diagrams so far will be described. In the third chapter we will collect a functional specification to be fulfilled by a concrete implementation. Therefore, we will first enumerate basic requirements for drawing UML class diagrams and select appropriate data formats for input and output. Afterwards, the problem of readability of UML class diagrams will be discussed intensively. From the four basic disciplines graph drawing, software engineering, software visualization and HCI arguments will be collected and compiled to form one set of structural and semantic rules.

Then, in the next chapter, based on a general graph drawing method, a layout approach for realizing the functional specification will be introduced. Thereby we specify the underlying graph model and present the basic architecture of *SugiBib*, a framework which realizes the layout algorithm.

In the fifth chapter, the formal notation for the graph model will be given and the individual steps of the algorithm will be discussed in detail.

The next chapter contains a practical comparison of existing layout implementations for UML class diagrams, deals with measuring aesthetics according to the aesthetic rules of the third chapter as well as problems in manual and automatic testing, specific applications of the framework and future optimizations of the implementation. Finally, hints for anticipated modifications of the implementation and approaches to other types of UML diagrams will be considered as well. In the last chapter issues deferred to future work and an overall conclusion will be given.

Some technical and notational details should be mentioned here. As it is usual in computer science, the author of this thesis will refer to himself by using plural form. Persons will always be referred as females. Items in the text meant to be emphasized will be displayed in italics. As regards to lists bold font face will be used. Names in algorithms, source code or in diagrams mentioned in the text will be given in teletype style.

Statements taken from literature will be indicated by reference in the bibliography, direct citations might turn up in text or in individual paragraphs.

UML diagrams, used to explain the relevant parts of UML or details of the implementation, will be drawn as such. In diagrams, which explain general issues of a layout algorithm, nodes will be drawn as circles, usually containing an identification, connected by solid lines. In diagrams, used to explain UML specific parts of a layout algorithm, visible edges will be drawn UML-like, invisible edges are drawn in other, appropriate styles, nodes will be depicted as rectangular boxes, visible nodes may be marked by simplified names, and invisible elements will be marked by diagonal lines in the node box.

Diagrams, relevant for explaining issues of UML class diagrams or the architecture of the implementation have been drawn manually. We made this decision to partly emphasize certain layout

requirements, some of them are currently not realized in the implementation, and partly to provide a comparison between automatic drawing and manual capabilities. In several sections as well as in the appendix some diagrams drawn by our algorithm will be presented.

## ACKNOWLEDGMENTS

Many thanks to my supervisors Jürgen Wolff von Gudenberg and Franz Josef Brandenburg for the environment in which this work evolved, the Gentleware Company for providing a license of Poseidon Professional and two layout plug-ins for free, Peter Eades and Helen Purchase for sending some of their articles, again Peter for the logo of *SugiBib*, Helen Purchase, Margaret-Anne Storey, Jonathan I. Maletic, Andreas Winter and Stephan Diehl for the fruitful influence, Yann-Gaël Guéhéneuc (Université de Montréal) for some larger test diagrams, proof-reading, procreative discussion and several encouraging words, Jorge Gomez Sanz (Universidad Complutense de Madrid) for using *SugiBib* and contributing some test diagrams, Fritz Kleemann for technical hints in particular on bash and L<sup>A</sup>T<sub>E</sub>X, Bernhard Gröhling, Per Pascal Grube and Dian Dochev and for plug-ins on metrics and XMI[DI], Iris Wagner for (unfortunately only partly) proof-reading my crowded English, Wolfgang Pfeiler (Druckerei Pfeiler, Stödtlen) for so much patience and enthusiastic while embedding missing fonts and preparing other details for his Indigo printing machine, Simone Albrecht, Doris Knopp and Roswita Albrecht for their silent and impressive words as well as for providing a light at the end of a (sometimes) dark tunnel and my mother, Marga Eichelberger, for always being there and guiding me all the time.



# 2 Diagram Basics

---

In this chapter we will outline an overview on the basic aspects related to UML class diagrams. In Section 2.1, we will start with an introduction to UML in general and to UML class diagrams in particular. While giving a brief overview on criticism on UML as well as on alternative visualizations of class diagrams, we will prepare arguments, which will help restricting the potentially broad range of functionality an implementation.

In Section 2.2, we will discuss the necessity of automatic layout in software engineering as well as the basic techniques for calculating the layout of graphs, abstract structures consisting of nodes connected by edges. Based on this knowledge, an evaluation of the current state-of-the-art in automatically drawing UML class diagrams will be given.

## 2.1 The Unified Modeling Language

Every tool carries with it the spirit by which it has been created.

Werner Karl Heisenberg (1901 – 1976)

The Unified Modeling Language (UML) has become the standard language for specifying and visualizing aspects of object-oriented software. Therefore, diagrams used in software engineering ought to be compliant to that visual language to reduce the costs of communication.

In this section, we will first give an overview of the different types of diagrams introduced by UML. Also, elements and relations of UML class diagrams, which are the main topic of our work, will be described. Having discussed the visual complexity and variability of these diagrams, we will briefly summarize some of the criticism on UML, alternative display approaches and relations of these issues to our work. At a first glance, parts of this section may appear to be a discussion on UML related details only, but going into some UML and software engineering details, natural boundaries of our work related to (aesthetic) layout issues, UML and CASE tool technology will become clear.

### 2.1.1 Diagrams of UML

Concern for man himself and his fate must always form the chief interest of all technical endeavor. Never forget this in the midst of your diagrams and equations.

Albert Einstein (1879 – 1955)

To simplify the user's understanding of the complex relationships of classes or objects in object-oriented software engineering, even the first object-oriented methodologies provided a graphical notation of the static and dynamic aspects of the software. Important features of UML include the unification of these diagrammatic notations, the specification of a standard instead of having several incompatible notations and an approach to support the entire software development process. UML [OMG 2003c] therefore provides:

- **Use case diagrams**, which collect and visualize the situations in which the software to be developed may be used. These diagrams are intended as a foundation for discussions with customers and stakeholders especially while capturing requirements. The diagrams show the global relations between individual use cases, but usually textual scenarios have to be provided for each use case to specify and to clarify the use cases and to describe different alternatives for realization.
- **Activity diagrams**, object-oriented extensions to flow charts, are intended to visualize a use case and its alternatives. A use case is separated into several more finely grained activities, which might then be partitioned according to their responsibilities. Finally, object-flows showing the migration of data between activities can be depicted.
- **Class diagrams**, showing the static structural relationships among classes, interfaces, packages or class instances. Depending on the development process, class diagrams might be derived iteratively from use case and activity diagrams but also from sequence or collaboration diagrams.
- **Sequence diagrams**, which visualize message passing between class instances according to the flow of time.
- **Collaboration diagrams**, which are semantically equivalent to sequence diagrams but show spatially arranged groups of instance collaborations.
- **Statechart diagrams**, which model the interactions within a single class as a state machine.
- **Package diagrams**, showing relations among modules, groups of classes semantically belonging together. Some software engineers and tool vendors strictly distinguish between class diagrams (containing classes and their relations only) and package diagrams (similarly containing packages only). UML itself makes no such separation because some model elements combine module notation and class notations [OMG 2003c, p. 3-34].

- **Component diagrams**, describing (nested) components, their interfaces and the dependencies between different components.
  
- **Deployment diagrams**, which show the distribution of components on different physical nodes.

The unified specification of the elements and relations, which might be used in a diagram, and the semantics, which arise from those elements and relations, is the foundation of a standardized communication among software engineers. On machine level, the communication can be done in terms of XML Metadata Interchange (XMI) which is specified along with UML.

On diagram level, UML specifies rules regarding the contents of the diagrams and some minor presentation options. These options can be seen as a kind of basic style guide defining stylistic issues like font faces and alignments. Changes to these default options may be given in individual profiles, a common extension mechanism. Unfortunately, further layout rules which can also help standardizing and simplifying the communication, are not discussed in UML. Therefore, apart from software engineering and software process issues, a concrete layout is one degree of freedom in creating UML diagrams.

Even if the specification diagrams in UML itself suggest a certain style of drawing and even if these diagrams are kept in mind by all software engineers who ever read the UML specification, every individual engineer might draw her diagrams according to her individual aesthetic principles. A company might, of course, introduce a style guide, as it is used for source code or administrative documents in the sense of quality engineering, e.g., according to ISO 9001 or a corporate identity of the company. But, in fact, the more different style guides exist (diagrams might be drawn that have no stylistic rules in mind, either) the more difficult the communication among different engineers will become, especially when they work in different companies or in different countries. In this work, we therefore will also extensively discuss the problem of aesthetic issues and propose our set of layout rules.

Furthermore, UML version 2.0 [OMG 2003d] combines component diagrams and class diagrams by introducing compound classifiers, provides revised activity diagrams, and introduces new diagrams like timing diagrams.

There are different reasons why we will use the older UML version 1.5 instead of the new version 2.0 in this thesis: Most tools dealing with UML are not yet able to support the complete specification of version 1.5 (in fact, some are far away from prior versions). Even if UML 2.0 is partly a simplification and partly an extension which, with regard to diagram layout, can be implemented on top of UML 1.5, a preview of version 2.0 was published in June 2003 is therefore too new to be considered in this thesis. Therefore, in the following, the term “UML specification” will refer to the specification documents of UML version 1.5.



## 2.1.2 UML Class Diagrams

Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things.

[OMG 2003c]

In this section, we introduce the UML notation for class diagrams and their basic features by example. Even if we aim to be complete, an exhaustive listing of all features and restrictions is far beyond the scope of this thesis. Readers interested in further details should turn to the UML specification [OMG 2003c] where a full description will be given.

The class diagram example to be constructed incrementally in this section will express that an usual graph from graph theory can be processed by a graph algorithm. We will assume that a graph consists of nodes which can be interrelated by edges. Furthermore, the “Sugiyama” algorithm will be an example of a specialized graph algorithm. Even if this example can also be seen as the core part of our graph model also showing the layout algorithm to be developed in this thesis, in this section it should be understood as a structural example only.

### Classes

Class diagrams are used to model the structure of classes and the relations connecting classes (and sometimes instances of classes) from a static viewpoint. Figure 2.1 gives an example of such a class in UML notation. Classes are displayed as rectangles, possibly partitioned into several subrectangles, the compartments. The top compartment, which contains the name of the class, must always be present. According to the standard UML style guide, the name of the class should be centered and printed in bold font face. If a class is abstract<sup>1</sup>, the name should be written in italics. Without going too far into details, the name compartment of a class might also contain

- **Constraints**, which represent conditions to be fulfilled by every implementation that realizes the diagram.
- **Tag-values**, lists of name–value pairs to directly change properties of the element, e.g., each class has internally an attribute `isAbstract` which might be switched on by `{abstract=true}` or simply `{abstract}`.
- **Stereotypes**, names enclosed in guillemets used as classification for the stereotyped model element and as well as a lightweight extension mechanism to the UML. A stereotype definition may also introduce an icon, constraints or tag-values which are automatically applied to the stereotyped model element. More details on different types of stereotypes were discussed in [Berner et al. 1999]. Stereotypes are usually shown centered above the name of the class.

As illustrated in Figure 2.1, a class has a name compartment, usually one for attributes and one for operation<sup>2</sup> signatures. Attribute and operation signatures must conform to the syntactical

---

<sup>1</sup>An abstract class cannot be instantiated.

<sup>2</sup>UML distinguishes between signatures of a service (operation) and its implementation (method).

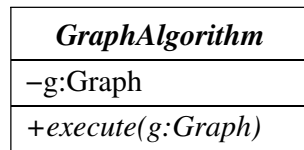


Figure 2.1: An abstract class named `GraphAlgorithm` having a private attribute `g` of type `Graph` and a public, abstract operation signature receiving a `Graph` as parameter.

rules defined in the UML, e.g., a signature starts with the visibility specifier. Abstract operations can be displayed in italics, static attributes or operations should be underlined. An exception are so called ellipses (. . .), which are used to indicate that not all attributes or operations are displayed or specified, respectively. The attribute as well as the operation compartment may be omitted, in particular, because it is obvious what signature<sup>3</sup> belongs to which kind of compartment. Additionally, further compartments for requirements, signals, etc., can be introduced. Every compartment can have a title, the entries might be furthermore partitioned according to their stereotypes.

A basic rule in UML is that elements or features which are not displayed are interpreted as unspecified, i.e., if attributes are missing in a class, especially in the early stages of a project, this means that they are currently not specified and might be added in future.

Interfaces<sup>4</sup> in UML are shown as usual classes flagged with the stereotype `«interface»`.

## Basic Relations

In object-oriented programming, two basic types of relations are common: is-a and has-one/has-many relations. Is-a relations express inheritance. In Figure 2.2, the class `SugiyamaAlgorithm` extends `GraphAlgorithm`, i.e., `SugiyamaAlgorithm` inherits all visible features (e.g., attributes and operation signatures) from `GraphAlgorithm` and may override or overload them. In Figure 2.2 the operation `execute` is overridden in `SugiyamaAlgorithm` to depict that the abstract operation of the superclass is implemented.

If a class implements an interface, a similar relation as in Figure 2.2 is drawn except that the line itself has dashed style. Inheritance edges can be labeled by a so called discriminator, which then shows to which sub-category of inheritance relations that specific one belongs to. For example, a graph can be derived to rooted or free trees but also to fully connected graphs and lattices. In this case `trees` and `nets` could be discriminators which label the individual inheritance relations. Additionally, constraints might be attached to several inheritance edges by visually connecting them with dashed connection lines. As mentioned for attributes or operations, an ellipse might be used instead of a class to emphasize that parts of the inheritance hierarchy have been left out. So far, the class `Graph`, mentioned as type in both classes in Figure 2.3, is missing in our example. Of course, it might be specified in any other diagram which relates to Figure 2.1 or Figure 2.2.

<sup>3</sup>Operations always have at least an empty parameter list.

<sup>4</sup>Collections of operation signatures to be implemented by the classes which are marked to be compliant to the interface.

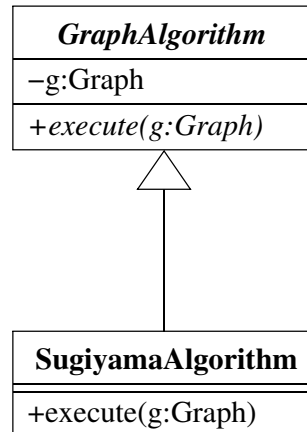


Figure 2.2: An is-a inheritance relation. `SugiyamaAlgorithm` inherits all visible features from its superclass `GraphAlgorithm` and specifies that `execute` implements the abstract signature of the superclass.

In Figure 2.3, we introduce that class using an association to show that a graph algorithm (temporarily) owns and works on a graph instance. Therefore, the attribute `g`, which played that role so far, is removed from `GraphAlgorithm`. Like a class, an association may have a name, stereo-

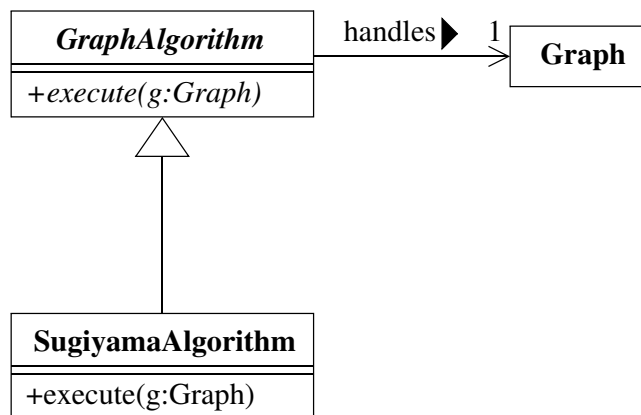


Figure 2.3: A has-one/many relation. At the right of the association name a reading direction indicator is shown.

types, constraints or tag-values. Both association ends of a binary association can furthermore be specified by a role name including a visibility, a multiplicity (the range(s) of numbers of occurrences), a qualifier (a kind of unique key) and a navigational indicator. In our example, an algorithm handles exactly one graph but a graph might be handled by different algorithms, the algorithm may navigate to the graph and access visible features but the graph does not know the algorithm instance which handles the graph. A directional identifier might be attached to the

name of the association to make the reading direction more obvious. An association is identified by its name or, if the name is missing, by its roles. If multiple associations occur between two classes, each of these associations must be named uniquely. As a special case, an association might connect a class with itself as a *reflective association*. In this case, the role names must be present.

Special types of associations are aggregations and compositions, dependent on the behavior of the collection and the contained elements when creating or disposing the instances. In Figure 2.4 nodes and edges, as usual in graph theoretic models, are attached by a composition to a graph, or, a graph is a composition consisting of nodes and edges. In this case, next to the side of the collection, a filled diamond (for aggregations a hollow diamond is used) is attached. The valid number of nodes and edges is specified but unbounded (a star symbol is used in this case as multiplicity).

It might happen that the algorithm somehow changes its behavior depending on additional infor-

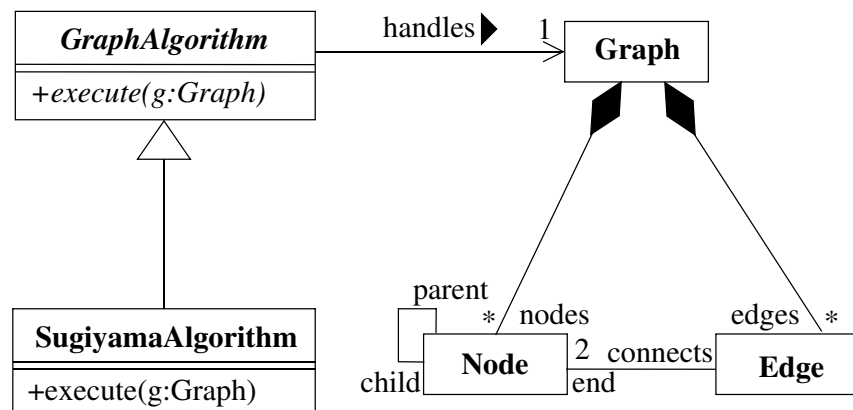


Figure 2.4: A graph, as usual in graph theory, consists of an arbitrary number of nodes and edges. Furthermore, a reflective association is used to express that the nodes may contain further nodes in a parent-child-relationship.

mation like the type of input the graph was created from. This information might be specified in the **Graph** class but not in the algorithm, because the algorithm is not that closely related to the graph: Semantically, the information is used by the algorithm but belongs to the graph (or the relation itself). Thereby, an algorithm instance may work reusable in sequence on several graph instances and it is not required that the lifetime of the algorithm is the same or longer than the lifetime of the graph. Hence, if that additional information relates more closely to the association between algorithm and graph, it can be attached to the association itself using a so called *association class* as shown in Figure 2.5.

For an association class, UML requires that the name of the class and the name of the connected association are equal. Mapping mechanisms, e.g., in OCL allow that the name of the class starts with a capital letter while the name of the association begins with a lower case letter. Both names may appear, at least one is required to be present.

Beside the dashed association class connecting line, further relations may be connected to an as-

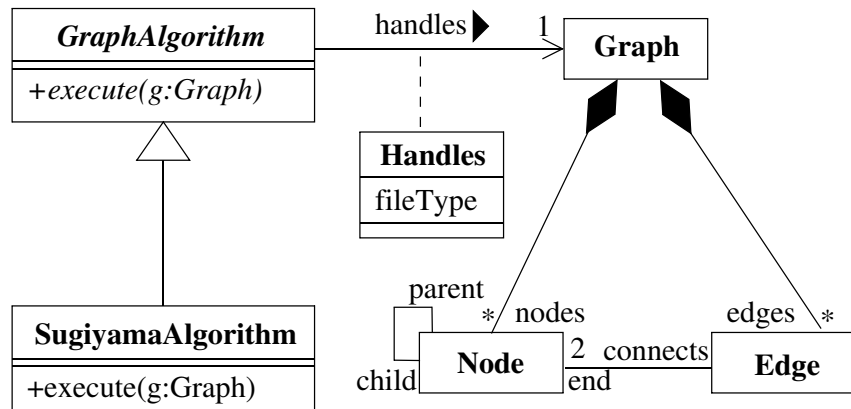


Figure 2.5: Specifying additional information at an association-like relation by attaching an association class.

sociation class. We like to refer to such association classes as *association classifiers*. In the early stages of the development process, when no attributes or operations are specified for classes, empty association classes might occur. If an empty association class inherits from another association class and no further relations are present as in Figure 2.6 (a), a short cut notation can be used. As shown in Figure 2.6 (b) an inheritance relation can then be drawn between both associations. As a special case an association class may also occur on a reflective association.

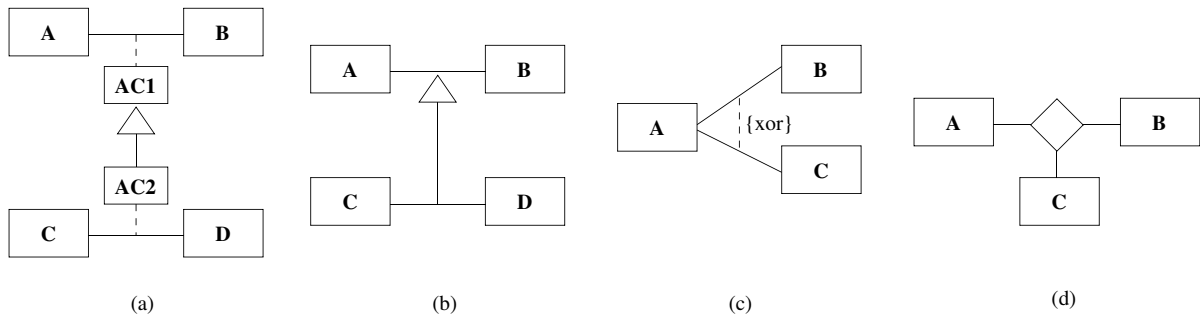


Figure 2.6: A structural example only: (a) inherited association classes, (b) equivalent shortcut to (a), (c) xor-constrained associations and (d) n-ary associations.

Associations can be constrained by an xor condition as depicted in Figure 2.6 (c) to specify that either the association between A and B or A and C may be realized.

In [Purchase et al. 2003], it is partly suggested to denote the name of the association in an association class without attributes or operations. We agree to that notation as long as it can be expected that the association class will receive further structural or behavioral features. Otherwise, especially considering certain consistency mechanisms of UML tools, the association class should be transformed to the equivalent association name. This also fulfills the rule in the UML specification that in this case the class symbol may be seen as subordinate detail which can be

suppressed [OMG 2003c, p. 3-78].

From one of the historic predecessors, Entity-Relationship (ER) diagrams, the higher or n-ary association has been taken over into object-oriented modeling and UML. In Figure 2.7 (d) a ternary association is shown.

## Model Management

Even if our example does not contain a large number of classes it makes sense to group elements together which have a close semantical relation. The partitioning of elements of a software system depends on the rules of modularization in software engineering. UML provides *packages*, which define an unique namespace for the contained elements, *models*, which are package capturing the physical view of a system, and *subsystems*, which provide detailed information on the access signatures to the contained elements. Figure 2.7 shows the usage of a package containing all directly graph-related elements. Therefore the fully qualified name of the class Graph accord-

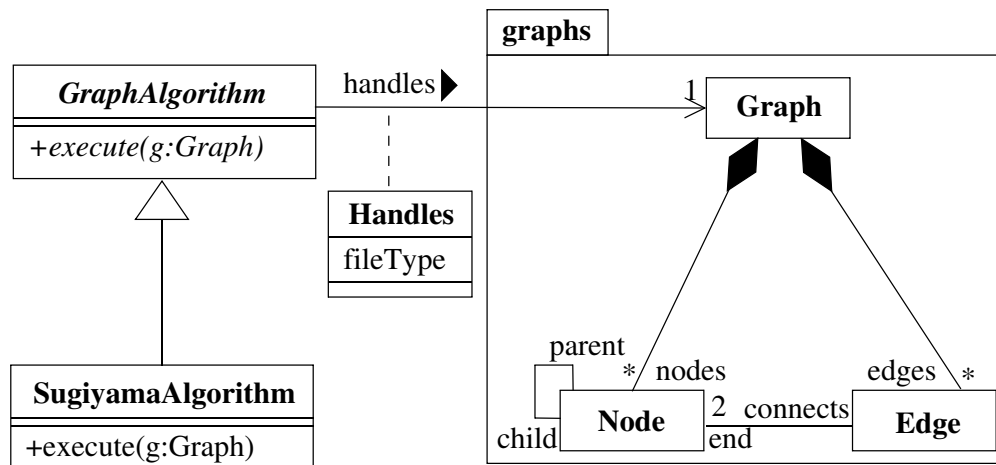


Figure 2.7: Modularization by packages.

ing to UML is now `graphs : : Graph` (similar to the scope operator in C++).

From the geometrical point of view, sometimes it would be desirable that the rectangular area of a model management element could be drawn as a polygonal line according to the convex hull of the contained elements. This option is not provided by UML because packages are explicitly characterized as a large rectangle with a small rectangle at the top, the “tab” [OMG 2003c, p. 3-16].

A subsystem may consist of three partitions, one for the interface of the subsystem, a second for the specification elements realizing the interface signatures and a third for the realization elements. As shown in Figure 2.8, use cases (the solid ovals in the specification elements compartment of graphs) might occur in conjunction with subsystems.

Generally, model management elements can participate in relations too, e.g., a package can be derived from another package.

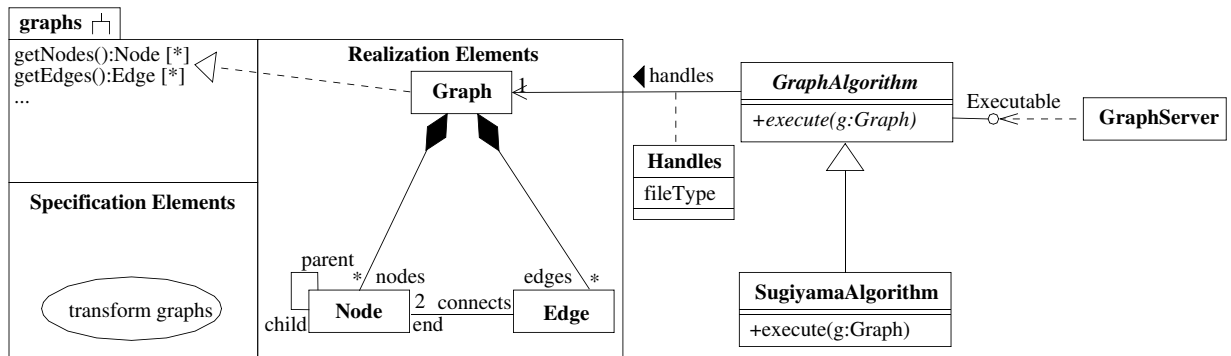


Figure 2.8: Figure 2.7 redrawn as a subsystem with explicit signature also containing a use case. A “lolly”, a shortcut to depict implemented interfaces, is shown at GraphAlgorithm which then is referenced by the class GraphServer.

### Comments (Annotations)

On the one hand, a comment, displayed as a kind of notepad as the one shown in Figure 2.9, may be attached to every element in a class diagram, i.e., classes, signatures within classes, relations, packages, etc. On the other hand, a comment might be attached to no element at all and may therefore provide additional information on a diagram as a stand-alone comment. Comments attached to a model element are connected by a dashed line to the target model element, which

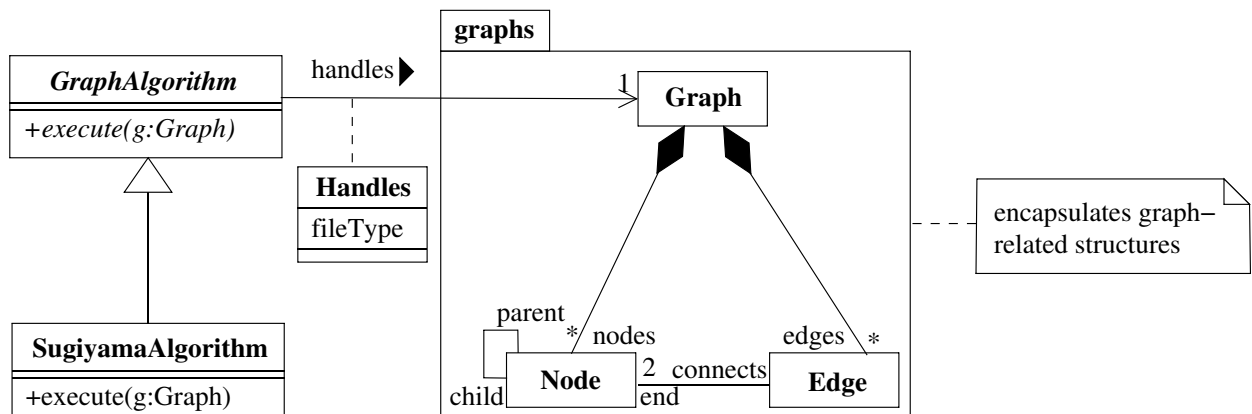


Figure 2.9: Comments can be attached to every model element, in this example to a package.

is similar to the connecting line of association classes. It is mentioned in the UML specification that, as with constraints, a comment may also be attached to multiple graph elements.

## Dependencies

When an element depends upon another element, a dependency relation emphasizing the (usually invisible) connection can be shown. A dependency is usually categorized by attaching a predefined stereotype, e.g. «import» for granting import access between packages. Especially, if template parameters on classes are specified, «bind» dependencies that show the mapping of template parameters can be used. Template parameters are displayed in a dashed rectangle at the upper right corner of the class.

Until version 1.5 of the UML, templates are a tricky issue: Templates are defined at the meta-model level of model elements and can therefore be specified for every model element including relations or packages. In fact, to avoid model level confusions, dummy classes holding the name of the parameters have to be declared and because of their nature, these classes then may violate the well-formedness rules of the UML. Consequently the use of templates is restricted in UML as follows:

A template is not a directly usable class because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a class. Only a class can be a superclass or the target of an association (a one-way association *from* the template to another class is permissible, however). A template may be a subclass of an ordinary class. This implies that all classes formed by binding it are subclasses of the given superclass.

[OMG 2003c, p. 3-52]

Fortunately, version 2.0 of UML introduces a completely redesigned template mechanism that is more appropriate to real world applications or programming languages like C++. For a detailed discussion of an extension of UML 1.x towards templates like those in C++ see [Eichelberger and v. Gudenberg 2000].

A dependency is drawn as a dashed line having an opened arrowhead from the client towards the supplier, the element on which the client depends on. A dependency is depicted in Figure 2.8 between the “lolly” at `GraphAlgorithm` and the class `GraphServer`. The interfaces implemented by a class can visually be emphasized by depicting the interfaces as an individual symbol, the so called “lolly”, and to connect it to the implementing class. The name of the interface is drawn next to the lolly. Dependencies may be attached to the lolly to show coupling or usage, e.g., message interaction via the signatures of the interfaces.

Multiple dependencies from or to a model element may be joined. If desired, a small dot can be shown as a junction point. In this case it, is recommended in the UML specification that an additionally clarifying comment is attached to the junction point.

## Display Alternatives

The composition in the `graphs` package in Figure 2.9 might alternatively be drawn in the rectangle of the class `Graph` as shown in Figure 2.10. The role names are then part of the class names of the classes in composition relations and the multiplicities are displayed within the names compartments.



Optionally, elements defined within a package can be depicted using the *anchor notation*. Because drawing classes within classes must be interpreted as a composition, a nested (inner)

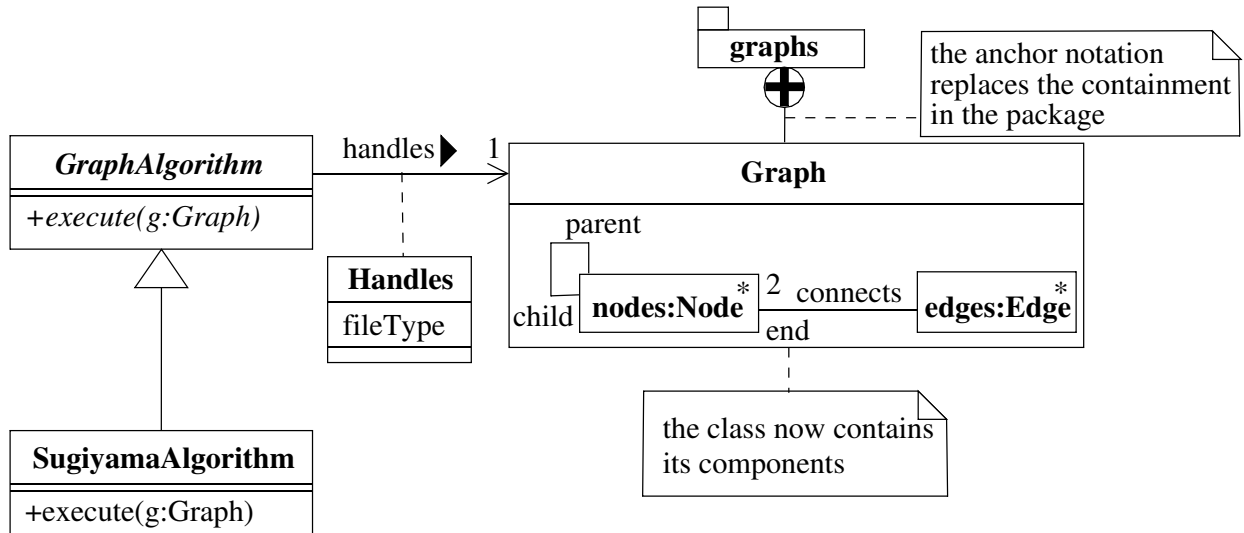


Figure 2.10: Alternative notations for composition, packages and nested package elements.

class must not be drawn in that way but the nesting can be denoted by applying the anchor notation [OMG 2003c, p. 3-82]. Furthermore, in Figure 2.10 the name of the package is drawn within

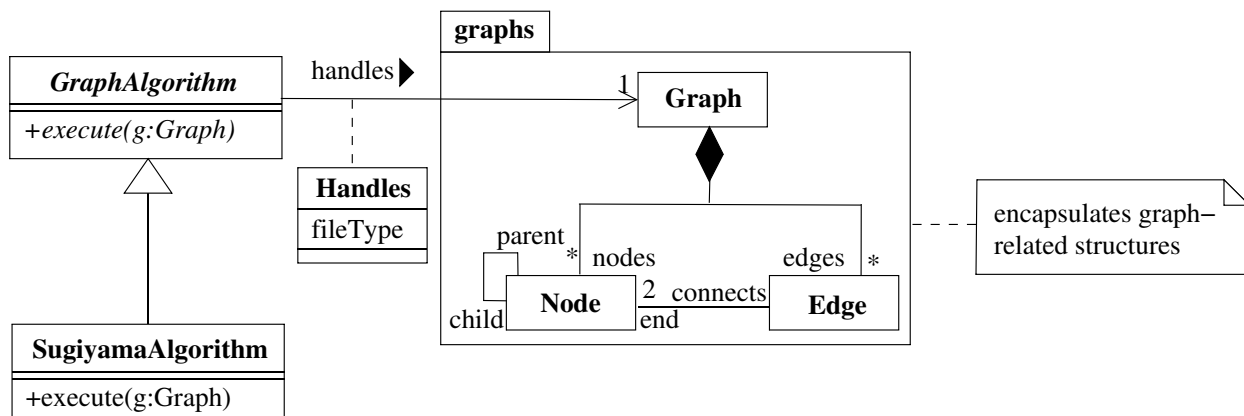


Figure 2.11: Figure 2.9 redrawn in shared target style.

the package rather than in the tab of the package.

The UML mentions explicitly that groups of edges (dependencies, inheritance arcs, aggregations or compositions) connected to the same class might be drawn as joined arcs (shared target style) as depicted in Figure 2.11 instead of drawing straight lines for each individual edge (separate target style) as shown in the figures before.

## Other Elements

To emphasize instances of classes, dependencies to instantiated objects may be shown in class diagrams. An *object* is drawn like a class, usually without operations but with concrete values for attributes, the class name is underlined and a unique name for the instance (similar to the composition notation in Figure 2.10) might be given. *Active objects* which own control over their own thread are drawn with a thick border line, *multi-objects*, specified with collaboration diagrams, representing collections are shown as a stack of two rectangles [OMG 2003c, p. 3-127]. In analogy to the class based composition notation, objects realizing such a collaboration might be drawn within the box of the collaboration. Beside dependencies, only instances of associations, *links*, can be specified as relations between objects.

In collaboration diagrams, objects interacting with each other in parallel or sequential flow of control are visualized. Since at least class based collaborations might be interesting in class diagrams, e.g., for emphasizing design patterns [Gamma et al. 2000], collaborations are shown as dashed ovals containing the name of the collaboration. The collaborating classes are connected by dashed lines, possibly adorned with role names. Additionally, because templates are available at model element level, generic collaborations in the sense of design patterns may have template parameters and the connected classes then play the role of a template argument.

## UML Default Layout

As mentioned in Section 2.1.1, the UML specification implicitly introduces a *default layout* by drawing certain elements in a common fashion. By comparing the class diagrams shown in the specification document [OMG 2003c] we have found the following conventions:

- Inheritance and realization relations appear most times in vertical direction, the arrow was directed to the top of the diagram. Most times shared target style, which induces joined relations, was applied and a hierarchy was emphasized.
- Anchors occur in hierarchical, horizontal style, the cross symbol was directed to the top of the diagram.
- Dependencies often also form a hierarchy directed to the bottom of the diagram while bidirectional dependencies were frequently drawn in horizontal fashion.
- Aggregations, compositions and directed associations were oftentimes drawn vertically, but appear also in horizontal direction like undirected associations.
- Relations were connected to all sides of classes whereby vertical directed relations most times connect to the horizontal sizes of a class box.
- Class rectangles were drawn often larger than the minimum area requirement of the interior elements to provide sufficient space for the connected relations.
- Edges are drawn as straight lines and, dependent on shared or separate target style, paths are drawn in orthogonal or directly connecting fashion, respectively.

- Association classes appear in close vicinity and (at horizontal associations) below the connected association.

We will keep this set of conventions in mind when comparing other diagram and layout approaches as well as when defining our own set of layout criteria in the next sections.

### 2.1.3 Criticism on UML

It has long been recognized that UML 1.x is too large and complex, making it unwieldy to learn, apply and implement.

[Kobryn 2002]

Beside many positive aspects of a unified and standardized modeling language, over the years different criticism on UML has been published. Some of this criticism directly affects how UML is implemented in tools: Many tools are not fully compliant to the UML specification, realize selected parts only or provide various mechanisms to (automatically) reduce the visual complexity of the diagrams. The main question in this section is, if we are allowed to follow that main stream, too, and if the result can then be called a UML tool or if we somehow have to indicate the differences. This section prepares some arguments to be applied when we will collect a basic set of requirements to our work in Section 3.1.

Multiple ways to display the same situation and the number of different types of elements in a diagram increase the cognitive load to read and to understand a diagram. Beside personal opinions, most of the currently implemented CASE tools seem to enforce a certain sub standard and do not fully implement one of the UML 1.x specifications but claim to be fully compliant according to their web sites and promotional materials.

In [Eichelberger 2002b], we considered 60 tools for an evaluation of the layout features of UML tools. For 18 tools we were not able to complete the installation procedure or we found out that no appropriate UML facilities were provided. For the other 42 tools, we applied the test diagram shown in Figure 2.12 consisting of a package, two association classes, two notes, two reflective associations and a ternary association. Even if the focus was more on layout, we found out that only 7 tools were able to take the complete diagram as input. This situation has remained the same since at least 2000:

As for modeling tools, the author knows of none that fully implements the UML 1.1 semantics and notation (adopted three years ago), let alone one that completely or correctly implements the current UML 1.3 specification (which was adopted a year ago).

[Kobryn 2000]

Some reasons for this discrepancy may be that the UML is too big for implementers to be realized as well as too big for developers to be remembered as a whole as mentioned in [Mellor et al. 1999; Egyed 2002]. Furthermore, UML appears to be unclear and ambiguous due to contradictory definitions, to be not precise and therefore leads to many conflicting interpretations. And finally,

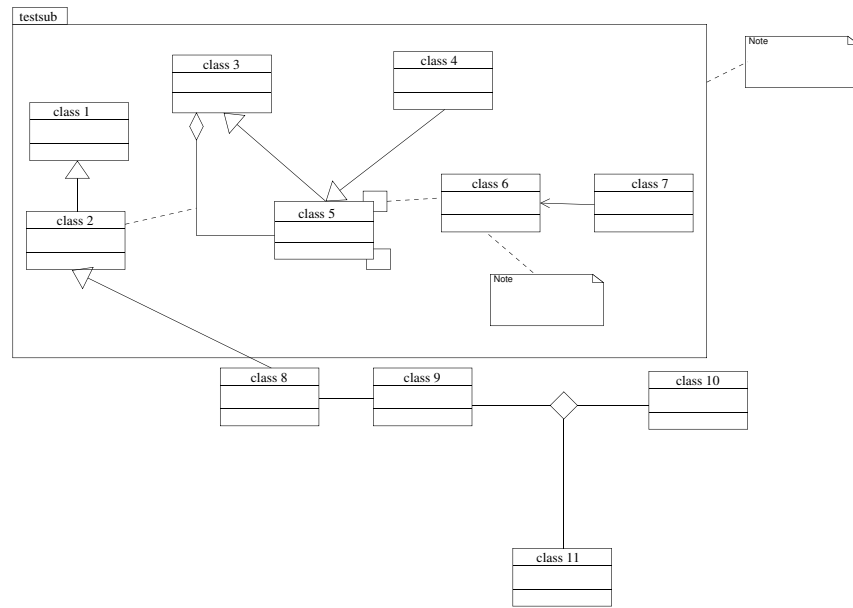


Figure 2.12: The test class diagram used in [Eichelberger 2002b]. To test an individual UML tool, the spatial distribution of the elements had to be reproduced by the tool’s editor and had to be laid out by the built-in layout mechanisms.

it provides a high degree of freedom due to presentation options and the variability of model elements in the diagrams.

Furthermore, Color is one of the features implemented by most UML tools and suggested by different research to improve UML diagrams in certain situations. On the one hand, color, a so called secondary attribute, does not change the semantics of the diagram itself. On the other hand, the interpretation of colors is dependent on various cultural aspects. This might be a reason why UML version 1.x does not specify any use of color while version 2.0 proposes shading (no concrete colors) for different visibilities of model elements within a package.

Of course, these facts are known to the OMG, too, and some ambiguities seem to be intended:

Dynamic tools need the freedom to present information in various ways and the authors do not want to restrict this excessively. In some sense, we are defining the “canonical notation” that printed documents show, rather than the “screen notation.” The ability to extend the notation can lead to unintelligible dialects, so we hope this freedom will be used in intuitive ways. The authors have not sought to eliminate all the ambiguity that some of these presentation options may introduce, because the presence of the underlying model in a dynamic tool serves to easily disambiguate things.

[OMG 2003c, p. 3-5]

“UML is not only a de facto modeling language standard; it is fast becoming a de jure standard.” [Kobryn 1999] Therefore, as far and as fast as possible, implementations should be compliant to

the complete specification or it should be clearly described what parts of the specification are not realized.

### Sub-Languages of UML

Even if the last section concluded with the need for a full compliance of an implementation to the UML, sometimes only parts of the UML are realized. For example, package notation might be replaced by anchor relations or less used elements (dependent on the perspective of the implementer) might be left out. But from the viewpoint of a modeler who learned to use the UML as a tool, these elements might be important in specifying the design of a system. Starting with an initial, possibly incomplete set of diagrams, UML elements like association classes, bidirectional associations, subsystems or higher associations might be used, even if they are not directly represented in the target programming language. But all these elements can somehow be implemented, e.g., using additional libraries, so that an abstract element like a higher association can be realized. When models are then transformed from an abstract target-independent to a more concrete (finally platform specific) model, these abstract elements can be substituted by a set of model elements in a top-down fashion. UML elements are then replacing other more abstract UML elements towards an implementation model, and target language elements are then be combined to realize the less abstract UML elements from the implementation model.

Therefore, if the implementation of an UML tool restricts the UML towards a sub-language of the UML because of the mentioned reasons, the degree of freedom and abstraction of software engineers is illegally restricted. The same is true for the other direction as described in [Koschke 2003].

Of course, from our perspective, a possible exception would arise, if the tool clearly mentions that it implements a sub-language of UML and specifies which parts are left out. But UML itself clearly states:

Note that a tool is not supposed to pick just one of the presentation options and implement it. Tools should offer users the options of selecting among various presentation options, including some that are not described in this document.

[OMG 2003c, p. 3-5]

Obviously, the same is meant for the non-optional model elements.

The set of presentation options should be seen as a style guide like rules for indentations in Java or C++ source code. An organization might specify its own style guide like for source code, but it should always be kept in mind that UML itself is a kind of primary style guide, too, and that to minimize the costs of communication, a tailored style guide or notation guide should not differ too much from the standard. Also, aesthetic conventions like those to be discussed in this work rather than font shapes or other basic graphical features should be part of such a style guide.

As a conclusion we can state that the UML provides many different ways to express individual software artifacts to allow different kinds of abstraction levels for various purposes. Even if highly abstract elements might thereby be used, these elements can be realized by transformation. Furthermore, this freedom in modeling is enriched by multiple styles for individual elements,

which might be modified by extension mechanisms even if this may increase the complexity of communication. Hence, while implementing a UML tool, there is no justification for implicitly restricting the standard. This is the reason why we focus more on the support of hopefully all model elements and variations of the UML dealing with class diagrams, than on a fast and more graph-drawing theoretical sufficient algorithm and implementation which ignores parts of the UML.

### Reduction of Visual Complexity

Another approach to get a much simpler diagram would be a transformation of the a diagram into a semantically equivalent but visually less complex diagram. A simple idea would be to collapse model elements containing other model elements to the containing model element only. Packages, subsystems or classes might be collapsed but the relations to the contained elements are kept or transformed as described in [Köth 2001; Köth and Minas 2002]. Alternatively the complete diagram might be transformed into an abstraction of itself applying a complex set of rules as proposed in [Egyed 2002].

The UML clearly distinguishes between diagrams as a graphic view and the underlying model of the entire software system which is described by all user diagrams. That underlying model is a consistent collection of all model element instances, possibly with history relations as in a Concurrent Versions System (CVS) repository and flagged according to the state of modeling, e.g., requirements capturing, analysis, design, implementation, etc. to which an individual model element belongs. A diagram as a view is a graphical description of a set of model elements collaborating to describe (visually) a certain static or dynamic situation of the software system being modeled. The decision about which subset of the elements and relations of the underlying model should be shown is made by the modeler dependent on the level of abstraction to be displayed. Therefore, a tool should transform a diagram only as part of a model or diagram transformation, e.g., as described for the Model Driven Architecture (MDA) [OMG 2003a]. On a single diagram, such a transformation should be initiated by the user or, according to a set of predefined transformation rules, by an automatic mechanism. But such a mechanisms is rather a part of a CASE tool than a layout tool or plug-in. A layout tool or plug-in is required to calculate the layout of whatever information is given as input.

Zooming and transforming mechanisms are far beyond the scope of this thesis, because our research project focuses on the aspects of aesthetic layout, even if we need a kind of browser to demonstrate our implementation. Of course, that browser is not intended as a UML or CASE tool, such as current UML or CASE tools often do not appear to be layout applications. That browser is therefore a viewer with demonstration purpose including file transformation and repository access facilities, but neither a diagram editor nor a code generator, an engineering tool, a CASE or a MDA tool.

### 2.1.4 Alternative Visualizations

The more alternatives, the more difficult the choice.

Abbe' D'Allanival

UML itself notes that most of its diagrams and some complex symbols should be described as graphs in two dimensions [OMG 2003c, p. 3-6]. Yet, from different perspectives, the diagrams of UML do not apply well to all situations in software engineering. In this section, we discuss some other approaches to the layout of (UML) class diagrams, in particular those relevant to automatic drawing of software engineering diagrams and visualizations. However, we keep in mind that UML is our main focus, because it is a standard, and we attempt to find a method for automatically laying out UML class diagrams that respect all features and options defined by the standard.

In [Diskin et al. 1999; Diskin et al. 2000; Diskin 2002a; Diskin 2002b], the universal arrow diagram logic was proposed as a more intuitive and precise description than the class diagrams of the UML. Visual models were seen as sketches (the specification format of the arrow diagram logic) in the language of generalized sketches from category theory. Even if this approach seems to be more formally founded from the viewpoint of HCI, reading of diagrams always depends on the personal skills at a certain graphical notation. From our viewpoint, diagrams dependent on the universal arrow diagram logic might be more precise but not more readable.

In [Teoh and Ma 2002] the ringed circular layout for trees, in which children are placed in equal sized circles around the center of the ring denoting the parent node, was presented. A similar technique is the radial layout, e.g., described in [Ellson et al. 2003], in which the levels of a hierarchy are placed on concentric circles around the root. Because class diagrams may contain large hierarchies, the ringed circular layout or a radial layout could be considered instead of the more traditional default UML layout. In [Gil et al. 2002] various other diagram types were suggested: "Spider diagrams" as extensions of Venn-diagrams, "Constraint diagrams", which extend the arrow notation to describe static system invariants, and "3D-diagrams" for conceptual modeling of dynamic system behavior, e.g., contract conditions, sequences or collaborations in the third dimension.

Different reverse engineering tools rely on a more proprietary notation in three dimensions. [Dwyer 2001] advocates the use of 3D layout for UML based on the results of a user study and general reasons for introducing 3D into graph drawing like coping with complexity [Eades and Feng 1997] or prevention of cluttering due to the quantity of information [Ware et al. 1993]. Hence, three dimensional drawings can be used to provide multiple abstraction levels and this may then simplify the orientation of the user in large information structures. An alternative notation optimized according to criteria of diagram perception is shown in Figure 2.13.

In two dimensions, due to the complexity of the diagrams, a class diagram often cannot be drawn without edge crossings. Because another dimension is added in 3D, edge crossings disappear automatically, even if, according to [Ware et al. 1993], also multiple views introduced by 3D, motion or stereopsis might be responsible for the reduction of recognition errors in 3D .

Especially in combination with color, 3D is advocated in [Ware et al. 1993] because it seems

to be effective for human perception to distinguish multidimensional discrete data. But, as discussed in Section 2.1.3, color as well as 3D are proprietary features with respect to the UML specification documents.

WilmaScope [Dwyer and Eckersley 2003] and CrocoCosmos [Lewerentz and Noack 2003] cal-

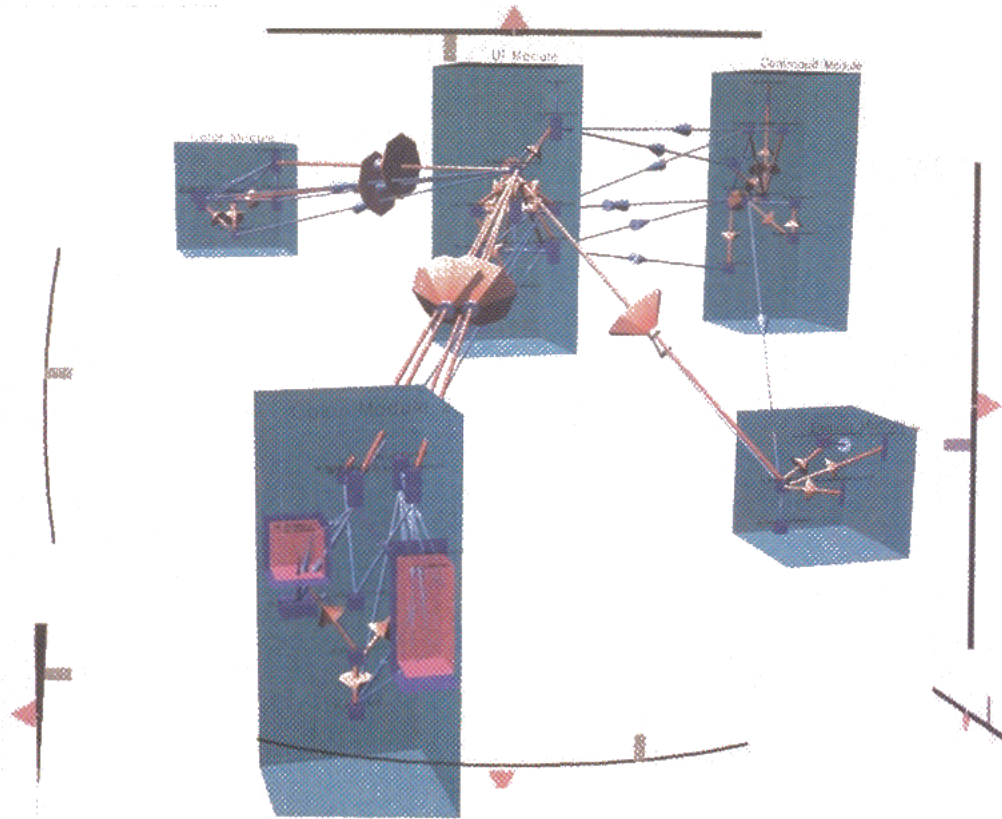


Figure 2.13: The structure of object-oriented software code in a 3D class diagram notation (from [Ware 2000]). The large boxes represent modules containing classes, variables and methods. The 3D spars interrelating the entities represent various kinds of relationships like inheritance, function calls and variable usage.

culate their layout based on different force functions applied to the relations between the entities, some rely on the simulation of different magnetic fields to influence the layout. For example, in CrocoCosmos, vertices represent general program entities, colored edges relations of different types, edge directions are visualized by color transitions, vertex properties are encoded by geometric shapes and the positions in 3D space encode the relational structure.

Even if pictures produced by tools like WilmaScope or CrocoCosmos might be appropriate for different specific software engineering situations like optimizing an existing program with respect to coupling and understandability, there is a large difference between the proprietary notation of these tools and the UML notation to be used when designing software. UML diagrams and such proprietary diagrams may coexist at the same time, when diagrams are used to under-



stand or optimize the source code, e.g., by reverse engineering existing code. Then, comparing and processing individual diagrams from both approaches obviously increases the complexity of mental operations.

## 2.2 Automatic Diagram Layout

Geometric representations of graphs have been investigated by mathematicians for centuries, for visualization and intuition, as well as for the pure beauty of the interplay between graph theory and geometry.

[Battista et al. 1999]

After introducing UML class diagrams in the last section, we will now describe basic aspects of automatically drawing diagrams. First, we will discuss different reasons, why it is desirable to draw UML class diagrams automatically. Then we will have a closer look at graph drawing, the discipline which is the foundation for calculating the layout of a general graph using computers. We will finally describe existing approaches to the automatic layout algorithms for UML class diagrams published so far.

### 2.2.1 Why to Draw a Class Diagram Automatically?

In our work we placed much more emphasis on manual layout.

[Ware et al. 1993]

Using tools in software engineering is common but even nowadays basic tools rather than sophisticated process enabling technologies are in use. According to the classification of CASE tools in [Fuggetta 1993], even editing tools (textual and graphical editors) are called CASE tools. When implementing a project, programming tools (editors, compilers, profilers, test tools, debuggers), verification, validation and configuration management tools are in use. These tools are then combined into workbenches to provide a unique user interface. Various plug-ins, which analyze source code and produce UML diagrams for integrated development environments (which are classified as workbenches in [Fuggetta 1993]) exist. In [Eichelberger 2002b], we considered 60 different tools which are intended to support analysis and design in the development process prior to implementation.

Due to changed requirements and more precise knowledge on the system to be created, design decisions in implementation have to be revised, and diagrams from the early phases, which might be a good documentation of the system, run out of date. To avoid these inconsistencies, the round-trip engineering cycle was invented: Source code is (partly) generated from diagrams which then are kept consistent by reverse engineering the source code after modifications. If this cycle would work perfectly in current tools, implementation, documentation and maintenance would be simplified [Ohst et al. 2003]. Even in visualization tools working on general data, roundtrip visualization is currently a one-way trip as noted in [Charters et al. 2003].

Still, due to the non-engineering practices of the early days of computer science, large systems are not well documented and as part of maintenance and changes to old software, first documentation has to be created from information gained by reverse-engineering methods. Even this information is transformed into diagrams.

Therefore, automatic layout of software engineering diagrams appears to be an important feature to CASE tools:

- **Standardization:** Automatically drawing diagrams simplifies the conformance to a certain style guide [Batini et al. 1985; Protsko et al. 1991; Eichelberger 2002a; Eichelberger 2003] even if this can be reached by manual editing, too. Furthermore, the expressive power of diagrams can be increased (automatically) [Batini et al. 1985].
- **Costs:** The use of diagrams increases the communication between designers, stakeholders, managers and (intended) users [Batini et al. 1985] but also, in collaboration with a standardized style guide, automatic drawing helps to reduce costs of communication [Eichelberger 2002a; Eichelberger 2003]. Also, generally, production and maintenance costs can be reduced [Batini et al. 1985].
- **Errors:** Nowadays, CASE tools relying on diagrams usually provide automatic diagram checks which are independent from layout and layout algorithms. But as described in [Eichelberger 2003] (and as discussed later when addressing class diagram aesthetics) the layout of a class diagram might provide information about the design so that a nice layout might arise from a good design. If a diagram is drawn according to a certain set of aesthetic rules, some visual indicators can help reducing design errors.
- **Incremental editing:** Small diagrams appear to be manually drawn faster than larger diagrams, but as requirements change and diagrams have to be adjusted, especially if elements in the center have to be inserted or deleted, layout algorithms help saving time. Of course, manual editing does not scale with the size of the diagram [Protsko et al. 1991; Sugiyama 2002; Eiglsperger 2003; Spinellis 2003].
- **Reverse engineering:** As long as diagrams are changed in an incremental process, the current editing step as well as some of the expected steps in the near future are known to the user. Therefore, even if manual editing of a diagram is tedious, it is a stepwise process. If data is initially gained from reverse engineering, complete diagrams are build from the information without any layout data. Especially on such diagrams but even on a large number of changes to a diagram after a round-trip code analysis, editing does not scale with the size of a project and its diagrams [Eiglsperger 2003].
- **Automatic documentation:** With data from a repository but also with information gained from source code analysis, documentation of a software engineering project can be kept synchronized with the development diagrams and the source code [Protsko et al. 1991; Eiglsperger 2003].

## 2.2.2 Graph Drawing – an Overview

In this way, for the visualization of graphs to function as a useful method for the communication of concepts, it is necessary to produce good diagrams; however, the generation of good diagrams is not so easy.

[Sugiyama 2002]

Interrelating abstract items to a complex structure is the basic idea of a graph, the fundamental construct in graph theory. Depending on the definition of the features of the items (nodes, vertices), the interrelations (edges, arcs) and the entire structure, different types of graphs can be identified. Graphs, usually consisting of a finite set of nodes or edges, respectively, can then be applied to various application domains, and, with an appropriate definition, a class diagram, as stated in the UML, can also be represented as a graph.<sup>5</sup>

A graph is often drawn on the plane to illustrate the problem or to sketch a proof of a certain feature. The tedious task of drawing a (larger) graph seems to be an appropriate problem to be processed by a computer. Sometimes prehistoric cave drawings are mentioned as the first (graph) drawings [Sugiyama 2002]. From the first work that dealt with the question of how to draw a graph in 1963 [Tutte 1963], graph drawing has emerged as a discipline with its own practical and theoretical results. We only introduce briefly the discipline which acts as a foundation for our work. Different books on graph drawing and applications [Battista et al. 1999; Kaufmann and Wagner 2001; Sugiyama 2002; Jünger and Mutzel 2003a], the graph drawing bibliography [Di Battista et al. 1994] and the proceedings of annual graph drawing conferences provide a wide range of literature for a more detailed overview on individual results of current work.

In this section, we focus on the graph drawing discipline itself but always keep our application domain in mind.

Graphs can be drawn according to various conventions using different algorithms. Therefore, for a given graph, usually plenty different drawings can be created. In [Feng 1997, p. 1] the *graph drawing problem* was defined as the activity of designing an algorithm which automatically assigns a position for each node and a route for each edge of the input graph. Some drawings of a graph might be perceived as nice diagrams if a “correct” algorithm was used to calculate the placement of nodes and edges.

Dependent on the structure of the nodes and edges, various types of graphs can be identified. Some of the basic types will be introduced in the following. An edge, usually connecting exactly two nodes, which occurs more than once is called a *multiple edge*. An edge having the same node as start and endpoint is called a *self-loop*. A graph, without self-loops and multiple edges is called a *simple graph*.

A drawing of a graph is called *planar*, if no two distinct edges intersect. Two planar drawings of the same graph are called equivalent if the drawings determine the same circular orderings of

---

<sup>5</sup>A definition of the type of graphs to be used in this work as well as a mapping of UML class diagrams to graphs will be given in Section 4.2.

the neighbor sets. A *planar embedding* is then defined as an equivalence class of planar drawings described by the circular order of the neighbors of each vertex. Unfortunately, due to the complexity of UML class diagrams, the structure of most class diagrams do not admit a planar drawing<sup>6</sup>.

A *directed graph* (or *digraph*) requires an orientation on each of its edges, therefore the set of edges incident to a node can be partitioned in a set of incoming and in a set of outgoing edges, respectively. A (directed) *path* of edges is defined as a sequence of distinct nodes each intercon-

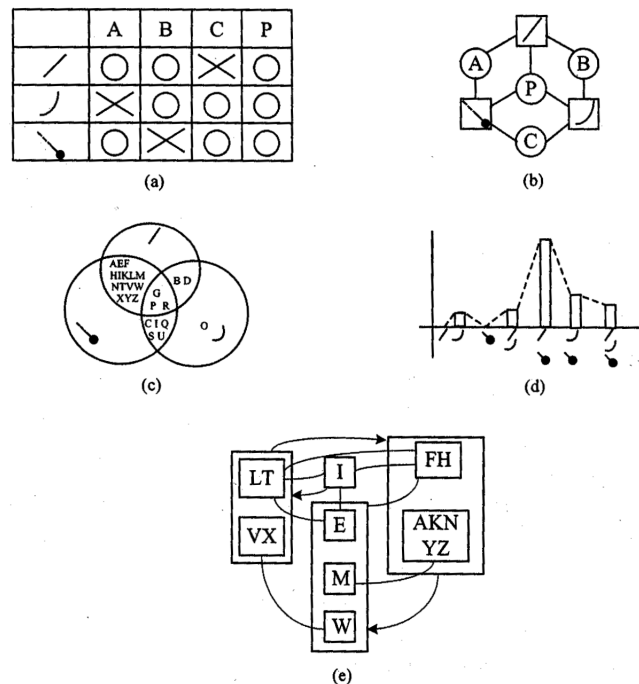


Figure 2.14: Basic and composite diagram languages: (a) matrix, (b) net, (c) region, (d) coordinate and (e) a composite of net and region (from [Sugiyama 2002, p. 2]).

nected by at least an edge of the graph. If a path ends at the same node where it started, the path is called a *cycle*, and, hence, a graph which does not contain cycles or self-loops is called an *acyclic graph*.

As described in Section 2.1.2, class diagrams may have reflective associations (as well as aggregations or compositions) which obviously map to self-loops and multiple associations (as well as aggregations or compositions) interrelating two classes which map to multiple edges. Therefore, a class diagram is not always a simple graph. Unfortunately, most non-application domain specific algorithms, e.g. those described in [Battista et al. 1999], deal with simple graphs. Furthermore, lot of the algorithms interpret nodes as pixel points and therefore have to be extended and tailored to fit the application on UML class diagrams. Before discussing these basic graph

<sup>6</sup>We will see examples of such diagrams when discussing the architecture or details of the implementation, e.g., in Figure 6.1 on page 286.

drawing algorithms, we will, however, have a look at different taxonomies and conventions for drawing graphs.

According to [Sugiyama 2002, p. 2], the combination of individual elements of diagrams and the rules of arrangements, the syntactic grammar, is called *diagram language*. Four basic types of diagram languages, matrix, net, region and coordinate type, are shown in figure Figure 2.14. Obviously, UML class diagrams appear as a combination of net and region type due to the opportunity of nesting elements, e.g., classes in packages. Unfortunately, it is mentioned in [Sugiyama 2002, p. 3] that realizing an algorithm for diagrams combining both types involves much work, because appropriate drawing methods are relatively underdeveloped, despite the fact that they are commonly used in everyday life.

For the edges of a graph, different drawing conventions [Tamassia 1998; Battista et al. 1999] or graphical standards [Tamassia et al. 1988; Nummenmaa and Tuomi 1990] can be found in the literature:

- **GS\_STRAIGHTLINE:** Each edge is drawn as a straight-line segment [Tamassia et al. 1988; Nummenmaa and Tuomi 1990; Tamassia 1998; Battista et al. 1999].
- **GS\_POLYLINE:** Edges are drawn as sequences of segments, called polygonal chains [Tamassia et al. 1988; Nummenmaa and Tuomi 1990; Battista et al. 1999].
- **GS\_CURVES:** Edges are depicted as curves, e.g., Bézier arcs as in [Gansner et al. 1993].
- **GS\_ORTHOGONAL:** The edges are routed as polygonal chains of alternating horizontal and vertical segments [Tamassia et al. 1988; Nummenmaa and Tuomi 1990; Battista et al. 1999].
- **GS\_GRID:** Vertices, edge crossings, and edge bends have integer coordinates [Tamassia et al. 1988; Nummenmaa and Tuomi 1990; Tamassia 1998; Battista et al. 1999]. The basic coordinates system of a graphics context of an usual computer implies a (1, 1)-grid. Therefore, a predefined lower grained (x,y)-grid might be given so that the elements somehow snap onto that grid.
- **GS\_UPWARD/GS\_DOWNWARD:** For directed acyclic graphs each edge is drawn monotonically increasing/decreasing in the vertical direction [Tamassia et al. 1988; Nummenmaa and Tuomi 1990; Battista et al. 1999]. Other directions more appropriate to the application domain might be defined similarly.

Even if the following conventions appear to be qualitatively different, they are also mentioned as graphical standards in the literature:

- **GS\_PLANAR:** No two edges cross [Battista et al. 1999].
- **GS\_LAYERED:** The drawing area is partitioned into several layers and each node is assigned to such a layer [Sugiyama et al. 1981].

The drawing conventions mentioned above should not be regarded as a set of orthogonal features, because mixtures of e.g. `GS_STRAIGHTLINE` and `GS_GRID` are in use, e.g., as mentioned in [Tamassia et al. 1988; Nummenmaa and Tuomi 1990]. Some of the drawing conventions like `GS_GRID` directly depend on the underlying coordinates system, some like `GS_PLANAR` do not. Furthermore, besides an usual graph, different types of graphs have been defined in the literature. The following list is not exhaustive and enumerates just a few types that are relevant to our work:

- **General graph:** As informally described above, a structure consisting of a set of nodes and a set of edges interrelating exactly two nodes is called a general graph. For example simple graphs, directed graphs or trees appear as more specialized types of general graphs.
- **Clustered graph:** As defined in [Feng 1997; Brockenauer and Cornelsen 2001], in a clustered graph the set of nodes is (recursively) partitioned so that each node is part of a cluster. Edges may connect two nodes, but clusters must not be involved.
- **Compound (directed) graph:** In [Sugiyama and Misue 1991; Sander 1996b; Brockenauer and Cornelsen 2001] a (directed) graph defining the adjacency relations and a second directed graph defining inclusion relations were used to introduced compound graphs. In this construction, a node in at least one inclusion relation defines a compound. Compounds may also be involved in adjacency relations. Therefore, edges between compounds or edges crossing the borders of nested subgraphs are allowed, in particular because a compound graph is not recursively defined as noted in [Sander 1996b]. Furthermore, compounds may intersect other compounds, similar to the region type in Figure 2.14 (c).  
A refined set of conventions as the following from [Brockenauer and Cornelsen 2001] is then used to determine the basic features of the final drawing of such a graph:
  - A vertex is drawn as a rectangle with horizontal and vertical sides.
  - An inclusion edge between the nodes  $u$  and  $v$  is drawn so that the rectangle corresponding to  $u$  geometrically includes the rectangle corresponding to  $v$ .
  - Vertices are laid out hierarchically in terms of both inclusive and adjacent relations on parallel-nested horizontal bends, called compound levels.
  - An adjacency edge between  $u$  and  $v$  is drawn as a downward arrow with possible bends, originating from the bottom side of the rectangle corresponding to  $u$  and terminating on the top side of the rectangle corresponding to  $v$ .
- **Higraph:** As defined in [Harel 1988], a higraph allows relations between more than two nodes (hyperedges) and multilevel blobs similar to clusters which may include or intersect each other.

Dependent on the type of a graph, specialized algorithms are known, which ensure certain features in the drawing. For example, a tree admits a hierarchy and such a feature might be that all edges are pointing in the same direction or algorithms working on general graphs only are not able to handle the special requirements for compound graphs.

Furthermore, each drawing algorithm conforms to a set of aesthetic criteria, some of them

arising from the drawing conventions, some to improve the readability of the results. A large set of those rules has been identified in graph drawing and will be discussed along with other aspects to be respected for UML class diagrams in Section 3.3.

It is known that in the last years, no new graph drawing algorithms have been developed. All work on general graphs or on application specific domains select one of the classical algorithms according to their features, their advantages and disadvantages, and combine these algorithms with several additional processing steps to tailor the algorithms towards the desired field of application.

To provide an overview of the work done in graph drawing so far, we will now point out some of the well-known basic algorithms relevant to our application domain. A more exhaustive overview on basic algorithms can be found in [Battista et al. 1999; Kaufmann and Wagner 2001; Sugiyama 2002]. Further details on extended algorithms and implementations for UML class diagrams will be given in Section 2.2.3.

We distinguish the several graph drawing methods into algorithmic and declarative approaches. A more detailed taxonomy was given in [Sugiyama 2002] by considering hybrid approaches and influences from artificial intelligence.

### Algorithmic approach

The layout is calculated according to a well specified set of aesthetic criteria and other requirements directly embodied in the implementation of the algorithm. Typically the algorithm is specific to the type of graph and graphical standards to be respected, because algorithm and implementation are hard-wired. These algorithms are well studied, usually computational efficient and tailored towards specific problems. When user-defined constraints are possible, they are restricted to the specific graph-theoretic class of graphs they are designed for and, like most graph drawing algorithms, it might be extremely difficult to test a concrete implementation.

- The **topology-shape-metrics approach** [Tamassia 1985; Tamassia et al. 1988] was tailored for the layout of UML class diagrams in GoVisual [Gutwenger et al. 2003b] and jarInspector/yWorksUML [Eiglsperger 2003].

An orthogonal drawing is characterized by three fundamental properties, defined in terms of the equivalence classes they establish among orthogonal drawings of the same graph:

- Topology: Two orthogonal drawings have the same topology if one can be obtained from the other by applying a continuous deformation that does not alter the sequence of edges surrounding the faces<sup>7</sup> of the drawing.
- Shape: Two orthogonal drawings have the same shape if they have the same topology, and one can be obtained by modifying only the lengths of the segments of the orthogonal edge chains, without altering the angles formed by them.

---

<sup>7</sup>A face is a topologically connected region bounded by edges. The unbounded face surrounding the graph is called external or outer face.

- Metrics: Two orthogonal drawings have the same metrics if they are congruent, up to a translation or rotation.

Usually, a stepwise implementation realizes the hierarchical relationships between topology, shape and metrics: A planarization step determines the topology of the drawing and reduces the number of edge crossings, then the crossings are eliminated by inserting dummy nodes at the crossing points, then an orthogonalization step determines the shape of the drawing and finally a compaction step calculates the final coordinates of the nodes and the edge bends. This approach is intended to construct orthogonal grid drawings and allows a homogeneous treatment of a wide range of aesthetics and constraints even if the stepwise strategy implicitly determines an order of importance among the aesthetic rules to be respected.

- The **hierarchical approach, Sugiyama approach or STT method** was invented by [Warfield 1977; Carpano 1980; Sugiyama et al. 1981] and reused for the layout of UML class diagrams in [Seemann 1997; Eichelberger 1999].

A digraph is processed by applying the following three steps in sequence:

- Rank or layer assignment: Each node is assigned to a distinct level according to the directed relations of the graph. Depending on the directional convention to be applied for the hierarchy, this step implicitly determines a part of the coordinates to be assigned, e.g., if the hierarchy should be displayed top-downwards, this step implicitly determines the vertical coordinates of the nodes. Edge chains are created by inserting dummy nodes into each layer such an edge spans across, because an edge can span over several layers and a proper layered digraph is required to reserve a corridor for each of these edges to prevent node-edge crossings.
- Crossing reduction: The rank assignment does not consider if the horizontal position of the nodes are chosen to reduce the number of edge crossings (which may influence the readability). Therefore, the nodes per layer are reordered according to a certain strategy to keep the number of edge crossings as small as possible. The output is a proper layered digraph but now the topology of the graph is fixed.
- Assignment of coordinates: According to an appropriate strategy, the coordinates of the nodes and the individual positions of the edges are determined. Equal distances for adjacent layers may occur in abstract graphs but are not appropriate to all application domains.

As described for the topology-shape-metrics approach, the stepwise processing of this approach determines also an ordering among the aesthetic rules to be applied by the algorithm. Furthermore, the approach described above requires the input graph to be acyclic. For a general digraph, an additional processing step which temporarily eliminates the cycles, e.g., by reversing several edges, can be prepended. The original direction of the edges can be respected in the result.

Different modifications and variants of this approach have been described in literature, e.g.,



in [Sander 1996b; Schreiber 2002], this approach was extended to compound digraphs and in [Baburin 2002], a modification towards the maximum number of edge bends, the minimum number of orthogonal edge crossings between any pair of adjacent levels of hierarchy and an initial coordinates assignment before crossing reduction was described. Alternatively, layers must not always be horizontal or vertical lines; concentric circle drawings [Sugiyama 2002] or ringed circular drawings [Teoh and Ma 2002] might also properly reflect hierarchies.

- The **visibility approach** [Di Battista and Tamassia 1988]: is a general purpose three-step methodology for drawing graphs with the polyline drawing convention (GS\_POLYLINE) and was applied to ER diagrams, one of the historical predecessors of UML class diagrams, e.g., in [Tamassia 1985].
  - Planarization: A planar graph is calculated by applying a similar processing step as in the topology-shape-metrics approach.
  - Visibility: A visibility representation is constructed, so that each node is mapped to a horizontal segment, each edge to a vertical segment in the representation and a vertical segment between two nodes lies in between the horizontal segments of the connected nodes and does not intersect any other vertical segment. This step produces a skeleton or a sketch of the final drawing.
  - Replacement: The final polyline drawing is then created by replacing the segments of the visibility representations by nodes and polyline edge representation, respectively, according to a replacement strategy.

This approach appears as a variant of the topology-shape-metrics approach.

- **Divide and conquer approach** [Reingold and Tilford 1981]: Considering inheritance forests or packages in UML class diagrams, an approach might be to split the diagram into several subgraphs, to recursively draw the subgraphs and to construct the entire drawing by assembling together the drawings of the subgraphs. Obviously, the later the edges between the subgraphs are considered, the less readable the entire drawing might become: the subgraphs are not ordered according to the edges between the subgraphs and the layout of the subgraphs is calculated without respecting these edges. With decreasing size of the subgraphs, the complexity of the composition of the result graph increases [Sugiyama 2002, p. 32].
- **Force-directed approach** [Eades 1984]: The input graph is virtually transformed into a mechanical system consisting of steel rings (vertices) and springs (edges). Starting with an initial layout, the spring forces move the system to a minimal energy state and determine the result layout.  
An adaption of this physical analogy to OMT (Object Modeling Technique)<sup>8</sup> class diagrams was described in [Noguchi and Tanaka 1998].

---

<sup>8</sup>OMT is an object-oriented methodology introduced by James Rumbaugh in 1992 prior to the UML. Rumbaugh is one of the “three amigos”, the core developers of the UML.

Some variations of the basic model are known from literature: Gravity may be considered to keep nodes close together or underlying magnetic fields may be appropriate to enforce the direction of (certain) edges. Combining randomly chosen nodes with an appropriate cooling function is called simulated sintering or annealing .

For example, partial differential equations [Kamada and Kawai 1989], iterative evaluation of attractive and repulsive forces [Fruchterman and Reingold 1991] as well as simulated annealing and simulated sintering [Davidson and Harel 1996] were applied as basic techniques to calculate the layout of general graphs.

But the use of this approach with real-world applications seems to introduce different further problems, in particular when non-point nodes have to be processed. Then, as remarked in [Gansner and North 1998], the results tend to be cluttered, nodes overlap and clear routes for edges are lost.

- **Genetic or evolutionary algorithms** [Kosak et al. 1994; Mäkinen and Sieranta 1994; Utech et al. 1998]: Genetic algorithms (simulations of the biological genetic reproduction mechanism) can be used to calculate graph layouts, in particular the layout of UML class diagram as shown by an inofficial prototype known to us. As a stochastic global search method, an evolutionary algorithm works on a population of candidate solutions (individuals) and tries to optimize these by applying three basic principles: selection, recombination and mutation. Usually, the initial population is chosen randomly. In every following generation a probabilistic selection function determines individuals to be recombined and mutated. Then the resulting individuals become the current population for the next iteration.

As mentioned in [Utech et al. 1998], the most important aspect is to choose an appropriate representation for the individuals and suitable genetic operators. In [Mäkinen and Sieranta 1994], genes were encoded as integer strings, a gene evaluation function based on metrics was suggested and usual genetic operations like crossover and mutation were described.

For directed acyclic graphs, which have point-sized nodes, the edge length representation for genes was introduced in [Utech et al. 1998] and a distributed island model was applied. The example drawings show relatively good results respecting crossings and layering but the algorithm seems to require a long runtime.

Due to the alignment with the default UML layout style, other techniques like the augmentation approach [Battista et al. 1999], which produces triangulated drawings, are not considered here.

### Declarative approach

A completely other methodology is used by declarative approaches. The properties of a drawing are specified by a user-defined set of requirements. These layout constraints are then considered by constraint solvers or operation research techniques, like graph grammars or visual languages. On the one hand, such non-fixed-wired approaches have a large expressive power, because the layout is described by its desired properties which gives much control to the user. Thus, it provides a wide range of applications due to the inherent flexibility of the approach. On the other hand, it is difficult to formalize some constraints like planarity, the implementations are often

computationally inefficient due to the high flexibility, the interfering effects of constraints are difficult to express and, generally, a powerful constraint language for graphs is still missing.

As noted in [Sugiyama 2002, p. 4] we also can conclude that in many cases of optimization of layout criteria efficient algorithms may not exist. Therefore, often effective heuristics are applied instead, and, despite great advancement in graph drawing, there are still many research questions remaining.

We will select one of the basic approaches mentioned above for adaption to UML class diagrams in Section 4.1.

### 2.2.3 State-of-the-Art in Drawing UML Class Diagrams

Algorithms that draw graphs well may be poorly suited to diagrams.

[Protsko et al. 1991]

We will now present an overview on the use of graph drawing algorithms in commercial UML tools and will describe algorithms tailored towards UML class diagrams and similar diagrams.

#### UML Tools

As mentioned above, we conducted in [Eichelberger 2002b] an evaluation on the layout facilities of 42 UML tools and summarized the conclusions in [Eichelberger and von Gudenberg 2003b]. Even if the evaluation was done in 2002, we reevaluated some of the tools at the end of 2003 and found out that the main facts did not change. In the meantime, some of the tool vendors decided to disable their own layout algorithm and to delegate the layout calculation to external plug-ins. The examples of professional tools shown in Figure 2.15 to 2.18 were all laid out automatically using Figure 2.12 as input.

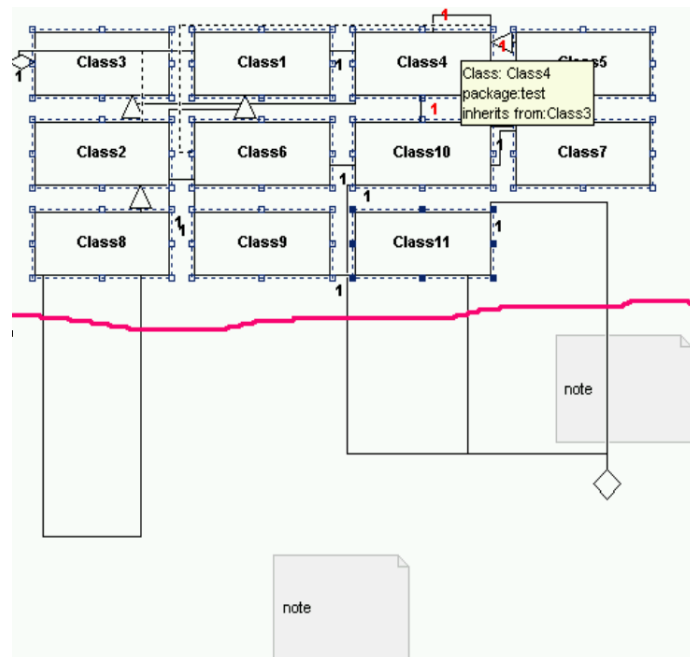


Figure 2.15: TNI OpenTool version 3.2.15 (horrible layout award): Class nodes are positioned with fixed distances, edges and adornments are not respected thereby. The diagram was cut because the lengths of some edges stretched the vertical extension and the diagram got too large.

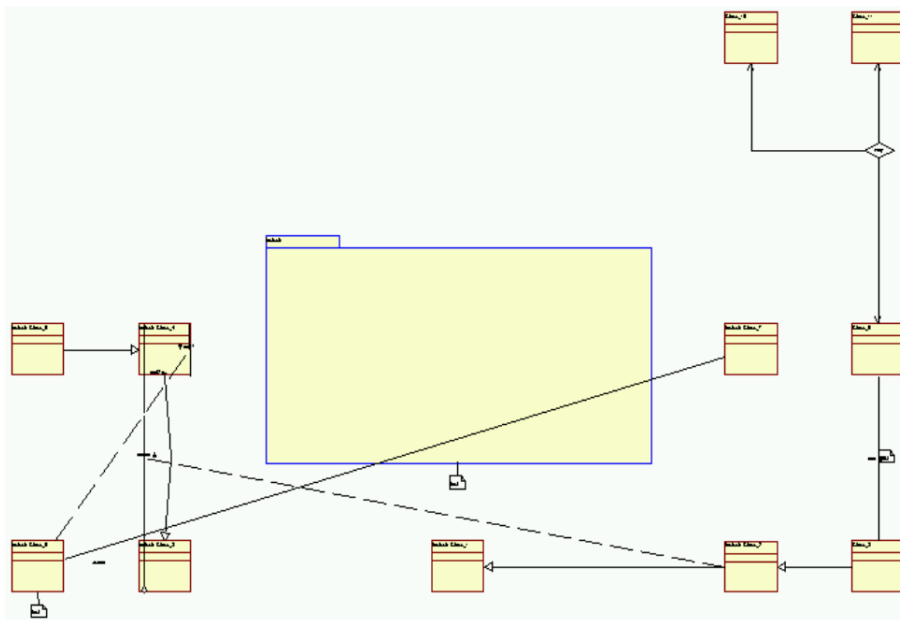


Figure 2.16: Popkin SystemArchitect version 8.5.16: A kind of orthogonal layout neither respecting different types of edges nor package containment.

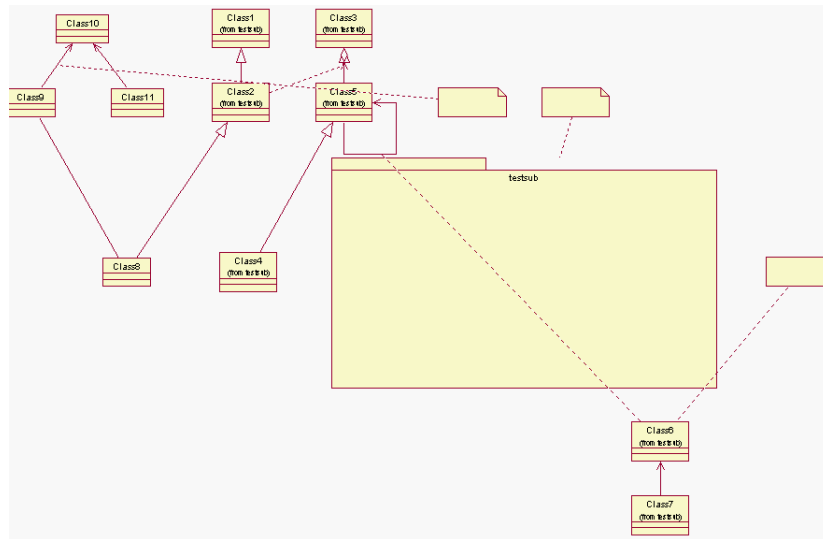


Figure 2.17: IBM/Rational Rose AnalystStudio version 2002.05.20: A kind of hierarchical layout where packages are not respected at all, some edges overlay other edges and connecting edges cross nodes.

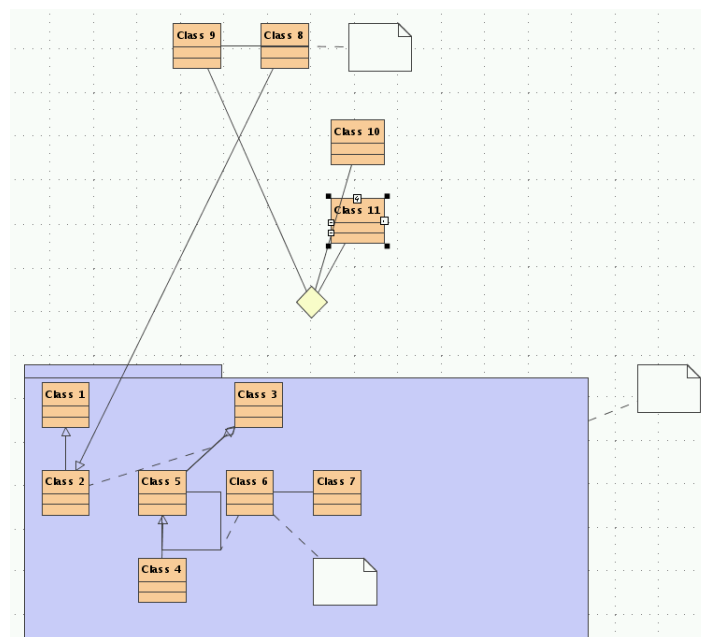


Figure 2.18: NoMagic MagicDrawUML version 7.0 (best layout award): A kind of hierarchical layout where package containment is respected, some edges overlap other edges and nodes.

### Research Tools on Similar Diagrams

Some work was done on one of the historical predecessors of UML class diagrams, ER diagrams: In [Batini et al. 1985; Tamassia 1985] some basic aesthetics were described and the topology-shape-metrics approach was applied. In [Nummenmaa and Tuomi 1990] visibility representations were used to handle planar ER diagrams only. A polynomial time algorithm based on the topology-shape-metrics approach with a constrained planarization and compaction for relational database schemas was presented in [Di Battista et al. 2002; Di Battista et al. 2003].

For network diagrams in [Kosak et al. 1994] two algorithms respecting certain visual-organization features are considered. The first algorithm incrementally augments a drawing by selecting and applying a layout rule until each node has been positioned. The second is a parallel genetic algorithm. As for all declarative approaches, it is not guaranteed that an algorithm is able to find a feasible layout compliant to the specified features, occasionally an unacceptable runtime due to the resolution-based search strategy may limit the overall performance and interferences between local and global user specified features as well as multiple features interacting at one node may lead to a mutually inconsistent layout.

Statecharts as well as bio-chemical pathways admit a hierarchical nesting of nodes and edges depicting transitions between states or pathways of the reactions, respectively. For statecharts, ViSta [Castelló et al. 2001; Castelló et al. 2002; Castelló et al. 2003] applies a variant of the hierarchical Sugiyama algorithm which was extended to handle compound graphs. A decomposition tree with three child node types acts as skeleton for the drawing. The drawings are oriented from left to right and Very Large Scale Integration (VLSI) techniques are taken into account for the floorplanning of the edges. The results show a low number of arc crossings, a natural hierarchical decomposition of states and a good aspect ratio. In [Brandenburg et al. 2003] the automatic layout of bio-chemical pathways based on a hierarchical approach respecting compound nodes, constraints in layer assignment and incremental drawing was described.

Regarding D-ABDUCTOR [Misue et al. 1995; Sugiyama and Misue 1996], it seems that this tool would have the power to be modified or extended towards UML class diagrams. Especially with all the browsing and scaling facilities implemented to preserve the mental map of the user, the tool seems to be reasonable for that task. Unfortunately, there is no attempt known to us to adapt the compound graph layout algorithm to UML class diagrams or state charts<sup>9</sup>.

### Research Tools on Class Diagrams

In 1997, Nakashima wrote a master's thesis on the automatic layout of OMT class diagrams. The inheritance subtrees of a class diagrams were treated as subgraphs, the metagraph consisting of the subgraphs was then processed by Walker's algorithm [Walker II 1990] and Eades' spring algorithm [Eades 1984]. In [Noguchi and Tanaka 1998; Noguchi and Tanaka 1999], an improvement of Nakashima's algorithm due to inefficient area usage and to missing relations between the subgraphs was proposed. It was recommended that associations be drawn from left to right and superclasses are placed above the subclasses in a top-down direction. The drawing algorithm

---

<sup>9</sup>When we started working on our project, we worried at least about the platform independence and made the decision for an own implementation.

itself was a modification of the magnetic spring model for multiple magnetic fields: Associations were modeled as bi-directional magnetic springs in a horizontal magnetic field, inheritance edges as uni-directional magnetic springs in a vertical top-down magnetic field and aggregations in a top-left to down-right diagonal magnetic field. The size of nodes was respected by adjusting the natural length of springs and the repulsive forces.

In the same year, Seemann proposed in [Seemann 1997] an extension of the hierarchical Sugiyama algorithm for the layout of UML class diagrams. The STT algorithm can be applied to the inheritance subgraph, because inheritance edges build a directed acyclic graph. Therefore, as preprocessing before executing the STT algorithm, all non-inheritance edges were temporarily removed from the graph and reflective associations were transformed to attributes. After calculating the rank assignment and the edge crossing reduction, the edges removed temporarily were reinserted in a step called incremental extension. Then, coordinates were assigned and in additional postprocessing steps the non-inheritance edges were routed. Thereby, inheritance edges were drawn according to GS\_STRAIGHTLINE while on the other edges GS\_ORTHOGONAL was applied. Inheritance edges were always connected to the horizontal sides, the other edges to the vertical sides of the nodes. The algorithm did not take the sophisticated model elements like model management mechanisms, association classes, n-ary associations or comments into account. A Java implementation of the “Seemann algorithm” called *SugiBib*<sup>10</sup> was then described in [Eichelberger 1999].

Diagen<sup>11</sup>, described in [Köth and Minas 2002], is an editor-generator toolkit which simplifies the implementation of language specific diagram editors. It works on a hypergraph model, consists of a reducer, a parser, a hypergraph transformer and an incremental layout mechanism. In the UML mode it respects most of the sophisticated UML elements described in Section 2.1.2, provides some features for semantical zooming [Köth 2001] and applies a force-directed layout algorithm which handles compound graphs and constraint propagation. Unfortunately, in our test, the incremental layout algorithm proposed unpleasing positions for several nodes.

GoVisual<sup>12</sup> [Gutwenger et al. 2002; Gutwenger et al. 2003a; Gutwenger et al. 2003b] relies on the topology-shape-metrics approach with hierarchical constraints calculated from the inheritance subgraph. It facilitates a mixed-upward and cluster planarization, where a cluster contains a separate hierarchy to avoid nesting of hierarchies. GoVisual is a C++ class library providing an API for C++, .NET or Java. Unfortunately, this approach does not fully consider the UML specification: Model management elements, association classes, dependencies, comments etc. are missing and the directions of edges are only locally consistent [Eiglsperger 2003, p. 33].

jarInspector/yWorksUML [Eiglsperger et al. 2003; Eiglsperger 2003; Wiese et al. 2002], built on top of the Java graph drawing library yFiles<sup>13</sup>, also applies the topology-shape-metrics approach with optimizations towards edge crossing minimization and hierarchical constraints calculated

---

<sup>10</sup>The term *SugiBib* was chosen as a combination of the name “Sugiyama” and the German word “Bibliothek” for library, because *SugiBib* was initially planned as class library rather than a framework. Furthermore the term denotes the bibliography of Kozo Sugiyama. Therefore the logo (by Peter Eades) represents the name Sugiyama in Japanese signs.

<sup>11</sup><http://www2-data.informatik.unibw-muenchen.de/DiaGen/>

<sup>12</sup><http://www.oreas.com/libraries.php>

<sup>13</sup><http://www.yworks.com/>

from the inheritance subgraph. Unfortunately, this approach also lacks an algorithm compliant to the UML specification because advanced elements like clusters and hyperedges are not considered:

We have chosen to not consider clustering since it is not supported by most modeling tools and is therefore used rarely in the context of class diagrams. Most tools do only allow to manipulate a flat diagram

[Eiglsperger 2003, p. 153]

In our UML tool evaluation [Eichelberger 2002b], exactly 50% of the tools allows classes to be placed in packages of arbitrary size even if most of the tools were not able to respect packages in the layout. Yet, we did not evaluate the code generation facilities of the tools.

One problem, selecting inheritance edges for the hierarchy exclusively is that there may be further hierarchical aspects relevant from the software engineering perspective, e.g., aggregation or composition hierarchies. Unfortunately, these mixed hierarchies are of less interest to the researchers:

It is even possible to visualize all hierarchical dimensions within one diagram. However, such an approach does not reveal enough analytical information, since in this case the graph is usually fully directed, including directed cycles. Thus a layout based on such a directed graph does not emphasize any hierarchical dimensions.

[Gutwenger et al. 2003a]

As we will evaluate later in Section 5.2, GoVisual and yFiles follow a philosophy, which is more influenced by graph drawing issues than structural and semantical aspects of the underlying software development diagrams.

Other approaches simply use the implementations of the graph drawing community and integrate the drawing tools into their own ones. In [Keller et al. 1999], the SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) environment is presented. SPOOL delegates the layout calculation to Dot [Gansner et al. 1993] for hierarchical and Neato<sup>14</sup> for force directed layouts. UMLGraph mentioned in [Spinellis 2003] tries to lay out diagrams using a mixed edges hierarchical approach also on top of Dot [Gansner et al. 1993] and has therefore to cope with overlapping nodes and edges.

We can conclude that there are different powerful extensions to traditional graph drawing algorithms, not only as theoretical results but also as implementations, but none of them was tailored towards the need of the software engineering community: The full compliance to the UML standard and readability with respect to the semantics of the diagram. This seems to be the state-of-the-art since 1998, because in [Markwitz 1998], appropriate layout algorithms were demanded, but only basic traditional graph drawing algorithms were proposed as solutions. From Section 2.2.2 we know that UML class diagrams are a combination of net and region, and now we can agree to the statement that, in particular for UML class diagrams, drawing methods are relatively underdeveloped [Sugiyama 2002, p. 2].

---

<sup>14</sup>Both tools belong to the same project at AT&T: <http://www.research.att.com/sw/tools/graphviz/>





# 3 Functional Specification

---

In this chapter, we will outline the functional specification of our layout algorithm for UML class diagrams. Therefore, we will collect and compile arguments from the four main disciplines involved in that topic: graph drawing, HCI, software engineering and software visualization.

First, we will compile a set of 12 basic requirements for a realization of a layout algorithm for UML class diagrams. In Section 3.2, input formats for UML class diagrams will be described and appropriate formats will be added to the set of requirements.

The next section will be on aesthetics for diagrams in general and UML class diagrams in particular. Therefore, first 26 structural and 11 semantic rules known from graph drawing literature will be listed. Then 11 general principles for diagram recognition from HCI, 9 semantical criteria related to good software design from software engineering and 3 requirements from the viewpoint of software visualization will be collected. All this interdisciplinary information will then be compiled to our unique set of aesthetic criteria for drawing UML class diagrams. It consists of 16 mandatory, 5 optional and 2 facultative user related criteria which are responsible for different aspects of readability of UML class diagrams. Finally, as a conclusion from our layout rules, in this section a comparison with other approaches to rules for aesthetic layout of UML class diagrams, 10 visual quality indicators to retrieve software design problems in UML class diagrams as well as issues of validating aesthetic rules will be given.

## 3.1 Requirements for UML Class Diagram Layout

There is no doubt that the first requirement for a composer is to be dead.

Arthur Honegger (1892 – 1955)

According to our roadmap for retrieving a functional specification for a layout algorithm for UML class diagrams, at this point of time we are able to collect a basic set of requirements taking into account the UML specification as well as the reasons for automatic layout and the foundations of graph drawing. The list of requirements will then be completed by a discussion on input formats and our set of aesthetic principles specific to UML class diagrams in the next

sections.

- **REQ\_COMPLETE\_UML:** According to the UML specification and the discussion in Section 2.1.3 our top-level goal is to develop an algorithm which respects all model elements which can be used in a UML class diagram. As stated in Section 2.1.1 and 2.1.3 we restrict ourselves to UML version 1.5.
- **REQ\_COMPLETE\_DIAGRAM:** Operations to reduce the visual complexity of a diagram to reduce also the complexity of the layout algorithm are seen as a part of the tool which invokes the layout mechanism. As pointed out in Section 2.1.3, such operations might be model-to-model transformations implemented as user commands as pre- or postprocessing operations before or after calling the layout algorithm.
- **REQ\_GRAPH\_TYPE:** The syntactic features of UML class diagrams influence the type of graph to be handled by the layout algorithm. As mentioned in Section 2.2.2, reflective associations induce self-loops and multiple edges may occur, so that graphs representing UML class diagrams may not be simple graphs. Some edges like generalizations are directed by definition, but e.g. associations are not always required to be directed. Therefore, we will operate on directed graphs, and as far as specified in the input, we try to use the semantic direction for the directions of the underlying edges. Hence, a direction is implicitly assigned to an undirected association but a directional indicator, like an arrow, will finally not be drawn. A graph that specifies model management elements like packages and contained classes or composite classes using the nested notation, is obviously a compound graph, because also relations between compounds like generalizations or dependencies are allowed by UML. If association classes or constraints are used, hyperedges connecting three nodes or, dependent on the implementation of the algorithm, simulated hyperedges may be part of the input graph. As depicted in Figure 2.9, elements outside any compound may occur also. Dependent on the implementation, the global namespace might be seen as the top level compound or *mixed compound graphs* can be considered. We will realize the latter alternative in this thesis, because handling compounds is usually time-consuming and a mixed processing might improve the runtime.  
In the following text, we will use the term “cluster” as a synonym for “compound” because intersections of compounds are not required for UML class diagrams and compounds therefore behave like clusters which then may act as start or end node of edges.
- **REQ\_HIERARCHY:** The default layout style in the UML specification suggests a kind of hierarchical layout where some edges are seen as hierarchical and other edges as non-hierarchical edges. Furthermore, this reflects the way of thinking of programmers and software engineers, because they are used to think in hierarchies to give their projects a certain structure like class, package, module or containment hierarchies. Therefore, we will rely on hierarchies to give the drawings of UML class diagrams a basic skeleton. A detailed discussion on various types of hierarchies will be given in Section 3.3.
- **REQ\_USER\_OPTIONS:** A layout algorithm should be as flexible as possible to be tailored to the user’s needs. Therefore, it should provide as many options as possible to give control

over the layout process to the user. But on the other side, the more options are provided to the user when invoking the layout mechanism of a CASE tool, the more time the user, who is usually not familiar with graph drawing issues and the concrete layout algorithm of the CASE tool, might need when searching for the correct layout options satisfying her needs respecting the concrete diagram. Consequently, an algorithm, which is capable of self configuration depending on the type of input, would be appreciated by real world users as noted in [Koschke 2003].

The basic idea seems to be understood in GoVisual, a layout tool for UML class diagrams, in which the algorithmic parameters are hidden behind an easy-access interface. Unfortunately this idea is then circumvented by providing five different layout styles and a variant of parameters that can be manipulated within each style [Gutwenger et al. 2003c].

Hence, the less layout decisions are left to the user, the better the integration into the CASE tool will be. Alternatively, the integration implementation might analyze the graph to be passed to the layout algorithm and select the appropriate layout options, or the layout algorithm itself is capable of sophisticated self-configuration features on a concrete input graph.

Of course, the opposite opinion is also known from literature: In [Henry and Hudson 1991] it is disbelieved that a single canonical layout algorithm will always produce the best results without interaction and customization by the user.

Two different types of options are relevant to UML class diagrams: Diagram specific options, like the type of hierarchy or even the individual edges to be considered and general options, like font faces, default distances or minimum extents.

On the one side, an interactive layout algorithm as well as a lot of options local to a certain diagram, have negative impact on the (automated) repeatability of the results in industrial applications and quality engineering of design documents. On the other side, there might be aspects improving the readability which cannot be specified in terms of a UML class diagram, like a certain sequence of nodes or the vicinity of nodes which cannot automatically be detected from looking on the specification of a diagram only. In some cases, e.g. to reflect dynamic aspects like the main execution sequence in the ordering of the elements of a class diagram, these facts might be deduced from other diagrams of different types. But any kind of such automatic deduction based on facts which might be changed independently from the diagram to be drawn, also may have a certain drawback on the stability of the mental map of the user. If such user defined options or constraints are stored along with the diagram, the repeatability of the layout result is not in danger. Therefore, we will also provide some user options towards interactive layout for features, where basic decisions may not always be left to the layout algorithm. A self-configuration dependent on the input will be deferred to future work.

- **REQ\_INCREMENTAL\_ALGORITHM:** A user working with a diagramming tool builds a so called mental map [Misue et al. 1995] of the diagram in mind. An automatic layout algorithm should respect that mental map and provide features for smoothly adapting a layout result, e.g., with respect to positional information in the input graph. Currently, this special layout feature called *incremental layout* is subject to different research projects. We

will provide basic support but we will defer a concrete mechanism for incremental layout to future work. Some ideas on the algorithmic issues to be considered when realizing this feature will be discussed at the end of this work in Section 7.1.

- **REQ\_GRID:** To support editing of class diagrams in a CASE tool, a snap-to-grid option which works on a user defined grid-size may be appropriate. In this case, the positions of all elements in the result graph are then fixed to positions  $(i_x \cdot g_x, i_y \cdot g_y)$  with  $i_x, i_y \in \mathbb{N}_0$  and  $g_x, g_y \in \mathbb{N}$  as grid sizes.
- **REQ\_SPEED:** Usually, an algorithm for automatic layout of UML class diagrams should support interactive application. On the one side, for incremental and interactive layout, a fast processing by the implementation of the algorithm and its integration is required. On the other side, the visual complexity of UML class diagrams induces a tradeoff with the speed of the algorithm. The more types of model elements and presentation options are supported by the algorithm, the more complex the algorithm is and the slower it reacts. In general, algorithms for the visualization of diagrams or graphs imply at least high complexity or belong to the sets of NP-hard or NP-complete problems. In particular, to draw a UML class diagram, several basic problems like proper hierarchization or crossing minimization allocate exponential runtime to obtain exact solutions. Therefore, heuristics will be investigated to reduce the complexity and to calculate amenable results. Hence, we are interested in algorithms of low theoretic complexity but high quality. Also on a prototypical implementation of our layout algorithm we are interested in fast execution and tuning of the implementation. Due to the prototypical character of the program, execution speed appears as an interesting goal of lower priority.

For a concrete implementation, the following requirements should be considered:

- **REQ\_DETERMINISTIC\_ALGORITHM:** As described in Section 2.2.2, some layout algorithms rely on random initial configurations or randomly select nodes to be processed at a time. Imagine such an algorithm which, each time the algorithm is invoked, produces for the same input graph a different result. In other words, a kind of layout animation would be produced, when the layout button of the CASE tool is pressed in sequence. Real-world examples were shown in [Eichelberger 2002b]. We are of the opinion that, to prevent the described behavior, a layout calculation used within our application domain, should rely on deterministic and repeatable decisions within the algorithm only.
- **REQ\_ARCHITECTURE:** Parts of the implementation should be reusable, exchangeable and across different stages of the algorithm as few as possible dependencies should exist. This seems to induce a component-like plug-in based implementation. We will discuss the architecture of the implementation in Section 6.1.
- **REQ\_IO:** The algorithm itself should receive input from standard formats as well as from easy-to-debug formats. The output produced, should be compliant to standard formats as well. CASE tools usually do not rely on files stored in the file system but on database-like repositories capturing more data on the design than shown in UML diagrams. A direct

access to the repository of a CASE tool would reduce the integration effort into individual tools, especially, if a common repository access interface is supported by the CASE tool vendors. Further issues on formats for UML class diagrams will be given in Section 3.2.

- **REQ\_PLATFORM**: The implementation of the algorithm should be executable on as much computing platforms as possible with as less effort for platform specific modifications as necessary. This will influence the programming language and the libraries to be used for an implementation and will be addressed in Section 6.1.

We can conclude that none of the graph drawing approaches mentioned in Section 2.2.2 directly meets the basic set of requirements and therefore none of these algorithms can directly be applied to UML class diagrams. Beside the fact that most of these approaches are not able to handle non-simple compound graphs with different types of edges, none of them considers the semantics of the input when calculating the layout [Fleischer and Hirsch 2001; Purchase et al. 2001b].

## 3.2 Input and Output

It is a capital mistake to theorize before one has data.

Sherlock Holmes

In the last section, we have mentioned REQ\_IO as one of our basic requirements. In this section, data formats, which may be used to describe UML class diagrams, will be listed. Thereby, we will discuss the advantages and disadvantages of the individual formats and select appropriate ones to be realized by a concrete implementation to meet REQ\_IO.

- **CDIF** (CASE Data Interchange Format) is a vendor-independent, method independent family of languages designed for the exchange between modeling tools. It was published in 1991, revised in 1994 and finally standardized by ISO/IEC. It was one of the archetypes for specifying the UML and there are still some mappings between CDIF and ancient versions of UML. The latest news on the CDIF homepage<sup>1</sup> were published in January 1998 and to our knowledge, most of the UML tool vendors decided to support different formats, but, according to the publication dates of the last specification documents at ISO/IEC and some research work like [Schauer and Keller 1998], it seems that CDIF is still in use.
- **GXL** (Graph eXchange Language) described in [Winter 2002] is a sublanguage of XML (Extensible Markup Language), a simple and flexible text format specified by the W3C (World Wide Web Consortium)<sup>2</sup>. Designed as a standard exchange format for graphs, GXL provides so called schemas to tailor GXL towards specific application domains. Furthermore, GTXL (Graph Transformation Exchange Language) was specified as a separate project along with GXL. For example, as mentioned in [Koschke 2003], the reverse engineering and re-engineering community adopted GXL as a standard interchange format.

<sup>1</sup><http://www.eigroup.org/cdif/index.html>, <http://www.cdif.org> is not available anymore

<sup>2</sup><http://www.w3.org>

Even if the specification of GXL uses UML class diagrams to describe GXL itself and its schemas, no official schema, which supports UML class diagrams especially on semantical level, was specified so far as regretted in [Reiniger 2003].

- **GraphML** [Brandes et al. 2002], also a XML dialect, is an extendible format for specifying graph structures tailored towards the needs of the graph drawing community. GraphML might also be used for specifying the structure of UML class diagrams, because it is a common graph format, but it is not designed to also provide semantical information as noted in [Reiniger 2003].
- **XMI** [OMG 2002] (XML Metadata Interchange), first specified in 1999, is the common metadata and metamodel interchange format specified by OMG. Basically, XMI is defined along with MOF and, because UML is, dependent on the version of UML, an alignment or an instance of MOF, respectively, XMI can be used to represent UML modeling information. As shown in Table 3.1, MOF, placed at the top of the model level stack, defines the

M <sub>3</sub>	meta-meta-model	MOF	XMI
M <sub>2</sub>	meta-model	UML	UML-XMI in terms of XMI
M <sub>1</sub>	model	UML diagrams	user diagrams in terms of UML-XMI
M <sub>0</sub>	instance level	realworld user objects	—

Table 3.1: Model levels of UML and XMI.

basic constructs of a modeling language.

MOF can then be used to describe the elements and relations of a concrete modeling language, in our case the UML. Basically, XMI is a XML based description of a meta-model in XML. More specific, XMI can be used to store the XML serialization of the MOF instances needed to describe the UML. Based on this meta-model, UML diagrams are then seen as instances of the M<sub>2</sub> model. Hence, UML diagrams can also be represented in XMI in terms of the XMI description of the UML metamodel.

On the one side, UML-XMI provides detailed information on the diagram elements. Usually, due to the alignment with the metamodel, more information, like stereotype or datatype definitions, than to be displayed is available. On the other side, also because of the alignment to the UML meta model, the relational structures and the additional information across the model levels lead to a high verbosity of UML-XMI.

- **UMLscript** is a programming language for object-oriented design. Similar to the declarative modeling approach described in [Spinellis 2003], in which a Java-like notation was advocated, UMLscript is designed as a programming language for class diagrams. For handicapped users, especially blind users, a textual notation provides different advantages over graphical notations and editors, where complex combinations on different input devices are required to specify a diagram.

UMLscript was first presented in [Seemann and von Gudenberg 1998], an adaption to more recent UML versions was given in [Eichelberger and von Gudenberg 2001]. It was

never intended that UMLscript should be used to describe other diagrams than class diagrams. More specialized versions like UMLscript-rt, the real-time version for sequence diagrams, ought to be used instead. Furthermore, from the viewpoint of compiler construction, UMLscript is a simple language (LL1) but it requires some effort to be implemented, because it admits a non-linear description (no fixed sequence of elements, opportunity of forward references and nested elements).

UMLscript can automatically be created from source code using *JTransform*<sup>3</sup> [Eichelberger and von Gudenberg 2004] or *Ptidej*<sup>4</sup> [Guéhéneuc 2003].

- To be more up to date, **XUMLscript** was developed as another XML dialect for UML class diagrams. In fact, it can be described as a XML version of UMLscript without being such verbose than UML-XMI. The structure of XUMLscript as well as a description of a transformation from class diagrams in UML-XMI to XUMLscript using XSLT and XPath were given in [Reiniger 2003]. Dependent on the content, the size of a XUMLscript file is less than 10% of the size of the corresponding UML-XMI file, and using UMLscript the size of the XUMLscript code can furthermore be reduced by 50%, because in both non-XMI formats only structural information relevant to the diagram is recorded.
- As long only structural, non-layout specific information is stored in an input file, the exchange of layout related data is left to proprietary formats and protocols. After the first version of UML-XMI has been specified, most tool vendors tried to be compliant to UML-XMI. But to retrieve the entire diagram information from exported and reimported data of the same tool, most times the XMI format was extended in a vendor specific way to store layout data, font sizes, etc. It was time to specify a format for also supporting the diagram interchange, because the additional proprietary information increased significantly the complexity of the interchange of diagram data between tools of different vendors. The logical consequence was **XMI[DI]** [OMG 2003b], a diagramming extension of the UML-XMI format which is currently being specified by the OMG. To the model data in XMI, an additional diagram section is attached. Each diagram element, specified in terms of graph elements and coordinates relative to the enclosing element, is backwards related by a so called model bridge into the XMI data.

By now we have to decide, which formats should be considered for input or output in a concrete implementation. Realizing an input or output filter if the format is not directly supported by the programming language or a library, requires a lot of time for implementation and testing as well as for maintaining it with respect to future versions of UML.

Because CDIF is rarely supported by current tools, GXL does not come along with a commonly accepted schema for UML class diagrams and GraphML is a graph structure format without semantical support for our application domain, we will not take these three formats into account. UMLscript was proposed in 1998 and the first version of XMI was published in 1999, the year when the first version of *SugiBib* was implemented. We deferred an implementation into

<sup>3</sup><http://jtransform.sourceforge.net/>

<sup>4</sup><http://www.yann-gael.gueheneuc.net/Work/Research/Ptidej/Download/>



future and made a decision for UMLscript, because in 1999 it was expected that UML and XMI, will change significantly in the next years. As (for us) it is much easier to specify class diagrams in UMLscript as in any of the languages mentioned above, UMLscript is still today the main debugging input format for *SugiBib*. With the decision to test an XML based variant of UMLscript, XUMLscript was implemented in 2003. Unfortunately, due to the vendor specific problems with XMI, the import of XMI usually still needs vendor dependent preprocessing. We are convinced that, with the advent of a commonly agreed diagram interchange format based on XMI, like XMI[DI], future UML tool implementations will provide a better compliance to the standard.

Even if we sometimes were in need to directly support XMI, we are convinced that deferring the implementation was a good decision, because in the last years, different approaches to generically bridging the gap between XMI and programming languages have been undertaken. One approach is JMI (Java Metadata Interface) [SUN JCP 2003]: A set of interfaces basically define the MOF structures ( $M_3$ ) in Java and a plug-in mechanism allows to attach the implementation of a concrete metamodel. At a first glance, JMI is only capable of defining a meta-model ( $M_2$ ) in terms of MOF, to maintain it using Java, to load and to store it via a XMI reader or writer, respectively. A concrete meta-model description for JMI, given in XMI, may contain references to concrete implementation classes attached as a plug-in to JMI (reflection classes). In other words, after loading the UML meta-model description into JMI and obtaining a repository extent, in which user models ( $M_1$ ) can be maintained, plug-in specific classes can be used to create, explore and modify user models. But also data exchange via XMI[DI] can simply be realized using this mechanism. Instead of a UML meta-model, a DI enhanced UML meta-model is loaded and a DI specific implementation of the JMI reflection classes is used as plug-in. Now, UML diagrams can be specified in terms of UML meta-model elements and DI graph elements, while JMI automatically handles XMI as well as JMI-compliant repository access. In fact, deferring the implementation was a good decision, because, using a general metadata repository with an appropriate plug-in, we are able to handle XMI, XMI[DI] and we can get direct access to compliant repositories using an external library. This also will support and simplify the interaction and integration with other tools.

Dependent on the programming language and available libraries, image and printing formats like PNG (Portable Network Graphics), JPEG (Joint Pictures Expert Group), TIFF (Tagged Image File Format) or EPS (Encapsulated Postscript) can be considered for output, too.

As a conclusion, an architecture implementing our work, requires, according to REQ\_ARCHITECTURE, a plug-in mechanism for input as well as for output formats. We will provide plug-ins for UMLscript, XUMLscript, XMI via XSLT and XUMLscript, XMI and XMI[DI] as files via a metadata repository and direct metadata repository access as input features. Beside the image and printing formats mentioned above, we will consider XMI[DI] as an output format. XUMLscript was designed to support extensions for layout data but we will not implement that feature, neither as input nor as output option. XMI as output format is implied in the export of XMI[DI] simply by preventing the creation of DI specific elements while interacting with the metadata repository.

## 3.3 Aesthetics

When diagrams are used to represent a piece of reality, the main quality desired for them is readability, where we say that a diagram is *readable* if its meaning is easily captured by the way it is drawn.

[Tamassia et al. 1988]

It is known that a picture is able to convey some thousand words. Therefore, it is necessary to gain readability of a picture or a drawing to understand the information depicted. This obvious rule is known throughout the disciplines dealing with diagrams. Especially when graphs are drawn automatically, clear rules should restrict the degrees of freedom in placing the nodes and edges. Thereby, the main problem is to characterize the quality of a diagram: A good diagram is often identified as a diagram which supports readability, the capability of clearly communicating information about the conceptual structure, as stated in [Tamassia 1985; Sugiyama 2002].

For drawing abstract graphs, various rules have been published in the literature. Unfortunately, for UML class diagrams there are currently no commonly agreed criteria to describe the features necessary for readability. Some formulae to objectively measure criteria known from graph drawing have been published in [Ware et al. 2002; Purchase 2004] after conducting perceptual experiments. Within the different communities dealing with diagrams, several individual features or aesthetic rules are known and accepted, but in our case there is few interaction across the involved communities.

In [Batini et al. 1985; Fleischer and Hirsch 2001] it was observed that automatic layout tools often adopt fixed weights in solving tradeoffs between aesthetic rules, while human beings with intimate knowledge on the semantics of the diagrams adapt their perception dependent on the application domain. The problem is to find a set of rules describing as close as possible the behavior of a human engineer while drawing manually a readable UML class diagram. As a course of action, the following two steps were proposed in [Batini et al. 1985]: Enumerate as many layout criteria as possible and then determine the preferences of designers for solving the conflicts between such criteria.

To derive the most important aesthetic principles for our target application domain, we will follow that procedure. Therefore, we will first clarify the terminology in Section 3.3.1 and will then summarize well-known criteria from several disciplines in Section 3.3.2 up to 3.3.5: We will look at rules from graph drawing which are in use for several years, we will consider HCI (Human Computer Interaction) for information on human perception, we will give an overview on common principles of good software design to find design indicators that map into diagrams and we will take software visualization into account. Combining this with the specification of UML class diagrams, we will then compose the knowledge accumulated so far to UML specific rules in Section 3.3.6.

### 3.3.1 Terminology

The term *aesthetics* is used for criteria that concern a diagram's readability from the graphical point of view.

[Nummenmaa and Tuomi 1990]

According to [Batini et al. 1985], the *readability of a diagram* is closely related to the ability of a layout of a graph to convey clear information about the structure of the associated conceptual graph. Two different types of readability can be considered:

- Conceptual readability that concerns the structural properties of the schema, independently of its graphic description.  
The associated example in [Batini et al. 1985] shows that the structure in combination with the semantics of the elements may admit a (vertical) hierarchy, while from the graphical viewpoint, e.g. area usage, horizontal flows are a better choice.
- Graphics readability that concerns the layout of the diagram.

Activities to generate readable diagrams can similarly be partitioned into

- Decomposition, how realworld knowledge is decomposed into graphical primitives.
- Layout, the positioning of the graphical primitives on the presentation space.

These two activities or similarly the two different types of readability have been evaluated in [Hahn and Kim 1999] considering sequence diagrams, activity diagrams and collaboration diagrams according to UML version 1.1 [OMG 1997]. Thereby, decomposition as well as layout organization decreased the average number of analysis and design errors.

Therefore, aesthetic criteria should capture these different types or activities and should provide clear rules to be respected by a layout algorithm. In our case, conceptual readability relies on the contents to be displayed, i.e., on the UML specification and on the design of the software artifact. For graphic readability, we are convinced that rules primarily designed to describe the beauty of a diagram independently from the semantics of the diagram and its individual elements can be combined with further knowledge on the underlying semantics to build a basic set of common aesthetic criteria.

More specifically, in [Tamassia et al. 1988], aesthetics are described as common layout rules that are not specific to an application, while layout constraints denote application specific rules to be fulfilled. Common layout rules, as described in the next section, as well as general constraints may conflict and selecting the most relevant one to gain a readable layout is a difficult task.

Especially for terms of graph drawing, we adhere to the following taxonomy composed from literature:

- **Static rules:** When a graph is laid out once without respecting a history of changes [Sugiyama 2002] the following types of rules may be respected.

- Syntactic rules [Coleman and Parker 1996], structural rules [Sugiyama 2002], aesthetic features [Batini et al. 1985] or general layout rules [Tamassia 1985] only rely on the basic visual principles and the structure of the graph
- Graphical standards [Tamassia 1985] as listed in Section 2.2.2, e.g. GS\_STRAIGHTLINE.
- Semantic constraints [Batini et al. 1985] or semantic rules [Coleman and Parker 1996] are related to the meaning of the elements of the graph. According to [Sugiyama 2002, p. 12] semantic rules, like the importance of a node or the strength of the relationship of an edge, can be automatically derived or be given by the user. In [Tamassia 1985], these rules are simply described as rules specific to the particular class of diagrams.

In our case, semantic rules may also be given by the application domain and can partly be deduced from the information attached to the elements of the graph.

- **Dynamic rules:** Rules to be respected, when a graph is modified over the time [Coleman and Parker 1996]. Therefore an algorithm is applied to a sequence of drawings [Sugiyama 2002]. These rules are useful for the restructuring of diagrams in the case of local changes [Tamassia 1985] or to ensure the relative positional relationship between diagram elements to preserve the human mental map and the continuity of cognition [Sugiyama 2002, p. 11]. Rules for incremental layout are difficult to integrate into basic graph layout algorithms and are currently topic of further research.

In this work, we will mainly focus on drawing a class diagram once without taking dynamic rules into account. Therefore, we will discuss exclusively syntactic and semantic rules as well as graphical standards in the following.

Dependent on the implementation of the layout algorithm, common layout rules and constraints might be implemented implicitly, e.g., as force formulae in force-directed algorithms or in a 2-layered implementation: A basic layout algorithm considering or ensuring some rules and a constraint solving mechanism dealing with dynamic or user defined constraints. While the term “constraint”, e.g., in UML, is a rule which always has to be fulfilled, a “constraint” in conjunction with a constraint solving mechanism suggests that if a constraint conflicts with other constraints, some are ignored and one is selected, e.g., based on some priority mechanism.

For UML layout aesthetics, which are semantically essential for the readability and the quality of the layout, we will refer to aesthetic rules, criteria or principles, which always have to be fulfilled and we will not consider the term “constraint” for these essential issues.

Furthermore, aesthetic rules can be classified according to their impact on the layout algorithm [Batini et al. 1985]. A layout rule may:

- Be local when it refers only to a part of the diagram or global otherwise.
- Be hierarchical when it concerns the relative position of a set of symbols, flat otherwise.
- Influence the topology, shape or metric of the diagram.

In [Sugiyama 2002], conventions and rules for graph drawing algorithms were related by priorities. Surprisingly, compared with most work in graph drawing, below placement conditions, routing conventions and drawing conventions, semantic rules were given a higher priority than structural rules.

### 3.3.2 Traditional Graph Drawing Aesthetics

In the development of graph drawing algorithms it is necessary to set *priorities* between convention and rules.

[Sugiyama 2002]

When developing a layout algorithm for UML class diagrams, basic approaches and experiences from graph drawing should be taken into account instead of designing new algorithms from scratch. Various advantages and disadvantages described in literature help avoiding inappropriate ideas and disappointing trials. Beside algorithmic descriptions, information on runtime complexity and memory usage, algorithms are often characterized enumerating the aesthetic criteria fulfilled or partially considered by an algorithm, respectively.

The set of traditional graph drawing aesthetics has emerged over a lot of years. Unfortunately, in the most recent publications, aesthetic criteria are often listed only without reasoning about the concrete impact on the readability.

#### Structural Rules

First we list the groups of rules which rely on the structural aspects of a graph only. Additionally, we give references to the literature, but we will mention only the first publication known to us. The listing is ordered according to the number of occurrences in the literature.

- **GDR\_EDGE\_CROSS**: Minimize or at least avoid edge crossings. The more edge crossings are present in a diagram, the harder is the task of the human eye to find out which nodes are connected. Unfortunately, only planar graphs or graphs in 3D can be laid out without edge crossings<sup>5</sup>. If hierarchical structures are present, minimize the crossings between them [Batini et al. 1985].
- **GDR\_OVERLAP**: Nodes should not overlap edges [Sander 1996a], other nodes [Sander 1996a] or other base nodes of clusters [Sander 1996b]. Additionally, edges should not overlap other edges.
- **GDR\_MIN\_BENDS**: Minimize or at least avoid bends (sharp corners) in edges [Sugiyama et al. 1981]. It is much easier to follow straight lines or orthogonal edge chains than chains wildly alternating the direction of the individual edges. This criterion relies on the underlying graphics standard. In **GS\_ORTHOGONAL**, bends are implicitly present but the more bends the harder edges between widespread nodes can be followed and the harder it is to

---

<sup>5</sup>This criterion was mentioned most times (24 occurrences) in literature

read the entire diagram. Alternatively, the variance of the number of bends can be minimized [Battista et al. 1999].

- GDR\_UNIFORM\_LENGTHS: Keep edge lengths uniform [Fruchterman and Reingold 1991].
- GDR\_MIN\_EDGES: Minimize
  - The average length of connections [Sugiyama 2002] to ensure that the edge lengths are short [Gansner et al. 1988] but not too short [Sugiyama et al. 1981].
  - The maximum length of an edge or the longest edge [Batini et al. 1985].
  - The global length of the edges [Tamassia 1985].
  - The difference in edge lengths [Sugiyama 2002]. This criterion relates to GDR\_UNIFORM\_LENGTHS.

Short edges prevent associated names to be properly positioned on an edge and may lead to excessive clustering, while long edges, in particular such with excessive turns, can complicate the perception of the flow as remarked in [Protsko et al. 1991].

- GDR\_DENSITY: As mentioned in [Batini et al. 1985], disomogeneous density leads to double observation, a discontinuity in the visual perception process. Therefore,
  - Place vertices on the boundary with uniform density [Sugiyama 2002].
  - Support uniform density of the placement and the routing [Batini et al. 1985].
  - Ensure uniform density of symbols along the boundary of the form [Batini et al. 1985].
- GDR\_NODE\_DISTRIBUTION:
  - Distribute vertices uniformly [Henry and Hudson 1991].
  - Nodes should be within a bounding box [Fruchterman and Reingold 1991] and more specifically, nodes should be distributed evenly within a bounding box, the page or the drawing area [Eades and Sugiyama 1990].
  - Nodes should not be laid out too close together and not too far apart [Coleman and Parker 1996] especially nodes connected to each other should be laid out as close as possible [Fruchterman and Reingold 1991].
- GDR\_FLOW: Support a consistent direction of edge flow [Eades and Sugiyama 1990].
- GDR\_EDGE\_DIRECTIONS: Different types of edges should be drawn with different specified directions [Sugiyama and Misue 1995]. If the different directions are chosen consistently, this might not conflict with GDR\_FLOW. However, this criterion seems not always to be appropriate as a structural rule, because specified directions may relate to the semantics of the diagram or at least to user preferences.

- **GDR\_ORTHOGONALITY:** Fix nodes and edges to an orthogonal grid [Tamassia 1987]. Obviously this criterion might be responsible for creating lengthy edges and therefore it might conflict with **GDR\_MIN\_EDGES**, **GDR\_DENSITY** and **GDR\_NODE\_DISTRIBUTION**.
- **GDR\_SYMMETRY:** Where possible, a symmetrical view of the graph should be displayed to simplify the recognition of similar or identical subgraphs [Eades and Sugiyama 1990], but most times the term symmetry is not further specified. In [Bachl and Brandenburg 2002], geometric symmetry is described as a drawing with a rotational or a reflectional invariant while the graph theoretic symmetry requires a non-trivial automorphism in the graph. The graph theoretic symmetry provides more flexibility and should be considered for subgraphs, because most graphs only have a trivial automorphism. Different types of rules considering symmetries are mentioned in literature:
  - Inherent symmetry should be reflected by a layout [Sugiyama et al. 1981].
  - Maximize local (subgraph) isomorphisms, axial and rotational symmetries [Tamassia 1998].
  - Balance the diagram with respect to the vertical or horizontal axis [Tamassia et al. 1988].
  - For trees and a left-to-right layout: Parent nodes should be placed to the left of their children [Coleman and Parker 1996].
  - For trees and a top-to-bottom layout: Parent nodes should be centered over children [Wetherell and Shannon 1979] or, more generally, the layout of children should be symmetrical [Batini et al. 1985].
- **GDR\_DRAWING\_SIZE:** The less large a drawing is and the more homogenous the nodes and edges are distributed, the better a drawing looks. This rule relates to **GDR\_DENSITY**.
  - The physical width of the drawing should be minimized [Coleman and Parker 1996].
  - The area of the drawing should be minimized [Batini et al. 1985].
  - Minimize the aspect ratio of the drawing, which is defined as the ratio of the length of the longest side to the length of the shortest side of the smallest rectangle with horizontal and vertical sides covering the drawing [Tamassia 1998].
  - The aspect ratio should be balanced [Sugiyama 2002].
  - The area of a geometric figure with given shape covering the diagram should be minimized [Batini et al. 1985].
- **GDR\_LABELS\_DIRECTION:** All text labels should be horizontal, rather than a mixture of horizontal and vertical [Purchase et al. 2001a].
- **GDR\_FONTS:** All text fonts should be the same rather than using different fonts for different types of labels [Purchase et al. 2001a]. Of course, the application of this rule depends

on the kind of diagram to be drawn. The UML describes font faces as a basic style guide, e.g., the font face for a class name should be bold, italics should be used for abstract classes or operations.

- **GDR\_ANGLE**: Maximize the smallest angle between two edges incident on the same vertex. This aesthetic is especially relevant for **GS\_STRAIGHTLINE** [Battista et al. 1999]. The less the physical resolution of a screen, the more it is important that edges are as far apart as possible. The angle between incident edges should not be too small [Coleman and Parker 1996].
- **GDR\_HIERARCHY**: If a hierarchical structure is present, the layout should expose hierarchical structure in some way [Sugiyama et al. 1981].
  - Hierarchical structures should be drawn vertically [Batini et al. 1985]. For example, on trees parents must be placed above their sons [Tamassia 1985].
  - Hierarchical structures should be drawn horizontally [Sugiyama 2002]. For example, on trees nodes on the same level should have the same vertical position [Wetherell and Shannon 1979].
  - Node positions are restricted to distinct layers [Fleischer and Hirsch 2001].
- **GDR\_NODES\_EDGES**: Nodes should not be placed too close to edges [Coleman and Parker 1996].
- **GDR\_ORDEREDTREE**: For ordered trees, the layouts should preserve the node ordering [Wetherell and Shannon 1979] and the layout for a tree and its reverse should be mirror images [Reingold and Tilford 1981].
- **GDR\_DIMENSIONS\_SYMBOL**: Minimize the difference between symbol dimensions [Batini et al. 1985].
- **GDR\_DIAGRAM\_CENTER**: Nodes should be placed near the center of a bounding box [Coleman and Parker 1996], objects with maximum or high degree should be placed in or near the center of the diagram [Batini et al. 1985].
- **GDR\_CLUSTERING**: Depending on the contents and the application domain of the graph, the graph's structure can be exposed if clusters are introduced [Fleischer and Hirsch 2001].
  - Clusters should be drawn as convex regions [Feng 1997], a border rectangle of a subgraph contains exactly the base nodes and the rectangles of subgraphs and border rectangles of two nonnested subgraphs should not overlap [Sander 1996b].
  - Connectivity edges may cross border lines of clusters but such crossings should be avoided [Sander 1996b].
- **GDR\_FACES**: The number of faces drawn as convex polygons is maximized [Tamassia et al. 1988].



- **GDR\_NODE\_SIZE**: Minimize differences among vertex dimensions [Tamassia et al. 1988].
- **GDR\_BALANCE\_EDGES**: Edges terminating on and originating from a vertex should be laid out in a balanced form [Sugiyama and Misue 1991].
- **GDR\_MANHATTAN**: Enforce the orthogonal layout of edges [Sander 1996a]: A horizontal segment of one edge and a vertical segment of another edge may share a point (legal kind of edge crossing), two horizontal segments of different edges never share a point, but they may share a subsegment, if they also share the vertical segment adjacent to the subsegment. Vertical segments of different edges may share subsegments, if and only if the segments are adjacent to a node, horizontal segments should have a minimal vertical distance.
- **GDR\_CONTOUR\_DIFFERENCE**: Maximize the difference between distances of lines which are contours of symbols and connections [Batini et al. 1985]. Minimize the length of contours of vertices and the length of edges in between [Sugiyama 2002].

### Semantic Constraints

As identified in Section 3.3.1, constraints in graph drawing are related to the meaning of the elements, while structural rules are defined without considering the semantics. In the following we will present some popular layout constraints. Some of them will be used as basic ideas for the deduction of UML specific rules in Section 3.3.6. As for structural rules, we will give only the first occurrences in literature known to us as references.

- **GDC\_NODE\_SYMBOLS**: Assign the dimensions of the symbols representing specified vertices [Tamassia et al. 1988]. This constraint might conflict with **GDR\_NODE\_SIZE**.
- **GDC\_IMPORTANCE**: The dimension of the objects should be chosen according to the importance of the objects, or more specifically, the greatest dimension is appropriate for the most important objects [Batini et al. 1985]. This constraint might conflict with **GDR\_NODE\_SIZE** and **GDC\_NODE\_SYMBOLS**.
- **GDC\_GROUP\_CURVE**: A group of objects is displayed on a straight line [Batini et al. 1985] or more generally on a specified curve [Sugiyama 2002] or on a stream established by an external semantic feature (e.g. flow of time) [Batini et al. 1985]. These objects might additionally be ordered as a sequence [Sugiyama 2002].
- **GDC\_GROUP**: Place close together a group of vertices according to some specified property [Tamassia et al. 1988].
- **GDC\_GROUP\_CENTER**: A given group of objects/symbols should be placed in the center of the diagram [Batini et al. 1985]. Some papers mention this constraint as a structural rule similar to **GDR\_DIAGRAM\_CENTER**.

- **GDC\_BALANCE\_DIAGRAM**: Balance the whole diagram with respect to a vertical or horizontal axis [Batini et al. 1985]. In some papers this constraint is given as a structural rule similar to **GDR\_SYMMETRY**.
- **GDC\_BOUNDARY**: A selected group of symbols should be placed in the external boundary of the diagram [Batini et al. 1985].
- **GDC\_SHAPE**: Draw a subgraph with a predetermined shape [Tamassia 1985].
- **GDC\_LIMIT\_CROSSINGS**: An upper limit to the number of edge crossings is specified [Sugiyama 2002].
- **GDC\_LIMIT\_BENDS**: An upper limit to the number of edge bends is specified [Sugiyama 2002].
- **GDC\_LIMIT\_LENGTH**: The lengths of specified edges have a given upper limit [Sugiyama 2002].

As a conclusion of this large set of 26 structural rules and 11 semantic constraints from graph drawing we can state that some rules are not applicable to all graphs, some rules exclude others and some interfere with others. Furthermore, the more rules should be fulfilled in one drawing, the higher is the probability that contradictions occur, e.g., **GDR\_HIERARCHY** can not fully be realized on a cyclic graph. By specifying the situations in which certain rules may be relaxed or preceded by other rules, conflicts can deterministically be avoided. For a concrete class of graphs, priorities and exceptions for certain situations on the aesthetic rules may be introduced. This can be implemented by the processing sequence of the layout algorithm, either by a fixed sequence or by a dynamic mechanism like a constraint solver.

In fact, except for the rules mentioned in [Batini et al. 1985], most of the rules listed above have been taken as axiomatic and have not been empirically tested as criticized in [Ware et al. 2002]. Unfortunately, most of the aesthetic criteria published in graph drawing so far relate to general graphs and most authors have in mind that edges are simply lines and nodes have point size. Therefore, usually these rules cannot directly be adopted to domain-specific applications and further rules to be identified, evaluated and validated [Purchase et al. 2001b]. In literature on empirical studies on layout aesthetics for UML class diagrams it was regretted that no domain-specific semantical rules and layout algorithms exist:

We believe that there are additional semantic issues that need to be considered when a layout algorithm is used in a domain-specific tool.

Automatic graph layout algorithms typically do not take the semantics of the diagram into account. As we wished our results to relate to the design of such algorithms, we did not consider the semantics of the diagrams when we created them according to the layout aesthetics.

[Purchase et al. 2001b]

Hence, it makes sense to collect the knowledge published so far, to combine it with ideas from other disciplines and to adapt, configure and aggregate them to more specific semantic criteria more appropriate for UML class diagrams.










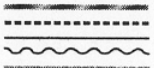
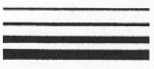

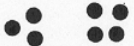
Graphical Code	Visual Instantiation	Semantics
1. Closed contour.		Entity, object, node.
2. Shape of closed region.		Entity type.
3. Color of enclosed region.		Entity type.
4. Size of enclosed region.		Entity value. Larger = more.
5. Partitioning lines within enclosed region.		Entity partitions are created, e.g., TreeMaps.
6. Attached shapes.		Attached entities. Part_of relations.
7. Shapes enclosed by contour.		Contained entities.
8. Spatially ordered shapes.		A sequence.
9. Linking line.		Relationship between entities.
10. Linking-line quality.		Type of relationship between entities.
11. Linking-line thickness.		Strength of relationship between entities.
12. Tab connector.		A fit between components.
13. Proximity.		Groups of components.

Figure 3.1: The visual grammar of diagram elements in node-link diagrams (from [Ware 2000, p.226]).

### 3.3.3 Human Computer Interaction and Cognitive Psychology

We have only borrowed a small part from the large pool of literature on graph layout since we feel that much of this literature is irrelevant to the most crucial issue in graph layout, namely the problem of semantic clustering. The most important criterion for information layout will usually be the *meaning* of the nodes and arcs and since most work on layout largely ignores this factor, we do not find it useful.

[Ware et al. 1993]

A lot of diagrams like data flow diagrams, organization charts, software structure and modeling diagrams consist of various kinds of entities and links, representing relationships between the entities. In [Ware 2000], this very large class of diagrams, which can be described by a visual grammar, was called *node-link diagrams*. The entity-relationship model, a precursor of the UML, is the most general data model that is expressed by node-link diagrams.

A visual grammar or the grammar of a visual language for diagrammatic representations is a set of rules translating relevant information into a diagrammatic representation. These rules restrict the (complex) decisions to be made to construct a valid diagrammatic representation [Hahn and Kim 1999]. Such a diagram convention tells the reader, e.g., how contours (lines) have to be interpreted.

Concluding from the visual grammar of diagram elements for node-link diagrams shown in Figure 3.1, the UML notation lacks in different common graphical codes. Neither the use of colors, which refers to the entity type, nor the size of the elements, which refers to the magnitude of the entity are specified in UML.

As mentioned in [Eichelberger 2003], partitions within enclosed regions, which obviously refer to entity partitions, e.g., in packages, are present at subsystems only. Shapes enclosed by contour introduce multiple confusingly meanings: packages in packages or classes in packages are contained, classes in classes are composed and inner or nested classes which are usually structurally enclosed are attached by the so called anchor notation (which is defined for packages as well). Furthermore, the thickness of links usually describes the strength of connections and neither the meaning of proximity nor spatial ordering of model elements are defined in UML.

In Section 2.1.4, we discussed some alternative notations which are more appropriate than the UML for some applications. A more pragmatic but also proprietary approach is to apply the Geon Theory developed by Biederman and Hummel to UML class diagrams. The Geon Theory aggregates a diagram throughout different layers: Edges from an input image are extracted, these edges are interrelated by vertices, axes and blobs, geon attributes (aspect ratio, horizontal and vertical position, size) are attached to the elements, relations like relative orientation or relative size are then assigned and the individual geons are assembled by respecting external interconnection information. The geon diagram in Figure 3.3 is the direct transformation of the class diagram shown in Figure 3.2 into geon space. Geon diagrams tend to enforce proximity, similarity and closure among their graphical elements and therefore combine Gestalt theory with the principles of the grammar of node-link diagrams shown in Figure 3.1. In a comparison between UML diagrams and the geon equivalent, participants performed faster and better in identification

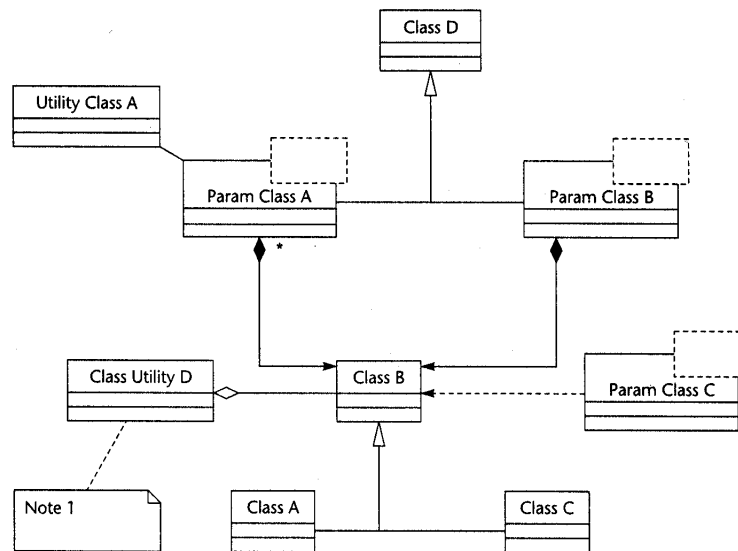


Figure 3.2: A UML class diagram to be represented as a geon diagram in Figure 3.3 (from [Ware 2000, p. 256]).

of substructures in geon diagrams and geon diagrams were easier to remember [Ware 2000, p. 255].

The framework of cognitive dimensions [Green and Blackwell 1998] intended to be applied to graphical user interface artifacts, can also be applied to tools for UML and to the UML itself as described in [Eichelberger 2003]. In the following, the mapping between cognitive dimensions printed as italicized terms and UML or tool issues, respectively, will be enumerated: *Viscosity* (resistance to local change) and *premature commitment* (based on early decisions in hand-crafted layouts) directly map to CASE tools and can be improved by good automatic layout algorithms. The *abstraction level* (grouping of elements), *consistency* and *role-expressiveness* map to UML itself, *error-proneness* and *hard mental operations* in interpreting diagrams can be reduced by further standardization also of layout rules and improved versions of UML. The experience of the software engineer in using UML and the viewpoint to be described by the diagrams affect *hidden dependencies* which also should be managed by tools, especially because some of these dependencies might be expressed as invisible hyperlinks specified in UML. *Closeness of mapping* and *diffuseness/terseness* are subject to the individual design as well as subject of discussions about the notation itself.

As mentioned earlier, our main goal is to improve the diagrammatic notation of UML class diagrams, but we do neither want to introduce another modeling framework nor a different kind of notation. Beside that the UML is fast becoming an international de-jure standard [Kobryn 1999], introducing changed notations into a standardized and widely accepted method increases *error-proneness*, the number of *mental operations* and influences *consistency*. Therefore *secondary notation and escape from formalism* like introducing colors or respecting extensions to realize the common graphical codes for node-link-diagrams may lead to confusion. Additionally colors

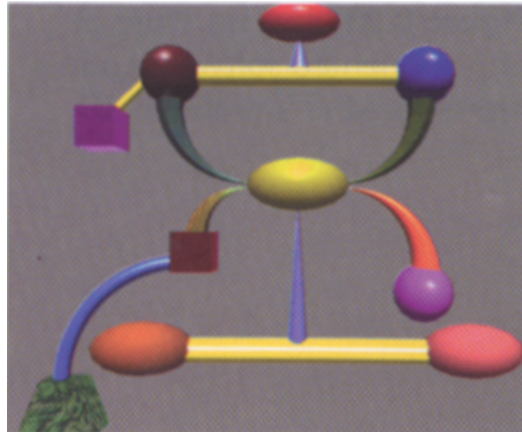


Figure 3.3: A geon diagram constructed using a subset of Biederman's geon primitives to represent the diagram in Figure 3.2 (from [Ware 2000, p. 256]). Major components of complex data objects were drawn as geons, architectural links were given as limbs consisting of elongated geons, minor subcomponents were attached as geon appendices and component attributes were given as color, texture and symbology mapped onto the geons.

introduce further problems in the case of color-blind users<sup>6</sup> or due to different cultural interpretation, e.g., red and green have contradictory meanings in European and Asian cultures. More generally, in [Schauer and Keller 1998], knowledge about the environment and the culture are mentioned as required information to find the correct interpretation of an artifact.

From the work in HCI described above and other publications we will now conclude several rules which are relevant for UML class diagrams from the HCI viewpoint. At a first glance, some of the HCI rules might repeat items or overlap with rules from graph drawing discussed in Section 3.3.2. In fact, this is not surprising, because some rules in graph drawing have been derived from the perceptual viewpoint and, of course, some aspects from graph drawing might have influenced HCI as well. Our overall goal in this section is to find a set of principles for UML class diagrams, which can be deduced from the other disciplines. Therefore, for each discipline, relevant rules are listed and assembled later in Section 3.3.6 regardless if they seem to overlap with criteria mentioned in sections before.

- **HCI\_HIERARCHY:** Nodes should be laid out in a top-down fashion in horizontal layers [Ware et al. 1993]. More generally, this relates to the logical flow in [Petre 1995].
- **HCI\_EDGES\_DIRECTIONS:** Arcs should point downward or horizontally on each layer. Obviously this interrelates with HCI\_HIERARCHY. There should be a minimum of upward pointing arcs [Ware et al. 1993]. More generally, this also relates to the logical flow mentioned in [Petre 1995].

<sup>6</sup>Obviously, many graphical representations, in particular UML class diagrams, induce problems for blind users. Textual alternatives were discussed along with input formats in Section 3.2.

- **HCI\_LAYER\_GRID**: Nodes in a layer should be laid out in a regular grid pattern [Ware et al. 1993]. This also supports grouping or alignment mentioned in [Petre 1995].
- **HCI\_SPATIAL**: Visibility and juxtaposability [Green and Blackwell 1998] can be realized by regarding and enforcing spatial relationships as shown for UML class diagrams in [Eichelberger 2003]. This was also mentioned as grouping or the use of white space in [Petre 1995], adjacency or the use of white space in [Noguchi and Tanaka 1998; Purchase et al. 2001a].
- **HCI\_CLUSTER**: Nesting nodes within nodes should be supported [Ware et al. 1993; Purchase et al. 2001a]. Nested nodes should be laid out within the parent nodes accepting the same criteria that are used for the parents [Ware et al. 1993].
- **HCI\_GROUP**: Model elements that belong together should be visually grouped [Petre 1995; Noguchi and Tanaka 1998; Purchase et al. 2001a]. This also relates to **HCI\_CLUSTER** but groups are distinguished visually by mechanisms for **HCI\_SPATIAL** while cluster are nodes having a certain shape.
- **HCI\_PATH\_CONTINUITY**: Based on the results of Gestalt psychologists, in [Ware 2000, p. 206] it was remarked that it is more easy to follow and understand smooth continuous contours instead of jagged ones. This has two direct consequences for graph drawing: Polyline paths with frequent changes of the direction are hard to follow and curved lines may simplify the perception of certain paths. Hence, **GS\_CURVES** is preferred over all kind of straightline drawings and **GDR\_MIN\_BENDS** appears to be important.
- **HCI\_NODES\_SYMMETRY**: Nodes should be positioned symmetrically, if they have properties in common or being equal in status [Dengler and Cowan 1998]. More generally, this relates to the alignment aspect in [Petre 1995] and adjacency in [Purchase et al. 2001a].
- **HCI\_NODES\_CENTER\_OR\_TOP**: Nodes should be placed centrally or nearer to the top of the layout, if they have special properties or are higher in status [Dengler and Cowan 1998]. More generally, this also relates to the alignment aspect in [Petre 1995] and adjacency in [Purchase et al. 2001a].
- **HCI\_NODES\_SEQUENCE**: Nodes should be positioned linearly, if (it should be assumed that) the sequence is significant [Dengler and Cowan 1998]. More generally, this relates to the alignment aspect in [Petre 1995] and adjacency in [Purchase et al. 2001a].
- **HCI\_MAGNITUDE**: The size of elements [Ware 2000] can be used to reflect the magnitude of the entity as shown for UML class diagrams in [Eichelberger 2003].

Another item in this list could be the replication of parts of the drawing: The lengths of edges can be shortened, the spacing between edges can be increased, the number of edge crossings can be reduced and the amount of edges with excessive turns and detours can also be narrowed when duplicating parts of the diagram. According to [Protsko et al. 1991], replication can help

achieving clarity but its overuse can lead to tedious complicated searches in the diagram. According to the UML specification, in a class diagram each named element may occur only once in a diagram due to the uniqueness of the fully qualified name. Hence, relations between those elements may also occur only once. Therefore, even if replication might be an interesting idea for adding clarity, it is not allowed in our application domain.

Beside HCI\_PATH\_CONTINUITY in [Ware et al. 2002] further issues for estimating the perceptual and cognitive costs of a diagram to refine usual graph drawing aesthetics were analyzed. The measurements were collected in the context of the shortest path  $p_s$  between two given nodes:

- The continuity, also called path bendiness, measured as the sum of angular deviations at all nodes of  $p_s$  from the straight line connecting the start and the end node of  $p_s$ .
- The number of crossings on  $p_s$ .
- The angle of each crossing on  $p_s$  to derive the average cosine crossing angle. It is expected that acute angles are more disruptive than more perpendicular angles.
- The number of branches so that if the total number of branches on  $p_s$  increases, the reading of the diagram becomes more difficult.
- The length of  $p_s$  as the size of the measurement context.
- The total geometric line length of  $p_s$ .

These values were compared, e.g., to the total number of crossings of the entire drawing. As a result, HCI\_PATH\_CONTINUITY was identified as an important factor in perceiving shortest paths. The total number of edge crossings was not a significant indicator of response time and the local number of crossings on the shortest path appears to be more interesting. Furthermore, another important factor seems to be the number of branches emanating from nodes on a path.

### 3.3.4 Software Engineering

There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C. A. R. Hoare

As introduced in Section 3.3.1, decomposition or conceptual readability determine main aspects of readability. Taking both aspect into account for UML class diagrams, we have to respect further aspects of object-oriented software engineering. Is therefore the beauty of a diagram, measured in dimensions of aesthetic criteria, somehow related to the quality the object-oriented design modeled by that diagram?

One direction of that question is obvious: If a well designed model is laid out by one of the tools compared in [Eichelberger 2002b] the result is probably a horrible layout. Therefore a horrible



layout usually does not always imply a poor quality of the design model.

We investigated literature from the software engineering community to find out what criteria have influence on the quality of an object-oriented model and how these criteria can be measured. Many publications, e.g., development processes [Jacobson et al. 1999], common software engineering [Summerville 1996], analysis and design patterns [Gamma et al. 2000] dealing with object-oriented design, describe commonly agreed methods on how to find classes and relations and especially how to focus on the relevant aspects of the software system to be modeled.

In [Genero et al. 2000], an overview of current measurements on design aspects was given. Most of these aspects are closely related to source code used to implement the target system. Source code related measurements will not be taken into account here, because in the early stages or iterations of most software process (at least informal) design documents are produced and source code is not available.

On the other side, coding-style independent source code metrics [Lorenz and Kidd 1994; Zuse 1998] might be interesting, when diagrams are produced in round-trip engineering and when synchronizing code with diagrams and v.v.

The following list enumerates criteria which may influence the layout of class diagrams respecting issues from the viewpoint of object-oriented system modeling:

- **SE\_FORESTS:** The depth of inheritance trees introduced in [Chidamber and Kemerer 1994] partially limits the physical dimensions of the drawing. It is known that well-designed OO systems are those structured as forests of classes, rather than as one very large inheritance lattice [Basili et al. 1996; Marchesi 1998]. Therefore these trees should be clearly visible and spatially separated from each other according to HCI\_SPATIAL.
- **SE\_INHERITANCE:** The in-degree restricted to inheritance relations should be minimized as described in [Lorenz and Kidd 1994], e.g., to restrict the use of multiple inheritance. Hence, the inheritance trees/implementation hierarchies are simplified.
- **SE\_NUMBER\_OF\_CHILDREN:** The limitation on the number of children introduced in [Chidamber and Kemerer 1994] relates to the difficulty to modify, maintain and test the implementation. Usually more testing is required because such classes potentially affect all of their children. Furthermore, a class with a large number of children may have to provide services in a larger number of contexts and must be more flexible [Basili et al. 1996]. Therefore the out-degree restricted to inheritance relations should be limited. Hence, the inheritance trees/implementation hierarchies are simplified.
- **SE\_CLASS:** Despite of obvious countings of class members, different other approaches have been proposed in [Lorenz and Kidd 1994; Brito e Abreu and Melo 1996; Marchesi 1998; Zuse 1998] to formalize the complexity of classes.  
One obvious point is that empty classes that do neither implement any methods nor define any attributes, are a lack in design quality [Lorenz and Kidd 1994]. They do not add any functionality except for usually inheriting from superclasses. This is not always that simple: While design takes progress there might be some classes which have not been fully

specified so far. This could be marked by ellipses instead of method or attribute signatures to show that these classes will not remain empty. Furthermore, empty interfaces are widely used as marker interfaces (e.g., `Serializable` in Java) and should not be treated like empty classes.

In [Genero et al. 2000], further measurements on class complexity were mentioned: Number of associations, height of a class within the aggregation hierarchy, number of multiple aggregations, number of in/out-dependencies and the number of (direct) parts/wholes respecting compositions. Furthermore, in [Marchesi 1998], the weighted number of responsibilities as well as the weighted number of dependencies were introduced.

Based on a combined measurement reflecting the different approaches, additional tool-specific tag-values or a magnitude-related decorative stereotype (see Figure 3.4) can be

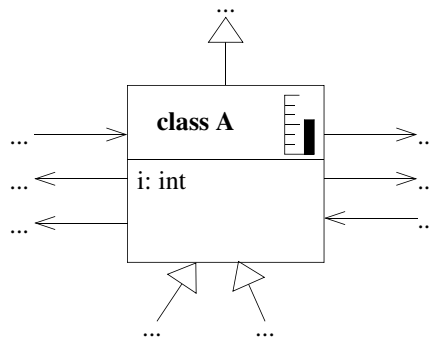


Figure 3.4: A class scaled to the magnitude of its coupling and a magnitude-related decorative stereotype.

displayed within the class rectangle. The area occupied by classes (`HCI_MAGNITUDE`) might be adjusted according to a weighted combination of the metrics, because changes of the model by a layout algorithm are usually not tolerable.

On the one side, this does not introduce conflicts with the UML, because the area requirements of individual classes are not specified. On the other side, this may introduce certain problems: The layout result appears as an unusual drawing due to the scaling of the nodes. Furthermore, in early analysis and design phases, classes are not always fully specified and class size metrics may lead to misinterpretations. Even if also alternative line styles would be appropriate to reflect the (hidden) complexity, *error-proneness* (e.g., visual conflicts with active objects), *hard mental operations* (by increasing the number of different line styles) and multiple substandards of the UML would be the consequence of introducing new line styles or even the stereotype shown in Figure 3.4 as a proprietary extension of the UML.

- `SE_PACKAGE`: In [Marchesi 1998], package coupling, in [Genero et al. 2000], the number of intra/inter-package associations, aggregations and in/out-dependencies were used to define the complexity of packages and therefore in principle of UML model management elements. In [Marchesi 1998], the weighted number of responsibilities as well as the weighted

number of dependencies were mentioned. Similar to `SE_CLASS` tag-values, stereotypes (see Figure 3.4) or the magnitude of the package according to `HCI_MAGNITUDE` might be used to reflect the complexity. Like `SE_CLASS`, scaling packages does not induce conflicts with the UML, but may also introduce similar problems.

- `SE_COUPLING`: Measuring coupling of classes and packages is a difficult task due to different viewpoints and definitions [Chidamber and Kemerer 1994; Li et al. 1995; Bansiya et al. 1999]. An overview of coupling measurements was given in [Briand et al. 1999]. Inheritance coupling is implicitly respected in `SE_FORESTS`, `SE_INHERITANCE`, `SE_NUMBER_OF_CHILDREN` and obviously present in UML class diagrams by generalization edges. Component coupling can be visualized by `SE_CLASS` and `SE_PACKAGE`. Interaction coupling is usually calculated from code (hidden method calls and scattered coupling) by static type analysis, but by introducing (hidden) dependencies from methods or attributes to the parameter, return or attribute types, respectively, class-attribute-interaction and class-method-interaction can be respected in a diagram. This information also increases the class and package complexity (see `SE_CLASS` and `SE_PACKAGE`). Within packages, intra- and inter-package coupled classes can be visualized by spatial distribution of classes (`HCI_SPATIAL`<sup>7</sup>). Shading or coloring of inter-package or intra-package coupling seems to be appropriate but this collides with the UML specification and the results in Section 3.3.3.
- `SE_DESIGN_PATTERNS`: The use of design patterns [Gamma et al. 2000] might be taken into account to measure the design quality. Thereby, the design pattern symbol and the related classes should be positioned in a close vicinity, other connected classes should be spatially separated from the design pattern collaboration according to `HCI_SPATIAL`. The size of the collaboration can be related to `HCI_MAGNITUDE`.
- `SE_RELATIONS`: Complexity of associations [Zuse 1998] and other relations could easily be displayed by applying the node-link-diagram grammar rules in [Ware 2000] but requires changes to the standardized UML notation. Therefore, `HCI_MAGNITUDE` does not seem to be an appropriate representation in UML class diagrams.
- `SE_DISCONNECTED`: On the one side, classes which are not interrelated to other classes are sometimes identified as a problem. On the other side, a software engineer might include e.g., a utility class having static accessors only into a diagram to emphasize that this class should be used when implementing the software. To reduce the visual complexity of the diagram, the designer may decide to draw this class without edges rather than showing all reasonable dependencies.  
In the underlying model, which captures all elements and relations of the entire software project, relations to the disconnected element, which are not displayed in the current diagram, may exist. If such relations are present, they can be seen as invisible dependencies inducing reasons for a vicinity to other model elements. If no invisible relations to elements of the current diagram exist, the disconnected model elements should be clearly

---

<sup>7</sup>Will be demonstrated along with the UML specific criteria in Figure 3.7

visible, e.g., at the boundary of the diagram, to emphasize their global meaning and to avoid misleading interpretations due to a vicinity to other model elements.

In fact, on the one side, this may also lead to misinterpretations considering HCI\_HIERARCHY, because these elements appear on hierarchical levels they do not belong to. On the other side, these elements are disconnected and the hierarchy is clearly induced by connected elements. From this viewpoint, we are convinced that exposing disconnected elements on the boundary is a valid layout style.

Even if the whole complexity and history of a project is accessible to a layout tool through the use of repositories, project metrics [Lorenz and Kidd 1994; Zuse 1998] like application size, staffing size and scheduling do not have influence on the relation between layout and design quality. Global class diagram complexity mentioned in [Marchesi 1998] could be taken into account, if the layout algorithm can be influenced by these measurements.

Of course, many of the effects described by the criteria listed in this section can simply be visualized by introducing alternative notations like mentioned above and in Section 3.3.3. According to our top-level goal, using the current version of UML without modification, we have to calculate visualizations with restricted instruments.

The combination of these metrics (see [Zuse 1998] for the mathematical background) leads to a formalized judgment of UML class diagrams with respect to object-oriented aspects. When properly incorporating the knowledge of the 9 rules from software engineering into a set of aesthetic criteria for the layout of UML class diagrams, a design, which reaches a high judgment in design metrics, should also have a high judgment respecting UML class diagram layout metrics on diagrams laid out according to our rules.

### 3.3.5 Software Visualization (SV)

Nevertheless, a big gap between desired aspects and the features of current SV tools was identified.

[Bassil and Keller 2001]

Few studies on engineer, developer and programmer preferences have been conducted so far in conjunction with issues on the display of diagrams in software engineering. In the last years two user studies [Bassil and Keller 2001; Koschke 2003] were published which will be taken into account here. As for the disciplines discussed in the last sections, we will collect some rules to be considered when our set of aesthetic principles for UML class diagrams will be assembled in the next section.

- **SV\_HIERARCHY:** Emphasize the inheritance hierarchy as the main structure of the visualization. Of 24 tools surveyed in [Bassil and Keller 2001], the following aspects were mentioned in decreasing order of priority: function calls showing complex recursion chains, inheritance graphs depicting deep inheritance braces, subsystem architecture graphs, inheritance graphs for detecting occurrences of multiple inheritance and further aspects of

function call graphs. The aspects related to visualization of function calls and of inheritance graphs were identified as the most wanted.

Because function calls are less (e.g., as invisible hyperlinks or dependencies) or not relevant to UML class diagrams, hierarchy and a proper display of subsystem architecture appear as the most interesting aspects. This was suggested in the second study, too.

Furthermore, in [Koschke 2003] trees were identified as a popular layout in software engineering tools. Due to the hierarchical nature of the underlying data it was suggested, that layout algorithms should be able to distinguish edges: Tree edges, which form the structure of the hierarchy, and further secondary edges. According to [Koschke 2003] this naturally applies to UML class diagrams with inheritance relationships as tree edges.

- **SV\_SUBSYSTEMS:** Show subsystem architectures as a separate diagram or, at least, emphasize subsystem architectures in a diagram. This directly relates to the results in [Bassil and Keller 2001] and **SV\_HIERARCHY**.
- **SV\_POLYMETRIC:** Nodes, when drawing graphs for software engineering, can be enriched with further semantical features, known as *polymetric views of nodes* [Lanza 2003]. As shown in Figure 3.5 up to 5 metrics can be visualized at a single node:

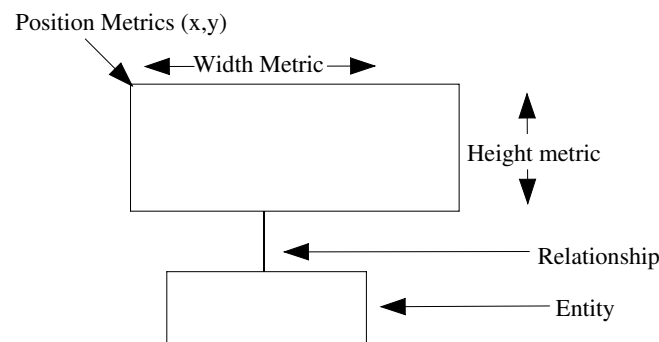


Figure 3.5: The principle of a polymetric view. (from [Lanza 2003])

- **NodeSize:** The width and height of a node can render two measurements, whereby the size of the extent relates to the value of the measurements.
- **NodeColor:** The color interval between white and black can display a measurement, whereby the darkness of the color relates to the measurement value. Thus light gray represents a smaller metric measurement than dark gray (not displayed in Figure 3.5 due to the principle nature of the illustration). Like discussed earlier, color is currently a proprietary feature in UML and will therefore be considered as an optional feature only.
- **NodePosition:** The horizontal and vertical coordinates of the position of a node can reflect two other measurements. This requires the presence of an absolute origin

within a fixed coordinate system. Therefore, not all layouts can exploit this dimension.

Obviously this is a more general approach than our proposition `SE_CLASS` which therefore not only relates to the issues of software engineering but also to known techniques in software visualization.

Finally, from the viewpoint of software engineers and software visualizers, drawing graphs requires different important features, which are often hard to realize:

The respondents of this research survey raise several issues specific to graph drawing, such as the need for scalable, incremental, and semantic layouts (i.e. layouts that take the semantics of nodes and edges into account).

[Koschke 2003]

Due to our basic set of requirements, we will try to realize issues of semantic layouts, but scalable (`REQ_SPEED`) and incremental layout (`REQ_INCREMENTAL_ALGORITHM`) will be part of future work.

### 3.3.6 Semantic Aesthetic Principles for UML Class Diagrams

These aesthetics and efficiency criteria stand in contrast to more intuitive criteria concerning the semantics and intended meanings of graphs.

[Fleischer and Hirsch 2001]

After discussing various influences on the readability of (UML class) diagrams and identifying 60 basic rules from graph drawing in Section 3.3.2, HCI in Section 3.3.3, software engineering in Section 3.3.4 and software visualization in Section 3.3.5, we are now ready to combine the knowledge with the structural specifications for class diagrams described in Section 2.1.2. The result of this section will be a set of 23 criteria assigned to three categories. The principles within a category will be ordered according to priorities due to our experience. To present these rules, we will list each individual rule, even if similar basic rules might have been enumerated in the sections before, we will describe and explain it using a class diagram example and we will give references to the basic rules from which the more specific UML rule is deduced.

To gain the set of criteria, we will first determine the three categories to be handled differently by a concrete implementation. Then, for each category, all assigned rules will be listed. Due to the degree of freedom of elements in UML class diagrams, we can expect that the set of rules will not be free of conflicts for a certain input graph. Therefore, priorities will be assigned and the reasons for assigning priorities to individual criteria will be given. After this, a conclusion on all the rules to be realized by the current version of our implementation, a comparison to previous work on layout rules for UML class diagrams, visual quality indicators to relate layout and underlying design as well as issues on validating our set of criteria will be given.

Aesthetic criteria for UML class diagrams are either *mandatory* for readability, *optional*, e.g., due to variants specified in the UML, or additional *user defined* features, which might be interesting to be enforced in certain situations. Mandatory and optional criteria, which, if enabled, have to be treated like mandatory criteria, should be realized according to the assigned priorities. As discussed for REQ\_USER\_OPTIONS, user defined layout constraints outside the UML might be relevant to additionally improve the readability and understandability of a diagram. These hints may, like graph drawing constraints, induce conflicts with the basic aesthetic criteria. The user should be informed about such problems, the conflicting constraints of lower priority should be ignored and optionally removed from the specification of the diagram.

Of course, all principles should be consistent to the style guidelines or the presentation options defined in UML like font faces, text alignment or underlining. On some (experimental) optional criteria, which may introduce conflicts with the UML, this will clearly be indicated.

The foundation of our work is therefore the UML specification itself [OMG 2003c, p. 3-6], which defines three important kinds of visual relationships: Connection, containment and visual attachment, where one symbol has to be placed near another one.

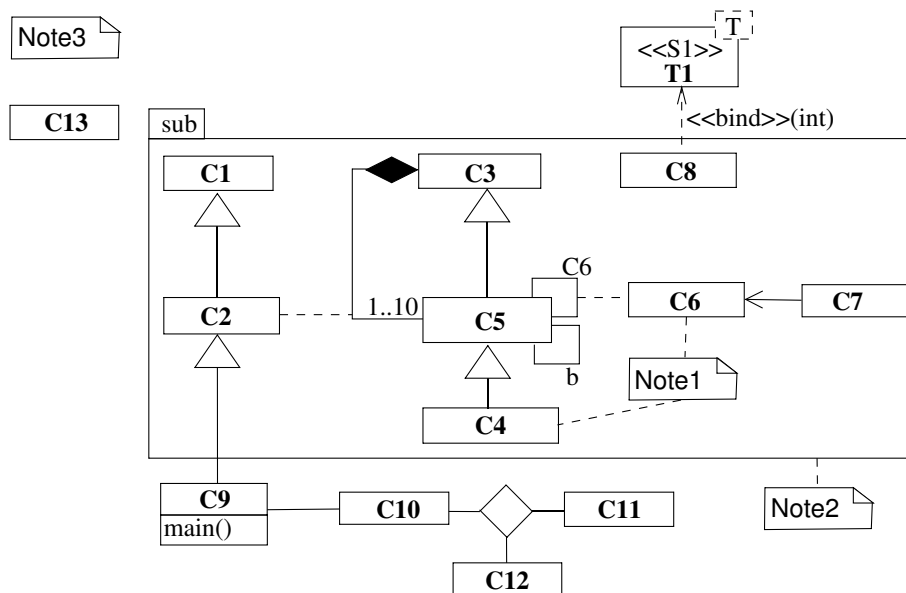


Figure 3.6: An example class diagram containing classes in a package, a template, a ternary association, comments, two association classes and two reflective associations.

### Mandatory Aesthetic Criteria

We will now list the UML specific criteria mandatory for the readability of UML class diagrams.

- **UML\_HIERARCHY:** In Figure 3.6 inheritance edges, composition relations, bind dependencies and the package containment were chosen as (*pseudo*) *hierarchy*. This

admits a clear structure of the drawing, conforms to the default UML layout style and realizes the basic UML rules for visual relationships as well as SE\_INHERITANCE and SV\_HIERARCHY.

The relations in a class diagram can be partitioned according to different rules. Because programmers and software engineers are used to think in hierarchies (SV\_HIERARCHY, SE\_FORESTS, SE\_INHERITANCE) to give their projects a certain structure like class, package, module or containment hierarchies, we recommend to partition the edges into a set of hierarchical and non-hierarchical edges. Containment relations, denoting the nesting of elements in other elements, are required to be hierarchical due to the UML specification. Furthermore, an usual partition would be to regard the inheritance and realization relations as hierarchical and the other relations as non-hierarchical relations.

Class nesting (by the anchor notation), partly aggregations, compositions, directed associations and some dependencies may be considered as members of the (pseudo) hierarchy as well.

According to different viewpoints, a user defined hierarchy may also be appropriate for individual diagrams.

Beside the basic rules mentioned above, UML\_HIERARCHY can be deduced from GDR\_FLOW, GDR\_HIERARCHY, HCI\_HIERARCHY and HCI\_SPATIAL to ensure that the hierarchy is clearly visible. This rule also partly supports GDR\_EDGE\_DIRECTIONS, HCI\_EDGES\_DIRECTIONS and HCI\_NODES\_CENTER\_OR\_TOP (roots of several hierarchies at the top of the drawing). Dependent on the layout mechanism for individual layers, HCI\_LAYER\_GRID might also be supported.

To distinguish visually between hierarchical and non-hierarchical edges, GDR\_MANHATTAN may be applied to the set of non-hierarchical edges opposite to GS\_STRAIGHTLINE for hierarchical edges [Seemann 1997; Noguchi and Tanaka 1998]. We do not incorporate HCI\_PATH\_CONTINUITY, because of the suggestive nature of the default UML layout style.

- **UML\_SPATIAL:** In the example drawing in Figure 3.6, the two inheritance trees rooted at C1, C3 as well as T1/C8 are spatially separated from each other. Furthermore, the elements of the ternary association C10 up to C12 all elements belonging to the package sub are grouped together, respectively.

According to HCI\_SPATIAL and most of the criteria identified from software engineering, spatial relations and a two-dimensional node distribution should be respected. Elements contained in other elements or collaborations (pattern notation) can be emphasized by respecting spatial relations and vicinity. Different hierarchy trees defined according to UML\_HIERARCHY should be laid out spatially separated (HCI\_SPATIAL, SE\_FORESTS). UML\_SPATIAL interrelates with GDR\_NODES\_EDGES and partly with GDR\_DIAGRAM\_CENTER.

- **UML\_SEMANTIC\_CLUSTERS:** In Figure 3.6, C1 up to C8 are members of the package sub, C10 up to C12 and the rhomb are members of a ternary association. These classes are members of separate clusters, the latter ones can be seen as members of an invisible



cluster.

Package membership, composition notation and invisible clusters like n-ary associations or patterns lead to an obvious clustering. Classes, in the case of the composition notation and UML model management elements (packages, subsystems and models) act as visible bordered clusters. According to the UML, both types of visible clusters induce rectangular regions, packages additional own a tab. Invisible clusters can be used as mechanisms to enforce semantical grouping.

Members of a cluster should be located in a close vicinity. UML\_SPATIAL and therefore HCI\_SPATIAL have to be respected. This rule arises from GDC\_GROUP, GDR\_CLUSTERING, SE\_PACKAGE, SV\_SUBSYSTEMS, HCI\_CLUSTER and HCI\_GROUP.

- **UML\_MEDIAN:** In Figure 3.6, for example C1 is centered above C2 which itself is centered above C9. This highlights the hierarchical relations.  
Especially in hierarchical relations, a parent node should be arranged as close as possible to the median position of its children. A child node should be located as close as possible to the median position of its parents. Of course, this may apply to neighbors in non-hierarchical relations as well.  
This rule can be deduced from SE\_FORESTS, SE\_NUMBER\_OF\_CHILDREN, SE\_INHERITANCE, SV\_HIERARCHY, UML\_SPATIAL, HCI\_NODES\_SYMMETRY and GDR\_SYMMETRY. It partly supports HCI\_NODES\_CENTER\_OR\_TOP but not HCI\_NODES\_SEQUENCE because UML does not impose an ordering on (child) nodes.
- **UML\_NODES:** To support readability, several issues on the size, the interior and the placement of individual nodes have to be respected. We will start with this general rule for all nodes and refine it rule for more specialized types like classes or packages.  
Except for nested nodes like compositions or nested packages which are not connected by anchor relations, nodes should not overlap other nodes or edges (GDR\_OVERLAP). Furthermore, interior elements of a node should neither overlap other interior elements of the same node nor the borders of the node. In Figure 3.6 all nodes are laid out properly with respect to a minimum node size which was chosen arbitrarily for that example.  
In principle, the area required by an individual node should be minimized (GDR\_NODE\_SIZE, GDC\_NODE\_SYMBOLS and GDR\_CONTOUR\_DIFFERENCE) with respect to the area required by the connected edges and the area occupied by the interior elements. Templates at classes (e.g., T1 in Figure 3.6) or at patterns should neither overlap (GDR\_OVERLAP) with the class interior (e.g., the contents of the name compartment) nor with edges connected to a class, but some edges might cross the border of a node, e.g., dependencies to operations.  
Nodes on the same hierarchy level should have related (e.g., a similar vertical or horizontal) positions (GDR\_HIERARCHY, GDR\_SYMMETRY and GDC\_GROUP\_CURVE), according to the graphical standard (e.g., top-down, left-right) with which the hierarchy is drawn. If hyperbolic or radial layout is applied, even if the result then does not fit to the current default UML layout style, the hierarchy levels are represented by circular-like

shapes (GDC\_GROUP\_CURVE).

On the one side, nodes should not be laid out too close to one another. For example, this can be realized by taking minimum distances between nodes into account. On the other side, nodes should not be drawn too far apart (GDR\_DIMENSIONS\_SYMBOL, GDR\_NODE\_DISTRIBUTION and GDR\_MIN\_EDGES) with respect to spatial relations according to the software engineering rules SE\_FORESTS up to SE\_RELATIONS. This aim can be obtained by compacting a drawing or by compressing unused areas.

A common font size similar to GDR\_FONTS should be used for interior elements to visually emphasize elements which belong together.

- UML\_CONTAINER (extends UML\_NODES): The borders of UML model management elements (packages, subsystems or models) as well as classes in composite notation (as shown in Figure 2.10) might be crossed by edges as depicted in Figure 3.6 but not by visible nodes. A tab of a model management element should neither be crossed by edges nor be overlaid by nodes (GDR\_OVERLAP). Furthermore, edges to a model management element should not connect to the tab even if this might globally require more drawing area. The border of a package should enframe the contained elements as close as possible. For subsystems, minimum sizes to ensure that compartments can properly be displayed, have to be respected (GDR\_NODE\_SIZE and GDC\_NODE\_SYMBOLS). The spatial arrangement of compartments in subsystems is not specified by UML and therefore the shape of a subsystem can dynamically and polymorphically be selected to improve the readability.
- UML\_CLASS (extends UML\_NODES): Font attributes for classes should be used according to the UML notation guide. A general font size should be used for class interior elements belonging to the same group (e.g., class names, stereotypes) or compartment type (e.g., operations, attributes, user-defined). The text labels used within a node should be directed towards the same direction (GDR\_LABELS\_DIRECTION). Furthermore, the interior elements should be aligned at least according to the basic UML styleguide, i.e., the elements in the names compartment should be centered, the attribute or operation signatures should be aligned to the left side, etc. To adjust the aspect ratio of the class rectangles, line breaks may (automatically) be inserted and the alignment of the subsequent lines should be adjusted to emphasize grouping.  
Similar or equal rules apply to class instances.
- UML\_CENTER: The rhomb of the ternary association in Figure 3.6 is centered upon its attached classes to also emphasize the relation between these elements as suggested by the default UML layout style.  
Centering of individual diagram should also be applied to the ellipse in the collaboration notation (SE\_DESIGN\_PATTERNS) or the junction point of multiple dependencies. This maps to HCI\_SPATIAL, HCI\_GROUP, GDC\_GROUP but not HCI\_NODES\_CENTER\_OR\_TOP.

- **UML\_EDGES:** As shown in Section 2.2.3, some layout mechanisms do not consider that all edges, in particular multiple edges between two nodes, should be visible as individuals. This was maintained in Figure 3.6. To attain readability, different edges should not overlap (**GDR\_OVERLAP**), this means that every edge should be visible and readable as an individual. Of course edges should not overlap nodes (**GDR\_OVERLAP**).  
In principle, also **GDR\_MIN\_EDGES** belongs to this rule, but optimizing a complex diagram, which contains association classes, hyper edges, comments and ordinary edges with respect to **GDR\_MIN\_EDGES** may lead to conflicts. Therefore, we will introduce this general and several specific rules which all map to **GDR\_MIN\_EDGES** but induce different priorities.
- **UML\_ASSOCIATIONCLASSES** (extends **UML\_NODES** and **UML\_EDGES**): An association class, specifying features of an association using a class-like notation, is attached by a dashed line to the association. In Figure 3.6, C2 is connected to the composition relationship between C3 and C5. C6 is attached to the reflective edge from C5 to itself. To simplify the perception of the relation and an associated class, both should appear in a close vicinity but not be located too near to either end of the relation [OMG 2003c, p. 3-78]. A location below a non-hierarchical association is preferred due to the default UML layout style (**UML\_SPATIAL,UML\_SEMANTIC\_CLUSTERS**).  
This rule supports **HCI\_SPATIAL**, **HCI\_GROUP** and **GDC\_GROUP**.
- **UML\_HYPEREDGES** (extends **UML\_EDGES**): Hyperedges, like xor-constraints, as introduced by Figure 2.6 (c), are not shown in Figure 3.6. But it is obvious that the length of hyperedges should be as short as possible. The connected model elements should be located in a close vicinity because of semantic reasons and improved readability (**HCI\_SPATIAL**, **HCI\_GROUP** and **GDC\_GROUP**).
- **UML\_REFLECTIVE** (extends **UML\_EDGES**): In Figure 3.6, two reflective associations are connected to C5. When reflective relations are handled like other edges by the drawing mechanism, e.g., it may occur, that a reflective relation is drawn from the left vertical to the right vertical side of a class and thereby it may cause crossings with other relations connected to the class as shown in [Eichelberger 2002b]. Depending on the size of the adornments of a reflective relation, it might be appropriate to draw it at a corner of the class or somewhere between two different relations adjacent to the class.  
Therefore, reflective edges must not overlap other elements and, in particular, it must not occur that reflective edges overlap each other as in Figure 2.17. Relations between reflective associations and other model elements like comments or association classes should not cross other model elements (especially reflective associations, **GDR\_OVERLAP**).
- **UML\_ADORNMENTS:** In Figure 3.6, a directed association to C6 and an aggregation at C3 are shown. The graphical symbols used to indicate the type of the relation as well as multiplicities like the one at the aggregation or the names of the reflective associations in Figure 3.6 are used to specify details on individual relations.  
The UML specification refers to all textual and graphical elements, which can be attached

to a model element, in particular to associations, as “adornments”. Examples are graphical adornments, like the hollow diamond at aggregations, role adornments or groups of adornments like end adornments of associations.

Generally, stereotypes, constraints and tagged values may be attached to every model element. These additional specifications as well as discriminators at inheritance edges should be clearly assigned to their model elements and should neither overlap other adornments nor other model elements (GDR\_OVERLAP). This, of course, depends on the strategy which determines the size of nodes, because, if a node is minimized in size without respecting the requirements of the connected edges, it might be necessary to scale down the edges and their adornments.

It is desirable that the text labels are oriented to one general direction (GDR\_LABELS\_DIRECTION). A general font size (similar to GDR\_FONTS) should be used for edge adornments belonging to the same group (e.g., constraints). Further font attributes should be used according to the UML notation guide.

The same holds for adornments optionally placed outside the model element, e.g., a constraint might be given outside the class rectangle. Thereby, these constraints should be located in a clear vicinity to the invisibly related model element.

Lollies, used to visualize interfaces realized by individual classes, appear as an adornment to be kept close to the connected class by considering a given minimum distance and the name of the interface.

To simplify reading, adornments should be compliant to basic relative positioning rules, e.g., multiplicities may always be placed above an association, the rolename below. Further examples are the following rules: The stereotype of an association, if present, should be placed at top, above the association name, both above the association. Constraints, if present, should be placed below the association, tag-values below the constraints. If the association is not drawn horizontally, the multiplicities should be on the left, roles on the right side. The stereotype, the association name, the constraints and the tag values should be arranged as described above, but should then be placed at one of the vertical sides of the association depending on the available space.

- **UML\_COMMENTS** (extends **UML\_NODES** and **UML\_EDGES**): The comments *Note1* and *Note2* in Figure 3.6 are located in a close vicinity to the connected model elements. In particular, if comments are connected to multiple model elements, like *Note1*, the comment should be centered between the connected model elements if possible. Comment nodes should be respected in package containment as well.

As written in the UML specification, comments should be placed with some vertical or horizontal offset to emphasize the distinct type of the comment. Therefore, based on **HCI\_SPATIAL**, alignment to ranks is not always appropriate for comments.

In some tools, like Innovator by MID, information on the editor and the relevant modification dates of a diagram are displayed in non-UML style at the top left corner as it is usual for other technical diagrams. Therefore, comments attached to the diagram, not to any model element, like *Note3*, should occur at the boundary of the drawing, (**GDC\_BOUNDARY**) preferable at a position defined in a style guide to avoid mislead-

ing interpretation based on a vicinity to other model elements.

This rule supports HCI\_SPATIAL, HCI\_GROUP, GDC\_GROUP. In fact, this rule relates to UML\_SPATIAL and UML\_SEMANTIC\_CLUSTERS.

- **UML\_DISCONNECTED**: According to the UML specific rules discussed above, elements are placed in the vicinity of their connected model elements. C13 in Figure 3.6, a disconnected class, is placed at the border of the diagram and not, e.g., next to C4 or below C9 to avoid erroneous perception due to nesting or vicinity to other model elements.

Considering SE\_DISCONNECTED, disconnected elements (except for comments handled by UML\_COMMENTS), which have invisible relations to other model elements, can be treated like usual model elements.

According to SE\_DISCONNECTED, we prefer the boundary of the graph instead of placing disconnected elements between related elements in the interior area of the drawing.

- **UML\_GRAPHDRAWING**: As mentioned along with UML\_EDGES, GDR\_MIN\_EDGES has to be respected for all ordinary edges and paths, also to keep the size of the drawing small and to simplify the perception of relations.

Fortunately, the diagram in Figure 3.6 can be laid out without edge crossings, but, due to the complexity of relations, class diagrams usually admit a non-planar drawing. Therefore, edge crossings cannot always be avoided but they should be circumvented whenever possible (GDR\_EDGE\_CROSS).

It is difficult to find a set of graph drawing principles which does not introduce conflicts with our UML specific rules, but some can obviously be enumerated: Edges should not have too much bends (GDR\_MIN\_BENDS) and especially superfluous bends should be avoided. The angle between (horizontal) incident edges should not be too small (GDR\_ANGLE). As much as possible, edge lengths should be uniform, in particular for hierarchical relations (GDR\_UNIFORM\_LENGTHS). Due to SE\_FORESTS, a generally compact drawing (with respect to UML\_HIERARCHY and UML\_SPATIAL) is more interesting than a balanced aspect-ratio (GDR\_DRAWING\_SIZE) or density (GDR\_DENSITY).

Table 3.2 explicitly shows the priority levels and the dependency relations between individual UML specific rules introduced above. Five groups of rules can be identified: Criteria for emphasizing hierarchy and nesting, border and interior specific rules for nodes and visible clusters, rules for edges and proximity, properties of disconnected elements and rules which influence the compactness of the drawing as well as the number of edge crossings.

At a first glance, classifying UML\_CENTER as a node-related criterion may appear unusual, but UML\_CENTER is intended to enforce a node related placement rather than a relation based coordinates assignment of the connected nodes, i.e., in this case, the positions of nodes is more important than optimizing e.g. GDR\_MIN\_EDGES. Also UML\_CENTER is the only rule, which interferes with other criteria when virtually swapping the priorities of all node related criteria with those of hierarchy and nesting related criteria. Now, central positioning would receive a higher priority than structural relevant criteria ensuring the correctness of nesting. This would conflict with the structural rules of UML.

Beside the UML default layout, the typical requirements of software engineers lead to the highest priority for UML\_HIERARCHY as the main principle. UML\_SPATIAL, UML\_SEMANTIC\_CLUSTERS and UML\_MEDIAN (on the same priority level as UML\_SEMANTIC\_CLUSTERS) enforce hierarchical relations in UML class diagrams and are assigned to the same group.

From the practical viewpoint of software engineers, the textual information represented in nodes seems to be of a high importance:

Another respondent complains that aesthetics are underrelated in software visualization and he would hope for visualization tools that present good-looking views as opposed to ones with minimum edge crossings. For instance, he would be happier if he could read the names of all nodes in a graph rather than have a better layout according to some abstract criterion.

[Koschke 2003]

This maps to our own experience with UML tools. Furthermore, it was shown in [Purchase et al. 2001a] that the exclusive application of traditional graph drawing criteria is not sufficient for UML class diagrams. Therefore, we prefer UML criteria over syntactical graph drawing rules and assigned the group of criteria directly influencing the proper display of the nodes to the next level of priority.

Considering GDR\_MIN\_EDGES helps reaching a compact drawing and enforces proximity between the connected elements. When GDR\_MIN\_EDGES is realized by a layout algorithm without distinguishing the type of an edge, the proximity of association classes, connected comments or start and end points of hyperedges cannot be ensured. Therefore, we defined a basic criterion for edges (UML\_EDGES) and introduced local levels of priorities for model elements to be kept in a close vicinity to the connected elements because of semantical reasons. Thereby, to us, the structural relevance of association classes and reflective edges to a concrete implementation appear to be more important than the constraints introduced by role names, association names, multiplicities or graphical adornments. Because comments are intended to document parts of a diagram for understandability, we assigned UML\_COMMENTS a lower priority than UML\_ADORNMENTS.

Due to the same reason for preferring node criteria over edge criteria, we placed UML\_DISCONNECTED on a higher priority level than UML\_GRAPHDRAWING.

Regarding the low priority of graph drawing constraints in our set of aesthetic criteria, we do not want to express that, e.g., the number of edge crossings is not important for the perception of the result. Imagine two diagrams,  $D_A$  and  $D_G$  showing the same UML class diagrams in two distinct layouts. If both diagrams are laid out according to all UML specific rules but in  $D_A$  more edge crossings or longer edge routes occur, we would prefer  $D_G$  over  $D_A$ . Let us now assume that the basic UML diagram contains an association classifier and laying it out according to our set of rules in a close vicinity to the connected association would induce several edge crossings in  $D_A$ . Furthermore, let  $D_G$  be optimized for edge crossings but without considering UML\_ASSOCIATIONCLASSES at that high level of priority so that, however, the dashed line to the association class appears as a long non-hierarchical path in the diagram. Then we would

<b>priority</b>	<b>name</b>	<b>depends on</b>
5.2	UML_HIERARCHY	
5.1	UML_SPATIAL	UML_HIERARCHY
5.0	UML_SEMANTIC_CLUSTERS	UML_HIERARCHY, UML_SPATIAL
5.0	UML_MEDIAN	UML_HIERARCHY, UML_SPATIAL
4.3	UML_NODES	
4.2	UML_CONTAINER	UML_SEMANTIC_CLUSTERS, UML_NODES
4.1	UML_CLASS	UML_NODES
4.0	UML_CENTER	UML_NODES
3.5	UML_EDGES	UML_NODES
3.4	UML_ASSOCIATIONCLASSES	UML_NODES, UML_EDGES
3.3	UML_HYPEREDGES	UML_EDGES
3.2	UML_REFLECTIVE	UML_EDGES
3.1	UML_ADORNMENTS	UML_NODES, UML_EDGES
3.0	UML_COMMENTS	UML_NODES, UML_EDGES
2	UML_DISCONNECTED	UML_NODES
1	UML_GRAPHDRAWING	UML_HIERARCHY, UML_SPATIAL, UML_COMMENTS

Table 3.2: Priority levels and dependencies among the mandatory UML criteria.

clearly prefer  $D_A$  over  $D_G$ , because  $D_A$  fulfills the rules described in UML regardless the higher number of edge crossings. This is compliant to the assignment of priorities to the set of mandatory rules.

### Optional Aesthetic Criteria

Beside the mandatory criteria introduced above, several optional criteria according to some of the presentation options in the UML specification might be respected for certain diagrams. The optional features should be understood as a defined set of features to the layout algorithm which can be selected, enabled or disabled, preferably as a kind of style guide independent of concrete diagrams (REQ\_USER\_OPTIONS).

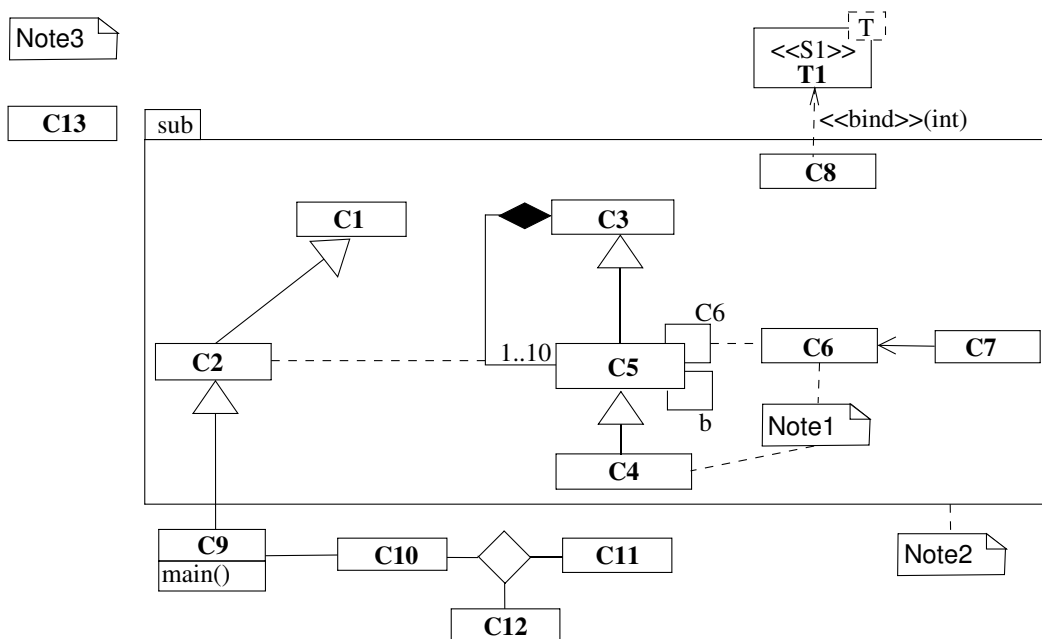


Figure 3.7: Figure 3.6 redrawn to emphasize spatial distribution. C2 and C8, the only classes which are inter-package related, have been moved towards a separate area, which is reserved for inter-package coupled classes. Subsequent displacements were required to fulfill UML\_MEDIAN and GDR\_MIN\_EDGES.

- **UML\_COUPLING:** In Figure 3.7 ideas from HCI and software engineering were combined to focus on coupling of package memberships. C2 and C8 are connected to elements outside the package sub and were therefore spatially separated in the drawing. An optional extension to the default UML layout style, coupling and inter/intra-package class relations can be pointed out by spatial relationships and vicinity. To emphasize SE\_COUPLING and SE\_PACKAGE, the rules for UML\_SPATIAL, UML\_NODES and UML\_CONTAINER must be relaxed.



Even if `UML_COUPLING` does not conform to the default UML layout style, it influences the display of hierarchies and therefore receives a priority directly below `UML_SPATIAL` but more than `UML_SEMANTIC_CLUSTERS` or `UML_MEDIAN`.

- `UML_JOIN`: Inheritance relations, aggregations and compositions should be joined as described in [OMG 2003c] wherever possible. This admits a kind of orthogonal layout for hierarchical relations: All children are connected to a horizontal bar which then is linked to the parent class. The adornments at the parent side of the relations are shown only once. On the one side, if, as suggested above, `GS_ORTHOGONAL` is used for non-hierarchical edges and `GS_STRAIGHTLINE` for hierarchical edges, this criterion does not support the visual separation anymore. On the other side, if this criterion is applied, possibly dependent on the kind of the edges, it may have a positive impact on the readability and on the aspect-ratio (`UML_GRAPHDRAWING` via `GDR_DRAWING_SIZE`). If this criterion is enabled, the sequence of edges reaching the same side of the nodes connected to the same join bar may be considered as illustrated in Figure 3.8 to increase

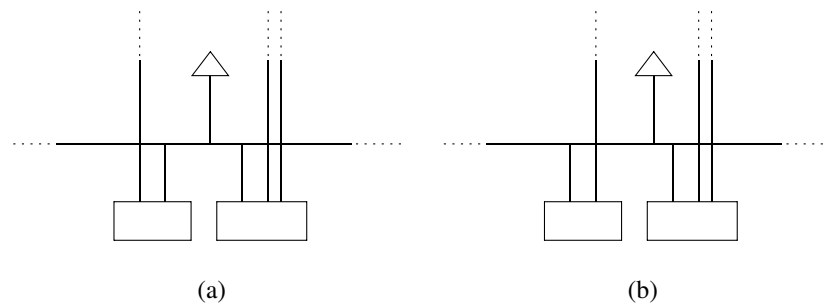


Figure 3.8: As illustration only: (a) arbitrary sequences (b) ordered sequences at joins.

regularity and therefore readability.

If enabled, the priority of `UML_JOIN` is the lowest in the group of hierarchy criteria, because `UML_JOIN` has influence on the display of hierarchies.

The question remains, if all hierarchical edges (of one kind, e.g., inheritance, aggregations, etc.) should either be drawn in shared or separate target style to gain uniformity, or, if, similar to `UML_HIERARCHY`, certain edges should be selected to appear in shared or separate target style. This might also be answered by an appropriate user study.

- `UML_SIZE_NODES`: Because rules to determine the size of nodes are not explicitly given in the UML, we also can take scaling according to some reasons into account. Thereby, the visible size of a node is used to reflect the magnitude, the importance or the complexity of the node via `HCI_MAGNITUDE`, `SE_CLASS` and `SE_PACKAGE`. This is a realization of `GDC_IMPORTANCE` or `GDC_NODE_SYMBOLS` with respect to software engineering and a partial implementation of `SV_POLYMETRIC`. This criterion receives the lowest priority in the group of node criteria because it is an optional rule from outside UML but influences the display of nodes.

- **UML\_SIZE\_EDGES:** In HCI, the line thickness is used to express the magnitude or the importance of an edge (HCI\_MAGNITUDE). A similar principle to SV\_POLYMETRIC might be applied to implement SE\_RELATIONS. Like the size of nodes, the thickness of edges is not explicitly specified in the UML, but, as mentioned along with SE\_RELATIONS this might require changes to the UML itself.

This criterion receives the lowest priority in the group of edge criteria because it is an optional rule from outside UML but influences the display of edges.

- **UML\_EDGE\_CROSSING\_SYMBOL:** Mark edge crossings by a small semicircular jog as it is usual in electrical circuit diagrams [OMG 2003c, p. 3-69]. This indicates that the paths do intersect.

UML\_EDGE\_CROSSING\_SYMBOL does not change the arrangement but is a criterion based on the UML specification. Therefore, its priority is greater than UML\_GRAPHDRAWING but it acts as the UML criterion with lowest priority.

Both UML based criteria, UML\_JOIN and UML\_EDGE\_CROSSING\_SYMBOL, appear in the UML specification as presentation options. Therefore, we have made a decision for separate target style and no crossing symbol in our mandatory UML specific criteria. From the personal point of view, one might see a higher relevance, e.g., in UML\_JOIN and prefer shared target style over separate target style as the default. This is left to user decision to be expressed by selecting appropriate options of the layout algorithms according to personal preferences.

Obviously, applying UML\_COUPLING, UML\_SIZE\_NODES or UML\_SIZE\_EDGES has a negative impact on GDR\_DRAWING\_SIZE, GDR\_NODE\_DISTRIBUTION, GDR\_NODES\_EDGES, GDR\_BALANCE\_EDGES and GDR\_CONTOUR\_DIFFERENCE.

### User Defined Aesthetic Constraints

In the description of REQ\_USER\_OPTIONS it was mentioned that certain user defined constraints might be appropriate to represent hints to the layout algorithm based on facts which cannot or should not be resolved from software engineering diagrams. In contradiction to the mandatory and optional UML specific criteria introduced above, user defined constraints are intended as hints which might be respected if no conflicts arise from their application.

- **UML\_CONSTRAINT\_SEQUENCE:** UML does not impose sequences on the display of nodes or edges in a diagram. In certain situations, e.g., if elements on the same level can emphasize an important sequence in the execution, it might be appropriate to display them in a given sequence. Thereby, aspects of the dynamic model might be reflected. Yet, to enable this feature, somehow additional information on sequences has to be provided.

This rule is related to HCI\_NODES\_SEQUENCE.

- **UML\_CONSTRAINT\_VICINITY:** Beside specifying visible edges according to the UML specification, hidden relations can be used also to enforce the position or the vicinity of several nodes, e.g., of comments or disconnected classes.

This rule is an aspect of UML\_SPATIAL.

Both optional constraints may help in generating a drawing which carries more but also unstandardized information. Therefore, to not disturb the requirements induced by the UML specification, we have to assign them to the lowest priority level. UML\_CONSTRAINT\_SEQUENCE and GDR\_EDGE\_CROSS as well as UML\_CONSTRAINT\_VICINITY and UML\_HIERARCHY, if hierarchical relations are used to realize aspects of UML\_CONSTRAINT\_VICINITY, can induce conflicts. Therefore, UML\_CONSTRAINT\_SEQUENCE and UML\_CONSTRAINT\_VICINITY are assigned to the same priority level than UML\_GRAPHDRAWING and concrete precedences are left to the preferences of the user.

priority	rule	restrictions for realization
6.4	UML_HIERARCHY	
6.3	UML_SPATIAL	not enforced on (sub)trees
6.2	UML_COUPLING	
6.1	UML_SEMANTIC_CLUSTERS	
6.1	UML_MEDIAN	
6.0	UML_JOIN	deferred to future work
5.4	UML_NODES	
5.3	UML_CONTAINER	tab conditions not guaranteed
5.2	UML_CLASS	
5.1	UML_CENTER	
5.0	UML_SIZE_NODES	
4.6	UML_EDGES	not guaranteed
4.5	UML_ASSOCIATIONCLASSES	
4.4	UML_HYPEREDGES	
4.3	UML_REFLECTIVE	at corners of classes only
4.2	UML_ADORNMENTS	predefined relative text positions
4.1	UML_COMMENTS	not complete
4.0	UML_SIZE_EDGES	via plug-in mechanism, no default plug-in
3	UML_DISCONNECTED	
2	UML_EDGE_CROSSING_SYMBOL	
1	UML_CONSTRAINT_SEQUENCE	deferred to future work
1	UML_CONSTRAINT_VICINITY	deferred to future work
1	UML_GRAPHDRAWING	GDR_EDGE_CROSS, GDR_MIN_BENDS GDR_MIN_EDGES are considered

Table 3.3: Priority levels of all UML specific criteria and restrictions for our concrete realization.

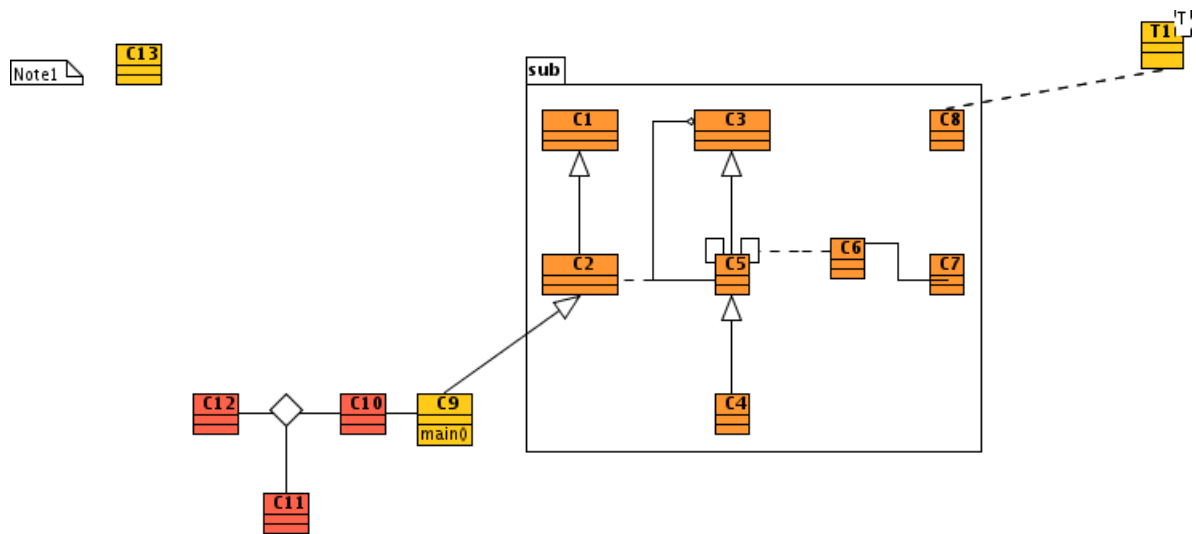


Figure 3.9: Figure 3.6 drawn by *SugiBib*: Containment, inheritance edges and dependencies were considered as pseudo-hierarchy. Comments are currently not fully supported. In this figure, T1 was not placed at a median position over C8 due to several area restrictions at clusters.

### 3.3.7 Aesthetic Conclusions

Essential to algorithmic graph layout is a set of rules that encode layout objectives. How these rules are related to inferences drawn from the graph by human observers is a largely unexplored issue. Thus, success or failure by algorithmic standards is only uncertainly related to perceptual effectiveness of the resulting layout.

[Dengler and Cowan 1998]

In this section we made an exhaustive study on aesthetics for general diagrams and UML class diagrams. We have collected 26 structural and 11 semantic rules from graph drawing literature, 11 principles from HCI, 9 issues on software design and 3 requirements from software visualization. This information was then compiled into a unique set of 16 mandatory, 5 optional and 2 facultative user related rules for UML class diagrams.

In this section we will first enumerate, which criteria will be realized by our concrete implementation. Then we will review other approaches to UML specific layout rules, highlight relations between software design and diagrams and finally discuss issues on validating UML specific layout criteria.

#### A Concrete Realization

The complete list of our UML specific rules sorted according to individual priorities is shown in Table 3.3. REQ\_COMPLETE\_UML as well as the priorities assigned to the individual rules can act as a primary guideline for a realization. Unfortunately, due to time restrictions, it was not

possible to consider all UML specific rules for our prototype. Therefore, we will limit ourselves to the rules and their individual restrictions mentioned in Table 3.3.

As an exception to the guideline on priorities, the optional rules `UML_SEMANTIC_CLUSTERS` and `UML_SIZE_NODES` (and thereby the basic mechanisms of `UML_SIZE_EDGES`) were realized as an experiment for [Eichelberger 2002a; Eichelberger 2003]. According to `REQ_COMPLETE_UML` we focused on proximity of model elements instead of `UML_EDGES`. Furthermore, we deferred the presentation option `UML_JOIN` as well as the user constraints `UML_CONSTRAINT_SEQUENCE` and `UML_CONSTRAINT_VICINITY` to future work.

Even if our implementation is currently evolving and several aspects have not implemented or the implementation was not finished so far, the automatic drawing of Figure 3.6 displayed in Figure 3.9 appears to be promising.

As introduced in Section 2.2.2 and discussed in Section 2.2.3, several basic algorithms can be tailored for UML class diagrams. Due to the high priority of `UML_HIERARCHY`, currently the hierarchical Sugiyama algorithm appears to be the best choice. By extending the basic Sugiyama algorithm, e.g., also nesting of nodes (`UML_SEMANTIC_CLUSTERS`) can be considered. Other basic graph drawing algorithms may be configured by constraints to produce hierarchical drawings as well. But in an application domain, in which further drawing rules like our aesthetic principles define the readability of the result drawing, a basic algorithm ensuring the most relevant aesthetic principles seems to be more appropriate.

## Related Work

In [Andersson 1998], the layout features of the CASE tool Rational Rose<sup>8</sup> were discussed but unfortunately that paper is more an overview on the application of general graph drawing algorithms to the problem of drawing class diagrams than an analysis of tool specific issues. The following aesthetic criteria were listed, but neither related to the algorithms nor to the layout mechanism implemented in Rational Rose: `GDR_EDGE_CROSS`, `GDR_SYMMETRY`, `GDR_NODE_DISTRIBUTION`, `GDR_FLOW` (on generalizations), `GDR_MIN_BENDS`, `GDR_DRAWING_SIZE`, `GDR_MIN_EDGES` and `GDR_OVERLAP`.

Three studies on user preferences for UML class diagrams have been described in [Purchase et al. 2001a; Purchase et al. 2001b; Purchase et al. 2002]. The subjects to be investigated in all studies were undergraduate students. The following general graph drawing rules have been taken into account in [Purchase et al. 2001b]: `GDR_MIN_BENDS`, `GDR_NODE_DISTRIBUTION`, `GDR_UNIFORM_LENGTHS`, `GDR_FLOW`, `GDR_ORTHOGONALITY`, `GDR_MIN_EDGES` and `GDR_SYMMETRY`. A different set has been analyzed in [Purchase et al. 2001a]: `GDR_MIN_BENDS`, `GDR_MIN_EDGES`, `GDR_ORTHOGONALITY`, `GDR_DRAWING_SIZE`, `GDR_LABELS_DIRECTION` and `GDR_FONTS`.

Referred as secondary notation features, two additional rules have been investigated in [Purchase et al. 2001a]:

---

<sup>8</sup><http://www.ibm.com/software/rational/>

- **OTHER\_JOIN\_INHERITANCE**: Apply shared target style for inheritance relations<sup>9</sup> only as a partly realization of the presentation option specified in the UML.
- **OTHER\_DIRECTIONAL**: Arcs should be labeled with two relationship labels and direction indicators, rather than one. This might be an improvement to the UML notation, especially if associations span over a long distance or over different levels, if present.

The summarized results of that work show that the common traditional graph drawing criteria are neither sufficient nor appropriate to be applied without considering further semantical issues to diagrams, in which readability highly relates to the semantic of the diagram rather than to syntactical features only. In [Purchase et al. 2001b] the evaluation produced no significant results or they were difficult to interpret reasonably and consistently.

For most algorithms tailored to UML class diagrams, aesthetic issues are neither described nor analyzed. But in both topology-shape-metrics approaches from the graph drawing community described in Section 2.2.3 the aesthetic features of the algorithm were mentioned and relations to the individual algorithmic steps were given. Despite the result of user studies, both highly relate to traditional graph drawing rules and realize only parts of the UML specification.

GoVisual [Gutwenger et al. 2003a] was implemented with respect to the following aesthetic principles: **GDR\_EDGE\_CROSS**, **GDR\_MIN\_BENDS**, **GDR\_LABELS\_DIRECTION**, **GDR\_HIERARCHY** (on generalizations) and **OTHER\_JOIN\_INHERITANCE**. Furthermore, two new rules were defined:

- **OTHER\_HIERARCHY\_NESTING**: Avoid the nesting of class hierarchies, i.e., a class hierarchy is not enclosed by a circle in the undirected sense of arcs of a different hierarchy
- **OTHER\_COLORS**: The various class hierarchies are marked by different colors and the generalizations also highlighted by color.

As argued in Section 2.1.3, the use of colors in UML is not specified prior to UML version 2.0. Therefore, the use of colors to highlight hierarchies, package containment (a non-standard option in our implementation) or other semantical groups of model elements depends on the features of the tool and on the preferences of the user. In general, the use of colors is currently a proprietary option for drawing UML diagrams.

In *jarInspector/yWorksUML* [Eiglsperger 2003], the class diagram layout problem was defined as an embedding according to the following aesthetic criteria: **GDR\_OVERLAP**, **GDR\_FLOW**, **OTHER\_JOIN\_INHERITANCE**, **GDR\_EDGE\_CROSS**, **GDR\_MIN\_BENDS**, **GDR\_MIN\_EDGES** and **GDR\_DRAWING\_SIZE**.

Beside user studies and concrete layout algorithms, mostly reusing traditional aesthetic rules, to our knowledge so far only little work has been done on a complete set of aesthetic criteria for UML class diagrams. The following rules for UML class diagrams have been listed in [Bernhart 2001]:

---

<sup>9</sup>At a first glance, this criterion seems to repeat **UML\_JOIN**. In fact, **OTHER\_JOIN\_INHERITANCE** is a partial realization of the presentation options defined by UML. This restricted version is often mentioned in literature. Therefore, an own name for that criterion was introduced here.

- Clusters should have an ideal size of  $7.5 \times 7.5\text{cm}$ , the number of the cluster members is restricted to 4 (where it does not become clear if the author means members or types of members) and non-cluster diagram elements are mapped to clusters having no contained elements to be treated equally. The clusters should be positioned according to the number of contained elements, where the cluster having the highest number of members should be positioned in the upper left corner while the cluster having the largest degree should be positioned in the center of the drawing.
- The number of elements in user-defined clusters should be inversely proportional to the relevance of the contents of the clusters.
- Cluster relations should be drawn in orthogonal style (GS\_STRAIGHTLINE).
- Compositions, aggregations and generalizations should be drawn hierarchically (GDR\_HIERARCHY). Additionally, they should be drawn in joined orthogonal style but they might be drawn directly when the angle of the edge respecting the horizon is between 25-30 degrees depending on the preference of the user.
- The number of edge crossings should be minimized (GDR\_EDGE\_CROSS).
- The lengths of edges should be short (GDR\_MIN\_EDGES).
- The aspect ratio of the drawing (GDR\_DRAWING\_SIZE) is required to be 3:2, because it relates to the size of DIN A4. Furthermore, the drawing space should be partitioned into 6 horizontal and 4 vertical ranks. Obviously, this does not take into account other international paper sizes like legal, letter, DIN A3 or larger sizes, which also might be handled by a printer.

Compared with Section 2.1.2 and the UML specification, many elements like subsystems, association classes, comments, higher associations, constraint-hyperedges are not neither discussed in [Bernhart 2001] nor in the other work mentioned above. Furthermore, some rules obviously introduce illegal restrictions, some rules cannot be applied due to missing precision and some rules introduce contradictions.

To our knowledge, no other work on UML class diagram aesthetics compiled an exhaustive set of rules, which is founded on other disciplines involved in drawing diagrams and captures syntactical as well as semantical aspects defined by the UML. Therefore, as done before, it is valid to call our set of rules unique.

### Layout and Design

If a layout algorithm respects at least all of our mandatory UML specific criteria listed in Section 3.3.6, it should produce a readable diagram. According to Section 3.3.4 the following (incomplete) list of indicators can be seen as warnings to highlight design problems:

1. **Huge inheritance/aggregation hierarchies** (SE\_FORESTS and UML\_HIERARCHY) can simply be identified in a class diagram. As an example, the Java library (version 1.4.2)

consists of more than 4000 classes and the maximum height of the inheritance tree is 9 according to the rules in [Chidamber and Kemerer 1994].

2. **Many classes at the borders of a package, few classes in the center**, when UML\_COUPLING is enabled, imply coupling problems (UML\_SPATIAL).
3. **Many children in hierarchy relations** signal a lack in the class structure, but when refactoring these classes the inheritance hierarchy may grow (UML\_HIERARCHY).
4. **A high percentage of the classes occupying relatively large areas**, especially when UML\_SIZE\_NODES is enabled: There might be a problem with the assignment of responsibilities to that classes and a problem of class complexity. This also affects GDR\_DRAWING\_SIZE in UML\_GRAPHDRAWING.
5. **Many inter-package-relations, fewer intra-package-relations**: The classes of such a package provide more services to classes outside that package than to the members of that package. This is a problem of coupling and can be identified in a class diagram, due to UML\_SEMANTIC\_CLUSTERS, but especially via UML\_SPATIAL when UML\_COUPLING is activated.
6. **A class with a high number of outgoing relations** indicates that it depends too much on other classes. This again is a problem of assigning responsibilities to a set of classes. It is partially visible via UML\_HIERARCHY, especially if container relations like aggregations, compositions or associations are member of the pseudo-hierarchy.
7. **Cross-relations between independent trees of the pseudo-hierarchy** indicate a low use of common services or attributes in top level classes (UML\_HIERARCHY and UML\_SPATIAL).
8. **Relations of a class cross the same package border(s) in direction of two different hierarchical layers** indicate an inappropriate modularization. Such a class might be refactored to act as a member of the connected package.
9. **Missing or inappropriate hierarchies** appear in hierarchical layout style as a one-layer layout. Figure 3.10 depicts two variants of the same class diagram, drawn in hierarchical fashion as one single layer in (a) and in a non-hierarchical style in (b). On the one side, Figure 3.10 (b) appears to be more appropriate according to traditional graph drawing aesthetics. By placing C1 below C2, a not intended (connected) hierarchical level is introduced in contradiction to HCI\_HIERARCHY. In a larger diagram, this may lead to a number of hard mental operations when the user searches for a non-existing hierarchy. On the other side, the hierarchical style, as depicted in Figure 3.10 (a), can indicate a missing or inappropriate hierarchy. This also reflects the priority of the UML criteria related to hierarchy over those of UML\_GRAPHDRAWING.
10. **Empty classes** are usually a design problem and should not occur except for marker interfaces, utility classes or empty classes in incremental design, which can be marked



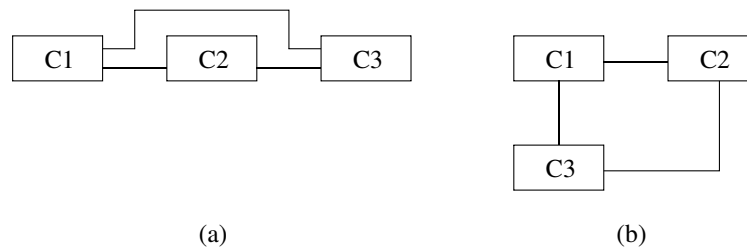


Figure 3.10: Layout of a class diagram, which does not admit a hierarchy, (a) applying the hierarchical layout approach, (b) using a non-hierarchical layout approach like the topology-shape-metrics method.

for example by ellipses. These exceptions were discussed along with `SE_CLASS` and `SE_DISCONNECTED` in Section 3.3.4.

If only layout issues like those mentioned in Section 3.3.6 are respected to measure the quality of a diagram, the design influences the layout, but a bad design does not automatically admit a bad drawing. Hence, a bad design does not automatically lead to lower metric values in the layout metrics. If there would be commonly agreed limits (always dependent on concrete project sizes like the overall number of classes, modules, etc.) for the design metrics listed in Section 3.3.4, the indicators discussed in this section might be used to judge the design quality of a diagram. Implicitly, all these indicators also affect the area occupied by the whole drawing or its aspect ratio. Therefore a better design would probably result in a smaller drawing area and in a better layout metric value [Eichelberger 2003]. Hence, there is a relation between quality and layout which is expressed in certain layout situations: The size of the drawing compared with alternative designs and the values of a composed design metric or a combined layout-design metric appear as an appropriate measurement technique. Applying a non-UML marking technique, like shading or coloring to highlight parts of the drawing according to the design indicators, would spot model elements and relations to be redesigned. Alternatively, classes could be flagged by additional (decorative) stereotypes like the one described for `SE_CLASS`.

## Validation

So far only little work has been done on validating aesthetics in graph drawing and related disciplines. To our knowledge, [Batini et al. 1985] was the first work which related graph drawing with aesthetic principles based on issues of cognitive psychology. In the following time there was little work in that field. On general graphs, Purchase started in 1996 with different user studies [Purchase et al. 1996; Purchase 1997; Purchase et al. 1997; Purchase 1998; Purchase 2000]. Specific to UML class diagrams only the studies described in [Purchase et al. 2001a; Purchase et al. 2001b; Purchase et al. 2002] were published. As regretted in [Ware et al. 2002], much more experimental work in this area could be done: More controlled experiments and a greater range of aesthetics were suggested.

Unfortunately, our set of UML specific rules is not validated by a user study. Respecting the large

---

degree of freedom in drawing a class diagram and the number of rules presented in Section 3.3.6, a user study and its evaluation would be a much harder task than the studies mentioned so far. Furthermore, we are of the opinion that a user study should not rely on students only. Seventy student volunteers, which were not generally proficient with UML, participated in the study described in [Purchase et al. 2001a]. To make the study subjects familiar with UML, in [Purchase et al. 2001b] a tutorial on UML class diagrams and notation was made available to the students as an introduction.

Even when teaching students more than the basic issues of the different diagrams in UML, usually much knowledge and experience in general software engineering is missing. Drawing a UML class diagram requires more than producing a fancy figure. It requires to be able to understand the problem to be specified, to select the elements appropriate to the specific solution and to distinguish between (abstract) design and (concrete) implementation. Similar experiences on other kinds of diagrams were described in [Petre 1995].

Therefore, we believe that if it is possible to design a user study and to be ready to evaluate the results respecting the syntactical and semantical issues of UML class diagrams, software engineers and practitioners should be invited to act as subjects of the study. This can be seen as the real world task mentioned as a future tendency in [Purchase 2004]. From personal communication, we know that many colleagues would be interested in participating in such a study. Of course, as a control group, students after a beginners course and/or after a course for advanced students ought to be considered, too.

Additionally, for analyzing user preferences of UML class diagrams with respect to the different features and our aesthetic criteria, at least tool support based on CASE tool file formats or repositories is desirable. We will discuss the foundations of such a program replacing rulers and manual tools as an application of our framework in Section 5.1.



# 4 The Layout Algorithm

---

After introducing UML class diagrams, basic techniques from graph drawing and a general functional specification for drawing UML class diagrams in the last chapters, we will now outline basic algorithmic and architectural issues of our layout algorithm and its implementation. This will prepare the description of details on the algorithm in the remaining chapter.

We will first introduce the process flow of our layout algorithm in Section 4.1 and the graph model, a mapping of UML class diagrams to graphs, in Section 4.2. Then the graph model is translated into appropriate definitions to be used by the pseudo-code algorithms, which will be used to explain the algorithmic details. In the 7 following sections, the macro steps and the assigned algorithmic steps will be described in detail. At the end of each section, a short conclusion will be given, in which also briefly the theoretical complexity of the macro step will be approximated. A final section will review the algorithmic issues of this chapter.

## 4.1 *SugiBib* – Just Another Hierarchical Algorithm?

I do not repeat my tactics but rearrange them to circumstances in an infinite variety of ways.

Sun Tzu

From our discussion in Section 3.3.6 we know that, respecting different viewpoints, a hierarchy determined according to the user's needs, induces the basic structure of the layout result. Combining this fact with the basic requirements from Section 3.1, it is suggested to tailor an algorithmic graph drawing approach instead of a declarative approach. More precisely, because it is the only basic algorithm which directly supports hierarchical edges, we will concentrate on the hierarchical approach. Instead of starting with a completely new algorithm, we will take the Sugiyama algorithm as foundation and consider further experience from the "Seemann algorithm" outlined in [Seemann 1997]. But we have to extensively modify the known algorithms to meet our requirements.

The basic algorithm in [Seemann 1997] consists of the following steps:

1. Remove reflective edges and insert them as attributes into the connected classes.

2. Compute the inheritance/realization subgraph and guarantee a virtual root in the case of a forest.
3. Remove association edges and nodes which are connected by association edges only.
4. Calculate the rank assignment.
5. Reduce the number of edge crossings.
6. Perform the “incremental extension” by successively reinserting the nodes and edges removed in step 3 and assigning these nodes to ranks.
7. Compute the sizes of the nodes, calculate the positions of the nodes due to inheritance edges as described in [Sugiyama et al. 1981; Gansner et al. 1993], prepare the orthogonal drawing of the associations and calculate the positions of the edges.

By temporarily removing the parts of the graph which cannot be used as input of an usual hierarchical layout algorithm and by reinserting these elements at that point of time when the algorithm is able to work on that information, the hierarchical drawing idea of Sugiyama can be reused and applied to UML class diagrams. This algorithm invented the technique of incrementally extending the graph in step 6 by reinserting removed nodes and edges.

As mentioned in [Eiglsperger 2003, p. 1], the force directed and the hierarchical approach are extremely successful in practice, because they produce fairly good results, are extensible and it seems that both are easy to implement. Therefore, the choice of a hierarchical algorithm seems to be valid. Beside the fact that different sophisticated model elements, which might be used in a class diagrams, are missing<sup>1</sup>, there is some criticism on that approach.

- A hierarchical algorithm runs into problems if the input graph does not provide an appropriate set of hierarchical edges. In the worst case no hierarchical edges are present, because the extensive use of inheritance is discouraged in software engineering. Furthermore, as noted in [Eiglsperger 2003, p. 31], drawing inheritance hierarchies as main skeleton of a UML class diagrams requires that the user prefers GDR\_FLOW over all other criteria. This is true for the basic algorithm as described above, but as discussed in Section 3.3.6, inheritance relations are not the only kind of relations which can be taken into account when deducing an appropriate hierarchy. In particular, aggregations or compositions can be respected when determining the members of a pseudo-hierarchy, because these edges should be preferred over inheritance relations from the software engineering viewpoint. Furthermore, we may have to remove cycles by applying an appropriate algorithm, because such a pseudo-hierarchy, retrieved from semantic principles only, is not guaranteed to be an acyclic graph required as input by the rank assignment.
- Most traditional edge crossing minimization techniques described in [Sugiyama et al. 1981; Gansner et al. 1993] work on two adjacent layers. For n-layer drawings, often a

---

<sup>1</sup>Please remember that the algorithm was presented in 1997 and UML version 1.1 was released as an OMG standard in November 1997.

2-layer algorithm is applied by successively sweeping up and down the ranks of the hierarchy. This may lead to edge crossings which are avoidable from the perspective of the user. Usually, edge crossing minimization algorithms consider one type of edges only and therefore, the non-hierarchical edges are not considered. This may lead to additional avoidable edge crossings and further long-span edges will occur in the drawing. Hence, an extended crossing number calculation or more appropriate crossing reduction heuristics are needed. Then, the rank assignment can be extended by a simplified “incremental extension” to calculate the layer assignment of all nodes at once. Determining individual positions for the nodes to be reinserted as in the original “incremental extension” is not required anymore, because this is done implicitly by the edge crossing reduction.

- Reflective edges in [Seemann 1997] are handled by transforming them to attributes in the connected classes. This transformation changes the underlying model of the class diagram, but, as discussed in Section 2.1.3, this is not allowed to be done by a layout algorithm. Either we have to respect loops and cycles in the whole implementation or we have to hide these edges somehow from the main algorithm.

The first implementation of [Seemann 1997] was described in [Eichelberger 1999] as an instantiation of a generic graph drawing framework called *SugiBib*. Our extensions and the implementation of our algorithm will also rely on the basic of that work but on a completely revised version of the framework.

To illustrate the individual steps of our algorithm, a concrete example and the individual modifications performed by the algorithm will be displayed as incremental changes applied to Figure 4.1. Figure 4.1 shows the input graph created by an arbitrary mechanism from Figure 2.9. The figures derived from Figure 4.1 should be seen as an instructive help, not as a complete introduction into the graph model. Details on the graph model will be given in Section 4.2.

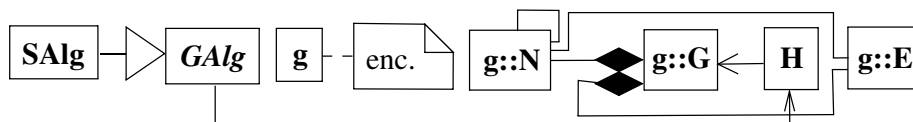


Figure 4.1: The input graph created by an arbitrary reading mechanism taking only the structural and semantical information in Figure 2.9 into account. Initially given coordinates for incremental layout are ignored.

In Figure 4.1 all names are shortened to obvious abbreviations due to space limitations<sup>2</sup>, the package is reduced to its base node and the containment is implicitly given by fully qualified names. According to an input convention of *SugiBib* to be described in Section 4.2, the association class is represented by an usual class splitting the connected association into two distinct edges. The association class H is internally flagged as an association class and the connecting line is not specified.

<sup>2</sup>Due to space limitations, hyperedges and disconnected nodes are not illustrated by the example.

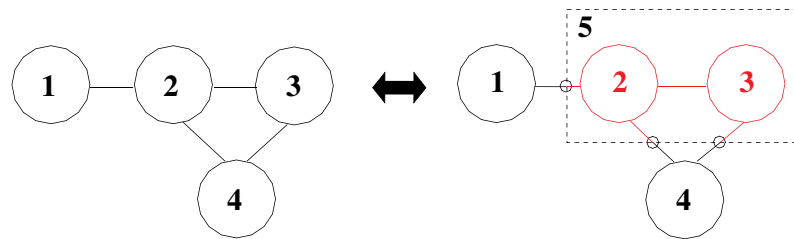


Figure 4.2: Transforming a set of nodes into a composite node and back to individual nodes.

Until layers and coordinates are not assigned by the algorithm, we will assume a simple 1-layer left-to-right drawing algorithm which produces the example drawings.

To guarantee the close vicinity of selected nodes, a set of nodes can be transformed into a *composite node* by mapping the edges of the individual nodes to the composite node and v.v. as depicted in Figure 4.2. Hence, association classes or comments, which have to be kept in a close vicinity with other nodes, can be treated by the implementation as usual nodes and expanded when neither the vicinity nor the following layout steps will be disturbed.

- S1: Prepare the input graph** and ensure consistency. The containment of nodes, given by fully qualified names, is additionally represented as invisible edges. Natural clusters like n-ary associations are retrieved and, as an optional operation, a relative scaling value can be attached to nodes and edges to prepare `UML_SIZE_NODES` or `UML_SIZE_EDGES`.
- S2: Order the nodes and edges** according to semantical issues based on fully qualified names. This step releases implicit dependencies on the sequence of definitions of the model elements between the input and the layout result. Furthermore, this step provides a simple structural and topological alignment to the mental map [Misue et al. 1995] of the user due to a deterministic normalization. Therefore, it basically supports `REQ_INCREMENTAL_ALGORITHM` and prepares `REQ_DETERMINISTIC_ALGORITHM`.
- S3: Identify a pseudo-hierarchy** by basic user preferences, heuristics or by considering a user defined hierarchy. This can simply be done by incorporating all edges belonging to a certain preselected group, e.g. inheritance edges, anchor relations, aggregations, etc. Identifying certain rules for heuristics, which automatically select edges for the pseudo-hierarchy according to the structure of the input graph, would require a detailed user study. Therefore, we mention this idea here only and will defer it to future work. Respecting a user specified hierarchy is then a tribute to interactive, diagram specific layout. Step S3 prepares `UML_HIERARCHY` and partly implements `REQ_USER_OPTIONS`.

In the example in Figure 4.3, two different pseudo-hierarchies are assumed. The diagrams marked by (a) are laid out using containment and inheritance edges as pseudo-hierarchy. The pseudo-hierarchy of the diagrams marked by (b) consist of containment, inheritance and composition edges. As illustration, visible hierarchical edges are marked by “h”.

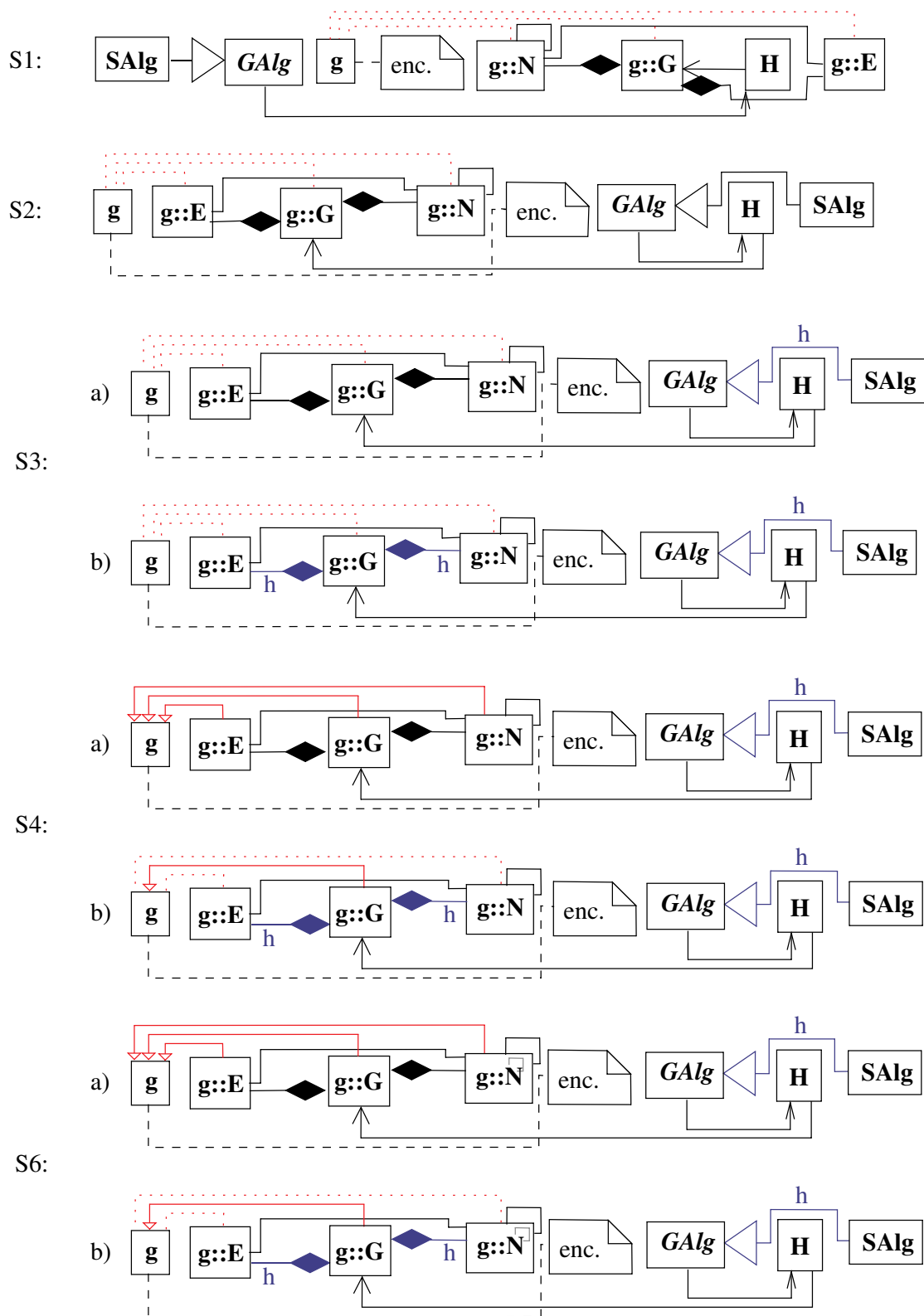


Figure 4.3: Step S1 to step S6 of the layout algorithm.



- S4: Insert containment relations of model elements as hierarchical edges.** To respect the containment hierarchy when calculating the rank assignment, we temporarily create hierarchical edges for containment relations whose nodes are not already connected by hierarchical edges. This step prepares `UML_SEMANTIC_CLUSTERS`.
- S5: Compress hyperedge connection nodes** to composite nodes. Hyperedges are simulated by connecting the hyperedge to two invisible nodes, which split the connected edges into pairs of edges. Two cases have to be considered: Hyperedges at hierarchical edges will be routed horizontally and by compressing the hyperedge connection nodes into composite nodes at either the start or the end, both connection nodes will appear somewhere between two ranks. Hyperedges at non-hierarchical nodes will be treated similar to hierarchical edges. When the hyperedge connects edges originating at the same node, a composite node will keep these three nodes in close vicinity. Thereby, optional semantical information to which node a hyperedge should be kept in vicinity can be considered. This step supports `UML_HYPEREDGES` and is not illustrated in Figure 4.3, because no hyperedges are present in the example.
- S6: Remove reflective associations** to simplify the implementation. The information of the reflective edges is stored in the connected node to be drawn later on. By now, the connected node is responsible for the size required by the reflective association, for further connections on that association and for the display of the edge in the final drawing. Because reflective associations may also be connected to association classes and the reflective association is physically removed from the graph in this step, these association classes are also handled in this step. In this case, the association class is encapsulated together with the node connected to the reflective association into a composite node. Existing composite nodes are reused. This step prepares `UML_REFLECTIVE` and `UML_ASSOCIATIONCLASSES`.
- S7: Compress association classes** by putting the association class and its connecting dashed edge into a composite node. Due to the input conventions, this step first explicitly inserts the dashed edge and a connecting node and encapsulates them then into a (existing) composite node. The connecting edge is not defined as hierarchical edge because this would force the association class into an other rank than the connecting node and the distance would accidentally be increased. Step S7 prepares `UML_ASSOCIATIONCLASSES`.
- S8: Compress annotations** and connected model elements. Similar to S7, certain composite nodes containing the annotation, the connected node and the dashed edge are created. An existing composite node, which contains the connected node, is reused. Even if our example might suggest that the composite node receives the size of all contained nodes that is not always true. Depending on the situation, member nodes acting as proxies for the required size can be specified. Currently, comments do not contribute their size to the extent of the composite node. This step prepares `UML_COMMENTS`.

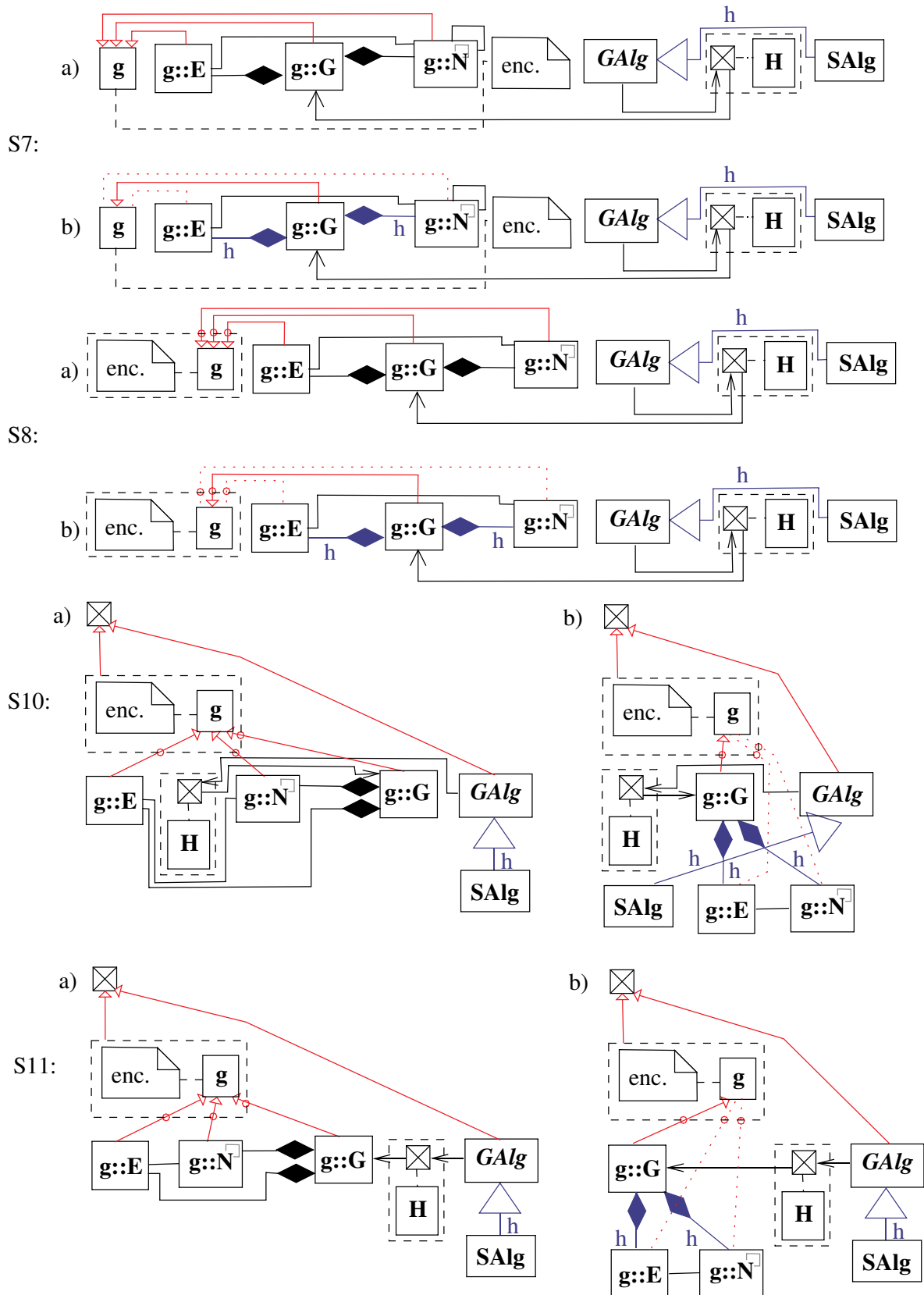


Figure 4.4: Step S7 to step S11 of the layout algorithm.

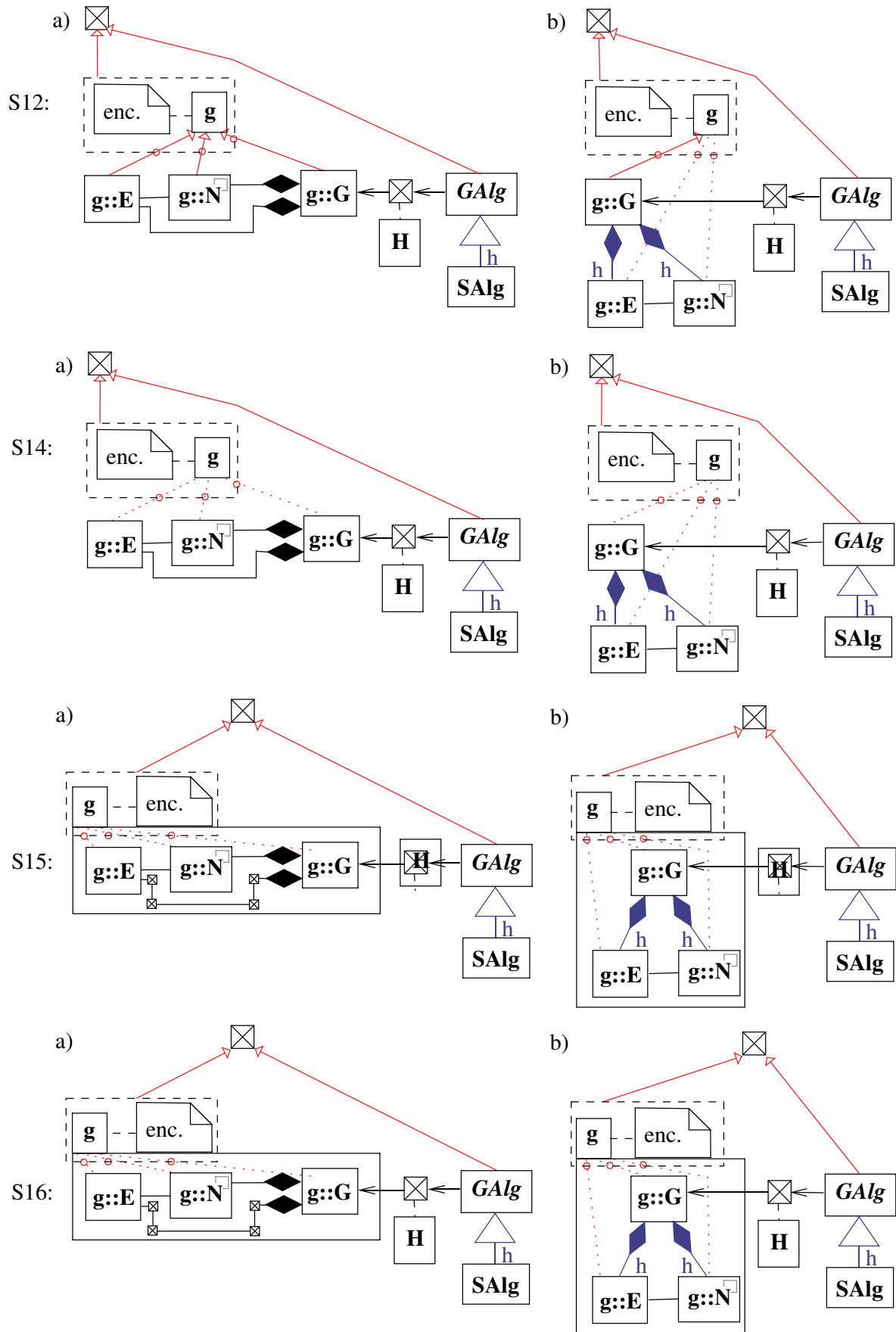


Figure 4.5: Step S12 to step S16 of the layout algorithm.

**S9: Remove disconnected nodes** not contained in packages. Usually, disconnected nodes can simply be reinserted after calculating the coordinates of all other nodes by respecting the shape of the result graph. If a disconnected node is member of a package, it must not be removed from the graph, because the rank assignment requires the node to compose the layered contents of all packages and the coordinates calculation needs the node to determine the extends of the package. This step prepares UML\_DISCONNECTED. Step S9 does not change the example graph, because no disconnected are present.

**S10:** Transform the hierarchically connected nodes into an acyclic subgraph and insert a virtual root, if required. Otherwise, this step and some of the following steps would not be able to handle a forest even if the subtrees are interconnected by non-hierarchical edges. Therefore it is required that, if the graph itself does not impose a unique root node, a dummy root node is inserted and connected by hierarchical edges to the roots of the subtrees. Then **calculate the rank assignment** for these nodes in one step. Calculate the layer positions of only non-hierarchically connected nodes in a second step (simplified “incremental extension”).

Correct and optimize the layered structure of the graph for UML class diagram layout, e.g., by ensuring several cluster specific rules. Then, replace long span edges by edge segments which connect nodes in adjacent ranks to prepare the routing of these edges in the coordinates assignment. Thereby, *dummy* nodes (also called virtual or hidden nodes) to connect the individual edge segments are inserted. The result of this step is a cluster-valid hierarchy, i.e. no overlapping clusters per rank must be present and similar cluster sequences in adjacent ranks are required.

This step realizes UML\_HIERARCHY via GS\_LAYERED and supports UML\_SEMANTIC\_CLUSTERS, UML\_CENTER, UML\_SPATIAL and UML\_COUPLING.

**S11: Reduce the number of edge crossings** respecting hierarchical as well as non-hierarchical edges, cluster and containment relations. Different heuristic approaches, each with its specific advantages and disadvantages can be taken into account.

By now the sequence of nodes in the individual ranks is kept stable. Depending on the concrete coordinates assignment, dummy nodes may be allowed to change their rank position even in coordinates assignment.

Step S11 respects UML\_SEMANTIC\_CLUSTERS, UML\_SPATIAL and UML\_GRAPHDRAWING. Furthermore it supports UML\_CENTER and it might partially ensure UML\_CONSTRAINT\_SEQUENCE.

**S12: Expand composite nodes for association classes.** The absolute position of the composite nodes association classes are contained in was determined by S10 and S11. The position of the association class relative to the connected association can be determined later. Hence, composite nodes for association classes are not required anymore. We have chosen to perform the expansion at this point of time, because the composite nodes for association classes must be expanded to individual nodes before concrete edge positions for

non-hierarchical edges are assigned and until then the individual nodes will not disturb the algorithm. UML\_ASSOCIATIONCLASSES is supported by this step.

**S13: Expand composite nodes for hyperedges.** Similar to step S12 the composite nodes containing hyperedges are expanded. This step supports UML\_HYPEREDGES.

Step S13 does not change the example diagram in Figure 4.5, because no hyperedges are present.

**S14: Remove containment information** inserted as edges in step S4, because the information is not needed to be represented as edges anymore.

**S15: Determine the sizes of nodes and edges and calculate the coordinates.** To respect adornments at edges, the area of the nodes is extended by an invisible outer node area. Contained model elements are treated in one step with the other nodes to respect non-hierarchical edges interrelating clusters and individual nodes. As mentioned in [Sander 1996b], most recursive bottom-up layout methods ignore that global connectivity.

An initial cluster-valid coordinates assignment is calculated. Then, with a priority to hierarchical and cluster relations, iteratively feasible coordinates for the nodes are determined. Afterwards, the result is augmented by adding dummy nodes for non-hierarchical edges to support GS\_POLYLINE on these edges. Thereby, hierarchical edges are always connected to the horizontal sides of the nodes, non-hierarchical edges to the vertical sides. Then the calculations for hierarchical and cluster relations are repeated to respect non-hierarchical edge chains spanning over multiple ranks. Finally, the non-hierarchical edges are orthogonalized.

The coordinates assignment is responsible for the realization of a lot of aesthetic principles: UML\_SEMANTIC\_CLUSTERS, UML\_MEDIAN, UML\_SPATIAL, UML\_CENTER, UML\_NODES, UML\_EDGES, UML\_ADORNMENTS, UML\_CLASS, UML\_CONTAINER and UML\_REFLECTIVE. Optionally it might realize UML\_JOIN, UML\_COUPLING, UML\_SIZE\_NODES and UML\_SIZE\_EDGES. As discussed above, dependent on UML\_JOIN it respects GS\_STRAIGHTLINE or GS\_ORTHOGONAL.

**S16: Augment the layout for association classes.** The dummy node, which simulates the hyperedge, has to be placed according to the positions of the edge. The association class is also positioned and the distances between adjacent layers may be adapted in this step. S16 finally realizes UML\_ASSOCIATIONCLASSES.

**S17: Calculate the layout of hyperedges** by considering all possible positions per hyperedge and by selecting the most appropriate one.

This is not shown in the example, because no hyperedges are present. S17 completes UML\_HYPEREDGES.

**S18: Expand annotations and augment the layout.** The size of the comments was not respected so far. Therefore, the connected nodes are analyzed for sufficient area in their vicinity with a preference to the border of the graph. Only if a comment cannot be placed

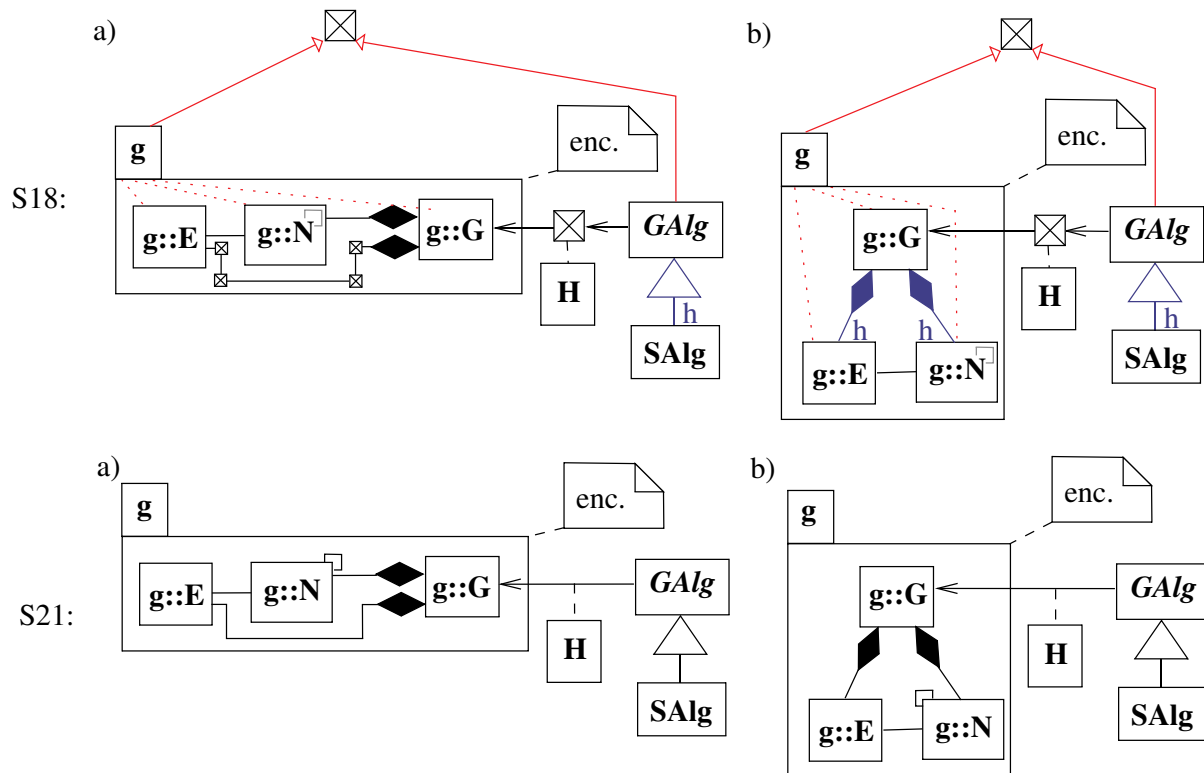


Figure 4.6: Step S18 to step S21 of the layout algorithm.

close to its connected node, the distances between adjacent layers are adjusted. This step finalizes the implementation of UML\_COMMENTS.

**S19: Re-integrate disconnected nodes** by searching for appropriate empty areas, preferably at the boundary of the drawing. This step completes UML\_DISCONNECTED.

In the example, the graph is not changed, because no disconnected nodes have to be processed.

**S20:** Optionally snap all nodes and edges to a specified **grid**. The step partially respects GS\_GRID and optionally fulfills REQ\_GRID.

For the example we assume that this option was not activated.

**S21: Create the result graph.** Further dummy nodes are inserted at logical bends, e.g. for reflective edges. Additionally, the coordinates and sizes of the elements are fixed to provide detailed information for communication with external tools (REQ\_IO). Optionally, the concrete position of edge crossings can be detected and additional information to realize UML\_EDGE\_CROSSING\_SYMBOL can be attached to the edges.

In Figure 4.6 the hidden nodes are omitted in the drawing.

To realize `UML_CONSTRAINT_VICINITY` another dedicated preparation step might be introduced before rank assignment.

According to our application domain, to automatically calculate the layout of UML class diagrams, we have extended a classical hierarchical algorithm, which consists of three steps, to an algorithm, which performs 21 steps. Dependent on the implementation, certain steps might be combined into a macro phase. In Table 4.1, several macro phases are introduced which will be used as underlying structure for the detailed description of the individual layout steps in Chapter 4. Hence, from this perspective, our layout algorithm comprises 6 steps and appears as a structural relative to the classical Sugiyama algorithm.

macro phase	aggregated steps	realized priority levels (Table 3.3)
preprocessing	S1-S9	4.5, 4.4, 4.3, 4.1
rank assignment	S10	6.4, 6.3, 6.2, 6.1, 5.1, 4.1
edge crossing reduction	S11	6.1, 5.1, 4.6, 4.5, 4.4, 4.1
intermediary processing	S12-S14	4.5, 4.4
coordinates assignment	S15	6.1, 6.0, 5.4, 5.3, 5.2, 5.1, 5.0, 4.6, 4.3, 4.2, 4.0
postprocessing	S16-S21	4.5, 4.4, 3, 2

Table 4.1: Macro steps of the *SugiBib* algorithm and aesthetics (as classified by priorities in Table 3.3) realized by the individual macro steps.

So far in this section, we have enumerated the individual steps of our layout algorithm for UML class diagrams. For each algorithmic step relations to our set of aesthetic criteria introduced in Section 3.3.6 were given. By comparing the criteria mentioned above with the complete list in Table 3.3, it can be shown that all rules selected for implementation will somehow be respected or realized by individual algorithmic steps. The general question remains, if our algorithm is capable of ensuring the priorities assigned to the UML specific rules.

We can point out that in the preprocessing macro phase, before executing the rank assignment in S10, various criteria which relate to hierarchy, containment and vicinity as well as basic issues of `REQ_INCREMENTAL_ALGORITHM` are prepared but no concrete positions are assigned. The sequence of these steps partly depends on dependencies arising from the valid encapsulation of nodes in composite nodes.

S10 determines the skeleton of the drawing by assigning abstract vertical positions and ensures criteria related to hierarchy and containment as well as to `UML_CENTER`. In S11, relative horizontal positions with respect to `GDR_EDGE_CROSS` are determined. Due to additional mechanisms, which, e.g., ensure cluster validity, basic edge crossing reduction algorithms are restricted. Hence, the higher priority of criteria related to hierarchy and containment as well as `UML_CENTER` over `GDR_EDGE_CROSS` can be realized. This is also true for the vicinity-related rules like `UML_ASSOCIATIONCLASSES`, because composite nodes avoid the cluttering of their members in S11. This may then lead to long association edges due to the restrictions introduced by composite nodes. Therefore, enhanced crossing reduction mechanisms considering containment and non-hierarchical relations are required.

In the intermediary processing macro phase, only technical transformations of the graph are

performed. The realization of the coordinates assignment S15 is then responsible for considering plenty rules and their individual priorities. Most of the rules specific to node interior can be ensured by a proper calculation of the node extents, overlappings of nodes can be avoided by a cluster-valid initialization of the node positions and by some basic node moving mechanisms. As mentioned in the description of S15, first coordinates are assigned considering hierarchical relations. Then non-hierarchical edges are processed and, finally, the graph is compacted by removing unused area.

Except of dummy nodes, the sequence of nodes determined by S11 is kept in all following steps. Hence, the priority sequence of our set of UML related criteria is ensured. The concrete realization of node-related criteria with a higher priority over edge-related criteria as well as individual priorities within one group of rules mainly depends on the (state of the) implementation and personal preferences.

Furthermore, the algorithm is designed to realize all basic requirements defined in Section 3.1, even if the realization of `REQ_USER_OPTIONS`, `REQ_DETERMINISTIC_ALGORITHM`, `REQ_PLATFORM`, `REQ_INCREMENTAL_ALGORITHM`, `REQ_ARCHITECTURE`, and `REQ_SPEED` mainly depend on the concrete implementation.

Therefore, we can conclude, that the algorithm introduced in this section is able to realize all of our requirements collected in Chapter 3.

## 4.2 Structural Conventions for Graphs

Conventionality is not morality. Self-righteousness is not religion. To attack the first is not to assail the last.

Charlotte Bronte (1816 – 1855)

To go more into detail, in this section we will describe the underlying diagram or graph model. As described in Section 4.1, the layout algorithm takes a UML class diagram given as a non-simple directed graph compliant to several rules as input, transforms that input graph to a non-simple directed composite graph to which coordinates are assigned and finally returns a graph which is able to draw itself onto a given graphics context. These three basic types of graph instances, input graph, intermediary graphs and output graph, will be described first. From the structural requirements drawn by these graph types, we will finally compose an object-oriented graph model which represents the core of the layout framework *SugiBib*. Therefore, some class or method names will occur in the description of the three graph instance types before the class diagram of the graph model realization will introduce them.

### Input Graphs

An input graph, representing an UML class diagram, is given in terms of visible nodes and edges. According to `UML_DISCONNECTED`, also invisible edges influencing the layout of the graph may occur. Specialized node types for classes, packages, n-ary associations, etc. as well as spe-



cialized edge types for generalizations, aggregations, dependencies, etc. are required.

As mentioned above and in the description of S1, we will take a directed non-simple graph as input, which also has to reflect nesting of nodes, if present. In general graphs, these relations might be given as additional edges of a certain type, as second graph containing the nesting tree or as external relations. In UML class diagrams this information is implicitly present via fully qualified names. To simplify arbitrary input mechanisms, we decided to calculate the nesting relations from the fully qualified names in S1. Hence, a concrete input mechanism specifies the input as a non-compound graph and simply assigns the available UML names to nodes. The compound itself, e.g. a package, is represented as a single node called the *base or parent node of a cluster*. Relations to compounds, should be connected to the base node of the compound.

According to the description of our layout algorithm, the rank assignment in S10 is the only part which may change the direction of some edges to gain an acyclic subgraph as input. Furthermore, S10 is (currently) the only part of the algorithm, in which the direction of the edges has major influence on the structure of the result. Hence, due to the requirements of the rank assignment, the main rule for specifying edge directions is that hierarchical edges must be directed from the root to the leaves, i.e. generalizations have to be specified in opposite direction as drawn in UML class diagrams. The other edges should be given according to their navigational direction. In the case of undirected or bidirected relations, the direction in the input graph is arbitrary, because it will be normalized in S1.

As another convention, we tried to avoid the use of too many different types of nodes and edges in the input graph. Therefore, only in the case of hyperedges, nodes occupying no area for the simulation of the hyper edge will be required. We will call these type of node `NullNode`. Concrete examples are shown in Figure 4.7. Due to successive conversion while executing the layout algorithm, the UML notation will appear instead of the input structures in the layout result.

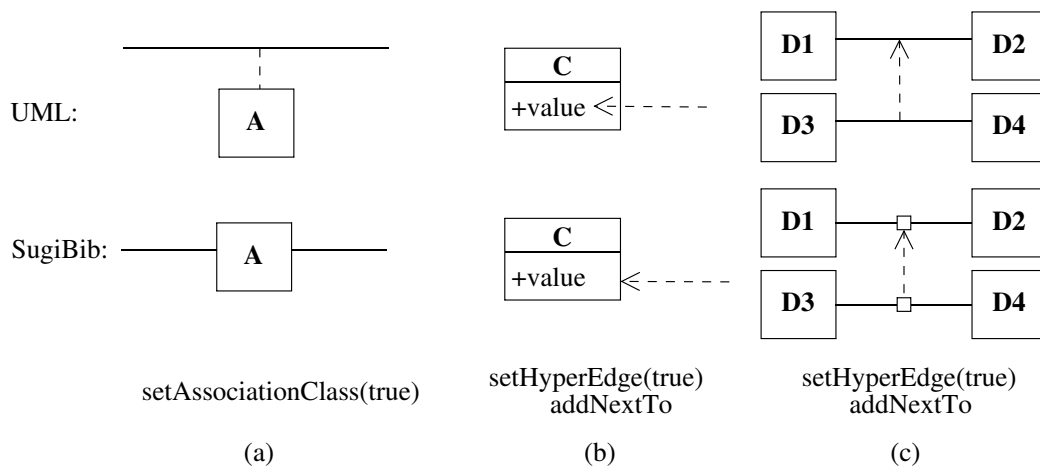


Figure 4.7: Structural rules on input graphs for (a) association classes and classifiers, (b) edges to a compartment entry and (c) hyper edges for xor-constrains or generalizations have to be specified by two instances of `NullNode`. The appropriate UML structures will appear in the result graph due to successive conversion while executing the layout algorithm.

Figure 4.7 (a) depicts that an association class is represented as an usual class, but the association class flag has to be set. The edge, to which the association class is attached to, is separated by the association class, i.e. one part of the original edge is an incoming edge to A, the other is an outgoing edge. The same holds for comments.

If a compartment entry should be referenced from outside as shown in Figure 4.7 (b), the edge is connected to the node containing the compartment entry, but the compartment entry is specified as original target using the method `addNextTo`. Additionally, the hyper edge flag has to be activated. In the case of an hyperedge, as shown in Figure 4.7 (c), both connected edges are separated by a `NullNode`, the hyperedge connects both `NullNodes` and the hyperedge flag should be activated. Additionally, if semantical reasons require the vicinity to D1, D2, D3 or D3, the appropriate nodes can be specified by `addNextTo`.

Collaborations, e.g. used for the UML design pattern notation, and n-ary nodes are specified by creating an appropriate node for the central element and by linking this node with individual edges to the connected model elements. Currently “lollies” are represented by an own edge type. Multiple edges to the same logical lolly will be unified by the algorithm.

### Intermediary Graphs

As shown in Section 4.1, the input graph is successively transformed towards the result graph. Thereby, temporarily deletable edges and nodes might occur and several dummy nodes may be inserted. Furthermore, in the preprocessing phase, composite nodes are created, existing nodes are transferred into composite nodes and input structures as shown in Figure 4.7 (a) are replaced. In S1 the nesting relationships, additional information which is not represented as individual edges, are initialized. This nesting information is internally used as a secondary access path to cluster containment data.

While executing the layout algorithm, nodes and edges may receive temporary data as well as information required to draw the result. Furthermore, the representation of clusters changes: In the rank assignment S10, hierarchical relations connecting to a cluster from hierarchical children have to be reconnected. A cluster border node is created and nested into the cluster, the external relations of that cluster are reconnected to the border node and the cluster border node is assigned to the bottom rank of the cluster. At the beginning of the coordinates assignment, clusters are bounded by cluster border nodes as depicted in Figure 4.8. An existing cluster border node from the rank assignment step is reused. The cluster border nodes are interconnected by cluster edges and nested into the cluster by appropriate relations to the cluster base node. To enforce spatial distribution of the contained elements, e.g., to emphasize coupling (UML\_SPATIAL) or to realize subsystems, a cluster can be partitioned by cluster separator nodes.

### Output Graphs

After all layout operations have been completed, the result graph is generated. To provide as much layout data as possible to mechanisms external to the layout algorithm like data exporters, even internally handled edges like reflective associations are explicitly transformed into graph elements. Furthermore, the `layout` method of all information instances is called to fix the posi-

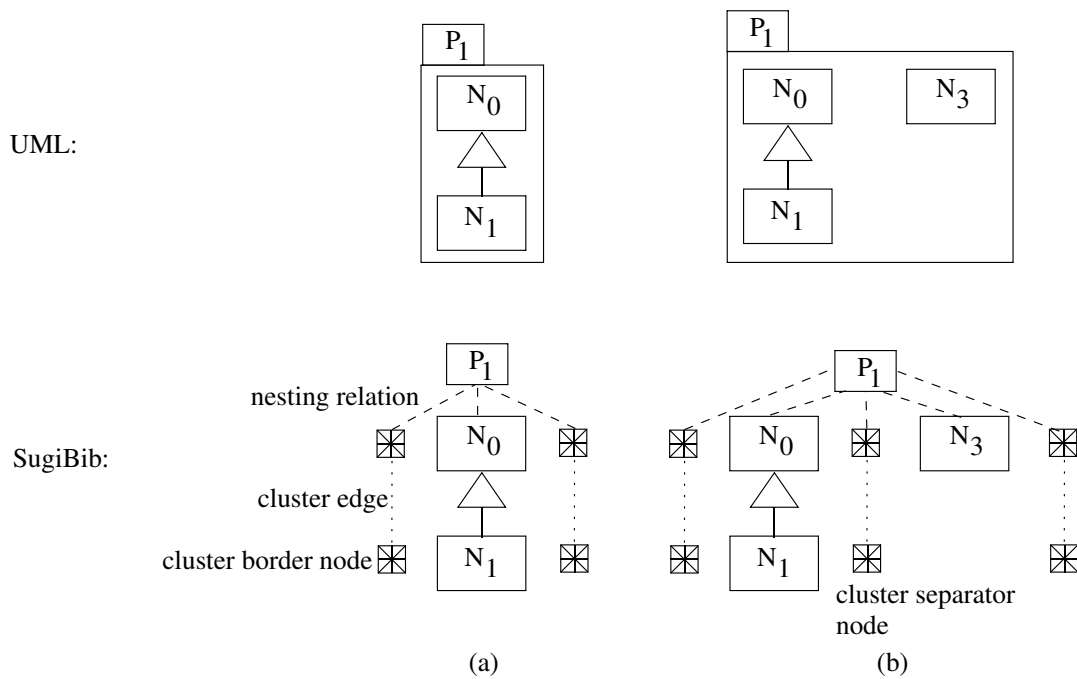


Figure 4.8: (a) simple cluster bounded by cluster border nodes, (b) partitioned cluster divided into two parts by cluster separator nodes.

tions of all attached elements, e.g. the position of association names or other textual adornments not enclosed in nodes. Cluster border nodes and cluster separator nodes are removed from the graph. Finally, the output graph is locked to prevent (accidental) writing access to the graph elements and their individual data, e.g. the type of a node should not be changed after the result graph was created.

## Graph Model

Figure 4.9 shows a simplified view of the graph model implemented in *SugiBib*. Packages and therefore different layers of implementation are omitted, class and interface names are shortened and some signatures are located not at the same place than in the implementation. Therefore, Figure 4.9 is to be seen rather as an illustrative introduction than as an exhaustive specification diagram.

A graph consists of its nodes and edges. A node knows its edges, i.e. its incoming and its outgoing edges. An edge knows its start and its end node. This allows multiple navigation paths through a given graph and requires different consistency issues to be respected. These issues are handled automatically by the graph, therefore, modifying accessors for nodes and edges should be used carefully.

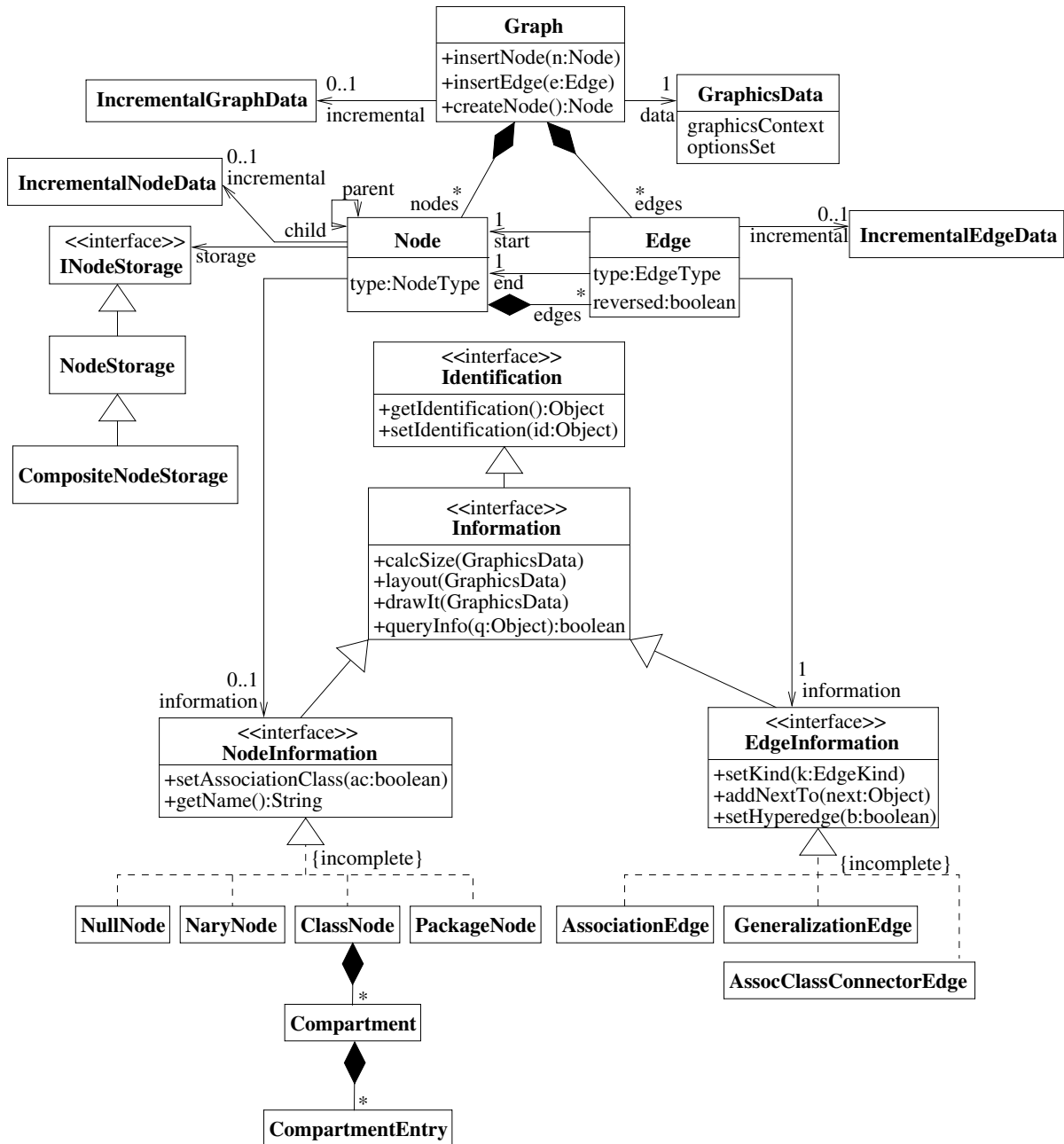


Figure 4.9: Simplified view of the graph model of *SugiBib*.

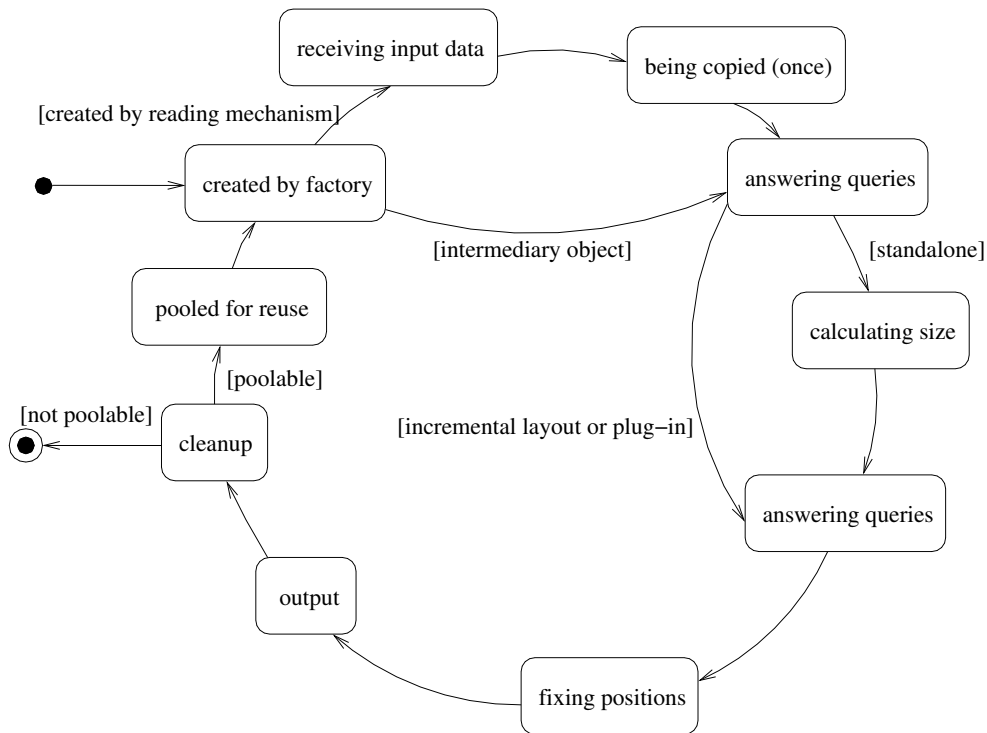


Figure 4.10: Information object life cycle.

Obviously, only one type of nodes or edges is specified in Figure 4.9, respectively, but to each node or edge an individual information object (Information) can be attached<sup>3</sup>. To allow the reuse of the basic classes like nodes and edges in other application domains, domain specific information is intended to be implemented in concrete information classes. In particular, the display of an individual node or edge is delegated to these information instances. As shown in the lower part of Figure 4.9, specific information classes represent the elements to be used in a UML class diagram.

Beside application domain specific types, nodes and edges may also be deletable, mark distinct parts of a cluster or may appear as dummy nodes. These (internal) algorithm specific types are directly stored in the nodes or edges, respectively.

To realize the composite nodes introduced by Figure 4.2 in Section 4.1, a storage strategy is assigned to each node. In contradiction to the usual `NodeStorage` strategy, the `CompositeNodeStorage` strategy automatically handles packing and unpacking nodes and the consistency of the connected edges.

Intended for incremental layout but also to inject graph element specific user preferences into the layout algorithm, graphs, nodes and edges may have attached instances (e.g.,

<sup>3</sup>In input graphs, which specify situations as shown in Figure 4.7 (a), both edges must have reference-identical information objects.

IncrementalNodeData) storing external sizes, coordinates, change information, etc.

As a secondary access path to nesting information, in intermediary graphs the nesting of nodes is also represented by node-node relations as denoted by the reflective association at Node.

Furthermore, nodes and edges are intended to store temporary data to be used within an individual algorithmic step and to transfer temporary but also permanent layout information across several algorithmic steps. Therefore, each algorithmic step is allowed to define its own specialized node, edge and graph classes, whose instances are created according to the input graph of the algorithmic step by a *graph copy mechanism*. Beside additional runtime and memory required by this mechanism, it allows a result graph to be independent from the input as well as the delegation of specific responsibilities to nodes and edges. This causes, that the instances of nodes and edges change while running the layout algorithm. Therefore, it may be difficult for an external application to find semantically equal elements of the input graph in the result graph. To solve this problem generically, all nodes and edges can be tagged by arbitrary identification information (Identification).

To hide the an arbitrary input mechanism from internal mechanisms, a dedicated input layer (not shown in Figure 4.9) restricts several accessors by defining subclasses of the basic Graph, Node and Edge class. On these classes it is not allowed, e.g., to change the internal type of a node. The classes of the input layer are then refined to support the creation of UML class diagram specific input graphs. Furthermore, the information objects to be attached by an input mechanism must be created by a factory which does not provide factory methods for internally used information objects.

*SugiBib* works on different types of graphs depending on the state of the layout process. It receives an input graph from an arbitrary graph reading mechanism, transforms it into intermediary graphs providing several additional facilities due to the individual layout step and finally creates a result graph. Starting with the creation of an information instance and finishing with its cleanup phase, information objects adhere to a certain life cycle which is depicted in Figure 4.10. After being created by a factory method in the application domain specific implementation, an information object (usually) receives its data from a graph reading mechanism. In the first processing step of the layout algorithm the information instance is copied to make the input independent from the result graph to be created. Then it enters an inactive state where it waits for queries to be answered. *SugiBib* provides a general querying mechanism represented by the method `queryInfo`. The information object reacts on an arbitrary object and returns a boolean value. Application domain independent queries can be defined and used by common code of the core implementation, while the query object and the implementation of `queryInfo` is located in application domain specific code. Furthermore, in most cases this mechanism is much faster than considering inflexible runtime type information.

Dependent on the general layout mode, the minimum size requirements for each graph element determined by individual information objects are calculated. If this information is provided from outside, e.g. by considering XMI[DI] layout information, it must not be calculated by the layout algorithm. While creating the result graph in S21 the positions of nodes, edges and adornments are fixed to provide the information for arbitrary output mechanisms, e.g. a XMI[DI] writer. In most cases, the graph is, however, considered for output, e.g. drawn on a graphical device by

visiting each graph element. Finally, due to memory reuse techniques, an information object is cleaned up and might be pooled.

In principle, a simplified and specialized version of this life cycle, dependent on the current layout step being executed, also applies to graph, node and edge instances.

## 4.3 Basic Definitions

If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them.

Henry David Thoreau (1817 – 1862)

Before discussing the individual algorithmic steps of the layout algorithm for UML class diagrams introduced in Section 4.1 in detail, in this section notational conventions for algorithmic descriptions, helpful primitive functions, the formal definition of a graph, its nodes and edges and operations on these elements will be given. Then, a hierarchical node naming function, used to translate between node names and node nesting relations will be described.

### 4.3.1 Notational Conventions

We will occasionally use this arrow notation unless there is danger of no confusion.

Ronald Graham, "Rudiments of Ramsey Theory"

The Roman letters  $i, j, k$  and the Greek letters  $\alpha, \beta$  will be used as index counters, the letters  $u, v, w, x, y, z$  will denote nodes of a graph and  $e, f, g$  will be used for edges. Uppercase letters will be used for sets or tuples, e.g.,  $G$  will denote a graph,  $E$  usually will be the set of edges and  $V$  the set of nodes (vertices) of a given graph. After determining the layered structure of an input graph,  $n$  will denote the number of levels (the height of the hierarchy) of a graph.

As usual in graph drawing, we will denote the theoretical complexity of some algorithm using the Oh-notation on formulae considering the number of nodes or edges, respectively. Further impacts, which also may influence the effective runtime, are assumed to be constant. A comparison between the theoretical complexity and the effective runtime, which can be retrieved by measurements, will be discussed in Chapter 5.

Furthermore,  $\mathbb{N}$  will denote the set of positive natural numbers without 0, while  $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ .  $\mathbb{Z}$  will be the set of positive and negative integer numbers. As usual, we will denote  $\mathbb{R}$  as the set of real numbers. The set of boolean constants will be given by  $bool := \{true, false\}$  and  $\perp$  will denote a pointer to nothing like `null` in Java. As common in computer science and programming languages, we will assume a short cut logic evaluated from left to right on boolean expressions

so that in an or-expression the evaluation is stopped when a subexpression evaluates to *true*. Similarly in an and-expression the evaluation is finished when a subexpression evaluates to *false*. We will make a clear distinction between assignments ( $:=$ ) and comparison for equality ( $=$ ). Furthermore, assignments will be used to also indicate that an object  $o$  probably has changed after performing an operation on  $o$ , which is usually specified by the first parameter<sup>4</sup>. Therefore, most operations will return at least  $o$ .

Similar to functors, sometimes functions instead of their return values will be used as parameters. If the type of an expression should be emphasized, the name of the function will occur without its parameter list. Furthermore, in some algorithms generic functions will be used. Let  $f$  be a generic function, then we will denote  $f(\cdot)$  in absence of relevant parameters or, e.g.,  $f(\cdot, p)$  if parameters are relevant to the context. If such a function itself will be used as a parameter, the relevant parameters may be given to “instantiate” the generic function similar to a template functor in C++, e.g.  $g(f(\cdot, true))$ . Furthermore, we assume for  $g(\cdot, f(\cdot, true))$  that all arbitrary parameters to  $g$  will automatically be passed to  $f$ .

### 4.3.2 Primitives

I am the primitive of the method I have invented.

Paul Cezanne (1839 – 1906)

To have some common constructs as well as some shortcuts at hand, in this section some specialized minimization and maximization notations and basic operations for lists and dictionaries will be introduced.

In some algorithms, explicitly lists or dictionaries (e.g., hashables) will be used. We will assume the semantics known from Java for these datastructures, e.g.,  $\perp$  is returned for keys, which do not have an assigned element in a dictionary or  $-1$  is returned as position of elements, which are not members of a list. Only some basic issues will be discussed here. A formal definition will be given in the appendix.

Let  $L, M$  be lists on arbitrary objects. Then  $L[i]$  returns the  $i$ -th element of  $L$  and  $L(l)$  returns the index position of  $l$  in  $L$ . Furthermore, the usual set operations like union, intersection or powersets will be available.  $L = M$  denotes that all elements of  $L$  are member in  $M$  and vv., while  $L \overset{*}{=} M$  also requires equal sequences of elements. To make obvious, which kind of data structure and operation is referenced, the remaining list operations will be prefixed by *list* and operations on hashables by *hash*.

We will implicitly assume that the sets used in this chapter are lists even if most times the property of index based access is not required.

---

<sup>4</sup>In object-oriented languages this is automatically done by the language implementation.



**Definition 1 (initialized minimization and maximization)**

Let *range* be a set of arbitrary elements and  $a \notin \text{range}$ . Then

$$\min_{r \in \text{range}}^a b(r) := \begin{cases} a & : \text{ if } \nexists_{r \in \text{range}} b(r) \\ \min\{r : r \in \text{range}, b(r)\} & : \text{ otherwise} \end{cases}$$

and

$$\max_{r \in \text{range}}^a b(r) := \begin{cases} a & : \text{ if } \nexists_{r \in \text{range}} b(r) \\ \max\{r : r \in \text{range}, b(r)\} & : \text{ otherwise} \end{cases}$$

denote the minimization or maximization, respectively, which are initialized by  $a$  if the condition represented by  $b(r)$  cannot be fulfilled at all. Analogously let

$$\min S := \begin{cases} a & : \text{ if } |S| = 0 \\ \min S & : \text{ otherwise} \end{cases}$$

and

$$\max S := \begin{cases} a & : \text{ if } |S| = 0 \\ \max S & : \text{ otherwise} \end{cases}$$

be the minimization or maximization of sets, respectively, which are initialized by  $a$  if the set is empty.

### 4.3.3 Graphs, Nodes, Edges and Operations

Most UML diagrams and some complex symbols are graphs containing nodes connected by paths. The information is mostly in the topology, not in the size or placement of the symbols (there are some exceptions, such as a sequence diagram with a metric time axis).

[OMG 2003c]

In literature, different kinds of definitions for graphs have been proposed. Usually, a graph is a tuple  $G = (V, E)$ , which consists of  $V$ , the set of nodes of  $G$  and  $E \subset V \times V$  the set of edges [Valls et al. 1996; Battista et al. 1999]. Such definitions are not capable to model non-simple graphs, which, according to REQ\_GRAPH\_TYPE, represent UML class diagrams. A more appropriate definition as the one in [Jünger and Mutzel 2003b] uses multisets to declare the set of edges.

A pragmatic way is a graph definition like that in [Noltemeier 1988], where a graph is a tuple, which consists of a set of nodes, a set of edges and individual functions returning the start or the end node of an edge, respectively. The elements, which represent nodes or edges, are seen as unique objects. This models implicitly the process of building an input graph. To model compound graphs (REQ\_GRAPH\_TYPE), we have to adapt that definition to introduce the nesting of nodes in nodes.

**Definition 2 (compound digraph)**

A compound (directed) graph is a 5-tuple  $G = (V, E, \alpha_G, \omega_G, nest_G)$ , in which  $V$  is the set of nodes and  $E$  is the set of edges with  $V \cap E = \emptyset$ . Let  $e \in E$ , then  $\alpha_G(e)$  returns the start node of  $e$  and  $\omega_G(e)$  the end node of  $e$ . Furthermore, let

$$nest_G \subseteq V \times V$$

be a partial order defining the nesting of nodes in nodes. For  $(u, v) \in nest_G$   $v$  is nested in  $u$ .

Due to our application domain, a partition of the set of edges into hierarchical and non-hierarchical edges (UML\_HIERARCHY) is required. Algorithmic steps like S9 may hide elements of a graph temporarily by removing them. A concrete implementation may hold the removed elements as well as other runtime related information in the graph data structure. We will not model this here.

Furthermore, a distinct node is declared as the “root”, which is intended to be used as an initial start position by some algorithms rather than to be a root in the sense of an usual hierarchical graph.

To be more compliant to the notation in graph drawing,  $\alpha_G$  and  $\omega_G$  from definition 2 will be assumed to be present implicitly. Due to notational convenience and a close relationship to the implementation, the compound relations defined by  $nest_G$  will be visible as operations on nodes only.

**Definition 3 (application domain specific digraph)**

Let  $G' = (V, E, \alpha_G, \omega_G, nest_G)$  be a compound graph according to definition 2. The 4-tuple

$$G = (V, r, E_H, E_N)$$

is called a hierarchical partitioned graph, if  $E$  can be partitioned into a set of hierarchical edges  $E_H$  and a set of non-hierarchical edges  $E_N$  so that  $E = E_H \cup E_N$  and  $E_H \cap E_N = \emptyset$  holds.

Often, the root is not relevant and therefore writing  $G = (V, E_H, E_N)$  is an appropriate shortcut. As long as a hierarchy is not determined, we simply write  $G = (V, E)$  and assume  $G = (V, E) = (V, \emptyset, E)$ .

The partition of hierarchical and non-hierarchical edges is given by the application domain. For UML class diagrams, this is the pseudo-hierarchy selected according to user preferences as described for UML\_HIERARCHY.

For the definitions in this section mathematical sets are sufficient. In an algorithm, when selecting an element from a set of elements all having a set of properties in common, this is usually left to chance. Therefore, due to REQ\_DETERMINISTIC\_ALGORITHM, we will assume that all sets are realized by lists, on which the element to be considered must explicitly be specified, e.g., the first element.

**Definition 4 (notation of edges)**

Let  $G = (V, E_H, E_N)$  be a graph and  $E := E_H \cup E_N$  the set of all edges of  $G$ . As usual, an edge  $e \in E$  is written as a pair which contains the nodes connected by  $e$ . All edges in  $G$  are directed by default, because  $G$  is a directed graph. We denote  $\vec{e} = (u, v) \in E$  as a declaration of the edge  $e$ , which connects its start node  $u$  to its end node  $v$  so that  $u = \alpha_G(\vec{e})$  and  $v = \omega_G(\vec{e})$ . Due to the graph model described in Section 4.2, we may navigate along the direction of  $\vec{e}$  as well as in opposite direction. Therefore, it is convenient to have the declaration  $e = \{u, v\} \in E$  for undirected edges at hands, e.g., to denote all edges connected to  $u \in V$  as  $\{e = \{u, v\} : v \in V\}$ .

In the following definitions we will introduce some basic operations on the elements of a graph. To avoid starting each of the following sections with a definition on new operations for nodes, edges or graphs, we will compile most of the operations into this section as a kind of global dictionary. Coordinates related operations on nodes and edges will be given as basic definitions in the description of the coordinates assignment.

For the pseudocode notation of algorithms it is convenient to assume that we do not have different types of nodes, edges or graphs and all operations defined below are available at once. It is also convenient to use these operations as reading as well as writing accessors, i.e., if a value is assigned to an operation, this means that the corresponding value is changed in memory. Furthermore, we will define required operations only, not the entire set of operations which may be available in a concrete implementation.

**Definition 5 (operations on nodes)**

Let  $G = (V, E_H, E_N)$  be a graph according to definition definition 3. Let  $v \in V$  be a node of  $G$  and  $E := E_H \cup E_N$  the set of edges of  $G$ . Then the following operations are defined on  $v$ :

- Let  $type(v) \rightarrow nodeTypes$  return the type of  $v$  with

$$nodeTypes := \{USUAL, CLUSTERBORDER, HYPEREDGE, ASSOCCLASS, COMMENT\}$$

- $out(v) := \{\vec{e} : \vec{e} = (v, u) \in E, u \in V\}$  is the set of outgoing edges of  $v$ , also called the out-star of  $v$ .
- $in(v) := \{\vec{e} : \vec{e} = (u, v) \in E, u \in V\}$  is the set of incoming edges of  $v$ , also called the in-star of  $v$ .
- $edges(v) := in(v) \cup out(v)$  is the set of edges connected to  $v$ , also called the star of  $v$ .
- $d^+(v) := |out(v)|$  is the number of outgoing edges called out-degree of  $v$ .
- $d^-(v) := |in(v)|$  is the number of incoming edges called in-degree of  $v$ .

- $d(v) := d^-(v) + d^+(v)$  is the number of edges adjacent to  $v$ , also called the degree of  $v$ .
- $V^+(v) := \{u : \vec{e} = (v, u) \in E, u \in V\}$  is the set of nodes connected by outgoing edges of  $v$ .
- $V^-(v) := \{u : \vec{e} = (u, v) \in E, u \in V\}$  is the set of nodes connected by incoming edges of  $v$ .
- $compoundParents(v) := \{u : (u, v) \in nest_G\}$  is the set of nodes  $v$  is nested in. Usually,  $|compoundParents(v)| \leq 1$  for non-intersecting compounds can be assumed due to our application domain.
- $compoundChildren(v) := \{u : (v, u) \in nest_G\}$  is the set of nodes nested in  $v$ .
- $llc(v) := \{v\} \cup compoundChildren(v) \cup \bigcup_{w \in compoundChildren(v)} llc(w)$  is the lower level compound closure of  $v$ .
- $ulc(v) := \{v\} \cup compoundParents(v) \cup \bigcup_{w \in compoundParents(v)} ulc(w)$  is the upper level compound closure of  $v$ .
- $r(v) \rightarrow \mathbb{Z}$  is the level (rank)  $v$  is assigned to. We do not use  $\mathbb{N}$  or  $\mathbb{N}_0$  here, because the rank number might temporarily be negative. Hierarchies and ranks will be introduced formally in Section 4.5.

#### Definition 6 (operations on graphs)

Let  $G = (V, E_H, E_N)$  be a graph according to definition 3. Then the following operations are defined on  $G$ :

- Let  $v \in V$  then  $remove(G, v) \rightarrow G$  changes  $G$  via  $V := V \setminus \{v\}$ ,  $E_H := E_H \setminus edges(v)$  and  $E_N := E_N \setminus edges(v)$ .
- Let  $e \in E_H \cup E_N$  then  $remove(G, e) := G(V, E_H \setminus \{e\}, E_N \setminus \{e\})$ .

### 4.3.4 The Node Naming Function

We do what we must, and call it by the best names.

Ralph Waldo Emerson (1803 – 1882)

In graph drawing, partitioning a given graph to gain appropriate clusters and compounds is one of the generally unsolved key problems. As discussed in Section 3.3.6, two basic types of clusters are relevant for UML class diagrams. Visible clusters, like model management elements or classes containing other classes, as well as invisible clusters, e.g., the elements involved in a n-ary association, can be used to realize UML\_SPATIAL or UML\_SEMANTIC\_CLUSTERS. According to our graph model described in Section 4.2, a visible cluster is initially specified by fully qualified names. Invisible clusters are intended to be detected by the layout algorithm and appear therefore as an implementation technique.

Obviously, two approaches for a concrete realization can be considered: Composite nodes as illustrated in Figure 4.2 as well as clusters, in which neither the cluster border nor the cluster base node are displayed.

According to our graph model, a visible cluster consists of the contained elements and a cluster base node, which will always be placed vertically above the contained elements. Even if the cluster base node might occupy no area to appear invisible, it will allocate a part of the top rank the cluster is assigned to and prevent other nodes from being placed there. Hence, the mechanisms for visible clusters cannot easily be reused for invisible clusters.

Applying a composite node to realize an invisible cluster implies a temporary substitution of the involved nodes. If these composite nodes are expanded before reducing edge crossings in S11, the vicinity according to UML\_SPATIAL and UML\_SEMANTIC\_CLUSTERS cannot be ensured without further mechanisms. If the composite nodes are expanded somewhere after S11, graph drawing rules like GDR\_EDGE\_CROSS or GDR\_MIN\_EDGES have to be considered while inserting the contained nodes by additional effort. This also leads to a tricky implementation.

As mentioned above, we need a rank assignment (S10) and an edge crossing reduction (S11), which generically consider the presence of clusters. This can be realized by taking the fully qualified names of the classes as a criterion for clusters into account on which certain cluster consistency issues have to be respected. As a side effect, invisible clusters can be realized by virtual qualified names.

A node naming function assigns every node in a graph a name, i.e.,

$$f : u \rightarrow c \text{ with } u \in V, c \in C^*$$

where  $C \neq \emptyset$  is an alphabet and  $C^*$  denotes all words that can be generated for the alphabet  $C$ .

To layout hierarchically nested nodes we need a node naming function which imposes hierarchical properties:

**Definition 7 (hierarchical node naming)**

A hierarchical node naming of graph  $G$  is a 3-tuple  $(C, crit, \preceq)$  where  $C \neq \emptyset$  is an alphabet,  $C^*$  the set of all words which can be created for  $C$ , and denotes the set of valid node names. Let  $G = (V, E_H, E_N)$  be a graph.

$$crit : V \rightarrow C^*$$

is the node naming function which assigns a name (a cluster criterion) to each node. Let  $c_1, c_2, c_3 \in C^*$  be node names, then a concrete hierarchical node naming defines the containment of names

$$c_1 \preceq c_2$$

so that  $c_1 \prec c_2 = c_1 \preceq c_2 \wedge \neg(c_1 =_N c_2)$  holds, where  $c_1 =_N c_2$  denotes the equality<sup>5</sup> of the two cluster names  $c_1$  and  $c_2$ .

The unique largest element  $g \in C^*$  with  $\forall_{c \in C^*, c \neq g} c \prec g$  denotes the (default) global name. The following conditions also have to be valid on a hierarchical node naming function:

- reflexivity:  $c_1 \preceq c_1$
- antisymmetry: if  $c_1 \preceq c_2 \wedge c_2 \preceq c_1 \Rightarrow c_1 =_N c_2$
- transitivity:  $c_1 \preceq c_2 \wedge c_2 \preceq c_3 \Rightarrow c_1 \preceq c_3$
- parent relation:  $c_1 \preceq c_2 \wedge c_1 \preceq c_3 \Rightarrow c_2 \preceq c_3 \vee c_3 \preceq c_2$

Therefore, a hierarchical node naming function induces a monotone, partial order.

Definition 7 is given in terms of node names and can be used to represent hierarchical nesting on nodes as cluster containment relations. Therefore, the node names represent cluster names and the global name  $g$  is the global (default) cluster, which contains all nodes not assigned to any other cluster. Invisible cluster can be modeled by defining a subset of  $C^*$  to be dedicated to invisible clusters.

**Example:**

A hierarchical node naming for UML can be defined as follows: Let

$$C = \{A \dots Z\} \cup \{a \dots z\} \cup \{0 \dots 9\} \cup \{::\} \cup \{*\}$$

be the simplified alphabet for fully qualified names in UML without respecting localized versions of UML e.g. for Japanese. The equality function  $c_1 =_N c_2$  is defined similarly to

<sup>5</sup>We use  $=_N$  instead of  $=$  to make a clear distinction between equality of node names and equality of objects.  $=_N$  can be extended for nodes, sets, lists, etc. by considering the node naming function instead of the objects themselves.

the string equality function. The containment function  $c_1 \prec c_2$  is given via the string prefix comparison so that  $c_1$  is contained in  $c_2$  if  $c_2::$  is a string prefix of  $c_1$ . The global cluster name  $g$  is the empty string.

The fully qualified name of a model management element is equal to its cluster criterion. For a class, the name of the containing namespace (fully qualified name without the name of the class) is appropriate.

As described above, the vicinity of certain nodes according to UML\_SEMANTIC\_CLUSTERS can be realized by invisible clusters. This can be done by defining a set of cluster names, which are dedicated to invisible clusters. For UML, the character  $*$ , which is not part of the usual UML naming system, can be used as prefix for virtual cluster names.

### Definition 8 (operations on a hierarchical node naming function)

The following functions are directly induced by definition 7. Let  $(C, crit, \preceq)$  be a hierarchical node naming function and  $c_1, c_2 \in C^*$  node names, then

- $c_1 \succ c_2 := c_2 \prec c_1$
- $c_1 \succeq c_2 := c_2 \preceq c_1$
- $c_1 \boxtimes c_2 := c_1 \preceq c_2 \vee c_2 \preceq c_1$
- $c_1 \not\boxtimes c_2 := \neg(c_1 \boxtimes c_2)$
- $global(c_1) := (c_1 =_N g)$

are introduced for notational convenience. Furthermore, the following operations are defined

- $\uparrow(c_1) := \begin{cases} c & : \text{ if } \exists_{c \in C^*} c_1 \prec c \wedge \forall_{c_1 \prec c_2} c \prec c_2 \\ g & : \text{ otherwise} \end{cases}$  as the least upper bound of  $c_1$  and  
 $\uparrow^0(c_1) := c_1, \uparrow^1(c_1) := \uparrow(c_1)$  and  $\uparrow^n(c_1) := \uparrow(\uparrow^{n-1}(c_1))$  if  $n > 2$
- the least common cluster  
 $LCC(c_1, c_2) := \begin{cases} c & : \text{ if } \exists_{c \in C^*} (c_1 \prec c \wedge c_2 \prec c) \wedge \forall_{c_1 \prec c_3, c_2 \prec c_3} c \prec c_3 \\ g & : \text{ otherwise} \end{cases}$

We also define a new equivalence relation  $=_N \subset V \times V$  via

$$u =_N v \Leftrightarrow crit(u) =_N crit(v) \text{ for } u, v \in V$$

and extend  $=_N$  for sets and lists. Let  $L$  be a list (or a set), then  $c \in S \Leftrightarrow \exists_{s \in S} c =_N s$  to simplify the notation.

On the one hand, using fully qualified names appears to be an obvious implementation of a hierarchical node naming for UML. On the other hand, in particular for long fully qualified

names, determining the containment and equality of cluster names via string operations is an inefficient realization. This can be circumvented by abstract node namings inducing the same structure, e.g., by

- a hierarchical huffman encoding [Heise and Quattrocchi 1995] of the cluster names.
- a preorder-size numbering [Li and Moon 2001; Pankowski 2004] as known for trees.

The hierarchical node naming for UML as described above is useful when debugging the implementation, because the abstract namings would require a remapping to the original names for readability. Even for UML class diagrams it is not required that each graph owns a hierarchical node naming. If no node naming function is present, S1 will not construct nesting relations and the input graph will be treated as an usual graph but not as a compound graph. If a node naming function is present, the rank assignment S10 as well as S11 will produce a sorted layering considering the specified node naming function.

Enabling and disabling a given node naming function can be used to realize further constraints in the layout. When considering the node naming in rank assignment and edge crossing reduction, but not in the coordinates assignment S15, an usual graph layout can be obtained, which admits a sorting of the nodes in layers according to clusters. Furthermore, (certain) invisible clusters can also be enabled or disabled, e.g., for experiments on aesthetic principles.

As a conclusion, the node naming function implies a generic, flexible mechanism to tailor the nesting but also the sorting of the nodes in layers.

**Corollary 1 (properties of stepping up node names)**

Let  $(C, crit, \preceq)$  be a hierarchical node naming function and  $c_1, c_2 \in C^*$  node names.

- $\uparrow^i(c_1)$  is convergent to  $g$
- Let

$$stepUpSet(c_1) = \{c : i \in \mathbb{N} \wedge c = \uparrow^i(c_1)\}$$

and

$$stepUpSet^g(c_1) = \{c : i \in \mathbb{N} \wedge c = \uparrow^i(c_1) \wedge \neg global(c)\}$$

be the set of parent cluster names of  $c_1$ . Then

$$c_1 \prec c_2 \Leftrightarrow c_2 \in stepUpSet(c_1)$$

and if  $c_2 \neq_N g$

$$c_1 \prec c_2 \Leftrightarrow c_2 \in stepUpSet^g(c_1)$$

hold.

Furthermore, if  $global(LCC(c_1, c_2))$  then  $c_1$  and  $c_2$  are not contained in a (named) common cluster (except for the global cluster). Hence,

$$notClusterRelated(c_1, c_2) := (c_1 \neq_N c_2 \wedge global(LCC(c_1, c_2)))$$

returns if two arbitrary node names are not related by nesting relations.



**Proof:**

According to definition 7  $\forall_{c \in C^* \setminus \{g\}} c \prec g$  and therefore with definition 8

$$\uparrow(c_1) = \begin{cases} c & : \text{ if } c_1 \prec c \prec g \wedge \nexists_{c_2 \in C^*} c_1 \prec c_2 \prec c \prec g \\ g & : \text{ otherwise} \end{cases}$$

hence  $\exists_{n \in \mathbb{N}_0} \forall_{l \geq n} \uparrow^l(c_1) = g$ .

If  $k_0 \prec k$  then there might exist a  $c$  so that  $k_0 \prec c \prec k$  because of the transitivity in definition 7. Let  $k_1, \dots, k_n \in C^*, n \in \mathbb{N}$  so that  $k_0 \prec k_1 \prec \dots \prec k_{n-1} \prec k_n \prec g$  and therefore  $stepUpSet(k_0) = \bigcup_{0 < i \leq n} k_i \cup g$  and  $stepUpSet^g(k_0) = \bigcup_{0 < i \leq n} k_i$ . Then  $\exists_{i \in \mathbb{N}} k =_N k_i$  and  $k \in stepUpSet(k_0)$ , especially if  $k \neq_N g$   $k \in stepUpSet^g(k_0)$ .

On the other side, let  $n := |stepUpSet(k_0)|$ ,  $k \in stepUpSet(k_0)$  and  $k_j = \uparrow^j(k_0)$   $0 < j < n$ . Then  $\exists_{0 < i \leq n} k =_N k_i$ . Because of the definition of  $stepUpSet$   $\uparrow(k_{j-1}) = k_j$ . Because of the definition of  $\uparrow$  follows  $k_{j-1} \prec k_j$  for all  $0 < j \leq n$  and because of the transitivity in definition 7  $k_0 \prec k_1 \prec \dots \prec k_i \prec \dots \prec k_{n-1} \prec k_n$ . Finally, because of the transitivity  $k_0 \prec k_i$  and hence  $k_0 \prec k$ . Especially if  $k \neq_N g$  and  $k \in stepUpSet^g(k_0)$   $k_0 \prec k$  follows similarly. □

## 4.4 Preprocessing Steps

The future belongs to those who prepare for it today.

Malcolm X (1925 – 1965)

In this section, the preprocessing steps S1 up to S9 of our layout algorithm for UML class diagrams, introduced in Section 4.1, will be described. We will also discuss some of the algorithms, which will be available as general implementation to be reused in various contexts, e.g. in the rank assignment step.

### 4.4.1 Adjust Semantical Issues

The UML notation is an essential element of the UML to enable communication between team members. Compliance to the notation is optional, but the semantics are not very meaningful without a consistent way of expressing them.

[OMG 2003c]

In the first step (S1) of our layout algorithm, some basic adjustments and modifications to the input graph are processed. As mentioned along with the graph model in Section 4.2, the input graph does not specify some main structural issues and relations, to keep arbitrary input mechanisms simple: Nesting relations are given as fully qualified names only and invisible clusters, like all elements involved in a n-ary association, should be detected by the layout algorithm.

---

#### Algorithm 4.1 *adjustSemanticalIssues*

---

**input:**  $G = (V, E)$  possibly with node naming

**output:**  $G$

$G := initializeCompoundHierarchy(G)$

$G := adjustVirtualClusters(G)$

$G := adjustNesting(G)$

$G := calculateComplexities(G)$

**return**  $G$

---

According to the graph model in Section 4.2, the (visible) nesting relations of nodes should additionally be represented as node-node relationships, because it is easier to realize some of the following algorithms using these relationships rather than cluster names issued by the node naming function. Therefore, as denoted in algorithm 4.1, first the nesting relations, are initialized. All cluster parents, e.g., packages, subsystems, models but also classes containing other classes, are identified. Thereby, the relations between the cluster names, issued by the node naming function, and the node instances are stored in a dictionary for fast access. By iterating over all nodes, to each individual node the bidirectional relationships between a nested node and its cluster parent are assigned.

As discussed in Section 4.3.4, virtual cluster names are used in *SugiBib* to realize invisible clusters, which ensure the vicinity of nodes belonging together in rank assignment S10 as well as in the edge crossing reduction S11. Each node being involved in, e.g., a n-ary association or a collaboration, is defined to be member of a virtual cluster.

As an issue of consistency, `adjustNesting` ensures that association classes or comments are assigned to the correct package, i.e. to the most common package of the connected elements. Finally, the individual design complexities of classes and relations are calculated to prepare the realization of the optional magnitude-related criteria. Due to the discussion in Section 3.3.4, a generic implementation of various complexity calculations is suggested.

S1 changes the structure of the input graph and represents thereby semantical issues, which are not explicitly present in the input graph. By initializing various nesting relations, `UML_SEMANTIC_CLUSTERS` and `UML_SPATIAL` as hierarchical aspects of our set of aesthetic criteria are prepared. Furthermore, information on the magnitude of diagram elements, which also may rely on external data, is collected to prepare the optional criteria `UML_SIZE_EDGES` and `UML_SIZE_NODES`.

Obviously, this step appears to be a preparation specific to the layout of UML class diagrams. When reusing *SugiBib* for other types of graphs, it may occur that the input graph contains proper nesting relations but no information to realize the hierarchical node naming. In this case, the node naming function can be constructed by assigning artificial hierarchical names to nested elements.

## 4.4.2 Semantic Ordering

A graph whose layout does not change much when it is newly layed out is called *stable*.<sup>[sic]</sup>

[Böhringer and Paulish 1990]

Deterministic algorithms, which work on lists of elements, heavily rely on the sequence of the elements in the lists. Changing the position of e.g. a node in an input file and assuming a strongly deterministic input mechanism, the position of the node in the set of nodes of the input graph also changes. Therefore, in the layout algorithm, the nodes are processed in a different order which will probably produce a different result. Obviously this is an unpleasing dependency on the input and, especially for UML diagrams, this can be circumvented by applying some ordering operations (S2) before proceeding with the layout algorithm.

In fact, reducing sequence dependencies is an issue of layout stability. *Structural stability* is usually concerned with user-specified layout constraints, while *dynamic stability* requires minimizing the differences between successive layouts of one graph [Tamassia et al. 1988; Böhringer and Paulish 1990]. Sorting the elements of a graph addresses dynamic stability and therefore, as mentioned in Section 3.3.1, can be classified as a dynamic aesthetic rule. Therefore, S2 closely relates to `REQ_INCREMENTAL_ALGORITHM`, but, of course, it does not provide a solution to the problem of incremental layout.

First, the ascending order of the nodes according to their fully qualified names is determined. In

the case of a node without a name, a normalized combination of the names of the connected nodes can be taken into account. According to REQ\_GRAPH\_TYPE, we work on directed graphs. As mentioned in Section 4.2, the concrete direction of the edges representing bidirectional or undirected UML relations, is arbitrary. The direction of these edges can be normalized by considering the order of nodes determined in this step. Thereby, when reversing the direction of edges, the position of the edge adornments may be exchanged. Then, the set of all edges is normalized by taking into account the positions of the start and the end node in the set of nodes, e.g., by sorting the edges according to

$$\text{numberEdge}(e = \{u, v\}) := |E| \cdot V(u) + V(v)$$

Similarly, the set of in and out edges of the individual nodes are normalized.

The result of S2 is a graph, in which the sequences of graph elements in all relevant set instances is normalized. Therefore, S2 prepares REQ\_DETERMINISTIC\_ALGORITHM for different sequences of graph element declarations in the input.

S2, as described above, does not consider previously determined positions of graph elements in incremental layout. This can induce problems maintaining the mental map, if the previous layouts were not created by *SugiBib*. In the case of incremental layout with *SugiBib*, obviously a problem ensuring stability occurs, if the name of a class is changed between two successive layout calculations. Then, the position of the node may change in the sorted sequence and the result graph might look completely different. This issue of stability can also be handled in this step considering difference information on graphs to be laid out in sequence. For example, proxy objects representing an old node but having a reference to the changed node can be processed instead of the nodes and replaced after sorting. We did not implement this proxy technique, because we decided in REQ\_INCREMENTAL\_ALGORITHM that deep issues of incremental layout should not be addressed in this work. We can conclude that this step helps

- producing the same result for input graphs, which are equal in contents disregarding the sequence of elements in the input. In particular, for different input mechanisms working on the same data (possibly in various file formats) from the viewpoint of the layout algorithm the same output is produced.
- maintaining the mental map when new elements are added or existing elements are removed, because the sets of nodes and edges differ only due to inserted or removed elements but not in their sequence.

### 4.4.3 Deduce Hierarchy

Hierarchical institutions are like giant bulldozers – obedient to the whim of any fool who takes the controls.

Edward Abbey (1927 – 1989)

In step S3, the pseudo hierarchy (UML\_HIERARCHY), which represents the main skeleton of the final drawing, is initialized. In a loop, each edge of the input graph is considered and tested for compliance to one of the following alternatives. An edge  $e$  is assigned to  $E_H$ , if  $e$

- belongs to a combination of predefined groups of edges like
  - inheritance and realization edges
  - anchor edges
  - aggregations
  - compositions
  - directed associations
  - dependencies

and is therefore included into the pseudo-hierarchy.

- should be part of the hierarchy due to an user selection. This might optionally be combined with some of the predefined groups of edges.
- is assigned to the hierarchy due to automatic selection by delegation to a hierarchy detection plug-in.
- was defined as an hierarchical edge by external mechanisms, e.g. because of incremental layout.

Otherwise,  $e$  is assigned to  $E_N$ .

Because of interferences between hierarchical edges, assigned to the pseudo-hierarchy, and hyperedges, which rely on  $E_H$  initialized in the first loop, a second loop determines the hyperedges, e.g., used for xor-constraints, which should also be regarded as hierarchical edges. As discussed in Section 4.1, the layout algorithm distinguishes between hyperedges at hierarchical edges and hyperedges at non-hierarchical edges. In fact, it is important, that hyperedges at non-hierarchical edges are not specified as hierarchical edges. Therefore, a hyperedge is classified as hierarchical edge, if start and end node have only hierarchical edges except for the hyperedge itself.

S3 realizes main (dynamic) aspects of UML\_HIERARCHY. The next step (S4) will ensure that the nesting relations are present as part of the pseudo-hierarchy.

#### 4.4.4 Insert Nesting Relations as Edges

If you choose not to live in a cluster, uh,  
dorm...

Jim Zelenka

The basic Sugiyama algorithm does not distinguish between different kinds of edges and is therefore neither able to handle compounds or clusters nor non-hierarchical edges. To also consider clusters, we can design a new rank assignment as in [Sander 1996b] or we can reuse a well-known rank assignment algorithm and ensure that the nesting relations will have major influence on the structure of the layout result. We made a decision for the second alternative, because we want to realize a layout algorithm for mixed compound graphs according to `REQ_GRAPH_TYPE`.

Therefore, in this step (S4), we have to create hierarchical edges to express the nesting relations as edges to be considered by the rank assignment in S10. Let  $u$  and  $v$  be connected by a nesting relation so that  $v$  is intended to be nested in  $u$ . If no hierarchical edge from  $u$  to  $v$  exists, a new (hidden) hierarchical edge is inserted into the graph. Thereby, cycles in the pseudo-hierarchy subgraph may be introduced, which then will be considered by a cycle breaking technique in S10 so that hierarchical edges denoting nesting are kept with a higher priority.

This step prepares the realization of `UML_SEMANTIC_CLUSTERS` and ensures that nesting relations are present in the pseudo-hierarchy according to `UML_HIERARCHY`.

#### 4.4.5 Compress Hyperedge Connection Nodes

MacDonald has the gift of compressing  
the largest amount of words into the  
smallest amount of thoughts.

Sir Winston Churchill (1874 – 1965)

Before calculating the rank assignment or reducing the number of edge crossings, putting hyperedge connection nodes together with their connected nodes into composite nodes is one of the preparation steps for keeping selected nodes in a close vicinity (`UML_HYPEREDGES`). This step (S5) should be performed before transforming reflective edges, because reflective edges might also be involved in hyperedge relations.

As described in Section 4.1, depending on the type of the hyperedge, the hyperedge connection nodes are encapsulated into two composite nodes. Figure 4.11 (a) depicts the input structure of a hyperedge at non-hierarchical edges, to which a comment is attached. In this case, the hyperedge will be visible as an individual edge between two of the four visible nodes as illustrated in Figure 4.11 (b). Which of the four visible nodes are selected to be encapsulated in composite nodes depends on additional information, which was specified using `addNextTo`, and on the normalized sequence of nodes determined by S2. The comment node  $v$  (as well as further comment nodes which might appear at a hyperedge) is packed into the first composite node.

In Figure 4.11 (c) the result for a hyperedge at hierarchical edges is shown. In this case, both hyperedge connection nodes are encapsulated either with the visible start or end nodes. It would also be possible to pack the two hyperedge connection nodes into a single composite node, but

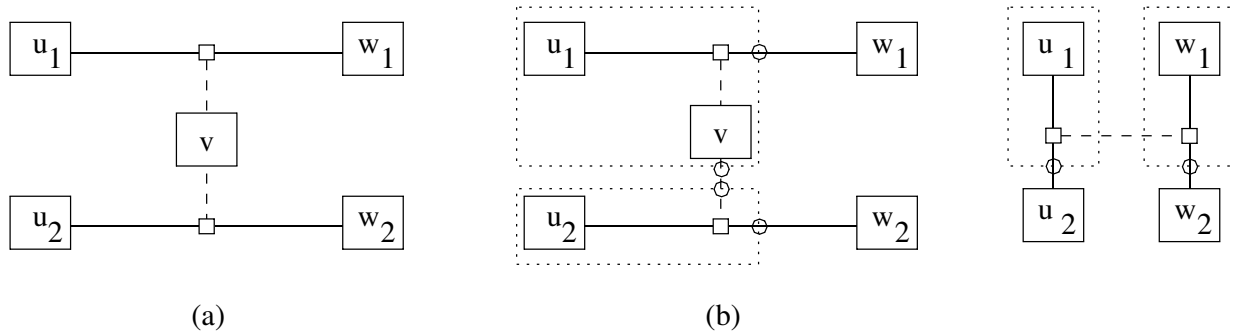


Figure 4.11: (a) input graph structure, which specifies that the annotation  $v$  is attached to a hierarchical hyperedge at non-hierarchical edges, (b) both hyperedge connection nodes and the comment are compiled into two composite nodes, (c) the composite nodes for a hyperedge at hierarchical edges.

this would induce an individual rank for the composite node in the rank assignment S10.

#### 4.4.6 Remove Direct Cycles

I'm out of the loop, and that's the way I like it.

Unknown, Graffiti

In most graph algorithms, handling cycles induces additional complexity in the implementation. As mentioned in Section 4.1, step S6 circumvents direct loops, which arise from reflective associations in the input graph. Thereby, the reflective information is transferred into the connected node and the edge representing the reflective association is removed. The connected node is then responsible for handling area and drawing specific issues of these edges.

Unfortunately, association classes introduce an additional dependency between S6 and S7, because association classes may be attached to reflective associations as well. In S7 association classes will be transformed to composite nodes. After executing S6, the reflective edge itself would be not visible anymore outside the composite node and an association class would structurally appear disconnected. Therefore, in the case of an association class connected to a reflective edge, this step connects the association class by an internal edge type to the class and puts the three elements into a composite node. As an obvious alternative, we might have decided to process association classes before reflective edges, but this would transform the graph structures forth and back by temporarily inserting several nodes and edges. In principle, comments can be handled similarly. A further discussion on comments will be given in Section 4.4.8.

This step prepares UML\_REFLECTIVE as well as certain aspects of UML\_COMMENTS and UML\_ASSOCIATIONCLASSES and prevents non-hierarchical cycles, which occur at reflective associations.

### 4.4.7 Compress Association Classes

An idea is a feat of association.

Robert Frost (1874 – 1963)

To keep association classes in a close vicinity with their connected association, in this step (S7) we will change the structure of the graph by inserting composite nodes.

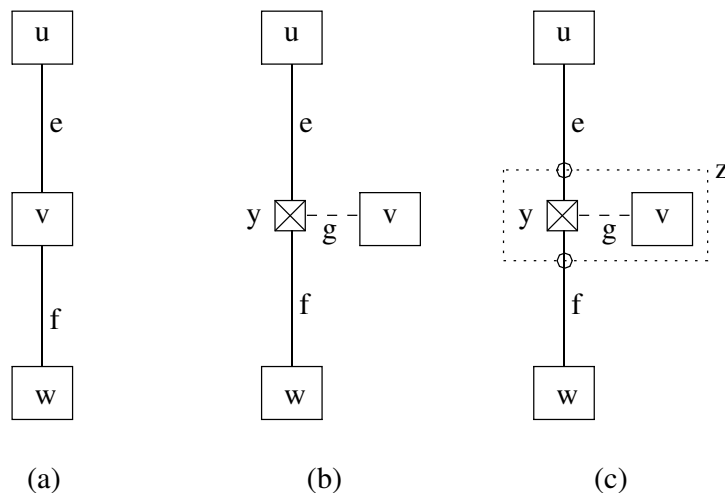


Figure 4.12: (a) input graph structure, which specifies that the association class  $v$  is attached to “an” edge, (b) node  $y$  and edge  $g$  which simulates the hyperedge are inserted, (c)  $v$ ,  $y$  and  $g$  are packed into the composite node  $z$  and all connections to elements outside are mapped.

So far, according to the graph model described in Section 4.2, the remaining association classes are at least connected by two edges (having identical information objects) which represent the underlying association. This situation is illustrated in Figure 4.12 (a). A hidden node is created and the edges formerly connected to the association class are reconnected to the hidden node. The dashed edge representing the connection between the association class and the association is then inserted as shown in Figure 4.12 (b). Finally, as depicted in Figure 4.12 (c), the association class and the hidden node is encapsulated into a composite node. Even if not mentioned above or shown in Figure 4.12, further connections to an association class are handled implicitly when creating the composite node.

If comments are attached to the association class, they are also encapsulated into the composite node. This is not shown in Figure 4.12. A further discussion on comments will be given in Section 4.4.8.

S7 prepares the realization of `UML_ASSOCIATIONCLASSES` and certain aspects of `UML_COMMENTS`.



### 4.4.8 Compress Comments

Comments are free but facts are sacred.

Charles Prestwich Scott

Comments in UML introduce a high degree of syntactical variation, because they might be attached to every model element. In Figure 4.13 some situations, in which comments are involved, are depicted.

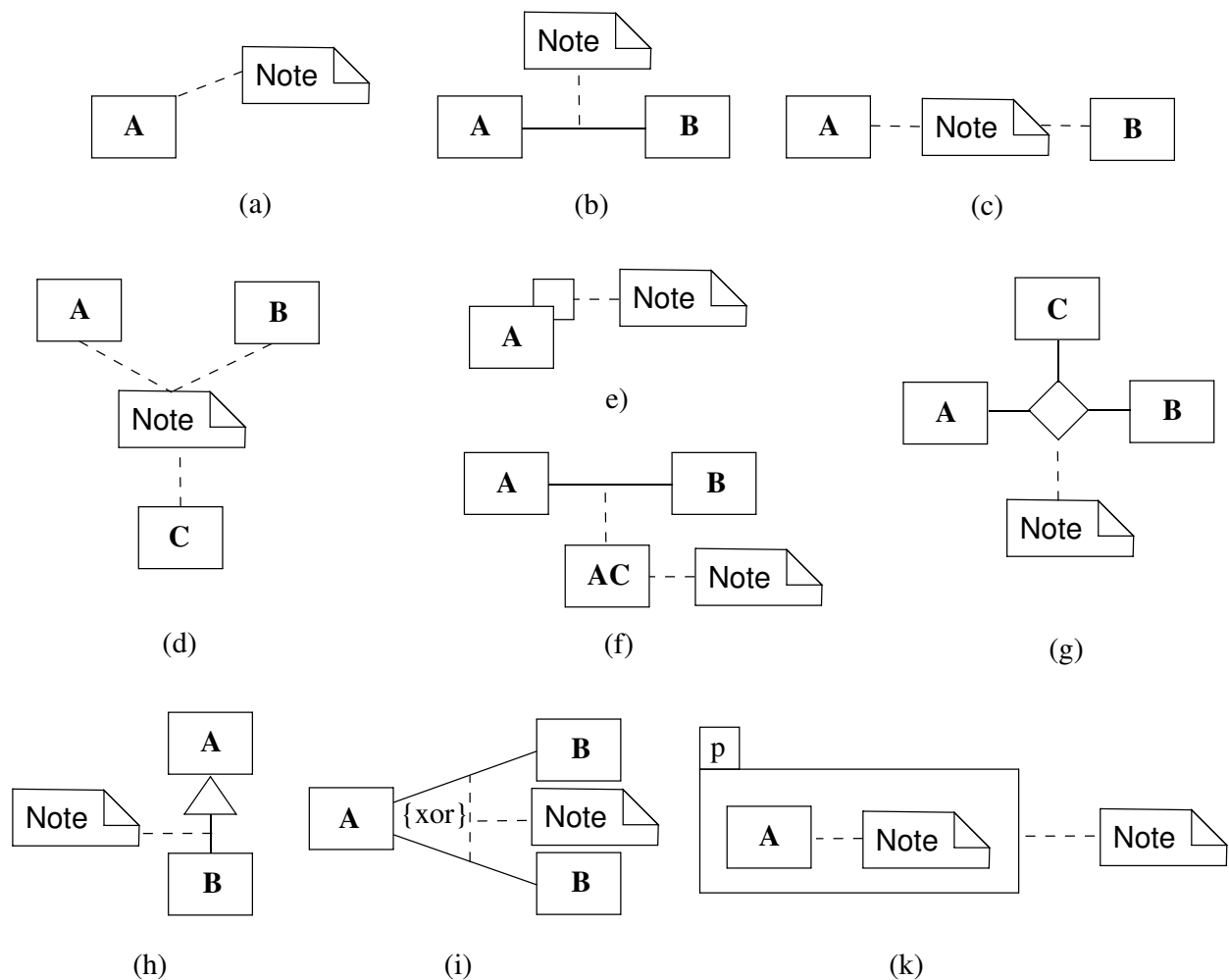


Figure 4.13: Various situations for a comment at (a) an usual class, (b) a non-hierarchical relation, (c) (d) multiple elements, (e) a reflective association, (f) an association class, (g) a n-ary node, (h) a hierarchical relation, (i) a hyperedge and (k) a model management element.

Except of the last one, the following items directly relate to Figure 4.13:

- a) To keep the close vicinity between a comment and its connected class, both elements are encapsulated into a composite node.
- b) In this case, simply all elements are compressed into a composite node.
- c) A comment at two other elements may be
  - encapsulated together with one connected node into a composite node.
  - packed together with both connected nodes into a composite node. This requires further knowledge on the rank assignment and should therefore not be processed here, but, e.g., in a postprocessing step of S10.
  - ignored in this step. It will then be treated as an usual node and hopefully will occur between both nodes depending on the edge crossing reduction.

It seems that the second alternative, postprocessing the rank assignment, is the most appropriate way.

- d) If the connected nodes are not involved in further visible hierarchical relations, a preprocessing step of the rank assignment can change the edges partition of the graph to redefine the dashed lines as hierarchical edges. After the rank assignment, multiple connected comments can be retrieved and the rank assignment as well as the edges partition can be adjusted.
- e) As discussed in Section 4.4.6, some of the preprocessing steps have to handle complex situations directly instead of deferring them to the dedicated layout step. Therefore, a comment attached to a reflective association, was processed in S6.
- f) Similar to e) this situation was handled in S7.
- g) A n-ary relation and a comment do not require further handling here. The comment was assigned to the invisible cluster in S1 and the connecting edge may be defined as hierarchical or non-hierarchical in the postprocessing of the rank assignment depending on the number of nodes involved in this situation.
- h) A hierarchical edge with a comment must not exist while executing the rank assignment. Therefore, the comment and its connecting node is compressed into a composite node with the start or the end node.
- i) A comment attached to a constraint edge should be encapsulated with either one of the hidden nodes which simulate the hyperedge, the attached start or end node. Therefore, this case was handled in S5.
- k) As mentioned along with S1, the correct nesting relations for association classes as well as for comments were determined in S1. Hopefully, the comment will be kept in vicinity to the package by the edge crossing reduction.

- l) Standalone comments, which relate to the diagram itself, are treated as usual nodes and because they are not connected, they will temporarily be removed in S9.

By combining the situations in Figure 4.13 further, more complex situations can be constructed, e.g. at reflective associations. Therefore, comments at a reflective association, which is furthermore described by an association class, comments at an association classes connected to a reflective associations or combinations of these situations have to be handled by an implementation. As a conclusion, in this step (S8) we have to handle the cases a), b) and h) by finding the appropriate subgraph and encapsulating the elements into a composite node to prepare UML\_COMMENTS.

#### 4.4.9 Remove Disconnected Nodes

Disconnected graphs occur rather frequently in real life applications either during the construction of a graph interactively or because of the nature of the application [...]

[Freivalds et al. 2002]

Nodes, which are not connected to any other node may disturb an iterative coordinate assignment algorithm, which places nodes according to priorities calculated according to the number of connected edges. Depending on the concrete implementation, such nodes, e.g., might be moved with their neighbors, but they may also remain at the border of the drawing somewhere far away from the other nodes. Therefore, in this step (S9), disconnected nodes are temporarily removed from the graph.

This step does not take nested nodes into account, because in S4 it has been ensured that nesting relations are present as hierarchical edges and therefore these nodes do not appear as disconnected nodes. Furthermore, this step prepares UML\_DISCONNECTED and disconnected comments according to UML\_COMMENTS.

#### 4.4.10 Virtual Root and Leaf

As the poet said, 'Only God can make a tree' – probably because it's so hard to figure out how to get the bark on.

Woody Allen

The rank assignment S10 requires the hierarchical subgraph to be connected and to be acyclic. The first requirement can be ensured by inserting a virtual root in the case of a disconnected forest. The virtual root will also act as starting point for some of the algorithms in S10. The second requirement will be treated in the next section.

If more than one node of the hierarchical subgraph has no incoming edges, a new hidden node is created and connected to these nodes by hidden hierarchical edges.

If an iterative coordinates assignment, which places nodes according to priorities calculated from their connected edges, is applied, a unique virtual leaf might be used to restrict the area required by the intermediary layouts. Therefore, nodes having no hierarchical outgoing relations are connected by hidden edges to a newly created hidden leaf. Creating a virtual leaf may also be applied after calculating the rank assignment in S10. In this case, it will be necessary to assign the virtual leaf to an own rank below the maximum rank of the graph.

#### 4.4.11 Breaking Cycles

For general directed graphs, one should compute a minimum size set of *feedback edges*; the direction of these is reversed, so changing it into a acyclic directed graph.

[Sugiyama 2002]

To realize UML\_HIERARCHY, we have built in S3 a pseudo hierarchy, which may contain cycles. As a precondition to the rank assignment S10, the hierarchical subgraph must be acyclic. The usual approach is to internally reverse certain edges and thereby to break the cycles in the graph [Rowe et al. 1987; Sugiyama 2002]. Furthermore, all the nodes in a cycle can simply be collapsed into one node, one of the nodes in the cycle can be duplicated to break the cycle or all nodes in a cycle might be placed in the same rank [Carpano 1980; Sugiyama et al. 1981; Sugiyama and Misue 1991]. While encapsulation using composite nodes might be an approach, the latter two ideas do not meet the requirements of our application domain considering the UML specification and UML\_HIERARCHY.

More formally, the *feedback arc set*  $F$  of a directed (cyclic) graph  $G = (V, E)$   $F \subset E$  contains at least every edge which is a part of a cycle in  $G$ . When  $F$  is removed from  $G$  or all edges in  $F$  are reversed, the resulting graph will be acyclic. Finding the minimal set of feedback arcs is called the *minimum feedback arc set problem* and, unfortunately, it was proven to be NP-complete in [Garey and Johnson 1983].

Different methods for calculating an appropriate feedback arc set or solving the problem of the minimal set on special types of graphs have been described in literature. The first discussion occurred in 1957 in a study of asynchronous logical feedback networks.

Different researchers tried to find an exact solution. In [Hackbusch 1997] the feedback arc set problem for planar digraphs was attacked by transforming the graph into a flow-problem. The flow-problem was then described as systems of linear equations which can be solved in linear time. A genetic method was given in [Jingwei and Zhuo 1994] and in [Saab 2001] an approach based on boolean expressions, which unfortunately requires exponential runtime, was mentioned. More practical and graph drawing related approaches are based on heuristics. From different experiments it is known that reversing the minimum set of feedback arcs does not necessarily lead to a better drawing. Therefore, simply reversing the edges, which participate in many cycles, appears to be a reasonable heuristic.

The basic idea is to traverse the graph in a depth-first search and to label thereby each node with

a unique integer value. Let  $l(v)$  be the label of  $v$ , then an edge  $\vec{e} = (v, w)$  with  $l(w) > l(v)$  is called upward edge. Upward edges may be reversed. An improvement to the simple approach from the perspective of graph drawing is to greedily move nodes having a large outdegree to the top of the drawing. The lowest label is assigned to the node having (locally) the largest outdegree. This step is then repeated recursively on the rest of the nodes [Eades and Sugiyama 1990]. Both ideas run in  $O(|V| + |E|)$ , because both directly rely on a depth-first traversal.

As noted in [Eades and Sugiyama 1990], the complexity can be reduced to  $O(|V|)$  by applying a divide & conquer approach, which also reduces the number of upward edges. According to [Eades and Sugiyama 1990], a topological sorting, which ignores backarcs, appears as a simple, fast but not very effective method. In [Saab 2001], a recursive algorithm based on a bisection function, which partitions a graph in two subsets of nearly equal size, was used to attack the problem. That bisection function was then determined by stochastic evolution or dynamic clustering. Reversing edges based on a set of rules to meet the down-arrow convention was presented in [Sugiyama and Misue 1991]. Finally, a greedy cycle removal algorithm by Eades, Lin and Smyth was described in [Battista et al. 1999].

Another interesting approach relies on the fact that an edge can be part of a cycle only if its start and end point are located in the same strongly connected component. Hence, according to [Sugiyama and Misue 1991; Gansner et al. 1993; Saab 2001] it is sufficient to reverse the edges which occur most in a cycle found by a depth-first traversal of a strongly connected component. It is known that reversing inappropriate edges disturbs the final drawing but from the perspective of stability, the depth-first cycle-breaking heuristic seems to be preferable [Gansner et al. 1993]. We will facilitate the depth-first cycle-breaking heuristic with a minor modification: In S3 various kinds of edges may be included into the pseudo-hierarchy. When breaking cycles, nesting edges should not be reversed at all to ensure UML\_HIERARCHY and UML\_SEMANTIC\_CLUSTERS. Therefore, the individual edge types can introduce an individual priority so that the candidates for reversal are selected according to that priority and the length of the cycle.

#### 4.4.12 Conclusions

Statistics: The only science that enables different experts using the same figures to draw different conclusions.

Evan Esar (1899 – 1995)

In this section, we have discussed the algorithmic steps S1 up to S9 of the preprocessing macro phase of our layout algorithm for UML class diagrams. Furthermore, two common preprocessing tasks, ensuring a virtual root/leaf and breaking cycles, useful for other layout steps, like the rank assignment, or layout algorithms have been described.

The preparation steps collect additional information, e.g., the nesting relations, the membership of edges in the pseudo-hierarchy, the design complexity, etc. and adjust selected subgraphs by changing the structure or encapsulating graph elements into composite nodes. The sequence of the preprocessing steps is determined by procedural and data dependencies rather than by the priorities of the aesthetic rules introduced in Section 3.3.6. Therefore, various aesthetic rules are

prepared, but neither criteria themselves nor their priorities are realized in this macro processing step. Furthermore, the result is a restructured graph, which currently does not impose a layout. This will be successively be introduced by the following macro phases.

Table 4.2 shows the complexities of the individual preprocessing steps. Several steps, like S1, S4, S5, S6, S7 and S8, change the structure of the graph by considering nodes and edges. All steps can be implemented so that each node is and each edge is regarded at most twice. S2 sorts the set of nodes, the set of edges and the partitions of in and out edges of each node. According to the description in Section 4.4.2, we can assume average runtime, e.g., of a quicksort algorithm. Currently, S3 does not provide sophisticated hierarchy detection mechanisms. Therefore, the pseudo-hierarchy can be deduced by taking the type of each individual edge into account. S9 simply searches for nodes without relations.

algorithmic step	runtime complexity
S1	$O( V  +  E )$
S2	$O( V  \cdot \log  V  +  E  \cdot \log  E )$
S3	$O( E )$
S4	$O( V  +  E_H )$
S5	$O( V  +  E_H \cup E_N )$
S6	$O( V  +  E_H \cup E_N )$
S7	$O( V  +  E_H \cup E_N )$
S8	$O( V  +  E_N )$
S9	$O( V )$
preprocessing macro phase	$O( V  \cdot \log  V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$

Table 4.2: Runtime complexities of the preprocessing macro phase.  $E$  denotes the initial set of edges of the input graph, which was not partitioned by S3 so far.

## 4.5 Rank Assignment

You can be a rank insider as well as a rank outsider.

Robert Frost (1874 – 1963)

The first step of the classical hierarchical layout algorithm is the rank assignment (S10 in our approach), i.e. partitioning the set of nodes into several (horizontal) layers and therefore a kind of basic (vertical) coordinates calculation. In this chapter we will adapt the basic definitions for  $n$ -level hierarchies from literature, we will introduce certain validity rules for node sequences and ranks respecting clusters and we will describe extensions to the classical algorithms to gain a specialized version for the layout of UML class diagrams.

### 4.5.1 Previous Work

The first pass finds an optimal rank assignment using a network simplex algorithm.

[Gansner et al. 1993]

In [Warfield 1977] a graph containing a hierarchy was represented as a reachability matrix. Furthermore, an algorithm to calculate a proper reachability matrix was given. That algorithm inserts dummy vertices to restrict every edge to a span<sup>6</sup> of 1.

To choose the minimum number of dummy nodes in [Gansner et al. 1988; Sugiyama 2002] a  $y$  coordinate and therefore the rank of a node satisfies the following properties:

1.  $r(u)$  is an integer for each node
2.  $r(u) \geq 1$  for each node
3.  $r(v) - r(u) \geq 1$  for each  $\vec{e} = (u, v) \in E_H$

Then the rank assignment was calculated by

$$\min \sum_{\vec{e}=(u,v) \in E_H} (r(v) - r(u)) \cdot w(\vec{e}) \quad \text{subject to: } r(v) \geq r(u) + 1 \text{ for all } \vec{e} \in E_H$$

where  $w(\vec{e}) \geq 1$  denotes the weight or the importance of the edge  $\vec{e}$ . Finally, the rank assignment should minimize the number of dummy nodes given by

$$f(y) = \sum_{\vec{e}=(u,v) \in E_H} (r(v) - r(u) - 1)$$

This can also be achieved by the integer program described above with the guarantee that the rank values are integer values. Minimizing the number of dummy nodes directly affects the runtime,

<sup>6</sup>The number of ranks spanned by an edge.

the area required by the drawing and prevents from additional effort in minimizing the number of bends. In [Gansner et al. 1988; Gansner et al. 1993] it was shown that the rank assignment problem can be solved in polynomial time by linear programming as well as transforming the problem into an equivalent min-cost flow or circulation problem [Goldberg and Tarjan 1986; Goldberg and Tarjan 1990]. Another technique, even if it does not necessarily run in polynomial time, is to apply the simplex method due to fact that the constraint matrix is totally unimodular [Cunningham 1976; Gansner et al. 1993].

A method respecting the extents of a page by applying the longest path layering was introduced in [Eades and Sugiyama 1990]: All sinks are placed in the top layer, each remaining node is placed in the layer where the node forms the longest path. On the one side, this method produces a layering of minimum height and it can be used to calculate a shortest-path layering in linear time. On the other side, it induces problems on the width of the drawing:

However, the longest-path layering may be very wide, especially near the bottom of the drawing [...] We need a tool for assigning each node to a layer such that the space used is neither too wide nor too high; this would avoid overcrowding in both directions.

[Eades and Sugiyama 1990]

Unfortunately, the problem of finding such a layering is NP-hard due to the multiprocessor scheduling problem [Garey and Johnson 1983; Eades and Sugiyama 1990].

In [Healy and Nikolov 2002a; Healy and Nikolov 2002b], a branch-and-cut approach to the directed acyclic graph layering problem was described. An integer linear programming formulation was given which combines the positive aspects of Coffman-Graham [Coffman and Graham 1972], longest path layering [Eades and Sugiyama 1990] and the Gansner method [Gansner et al. 1988; Gansner et al. 1993] with respect to given width and height bounds.

While the classical Sugiyama algorithm [Sugiyama et al. 1981] relies on [Warfield 1977] respecting hierarchization, in [Sugiyama and Misue 1991; Brockenauer and Cornelsen 2001] the hierarchization for compound graphs was performed by creating a so called derived graph. The adjacency edges were replaced by two types of edges representing less-than and less-than-equal relations arising from the down-arrow convention for compound graphs. After the substitution of certain edge combinations and cycles, the derived graph was then used for assigning compound levels to all the vertices. Finally, as in most approaches, the hierarchy had to be normalized by introducing an appropriate number of dummy nodes and edges.

Another rank assignment algorithm specific to compound graphs was presented in [Sander 1996b]. The nesting graph (similar to the derived graph in [Sugiyama and Misue 1991]) of a compound graph was created. Cycles were removed by applying certain rules and finally the nesting graph was traversed in topological order to produce a legal rank assignment.

Further and more sophisticated approaches addresses the problem of *local layering* [Schreiber 2002]. Hereby a node with a large height is spread over several (sub)layers and therefore reduces the overall height of the drawing. This appears interesting, in particular for UML class diagrams, in which the variance of node sizes might be relatively high, due to the individual number of attributes and operations of the classes. On the one side, this may be a future improvement to reduce the height of a drawing (UML\_GRAPHDRAWING via GDR\_DRAWING\_SIZE) and to



improve compaction. On the other side, the clarity of the hierarchy (UML\_HIERARCHY) might be disturbed.

## 4.5.2 Basic Definitions

To freely bloom - that is my definition of success.

Gerry Spence

In [Sugiyama et al. 1981], the term *n-level hierarchy* was introduced as a partition of the nodes of a graph according to levels, whereby a level (or rank) was defined as a sequence of nodes belonging to the same hierarchical layer. Furthermore, the edges were also partitioned so that each edge connects adjacent ranks only. Edges, which span over more than one layer, were split by dummy nodes into edge chains to gain a *proper n-level hierarchy*.

We have to adapt that basic definition to our application domain, because non-hierarchical edges are not considered by the Sugiyama algorithm and therefore not treated by the usual definition of a *n-level hierarchy*.

### Definition 9 (*n-level hierarchy*)

A *n-level* ( $n \geq 1$ ) hierarchy  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  is a directed graph  $G = (V, E_H, E_N)$  which satisfies the following conditions:

1.  $V$  can be partitioned into  $n$  subsets<sup>7</sup> that is  $V = V_0 \cup V_1 \cup \dots \cup V_{n-1}$  ( $V_i \cap V_j = \emptyset, i \neq j$ ).  $n$  is called the length of the hierarchy.
2. For edge  $\vec{e} = (v_i, v_j) \in E_H \cup E_N$  with  $v_i \in V_i$  and  $v_j \in V_j$ ,  $i < j \leq i + 1$  if  $e \in E_H$  or  $i \leq j \leq i + 1$  if  $e \in E_N$  holds, respectively. Long span edges are assumed to be split by invisible nodes to fulfill this condition.
3. The set of edges  $E = E_H \cup E_N$  can be partitioned as follows:

$$\begin{aligned} E &= E_H \cup E_{NH} \cup E_{NF} \\ &= (E_{H_0} \cup \dots \cup E_{H_{n-2}}) \cup (E_{NH_0} \cup \dots \cup E_{NH_{n-2}}) \cup (E_{NF_0} \cup \dots \cup E_{NF_{n-1}}) \end{aligned}$$

with

$$\begin{aligned} E_{H_i} &:= \{\vec{e} : \vec{e} = (v, w) \in E_H, v \in V_i, w \in V_{i+1}\} \\ E_{NH_i} &:= \{\vec{e} : \vec{e} = (v, w) \in E_N, v \in V_i, w \in V_{i+1}\} \\ E_{NF_i} &:= \{e : e = \{v, w\} \in E_N, v \in V_i, w \in V_i\} \end{aligned}$$

4. An order  $\sigma_i = v_0 v_1 \dots v_{|V_i|-1}$  of  $V_i$  with  $0 \leq i \leq n - 1$  and  $v_j \in V_i, 0 \leq j < |V_i|$  is given for each  $i$ , where the term "order" means a sequence of all vertices of  $V_i$ , e.g., after permutating  $V_i$ .  $\sigma_i$  is called the *i-th rank* (or level) of graph  $G$ .

For node  $v \in V_i$  and therefore  $v \in \sigma_i$  we denote  $r(v) = i$  as the rank number of  $v$ . For the sequence  $\sigma_i = v_0 v_1 \dots v_k \dots v_{|V_i|-1}$  with  $v_k = v$  we denote  $\sigma_i(v) = k$  as the 0-based index of  $v$  in  $\sigma_i$  and as the counterpart  $\sigma_i[k] = v$  as an index access to  $\sigma_i$ .

We allow  $n = 1$  in definition 9, because graphs having non-hierarchical relations only or exactly one node should be processed, too.

As mentioned above, edges, which span over multiple ranks, have to be split up to meet definition 9 and to be handled by the following parts of the layout algorithm. An edge reaching from rank  $i$  to  $j$ ,  $i < j \wedge j - i > 1$  is split up into  $j - i$  edges by  $j - i - 1$  hidden nodes positioned in each  $V_k$   $i + 1 < k < j - 1$ .

**Example:**

Figure 4.14 (a) shows a simple digraph a  $n$ -level hierarchy should be calculated for. In Fig-

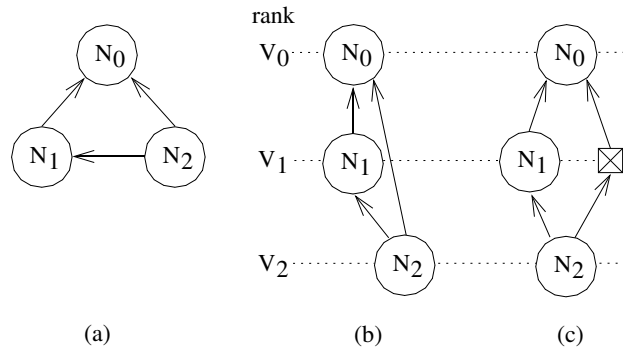


Figure 4.14: Building a 3-level hierarchy from the graph in (a) by (b) rank assignment and (c) insertion of dummy nodes to be compliant to definition 9.

ure 4.14 (b) the nodes are partitioned into 3 ranks but yet the graph is not an  $n$ -level hierarchy as introduced by definition 9. In Figure 4.14 (c) the hidden node, which is necessary to meet definition 9, is inserted.

**Definition 10 (length, minimum length and weight of an edge)**

Let  $\vec{e} = (v, w)$  be a directed edge of graph  $G = (V, E_H, E_N)$ ,  $\vec{e} \in E_H \cup E_N$ ,  $v, w \in V$ . The length  $l_r(\vec{e})$  is defined as  $l_r(\vec{e}) := r(w) - r(v)$ . Let the (externally given) minimum length constraint be  $\delta_r(\vec{e}) \geq 1$ . Furthermore, let  $w(\vec{e}) \geq 1$  be the (externally specified) weight of an edge, which describes the edge's importance. Usually  $w(\vec{e}) = 1$ .

The minimum length constraint and the weight of an undirected edge  $e$  are introduced similarly. The length of an undirected edge  $l_r(e)$  is defined by  $l_r(e) := |r(w) - r(v)|$ .

<sup>7</sup>In this work, indices run over  $0 \dots n - 1$  on a set of  $n$  elements according to the implementation.

### 4.5.3 Validity Rules

The rule is perfect: in all matters of opinion our adversaries are insane.

Mark Twain (1835 – 1910)

The nodes can be partitioned into hierarchically nested clusters by a hierarchical node naming

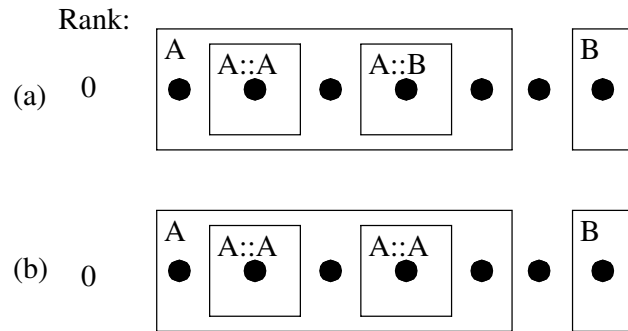


Figure 4.15: Cluster sequence within one rank: (a) is valid, (b) is invalid

function, which was introduced in Section 4.3.4. According to `UML_SEMANTIC_CLUSTERS`, `UML_NODES` and `UML_EDGES` the clusters of a graph should

- be clearly visible as individuals.
- not overlap other edges, other nodes and clusters except for nested clusters/nodes.
- not be split up into different subsets within one rank if the cluster does not contain further subclusters.

These intuitive aesthetic rules have to be specified as validity rules on the ranks of a graph to be respected by all parts of the layout algorithm.

In [Sander 1996b; Forster 2002] the following common rules to ensure validity are suggested:

1. The nodes on a layer that belong to the same compound node must be placed next to each other with no other nodes between them.
2. The relative position of two compound nodes must be the same on all layers, i.e., compound nodes must not “cross” each other.

In this section, we will formalize these rules by specifying them in terms of the node naming function.

Figure 4.15 (a)<sup>8</sup> depicts a valid sequence of clusters within a rank. Members of the global cluster

<sup>8</sup>In this section, nodes are drawn as filled circles and clusters as rectangles, because individual names for nodes are not required here.

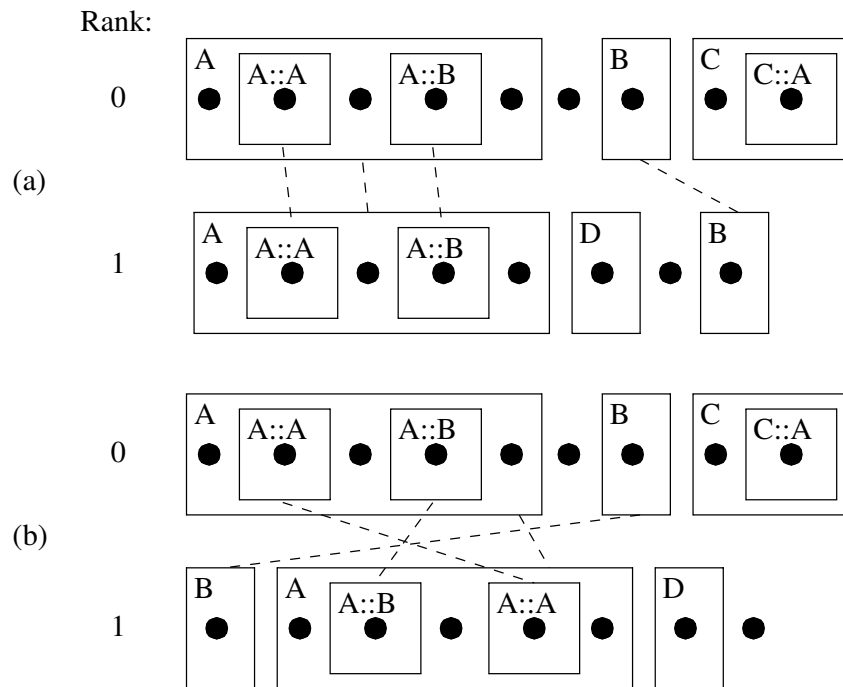


Figure 4.16: Cluster sequences considering adjacent ranks: (a) is valid, (b) is invalid, because the sequence of the common clusters are not equal.

may appear between two different clusters but not inside a cluster. The nesting of  $A : A$  and  $A : B$  inside of  $A$  is valid. Figure 4.15 (b) shows an invalid cluster sequence, because  $A : A$  must not be separated by a node of  $A$ . This kind of rule takes only one rank at a time into account and is therefore called *intra-rank validity* in the following.

Cluster may span over adjacent ranks and so far the *intra-rank validity* respects the cluster sequence in one rank only. At least the sequences of the clusters common to both ranks have to be equal. Clusters which are not mentioned in both ranks may occur at positions according to the *intra-rank validity* rules. Figure 4.16 (a) shows two adjacent ranks having equal sequences of the common clusters. We assume that the upper rank is the reference rank the lower rank has to be aligned to or to be tested on. Figure 4.16 (b) is invalid because  $B$  occurs at the left side of  $A$  and  $A : B$  at the left side of  $A : A$  in contradiction to the sequence of the upper rank. This kind of rule takes cluster-relations between adjacent ranks into account and is called *inter-rank validity* in the following.

Obviously, a graph is valid if all of its ranks are *intra-rank* and *inter-rank* valid.

Next, formal definitions, which specify the *intra-rank*, *inter-rank* and *graph validity*, will be given.

**Definition 11 (intra-rank validity)**

Let  $\bar{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy of  $G$  and  $(C, crit, \preceq)$  a hierarchical node naming function on  $G$  according to definition 7. Let

$$pairs(r) := \{(v, u) : v, u \in \sigma_r, v \neq u \wedge v \bowtie u \wedge \sigma_r(v) < \sigma_r(u)\} \quad (4.1)$$

be the set of cluster-related nodes in rank  $r$ . Without loss of generality it is required that  $v$  occurs before  $u$  in  $\sigma_r$ . The sequence of nodes in  $\sigma_r$  is intra-rank-valid if

$$rankFit(r) := \forall_{(v,u) \in pairs(r)} \forall_{\sigma_r(v) < i < \sigma_r(u)} \neg notClusterRelated(crit(\sigma_r[i]), crit(v)) \wedge \sigma_r[i] \preceq LCC(v, u) \quad (4.2)$$

holds so that for each pair of cluster-related nodes there is no node in between which is not comparable or assigned to a parent cluster. Finally, the entire hierarchy is intra-rank-valid if

$$\forall_{0 \leq r < n} rankFit(r) \quad (4.3)$$

**Example:**

In Figure 4.15 (a)

$$pairs(0) = \{(A, A :: A), (A, A), (A, A :: B), (A, A), (A :: A, A), (A :: A, A :: B), (A :: A, A), (A, A :: B), (A, A), (A :: B, A)\}$$

Even if multiple occurrences of the same pair seem to be returned, each node (here represented by its cluster name) is an individual and therefore different parts of the rank are considered for each pair in  $rankFit$ . Neither parent cluster names of a  $A$  nor not comparable cluster names occur between the nodes of the subranks marked by the individual pairs. Therefore,  $rankFit(0)$  holds.

In Figure 4.15 (b)

$$pairs(0) = \{(A, A :: A), (A, A), (A, A :: A), (A, A), (A :: A, A), (A :: A, A :: A), (A :: A, A), (A, A :: A), (A, A), (A :: A, A)\}$$

is returned. Between  $(A :: A, A :: A)$  the parent cluster name  $A$  occurs, which prevents the validity.

**Definition 12 (inter-rank validity)**

Let  $\bar{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy of  $G$  and  $(C, crit, \preceq)$  a hierarchical node naming function on  $G$  according to definition 7. Let  $d \in \{-1, 1\}$  be the direction of the reference rank  $\sigma_{r+d}$  rank  $\sigma_r$  should be tested on and  $0 \leq r < n, 0 \leq r+d < n$ . Let  $augmented(r)$  be the sequence of first-occurrences of all containing cluster names of each node in  $\sigma_r$ . The global cluster

must not be considered in  $augmented(r)$ , because members of the global cluster may occur everywhere at intra-rank valid positions. Then  $\sigma_{r+d}$  and  $\sigma_r$  are inter-rank valid if  $augmented(r)$  and  $augmented(r+d)$  are equal regarding elements which occur in both ranks only.

Augmenting a given sequence of node names simulates the presence of cluster border nodes, which are not required to be member of the ranks until coordinates assignment in *SugiBib*.

**Example:**

In Figure 4.16 (a),  $augmented(0) = A A::A A::B B C C::A$  and  $augmented(1) = A A::A A::B D B$ . Let  $augmented(r_1, r_2)^R$  be the sequence of first-occurrences of all containing cluster names of each node in  $\sigma_{r_1}$  restricted to elements, which occur in  $\sigma_{r_1}$  and in  $\sigma_{r_2}$ .  $augmented(0, 1)^R = A A::A A::B B =_N A A::A A::B B = augmented(1, 0)^R$ . Hence, the depicted ranks are inter-rank valid.

In Figure 4.16 (b)  $augmented(0) = A A::A A::B B C C::A$ ,  $augmented(1) = B A A::B A::A D$  and  $augmented(0, 1)^R = A A::A A::B B \neq_N B A A::B A::A = augmented(1, 0)^R$ . Therefore, these ranks are not inter-rank valid.

**Definition 13 (graph validity)**

Let  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy of  $G$  and  $nodeNamingTupelCN$  a hierarchical node naming function on  $G$  according to definition 7.  $\tilde{G}$  is valid respecting the given naming function, if all ranks are intra-rank valid according to definition 11 and the hierarchy is inter-rank valid according to definition 12.

#### 4.5.4 The Core Algorithm

At the innermost core of all loneliness is a deep and powerful yearning for union with one's lost self.

Brendan Francis

In [Gansner et al. 1993], the rank assignment problem for graph  $G = (V, E_H, E_N)$  was formulated as

$$\begin{aligned} \min \quad & \sum_{\vec{e}=(v,w) \in E_H} w(\vec{e}) \cdot l_r(\vec{e}) \\ \text{subject to:} \quad & \forall_{\vec{e}=(v,w) \in E_H} l_r(\vec{e}) \geq \delta_r(\vec{e}) \end{aligned}$$

Since the first version of *SugiBib* [Eichelberger 1999] we apply the network simplex method and due to our experience we can agree to the statement that “Although its time complexity has

not been proven polynomial, in practice it takes few iterations and runs quickly” [Gansner et al. 1993].

---

**Algorithm 4.2** rankAssignment
 

---

**input:**  $G = (V, r, E_H, E_N)$

**output:**  $(G, \tilde{G} = (V, E_H, E_N, n, \sigma))$

$G' := select(G)$

$G' := preprocess(G')$

$G' := hierarchicalRankAssignment(G', 0, true)$

$G := nonHierarchicalRankAssignment(G)$

$G := postprocess(G)$

$G := createEdgeChains(G)$

**return**  $(G, createRankStructure(G))$

---

Algorithm 4.2 selects the subgraph, which represents the pseudo-hierarchy to be processed by the hierarchical rank assignment algorithm, and performs some application domain preprocessing. Due to the nature of the graph, the preprocessing step may ensure an acyclic graph as input. Additional rank assignment transformations for UML class diagrams will be discussed later in Section 4.5.5. Then the hierarchical and the non-hierarchical rank assignment are calculated. In an application-domain specific postprocessing step, corrections to the basic rank assignment are carried out. `createEdgeChains` inserts dummy nodes to fulfill definition 9 as depicted in Figure 4.14.

Finally, `createRankStructure` determines the vertex sequences for each  $\sigma_i$   $r(v)$  on a node  $v \in V$  was introduced in definition 9 via the membership of  $v$  in a  $V_i$  or  $\sigma_i$ , respectively. In the description of the rank assignment algorithms, we will use  $r(v)$  to also denote the desired target rank assignment of individual nodes, even if a concrete partition was not created physically so far. As long as `createRankStructure` is not called, we assume, because of notational convenience that the individual  $r(v)$  values are, however, stored in  $G$ .

If no node naming function is given, the node partitions  $V_i$  can simply be obtained by iterating through all nodes in  $V$  and considering  $r(v)$ . In this case, the sequences in  $\sigma_i$  are determined indirectly by the sequence of  $V$  initially calculated in step S2 of the main layout algorithm.

If a node naming function is given, an ordering on the node names is implied by definition 7. Therefore, the properties of the node naming function can be used to sort each  $\sigma_i$  to be compliant to definition 13. In the case of the hierarchical naming system defined by UML, representing cluster names as strings and applying an ascending sorting function with respect to the general cluster name is appropriate. Obviously the intra-rank validity from definition 13 holds, because all nodes, which are member of the same cluster, are in sequence and different clusters are ordered according to the ascending string sorting. Members of the general cluster are then located to either the left or the right side of a rank. Nodes within the same cluster might be ordered similar to step S2 or according to `UML_CONSTRAINT_SEQUENCE`. The inter-rank validity from definition 13 holds, because the order of adjacent ranks restricted to cluster names occurring in both ranks are the same due to the ordering properties of the node naming function.

**Algorithm 4.3** hierarchicalRankAssignment**input:**  $G = (V, r, E_H, E_N), min, bal$ **output:**  $G$ 


---

```

T := feasibleTree(G)
while  $e := \text{leaveEdge}(G, T) \neq \perp$  do
     $f := \text{enterEdge}(G, T, e)$ 
     $G := \text{exchange}(G, T, e, f)$ 
end while
G := normalize(G, min)
if bal = true then
     $G := \text{balance}(G)$ 
end if
return G

```

---

Algorithm 4.3 shows the rank assignment algorithm which was described in [Gansner et al. 1993] based on the network simplex method. A graph is called feasible, if  $\forall_{e \in E_H} l_r(e) \geq \delta_r(e)$ . `feasibleTree` constructs an initial, feasible spanning tree. The simplex method starts with a feasible solution and maintains the feasible constraint as an invariant. `leaveEdge` returns an edge of the spanning tree which should be replaced by a tight edge having  $l_r(e) = \delta_r(e)$  to probably reduce the weighted edge length sum of all edges towards an optimal rank assignment. `enterEdge` selects an appropriate edge to replace  $e$  and to keep the graph feasible. `exchange` swaps  $e$  and  $f$  and updates internal structures for an efficient computation. `normalize` ensures that the minimum rank number is zero and that all integer numbers between zero and the maximum normalized rank number are used. `balance` is a standard postprocessing function described in [Gansner et al. 1993]: It performs an aspect-ratio optimization by moving nodes having equal in- and out-edge weights and multiple feasible ranks to the feasible rank with the fewest members.

### 4.5.5 UML and Cluster Specific Adjustments

The art of life lies in a constant readjustment to our surroundings.

Okakura Kakuzo

In this section, adjustments of the basic rank assignment due to clusters and UML class diagrams will be given. In the description of algorithm 4.2, some methods were left out so that they can act as hot spots to plug in application domain specific algorithms. In the first part, preprocessing steps will be discussed. Then the rank assignment for non-hierarchically connected nodes based on the “incremental extension” from [Seemann 1997] is presented. Finally, various postprocessing tasks due to the compound nature of the graph will be pointed out.



## Preprocessing

---

### Algorithm 4.4 preprocess

---

**input:**  $G = (V, E_H, E_N)$

**output:**  $G$

$G := \text{transformToAcyclicGraph}(G)$

$G := \text{ensureVirtualRoot}(G)$

$G := \text{enforceHierarchicalClusterDependencies}(G)$

$G := \text{respectHierarchicalClusterRelations}(G)$

**return**  $G$

---

Algorithm 4.4 shows the steps to be executed on the subgraph representing the pseudo hierarchy. Therefore, `select` in algorithm 4.2 is redefined to deliver the members of the pseudo hierarchy only.  $G$  is transformed into an acyclic graph as described in Section 4.4.11. Then the presence of a virtual root in the case of hierarchically disconnected forests is ensured.

In Section 4.2 we pointed out that a cluster consists of a cluster base node, contained nodes and, dependent on the state of the layout process, cluster border and cluster separator nodes. The nesting relations have been temporarily represented by inserting hierarchical edges in step S4, and therefore clusters and their contained nodes can be considered in the rank assignment as usual nodes.

Because of hierarchical relations it may occur that contained nodes across clusters are connected but not the clusters themselves. This would lead to a side-by-side placement of the clusters

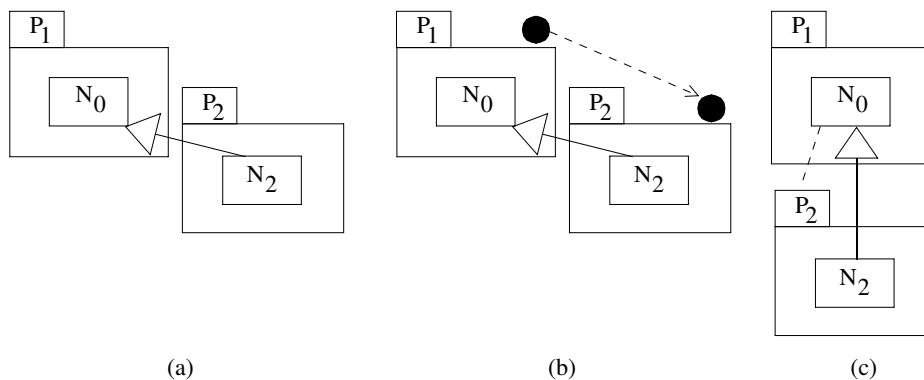


Figure 4.17: (a) unpleasing situation after default rank assignment considering nesting relations, (b) the compound dependency graph admits a leaf to be moved, (c) rank assignment with temporary hierarchical relation due to invisible compound dependencies.

as depicted in Figure 4.17 (a) and `UML_HIERARCHY` on the semantically implicit relations between clusters would not be emphasized. To prevent this situation, the cluster dependency graph illustrated in Figure 4.17 (b) can be calculated by also considering implicit relations which

arise from edges between members of the clusters. Considering the structure of the dependency graph, additional hierarchical edges can be temporarily inserted into the original graph. This has to be done carefully, because no cycles must be introduced by new edges. Therefore, for each leaf in the dependency graph, appropriate connections are inserted and the leaf is removed from the dependency graph. This is repeated until no leaves are contained in the dependency graph. Then, after calculating the rank assignment, these additional hierarchical edges will be removed.

---

**Algorithm 4.5** enforceHierarchicalClusterDependencies
 

---

**input:**  $G = (V, E_H, E_N)$ 
**output:**  $G$ 
 $V_d := \{\}$ 
 $E_d := \{\}$ 
 $G_d = (V, E_d)$ 
**for all**  $\vec{e} = (v, w) \in E_H$  **do**
 $c := LCC(v, w)$ 
 $v_c := listGet(\{x : x \in ulc(v), crit(x) =_N c\}, 0)$  {at maximum one by construction}

 $w_c := listGet(\{x : x \in ulc(w), crit(x) =_N c\}, 0)$ 
**if**  $v_c \neq \perp \wedge w_c \neq \perp \wedge |\{f : f = \{v_c, w_c\} \in E_d\}| = 0$  **then**
 $V_d := V_d \cup \{v_c, w_c\}$ 
 $\vec{f} := new\ Edge(v_c, w_c)$ 
 $E_d := E_d \cup \{\vec{f}\}$ 
**end if**
**end for**
 $D = \{\}$ 
**repeat**
**for all**  $v \in \{w : w \in V_d, |out(w)| = 0\}$  **do**
**for all**  $w \in V^-(v)$  **do**
**if**  $|\{f : f = \{v, w\} \in E_H\}| = 0$  **then**
 $D := D \cup \{(v, w)\}$ 
**end if**
**end for**
 $remove(G_d, v)$ 
**end for**
**until**  $|\{w : w \in V_d, |out(w)| = 0\}| = 0$ 
 $sort(D, decendingLCC(.))$ 
**for all**  $(v, w) \in D$  **do**
 $E_H := E_H \cup \{\vec{f} := new\ Edge(deepestContained(G, v), w)\}$ 
**end for**
**return**  $G$ 


---

Algorithm 4.5 shows the construction of the dependency graph  $G_d$  in the first loop. In the second loop the leaves are processed successively. Thereby, a leaf is connected to the “deepest” node

in the depending cluster: Let  $w$  be a cluster dependent on cluster  $v$  as shown for  $P_2$  and  $P_1$  in Figure 4.17. To ensure that  $w$  will occur below  $v$  if  $w$  is a leaf in the dependency graph,  $w$  has to be connected to the “deepest” node in  $v$ . By successively inserting cluster dependencies, also the currently deepest node changes. Therefore, the dependencies are inserted in sequence of the most specific edges given by a descending sort on the least common cluster criteria. The deepest node can then be retrieved by a depth-first traversal also considering cluster dependencies inserted so far.

Similar to the preprocessing of invisibly connected clusters, also nodes considered for UML\_CENTER or UML\_COMMENTS can be handled. Several nodes, like the central node of  $n$ -ary associations, collaborations or comments should be centered upon their connected nodes. A node which is exclusively connected to its central node can later easily be assigned to adjacent ranks. If too few of the exclusively connected nodes are present, also nodes in further relationships have to be taken into account. These nodes can be forced into a lower or an upper rank by connecting them by invisible hierarchical edges to the desired centered node. Thereby, as mentioned above, cycles have to be avoided. According to the different priorities of aesthetic principles, as visualized in Table 3.3, first hidden edges due to nesting, then due to UML\_CENTER and finally due to UML\_COMMENTS are considered for insertion.

Without preprocessing, the class diagram in Figure 4.18 (a) would lead to the result in Figure 4.18 (b), which does not emphasize hierarchical aspects according to UML\_HIERARCHY and UML\_SEMANTIC\_CLUSTERS. In this preparation step, we add a cluster border node to the cluster and reconnect the hierarchical edges (especially those inserted by algorithm 4.5) from the cluster parent to the newly inserted cluster border node. This is illustrated in Figure 4.18 (c). Accidentally, the rank assignment algorithm then might place the cluster border node in one of the ranks together with other contained nodes. Therefore another postprocessing step will have to ensure that the cluster border node is assigned to the maximum rank of the cluster.

Another cluster situation has to be considered if UML\_COUPLING is enabled. In this case, it has to be ensured that nodes contained in a cluster, which have no incoming edges, are connected to a newly created hidden node. Otherwise, the separation of outside coupled and not-outside coupled nodes in the top rank of a cluster cannot be guaranteed<sup>9</sup>.

### Rank Assignment for non-hierarchically Connected Nodes

In [Seemann 1997], an extension of the Sugiyama algorithm was proposed, which partitions the set of edges into the inheritance subgraph and the subset of association edges. As described in Section 4.1 and also mentioned above, there are two basic variants to perform that algorithm: The Sugiyama-Seemann combination, in which non-hierarchical edges and non-hierarchically connected nodes are removed before calculating the rank assignment and the edge crossing reduction, and the integrated version, which performs the ranking on the hierarchical subset of the edges and does the edge crossing minimization on the entire graph. In both variants, the nodes only connected by non-hierarchical edges have to be assigned to the  $n$ -level hierarchy some-

<sup>9</sup>This will be illustrated with a cluster specific postprocessing of the rank assignment in Figure 4.20.

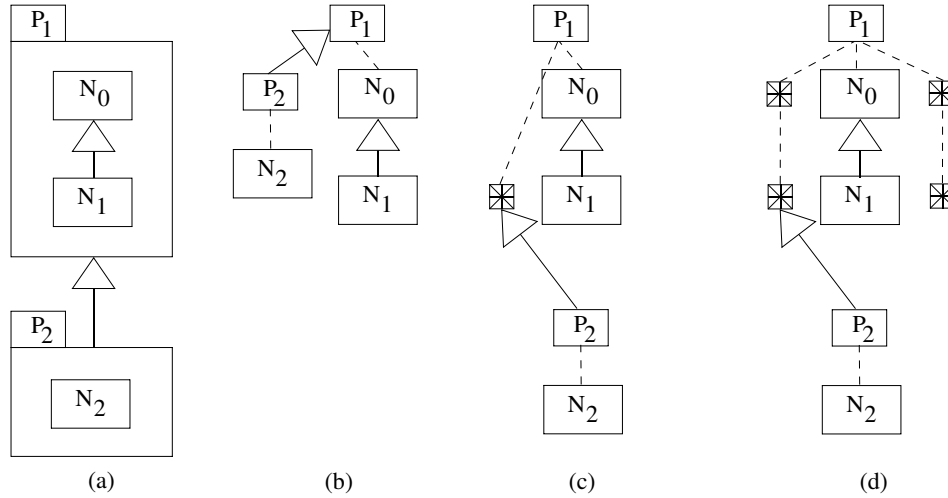


Figure 4.18: (a) derived packages as UML class diagram (b) erroneous rank assignment without cluster border nodes (c) graph with compound hierarchy and the mapped inheritance edge connected to a cluster border node (d) explicit cluster borders ready for coordinates calculation.

when. In [Seemann 1997], this is done by the “incremental extension” step, which reinserts the removed edges and nodes into the graph at appropriate rank and index positions directly before coordinates calculation. As a drawback, in this variant non-hierarchical edges are not considered by the edge crossing reduction. To realize the integrated version, the rank assignment of the non-hierarchically connected nodes has to be calculated immediately before executing the UML specific rank assignment. Concrete positions of nodes in ranks are initialized at the end of the basic rank assignment algorithm 4.2 and final positions are determined by the edge crossing reduction (S11). Therefore, in the integrated version, positions of nodes in ranks are not relevant. The following algorithm is an adaption of the “incremental extension” algorithm proposed in [Seemann 1997] for the integrated version.

1. Partition the set of nodes  $V$  into  $V_H$ , the set of nodes which are hierarchically connected by edges in  $E_H$ , and  $V_N$ , the set of nodes connected by non-hierarchical edges to nodes in  $V_H$ .
2. Assign the nodes in  $V_N$  to the nodes in  $V_H$  by constructing sets  $S(v)$ ,  $v \in V_H$  in two steps:  $w \in V_N$  is inserted into  $S(v)$  if
  - (a)  $edges(w) \cap E_N = \{e = \{v, w\}\}$ , i.e.  $w$  is connected to  $v$  only
  - (b)  $|edges(w) \cap E_N| > 1 \wedge e = \{v, w\} \in edges(w) \cap E_N \wedge S(v) = listGet(\{S(x) : |S(x)| = \min_{f=\{y,w\} \in edges(w) \cap E_N} |S(y)|\}, 0)$ , i.e.,  $w$  is connected to multiple nodes in  $V_N$  and  $S(v)$  is one of the sets having the minimum number of elements. As denoted above,  $S(v)$  is selected deterministically from a candidate list, which was calculated according to the implicit sequence of nodes in  $V_H$ .

3. Determine if the nodes in  $S(v)$  can be inserted into rank  $r(v)$  or if further sub ranks have to be introduced.

- (a) Let  $V_H^C := \{w : w \in V_H, x \in S(v), e = \{x, w\} \in E_N, r(w) = r(v)\}$  be the set of nodes in  $V_H$  connected by non-hierarchical edges to the nodes in  $S(v)$ . Temporarily build a composite node  $v_c$  as shown in Figure 4.2 from the nodes in  $V_H^C$  to keep these nodes in the same rank. Let  $V_C := S(v) \cup \{v_c\}$  and  $E_C := \{e : e = \{w, x\} \in E_N, w, x \in V_C\}$ . Then the subgraph

$$G_C = (V_C, v_c, E_C, \emptyset)$$

consists of  $S(v)$  and all nodes in  $V_N$  connected to the nodes in  $S(v)$  and all edges between these nodes.

- (b) Transform  $G_C$  to an acyclic graph by applying the algorithm described in Section 4.4.11.
- (c) Compute a rank assignment e.g., by algorithm 4.3, without balancing and UML optimizations.
- (d) Because all edges in  $G_C$  are non-hierarchical edges in  $G$ , nodes connected to at least two different nodes would require extra space. Process the nodes ordered according to the currently assigned rank numbers and move the node  $w$  to rank
- $r(x)$  if  $w$  is connected to  $x$  only.
  - $\min\{r(y), r(x)\}$  if  $w$  is connected to  $x$  and  $y, x \neq y$  and  $r(x) \leq r(w) \wedge r(y) \leq r(w)$ .
- (e) Release the constraints on the nodes of  $V_N$  by breaking the temporarily inserted composite node  $v_c$  and restoring the reversed flags according to  $G$ .
- (f) Shift the existing ranks of  $V_C \setminus V_H^C$  downwards, thereby reuse existing subranks and mark new ranks as subranks.

This algorithm realizes the edge partition aspect of UML\_HIERARCHY. The hierarchical edges and some aspects of UML\_SEMANTIC\_CLUSTERS have been prepared by UML specific steps as described above and induced the main skeleton of the graph while processing a usual rank assignment method. Now, also the non-hierarchical edges as the second partition have been considered and all nodes are assigned to ranks, even if further corrections to ensure UML\_SEMANTIC\_CLUSTERS and UML\_SPATIAL, which will be described below, have to be executed.

As discussed in Section 4.1 and Section 4.4.8, selected edges can be defined as hierarchical edges to realize certain aspects of UML\_CENTER and UML\_COMMENTS. With respect to the current rank assignment, nodes exclusively connected to a n-ary association, a collaboration or a comment can easily be assigned to adjacent ranks. In the case that no such exclusively connected nodes exist, hidden hierarchical edges have been inserted in the UML specific preprocessing.

## Postprocessing

Due to the nature of UML class diagrams, some edges, which are connected to the virtual root, can be elongated to improve the layout of the graph. Then, depending on the presence of a

hierarchical node naming function, certain transformations of the rank assignment are carried out to reflect a correct nesting. Finally, the rank assignment is compacted and normalized. The individual steps, shown in algorithm 4.6, will be described in detail below.

---

**Algorithm 4.6** postprocess
 

---

**input:**  $G = (V, E_H, E_N)$ 
**output:**  $G$ 

```

 $G := stretchEdges2VirtualRoot(G)$ 
 $G := adjustClusterBorderNodes(G)$ 
 $G := adjustCoupledClusterMembers(G)$ 
 $G := redistributeDisconnected(G, V)$ 
 $G := removeEmptyRanks(G)$ 
 $G := normalize(G, 0)$ 
 $G := removeClusterDependencyRelations(G)$ 
 $G := insertClusterBorderNodes(G)$ 
return  $G$ 

```

---

The basic rank assignment algorithm usually shortens the edges, in particular those connected to the root of the graph. In the case of a virtual root, the edges between the virtual root and the connected edges will not be drawn. Therefore shortening these edges is not required. The separate sub roots of the subgraphs in the initial forest can be emphasized (UML\_HIERARCHY via SE\_FORESTS) and the lengths of the visible edges can be shortened by moving the sub roots down towards their visible children. In Figure 4.19,  $v$  and  $y$  are sub roots below the virtual

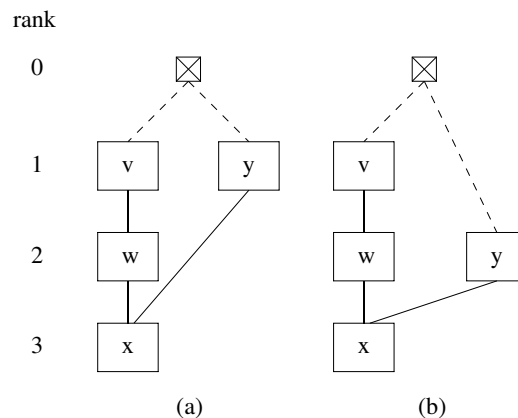


Figure 4.19: (a) result produced by the basic rank assignment (b) more appropriate rank assignment for UML class diagrams

root, the edges between the virtual root and  $v$  or  $y$ , respectively, are dashed because they will finally not be drawn. Even if in Figure 4.19 (b) the edge from the virtual root to  $y$  is longer than in the theoretical optimal situation, it is more appropriate because the edge between  $y$  and its

descendants ( $x$  in this case) is shortened.

As described as UML specific preprocessing of the rank assignment, cluster border nodes are inserted into a cluster if the cluster acts as start node of hierarchical relations. Unfortunately, there is no constraint in the basic rank assignment algorithm, which keeps the cluster border at the bottom of the cluster as depicted in Figure 4.18. The externally specified minimum edge length introduced by definition 10 cannot be used, because the number of ranks allocated by a compound would only be available after a kind of pre-rank assignment.

`adjustClusterBorderNodes` searches for a cluster border node in each cluster of the graph, finds thereby the maximum rank of each cluster and moves a present cluster node into this rank. Thereby, `adjustClusterBorderNodes` realizes `UML_HIERARCHY` for clusters. To realize `UML_SPATIAL`, in particular `UML_COUPLING`, cluster members connected to elements outside the cluster should be placed at the boundary of the containing cluster. Similarly, cluster members not connected to any elements outside should be located near to the center of the cluster. Additionally, an optional invisible border area between coupled and non-coupled cluster members can be introduced. As discussed in Section 3.3, from the viewpoint of the UML specification, such a semantic based spatial distribution is permitted, because the UML does not specify advanced layout issues.

To realize coupling induced spatial distribution, we may have to adjust the rank assignment of the cluster members, which have no incoming or no outgoing edges. Figure 4.20 (a) shows a drawing based on the basic rank assignment algorithm without preparing the spatial distribution. Figure 4.20 (b) depicts the situation after correcting several nodes when spatial distribution is

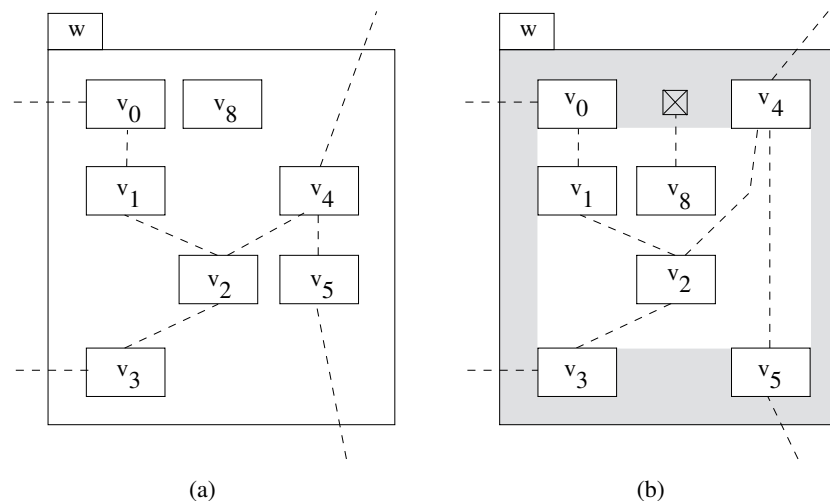


Figure 4.20: (a) result produced by the basic rank assignment algorithm (b) more appropriate rank assignment if coupling of nodes should be emphasized. Outside coupled nodes are marked by a shaded region here, Furthermore, the hidden node required to allow the separation in the top rank is shown.

activated.

For each cluster, the minimum and maximum rank is determined and nodes connected to outside the cluster are reassigned to the appropriate rank.

So far, nodes connected by invisible hierarchical relations arising from S14 may not be distributed evenly over the ranks of a cluster. By considering each cluster, the number of connected nodes in each rank of a cluster can be determined and the ranks containing a low number of nodes can be filled up. On the one side, the sizes of nodes and therefore the area required by a rank is currently not accessible to the rank assignment implementation but having access to the extents here would probably improve the result. On the other side, in particular scaling of nodes, like due to UML\_SIZE\_NODES, which will be performed in S15, would introduce uncertainty relying on that information here.

As described above, globally empty ranks might occur because of the extensive postprocessing operations. Therefore an algorithm, which successively fills empty ranks by shifting higher levels downward is required. A similar algorithm was also mentioned in [Sander 1996b].

In the preprocessing or the UML specific rank assignment, various edges and some hidden nodes may have been inserted to enforce a hierarchical layout of implicitly connected clusters and to prepare UML\_COUPLING. These edges have been considered by the basic rank assignment algorithm and are not required anymore.

Obviously, invisible cluster dependencies play an important role. Especially when calculating the coordinates in S15, these invisible relations between clusters will be considered again. Therefore, in a concrete implementation, it makes sense, to store the relations as additional information (but not as edges).

Cluster border nodes were mentioned above to handle hierarchical relations to clusters. Now, similar to the artificial dummy nodes inserted at each vertical side of compounds in [Sander 1996b], these dummy nodes are inserted to effectively ensure the cluster validity rules (UML\_SEMANTIC\_CLUSTERS) in the following processing steps. Cluster border nodes were illustrated in Figure 4.8 and Figure 4.18 (c). The additional edges and nodes introduced by this step may slow down the edge crossing reduction on large graphs. Therefore, as an option for lower quality layout calculation, this step can be deferred to the preparation of the coordinates assignment.

Cluster separator nodes are handled similarly except that these nodes are located inside a cluster, i.e., somewhere between two cluster border nodes of a cluster. Separator nodes can be used to spatially partition a cluster for the use as a UML subsystem and, therefore, to realize specific aspects of UML\_CONTAINER, or to implement aspects of UML\_SPATIAL, e.g., UML\_COUPLING.

For a cluster, first an existing cluster border node  $x$  is identified. Then, the cluster border nodes and the edges for the cluster border chains are inserted at both sides of the cluster at appropriate positions into the affected ranks.  $x$  was kept in the bottom rank of its cluster by the algorithms described above and can therefore be reused in that rank on either the left or the right side.

The insertion of cluster separator nodes is delegated to a plug-in, because it relies on application-domain specific information. Depending on the concrete shape of a UML subsystem, a vertical



separation line is represented by cluster separator nodes. The extents of these nodes will be adjusted in the coordinates assignment to reserve space for the interface compartment. Furthermore, cluster separator nodes can be inserted to realize the horizontal partitions, if UML\_COUPLING is enabled.

Even if creating edge chains by calling `createEdgeChains` seems to be a trivial task, this processing step is responsible for assigning the newly created hidden nodes to the appropriate clusters. Some of the predictions made by `createEdgeChains` might be corrected later, but the less this step is able to ensure a proper nesting, the more complex the postprocessing will be. The initial start and end node of the edge to be split are taken into account and while stepping up both cluster names, the greatest common cluster criterion limits the search for the appropriate containing cluster for each individual hidden node.

### 4.5.6 Conclusions

A conclusion is the place where you got tired of thinking.

Harold Fricklestein

In this section, we discussed details on the rank assignment S10, which is responsible for retrieving and ensuring the main structure of the layout result according to UML\_HIERARCHY. Furthermore, as described in Section 4.5.5, cluster structures as well as cluster relations (UML\_SEMANTIC\_CLUSTERS), UML\_CENTER for selected nodes and UML\_SPATIAL, if UML\_COUPLING is enabled, are prepared. Due to the order properties of the node naming function, the result of this layout step is a  $n$ -level hierarchy, which also is required to be cluster-valid if nested elements should be displayed.

Table 4.3 shows the complexities of the individual subalgorithms involved in the rank assignment. Breaking cycles and ensuring a virtual root were discussed along with the preprocessing macro phase in Section 4.4. Inserting additional hidden edges between clusters can be implemented in linear time due to certain precalculations. Correcting edges to realize hierarchical relations to clusters considers all nodes and edges and performs a sorting of the cluster dependences for at most all nodes.

The runtime of the network simplex algorithm, which realizes the core hierarchical rank assignment, was not proved to be linear so far. As mentioned in [Gansner et al. 1988], the network simplex algorithm runs in  $O(I \cdot |V| \cdot |E|)$  time. The value  $I$  represents some number of simplex iterations, which might be exponential in the size of the graph. However, also in our application domain,  $I$  is usually  $O(|E|)$  in practice.

The non-hierarchical rank assignment mainly depends on the local rank assignment of some subgraphs. Here, we will assume that also a network simplex algorithm does this task.

Most postprocessing operations exclusively consider nodes and `createEdgeChains` as well as `adjustCoupledClusterMembers` may change the structure of the entire graph. `createRankStructure` initializes the rank structures. Concrete sequences are required for diagrams, which should display nested elements, because then the resulting  $n$ -level hierarchy must

<b>algorithmic step</b>	<b>runtime complexity</b>
transformToAcyclicGraph	$O( V  +  E_H )$
ensureVirtualRoot	$O( V )$
enforceHierarchicalClusterDependencies	$O( V  \cdot \log  V  +  E_H )$
respectHierarchicalClusterRelations	$O( V  +  E_H )$
network simplex	$O( V  \cdot  E_H ^2)$
normalize	$O( V )$
balance	$O( V )$
non-hierarchical rank assignment	$O( V  \cdot  E_H \cup E_N ^2)$
stretchEdges2VirtualRoot	$O( V )$
adjustClusterBorderNodes	$O( V )$
adjustCoupledClusterMembers	$O( V  +  E_H \cup E_N )$
redistributeDisconnected	$O( V )$
removeEmptyRanks	$O( V )$
createEdgeChains	$O( V  +  E_H \cup E_N )$
createRankStructure	$O( V  \cdot \log  V )$
rank assignment macro phase	$O( V  \cdot \log  V  +  V  \cdot  E_H \cup E_N ^2)$

Table 4.3: Runtime complexities of the rank assignment phase.

be cluster valid. Final sequences for visible elements will be determined in the next layout step, the edge crossing reduction S11.

## 4.6 Edge Crossings

When you build bridges you can keep crossing them.

Rick Pitino

The second step of the classical hierarchical layout algorithm is the reduction of the number of edge crossings. Thereby, the nodes in a layered graph are repositioned to get as close as possible to the theoretical minimum number of edge crossings.

### Definition 14 (bipartite graph)

A graph  $G = (V, E)$  is called bipartite if the set of nodes can be partitioned into two sets  $U$  and  $L$  with  $U \cup L = V$ ,  $U \cap L = \emptyset$  and  $\forall_{e=\{u,l\} \in E} u \in U \Rightarrow l \in L \vee u \in L \Rightarrow l \in U$  holds.

Minimizing the number of crossings of a bipartite graph is called the *bipartite drawing problem* (BDP). It was proved to be NP-complete [Johnson 1982; Garey and Johnson 1983; Eades et al. 1986]. Therefore, various heuristical approaches have been proposed in literature. Most of the algorithms for graphs, which admit a  $k$ -level hierarchy, iteratively apply a crossing reduction algorithm designed for the 2-level crossing problem.

Definition 14 does not permit flat edges with  $(u \in U \wedge l \in U) \vee (u \in L \wedge l \in L)$  while our definition 9 for  $n$ -level hierarchies includes that case. Therefore, algorithms, which should be applied to UML class diagrams, may have to be redesigned. So far only few work was published which took crossing minimization for compound graphs into account [Sander 1996b; Forster 2002]. Furthermore, crossings, which arise from non-hierarchical edges, are seldom considered [Waddle 2001].

After an overview on previous work, some aspects of hierarchical and flat-edge crossing theory will be discussed. Then algorithms to guarantee cluster relations when minimizing the number of edge crossings will be described. After that, adaptations to well-known edge crossing strategies as well as methods specific to our application domain will be given. The question, which concrete algorithm is most appropriate to our application domain, will be left unanswered until measurements on the implementation are presented in Section 5.3.

### 4.6.1 Previous Work

A pessimist is a man who looks both ways before crossing a one way street.

Laurence J. Peter (1919 – 1988)

In [Warfield 1977], matrices were used to represent the interconnections between nodes on adjacent levels. Furthermore, basic crossing formulae were defined and an algorithm on reducing crossings by permutation was given. This work can be seen as the foundation for the research on

edge crossing reduction.

Concrete implementations can be categorized as

- **2-layer algorithms** which assume one layer as fixed while reordering the second layer (one sided crossing minimization problem). According to [Eades et al. 1986], even this reduced problem is NP-hard. The problem of determining the minimum set of edges whose removal allows the graph to be drawn with no crossings is also NP-hard, whether or not the order on one of the layers is fixed [Eades and Whitesides 1994]. Other algorithms work on both layers, e.g. by reordering all vertices in one step in parallel.
- **layer-by-layer sweep** methods, which iteratively apply a 2-layer algorithm on two adjacent ranks. The other layers not processed by the BPD algorithm at a time are kept fixed [Eades and Sugiyama 1990; Battista et al. 1999; Matuszewski et al. 2000].
- ***n*-layer algorithms** natively working on all ranks of a hierarchy. A generic approach to the *n*-level BDP was given in [Sugiyama et al. 1981].

Furthermore various *hybrid approaches*, combining different algorithms into one, seem to be popular. Orthogonal to the treatment of layers, algorithms can be classified according to the handling of context-dependent information as mentioned in [Mäkinen and Sieranta 1994]:

- **Context-free heuristics** assign the same approximate value to each vertex independent from other vertices. The solution is then obtained by ordering the vertices according to these values.
- **Context-sensitive heuristics** determine the exact numbers of edge crossings connected to the relative orders of vertex. Then, the algorithm tries to find a linear order which minimizes the number of edge crossings.

In our application domain a context-sensitive algorithm may also partly respect UML\_CONSTRAINT\_SEQUENCE: In the case that the main ordering criterion admits the same value for multiple vertices in one rank, a secondary criterion might be used to realize UML\_CONSTRAINT\_SEQUENCE. According to the discussion in Section 3.3, also a predefined sequence of nodes may supersede the sequence determined by an edge crossing reduction method.

Furthermore, edge crossing reduction strategies can be distinguished into

- **incremental heuristics**, which successively change the position of individual nodes. Starting with a cluster-valid input, individual node positions can be excluded to guarantee validity.
- **non-incremental heuristics**, which may modify the positions of multiple nodes at a time. If clusters should be considered, complete ranks or the entire hierarchy have to be postprocessed to ensure cluster validity.

In [Carpano 1980] and [Sugiyama et al. 1981], similar algorithms, based on ordering the nodes according to *barycenter* weights, were given. The position of a vertex was calculated as the arithmetic mean of the positions of its adjacent vertices. This follows the basic principle that crossings are likely to be minimized by increasing the number of horizontal arcs. A similar approach was also mentioned as *averaging* or *relative degree method* in [Eades and Kelly 1986]. Different variants of the basic idea have been proposed, as the refinement in [Gansner et al. 1988], left-barycenter (average position of the immediate predecessors in the previous layer) or right-barycenter (average position of the immediate successors in the next layer) as described in [Rowe et al. 1987]. According to [Rowe et al. 1987], sweeping from left-to-right and backwards in succeeding iterations may cause vertices to move forth and back if different barycenter values are calculated. Therefore, in the third sweep, the average of the barycenters may be considered. Instead of the barycenter calculation, the average (median) can be employed similarly. The so-called *median method* was refined in [Eades and Wormald 1994] and both techniques, barycenter and median, have been analyzed for their crossing performance. It was proved for the median method that the number of edge crossings is never more than three times larger than in the optimal drawing. Furthermore, the median method can be implemented more efficiently as mentioned in [Stallmann et al. 2001]. Similar to the barycenter method, different variants have been published, like considering a median value based on a weight function, the average median relying on the arithmetic mean of the positions occupied by the two middle adjacent vertices or different relative degree algorithms considering the degree of the nodes. Another variant is to apply the transpose heuristic as a postprocessing step: It exchanges nodes as long as it finds direct neighbored nodes for which an exchange of the position improves the number of crossings as described in [Gansner et al. 1988; Gansner et al. 1993].

The *semimedian heuristic*, a hybrid combination of median and barycenter method, was discussed in [Mäkinen 1990]. Another hybrid approach is the *dot heuristic* [Gansner et al. 1993; Stallmann et al. 2001] which combines an initial depth first ordering and a fixed-layer median implementation.

Median and barycentric method appear to be the most popular heuristics in concrete implementations. Therefore, we will mention the other approaches only briefly here. The *greedy insertion* [Eades and Kelly 1986] successively removes the node  $u$ , which minimizes the local crossings between edges adjacent to  $u$  with edges adjacent to vertices below  $u$ , and appends  $u$  to the result. In the *greedy switching* method [Eades and Kelly 1986], the *switch* or the *adjacent exchange* [Eades and Sugiyama 1990; Battista et al. 1999], consecutive vertices in one rank are exchanged if the number of crossings can locally be improved. A pivot element is selected in the *splitting approach* [Eades and Kelly 1986], then every other vertex is (recursively) placed above or below the pivot element with respect to the number of crossings. All vertices of one layer are positioned simultaneously in the *assignment method* [Catarci 1995] by considering the BDP as assignment problem, which was originally defined for computing the best assignment of tasks to workers under certain conditions. The *branch and bound* algorithm [Valls et al. 1996] operates on a structured search tree, which allows in some cases to directly obtain the optimal solution associated with a node. In [Mutzel 1997; Jünger and Mutzel 1996; Jünger and Mutzel 1997; Mutzel and Weiskircher 1998] a (minimal) number of edges in the input graph was removed to gain a  $k$ -level planar graph and in the final drawing these edges were reinserted. Unfortunately, the extraction

of a 2-level planar subgraph is NP-hard and even if this approach ensures the optimum solution, there is no guarantee that it can be found in polynomial time. *Tabu search* and the greedy randomized adaptive search procedure (GRASP) approach have been proposed in [Laguna et al. 1997; Laguna and Martí 1999; Martí and Laguna 2003]: A meta-heuristic tries to hold the balance between two phases, one searching a near-optimal solution in one layer, the other selecting layers for intensification and switching of randomly selected vertices to escape from optimality. The *vertex-exchange method* [Healy and Kuusik 2000] is a linear programming method, which operates on the vertex-exchange graph. That graph represents all distinct pairs of same-level vertices of the original graph as nodes, which are connected by edges, if the vertices of the corresponding graph are connected by non-adjacent edges. The *sifting method* [Günther et al. 2001] exclusively exchanges neighbor vertices in alternating directions. Global sifting was considered in [Matuszewski et al. 2000] for the  $k$ -layer straightline crossing minimization. In contradiction to sifting, *adaptive insertion* [Stallmann et al. 2001] does not allow a node to stay in its current position and nodes are considered in their current right-to-left order rather than based on their degrees.

Some of the approaches mentioned above were furthermore combined to hybrid methods: Sifting and barycenter [Günther et al. 2001], adaptive insertion and barycenter [Stallmann et al. 2001] or *Guided breadth-first search* and adaptive insertion in [Stallmann et al. 1999].

As a postprocessing step, windows optimization [Eschbach et al. 2002] can be applied to the other crossing reduction algorithms. Thereby, a series of subsets of nodes with constant size typically spreading over several layers are considered and optimized with respect to their adjacent nodes. A study showed that this additional processing may enhance the quality by more than 10%.

Detailed overviews on reducing edge crossings in graph drawing especially on the BDP can be found in [Di Battista et al. 1994; Catarci 1995; Laguna et al. 1997; Bastert and Matuszewski 2001; Martí and Laguna 2003], runtime complexities of the algorithms were mentioned in [Stallmann et al. 2001].

The question remains, what edge crossing reduction approach should be considered when designing an own layout algorithm. In [Jünger and Mutzel 1997], iterated application of the barycenter method for large graphs was recommended. In [Battista et al. 1999], a hybrid approach using the median method for initial and then adjacent exchange/switch for the final reduction was advised. A combination of the barycenter heuristic followed by a weighted variant of subsequent sifting was advocated in [Brandes and Wagner 2003], because appeared to be fast and good at separating biconnected components. In [Protsko et al. 1991] replication of parts of the graph to achieve clarity, low edge crossings and short edges was suggested. As discussed in Section 3.3.3 this idea does not fit to our application domain.

Unfortunately, the most algorithms deal with solving the BDP. Except of the obvious layer-by-layer sweep method, only few approaches directly attack the multilevel BDP. Furthermore, the methods mentioned so far do neither respect clusters, non-hierarchical edges nor multiple edges between the same nodes. In [Sander 1996b], the results of a barycenter method were post-processed to force the the individual layers to be cluster-valid. An extension to the barycenter approach to consider same level edges was given in [Waddle 2001]: Constraints restrict the

edge crossing minimization and the reversal of edges due to cycles, further constraints can impose a total ordering on the node level assignment, which would be helpful when realizing UML\_CONSTRAINT\_SEQUENCE. In [Forster 2002], an in-order justification according to the compound nodes was suggested. Thereby, each possible edge crossing is associated with a unique compound node. To ensure intra-rank validity, a crossing reduction graph is computed and a conventional algorithm for weighted crossing reduction to the crossing reduction graph is applied. To ensure the inter-rank validity, a constrained 2-layer crossing minimization is needed.

We can conclude that currently no optimal edge crossing reduction algorithm for our application domain exists. Additionally, some of the approaches mentioned above can not be applied according to REQ\_DETERMINISTIC\_ALGORITHM, because they rely on random processing, e.g., selection of nodes to escape local optimal situations. Therefore, we will modify the standard barycenter and median to be forced to be cluster-valid. Algorithms which are useful implementing this task will be presented in Section 4.6.4. Furthermore, an own idea, the hierarchical crossing minimization, will be described.

## 4.6.2 Basic Definitions

A perpetual holiday is a good working definition of hell.

George Bernard Shaw (1856 – 1950)

The definitions in this section are adapted from [Sugiyama et al. 1981]:

### Definition 15 (matrix realization of a n-level hierarchy)

For a  $n$ -level hierarchy  $\bar{G} = (V, E_H, E_N, n, \sigma)$ , the matrix realization of  $\bar{G}$  is defined as follows:

1. A matrix  $M^{(i)} = M(\sigma_i, \sigma_{i+1})$  is a  $|V_i| \times |V_{i+1}|$  matrix whose rows and columns are ordered according to  $\sigma_i$  and  $\sigma_{i+1}$ , respectively<sup>10</sup>.
2. Let  $\sigma_i = u_0 \dots u_k \dots u_{|V_i|-1}$  and  $\sigma_{i+1} = v_0 \dots v_k \dots v_{|V_{i+1}|-1}$ . Then the  $(u_k, v_l)$  element of  $M^{(i)}$ , denoted by  $m_{kl}^{(i)}$ , is given by

$$m_{kl}^{(i)} := \begin{cases} w(e) & : \text{ if } e = \{u_k, v_l\} \in E_{H_i} \\ 0 & : \text{ otherwise} \end{cases} \quad (4.4)$$

where  $w(e) > 0$  is the externally defined weight of the edge.  $M^{(i)}$  is called an interconnection matrix.

3. A matrix realization  $\bar{\mathbf{g}}$  of  $\bar{G}$  is given by the formula

$$\bar{\mathbf{g}}(V, E_H, E_N, n, \sigma) := \bar{\mathbf{g}}(\sigma_0, \dots, \sigma_{n-1}) := M^{(0)}, \dots, M^{(n-2)} \quad (4.5)$$

We define the contents of the interconnect matrix as the weights of the edges instead of binary values

$$m_{kl}^{(i)} := \begin{cases} 1 & : \text{ if } e = \{u_k, v_l\} \in E_{H_i} \\ 0 & : \text{ otherwise} \end{cases}$$

as in [Warfield 1977; Sugiyama et al. 1981], because external weights might be permitted to gain influence on the result of the computation. The implementation may provide a configuration flag to initialize the matrices in the binary version.

**Example:**

Let  $G = (V, E_H, E_N)$  be a graph,  $V = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$ ,  $E_H = \{\{N_1, N_6\}, \{N_2, N_6\}, \{N_3, N_5\}, \{N_3, N_7\}, \{N_4, N_7\}\}$  and  $E_N = \emptyset$ . Let  $V_0 = \{N_1, N_2, N_3, N_4\}$ ,  $V_1 = \{N_5, N_6, N_7\}$ ,  $\sigma_0 = N_1, N_2, N_3, N_4$  and  $\sigma_1 = N_5, N_6, N_7$  be the representation of

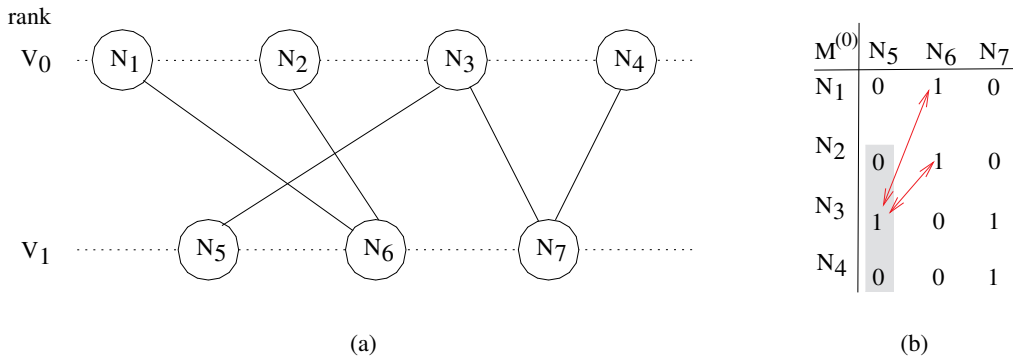


Figure 4.21: (a) a 2-level drawing with 2 crossings, (b) matrix realization of (a). In (b), matrix entries corresponding to edge crossings are marked by arrows.

$\bar{G} = (V, E_H, E_N, 2, \sigma)$ , the 2-level hierarchy of  $G$  according to definition 9. Figure 4.21 (a) shows a 2-level drawing of  $G$  according to the sequences given for  $\sigma_0$  and  $\sigma_1$ . Figure 4.21 (b) is the interconnect matrix  $M^{(0)}$  of  $\sigma_0$  and  $\sigma_1$  assuming that all edges have weight 1. In this case the matrix realization according to definition 15 of  $\bar{G}$  is  $\bar{\mathbf{g}}(V, E_H, E_N, n, \sigma) = M^{(0)}$ . In Figure 4.21 (b) the matrix entries, which correspond to the edge crossings in (a), are marked by arrows.

The next definition formalizes the fact that edge crossings induced by an individual edge  $e$  can be encountered by multiplying the matrix entry of  $e$  to all entries of its lower left sub matrix. In Figure 4.21 (b) the lower left sub matrix of  $e = \{N_1, N_6\}$  at  $m_{0,1}^{(0)}$  is shaded.

From [Warfield 1977; Sugiyama et al. 1981] the following formalization of the number of edge crossings is known:

**Theorem 1 (number of hierarchical crossings in a  $n$ -level hierarchy)**

Let  $\bar{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy and  $\bar{\mathbf{g}}$  be the matrix realization of  $\bar{G}$ , in which edge weights have maximum value 1. Let  $\sigma_i = v_0 \dots v_j \dots v_k \dots v_{|V_i|-1}$  be the  $i$ -th rank in

<sup>10</sup>Matrix indices run on  $0 \dots |V_i| - 1$  or  $0 \dots |V_{i+1}| - 1$ , respectively, according to the implementation.



interconnection matrix  $M^{(i)} = M(\sigma_i, \sigma_{i+1})$  of  $\bar{\mathbf{g}}$ . Furthermore, let the row vector of  $M^{(i)}$  corresponding to a vertex  $v \in V_i$  be denoted by  $\rho(v)$ . Then the number of crossings  $k(\rho(v_j), \rho(v_k))$  produced by the ordered pair of row vectors  $(\rho(v_j), \rho(v_k))$  is given by the formula

$$k(\rho(v_j), \rho(v_k)) := \sum_{\alpha=0}^{|V_{i+1}|-2} \sum_{\beta=\alpha+1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)}. \quad (4.6)$$

Consequently the formula

$$K_h(M^{(i)}) := \sum_{j=0}^{|V_i|-2} \sum_{k=j+1}^{|V_i|-1} k(\rho(v_j), \rho(v_k)) = \sum_{j=0}^{|V_i|-2} \sum_{k=j+1}^{|V_i|-1} \left( \sum_{\alpha=0}^{|V_{i+1}|-2} \sum_{\beta=\alpha+1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)} \right) \quad (4.7)$$

determines the number of crossings of  $M^{(i)}$ , i.e., the number of crossings induced by adjacent ranks  $\sigma_i$  and  $\sigma_{i+1}$ . The total number  $K(\bar{\mathbf{g}})$  of crossings of  $\bar{\mathbf{g}}$  is given by

$$K_h(\bar{\mathbf{g}}) := K_h(M^{(0)}) + \dots + K_h(M^{(n-2)}) \quad (4.8)$$

As defined above, the edge weights may have values larger than 1. In this case the number of edge crossings will differ from theorem 1. Therefore, we will implicitly work on the *number of pseudo edge crossings* and we will also refer to that pseudo number by the term “number of edge crossings”.

### 4.6.3 Crossing Theory

In theory, there is no difference between theory and practice. In practice, there is.

Chuck Reid

Because self-loops are removed from the graph in S6, we do not treat self loops in the following part of this section. First we will look on different properties of hierarchical crossing theory, in particular for slightly improving runtime performance, then issues needed to calculate the number of crossings on non-hierarchical edges will be discussed.

#### Hierarchical Edges

As mentioned above, a naive implementation calculating the number of edge crossings according to theorem 1 would take  $O(|V|^4)$ . Using a second matrix, which stores the sub matrix sums of the second part of (4.7) duplicates the memory usage, but reduces the computational complexity to  $O(|V|^2)$ . The sub matrix sums always run over the lower left sub matrix of the element which is currently regarded, i.e., for  $m_{j\beta}$  the sums run over  $j+1 \dots |V_i|-1$  and  $0 \dots \beta-1$ . Below, the relation between the matrix element  $m_{j\beta}$  and the area of the submatrix sums, displayed as a boxed sum sign, is depicted:

$$\left( \begin{array}{c} \boxed{\Sigma} \quad m_{j\beta} \end{array} \right)$$

Incremental approaches like successive node insertions or node sifting require insertion and update operations, which, in principle, also run in  $O(|V|^2)$ . The effective runtime is significantly faster as we will show in this section.

To our knowledge, even if work like [Sugiyama et al. 1981; Catarci 1995] suggested this idea, it was not explicitly formulated so far. A non-matrix based approach was described in [Valls et al. 1996] but on an entire graph it also takes  $O(|V|^4)$ .

In [Eades and Kelly 1986] a crossing matrix for the BDP was defined as a triangular matrix containing the crossings caused by the sequence of nodes of one rank only (not of two adjacent ranks as in definition 15 and theorem 1). That approach required  $n$  matrices on a  $n$ -level matrix (instead of  $n - 1$  in our approach), but it might save memory using triangular matrices depending on the input hierarchy. It was mentioned that building the matrix lies in  $O(|V|^2 \cdot |E|)$  and calculating updates in  $O(|V|^2)$ .

In [Grohe 2001] it was shown “[...] that for every fixed  $k \geq 0$  there is a quadratic time algorithm that decides whether a given graph has crossing number at most  $k$  and, if this is the case, computes a drawing of the graph in the plane with at most  $k$  crossings.” Unfortunately, there is a hidden constant in the quadratic upper bound which heavily depends on  $k$  so that the running time is  $O(f(k) \cdot n^2)$  where  $f$  is a doubly exponential function which limits the given algorithm to theoretical interests only.

Calculating the entire number of crossings can also be attacked from the geometrical point of view. An obvious algorithm can be given in  $O(|E|^2)$ . Let  $C$  be the set of pairwise crossings, then the algorithm proposed in [Chazelle 1986] runs in  $O(|E| \cdot \log |E| + |C|)$  time and  $O(|E| + |C|)$  space. An interesting fact for UML\_EDGE\_CROSSING\_SYMBOL is that reporting all  $k$  intersecting pairs among  $n$  arbitrary segments lies in  $O(|E| \cdot (\log^2 |E| / \log \log |E|) + |C|)$  time using  $O(|E| + |C|)$  space [Chazelle 1986]. A faster but more complicated algorithm calculating the number of crossings in  $O(|E|^{1.695})$  and  $O(|E|)$  space was given in [Chazelle and Edelsbrunner 1992]. Sander suggested 1996 an algorithm in  $O(|E| + |V|)$  time and  $O(|E|)$  space. In [Waddle and Malhotra 2000] the problem was solved in  $O(|E| \cdot \log |U \cup L|)$  time (see definition 14) and  $O(|E|)$  space and in [Barth et al. 2002] an improved algorithm in  $O(|E| \cdot \log(\min\{|U|, |L|\}))$  time was given.

Despite the theoretical improvements on bipartite crossing numbers, we still rely on simple improvements of the matrix based crossing number calculation. Most of the theoretically more sophisticated ideas do not discuss the problem of incrementally inserting, deleting or moving individual nodes which is more interesting to our approach. Similar experience with crossing matrices and fast results for incremental changes to the vertex sequence were also mentioned in [Eades and Kelly 1986] and [Günther et al. 2001].

In fact, in the implementation, the crossing number calculation is realized as a plug-in to the edge crossing reduction algorithm so that probably better approaches can easily be integrated.

### Lemma 1

Let  $M_{(n,o)}$  be a  $n \times o$  matrix with 0-based indices. Then

$$\sum_{d=1}^{o-1} \sum_{e=0}^{d-1} m_{jd} m_{ke} = \sum_{e=0}^{o-2} \sum_{d=e+1}^{o-1} m_{jd} m_{ke} \quad (4.9)$$

with  $0 \leq j \leq n$  and  $0 \leq k \leq n$ .

**Proof:**

Complete induction over  $o$  with fixed  $j$  and  $k$ :

- $o = 1$ :  $\sum_{d=1}^0 \sum_{e=0}^{d-1} m_{jd}m_{ke} = 0 = \sum_{e=0}^{-1} \sum_{d=e+1}^0 m_{jd}m_{ke}$  via the definition of the empty sum.
- $o = 2$ :  $\sum_{d=1}^1 \sum_{e=0}^{d-1} m_{jd}m_{ke} = m_{j1} \cdot m_{k0} = \sum_{e=0}^0 \sum_{d=e+1}^1 m_{jd}m_{ke}$
- $o = 3$ :  $\sum_{d=1}^2 \sum_{e=0}^{d-1} m_{jd}m_{ke} = m_{j1} \cdot m_{k0} + m_{j2} \cdot m_{k0} + m_{j2} \cdot m_{k1} = \sum_{e=0}^1 \sum_{d=e+1}^2 m_{jd}m_{ke}$
- $o \rightarrow o+1$ :  $\sum_{d=1}^o \sum_{e=0}^{d-1} m_{jd}m_{ke} = \sum_{d=1}^{o-1} \sum_{e=0}^{d-1} m_{jd}m_{ke} + \sum_{e=0}^{o-1} m_{jo}m_{ke} \stackrel{1A}{=} \sum_{e=0}^{o-2} \sum_{d=e+1}^{o-1} m_{jd}m_{ke} + \sum_{e=0}^{o-1} m_{jo}m_{ke} = \sum_{e=0}^{o-2} \sum_{d=e+1}^{o-1} m_{jd}m_{ke} + \sum_{e=0}^{o-2} m_{jo}m_{ke} + m_{jo}m_{k,o-1} = \sum_{e=0}^{o-2} \sum_{d=e+1}^o m_{jd}m_{ke} + \sum_{d=(o-1)+1}^o m_{jd}m_{k,o-1} = \sum_{e=0}^{o-1} \sum_{d=e+1}^o m_{jd}m_{ke} \quad \square$

**Corollary 2 (alternative edge crossing formula)**

(4.6) and (4.11) can alternatively be written as

$$k(\rho(v_j), \rho(v_k)) = \sum_{\beta=1}^{|V_{i+1}|-1} \sum_{\alpha=0}^{\beta-1} m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)}. \quad (4.10)$$

$$K_h(M^{(i)}) = \sum_{j=0}^{|V_i|-2} \sum_{\beta=1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot \left( \sum_{k=j+1}^{|V_i|-1} \sum_{\alpha=0}^{\beta-1} m_{k\alpha}^{(i)} \right) \quad (4.11)$$

**Proof:**

Formula 4.10 can simply be obtained from (4.6) by applying the result of lemma 1.

$$\begin{aligned} K_h(M^{(i)}) &\stackrel{(4.7)}{=} \sum_{j=0}^{|V_i|-2} \sum_{k=j+1}^{|V_i|-1} \left( \sum_{\alpha=0}^{|V_{i+1}|-2} \sum_{\beta=\alpha+1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)} \right) \stackrel{\text{lemma 1}}{=} \sum_{j=0}^{|V_i|-2} \sum_{k=j+1}^{|V_i|-1} \left( \sum_{\beta=1}^{|V_{i+1}|-1} \sum_{\alpha=0}^{\beta-1} m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)} \right) = \\ &\sum_{j=0}^{|V_i|-2} \sum_{\beta=1}^{|V_{i+1}|-1} \sum_{k=j+1}^{|V_i|-1} \sum_{\alpha=0}^{\beta-1} m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)} = \sum_{j=0}^{|V_i|-2} \sum_{\beta=1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot \left( \sum_{k=j+1}^{|V_i|-1} \sum_{\alpha=0}^{\beta-1} m_{k\alpha}^{(i)} \right) \end{aligned} \quad \square$$

**Definition 16 (condensed interconnection matrix)**

Let  $M^{\Sigma(i)}$ , the condensed interconnection matrix, be a  $|V_i| \times |V_{i+1}|$  matrix induced by  $M^{(i)}$  as follows:

$$m_{kl}^{\Sigma(i)} := \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)}$$

**Corollary 3 (type of  $M^{\Sigma(i)}$ )**

$M^{\Sigma(i)}$  is always of type

$$\left( \begin{array}{c|ccc} 0 & & * & \\ \vdots & & & \\ \hline 0 & \dots & & 0 \end{array} \right)$$

**Proof:**

$$m_{k0}^{\Sigma(i)} = \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{-1} m_{\beta\alpha}^{(i)} = 0 \text{ and } m_{|V_i|-1,l}^{\Sigma(i)} = \sum_{\beta=|V_i|}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} = 0$$

via the definition of the empty sum. □

**Corollary 4 (recursive calculation of  $M^{\Sigma(i)}$ )**

$M^{\Sigma(i)}$  can be calculated recursively by

$$m_{kl}^{\Sigma(i)} = \begin{cases} 0 & : \text{ if } k = |V_i| - 1 \\ 0 & : \text{ if } l = 0 \\ m_{k+1,l}^{\Sigma(i)} + \sum_{\alpha=1}^{l-1} m_{k+1,\alpha}^{(i)} & : \text{ otherwise} \end{cases}$$

**Proof:**

$m_{kl}^{\Sigma(i)} = 0$  if  $k = |V_i| - 1 \vee l = 0$  directly follows from corollary 3. For all other matrix entries:

$$m_{kl}^{\Sigma(i)} \stackrel{\text{def. 16}}{=} \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} = \sum_{\beta=k+2}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} + \sum_{\alpha=0}^{l-1} m_{k+1,\alpha}^{(i)} \stackrel{\text{def. 16}}{=} m_{k+1,l}^{\Sigma(i)} + \sum_{\alpha=0}^{l-1} m_{k+1,\alpha}^{(i)} \stackrel{\text{cor. 3}}{=} m_{k+1,l}^{\Sigma(i)} + \sum_{\alpha=1}^{l-1} m_{k+1,\alpha}^{(i)} \quad \square$$

**Corollary 5 (crossing calculation with  $M^{\Sigma(i)}$ )**

$$K_h(M^{(i)}) = M^{(i)} \odot M^{\Sigma(i)} \quad (4.12)$$

where  $M^{(i)} \odot M^{\Sigma(i)} = \sum_{j=0}^{|V_i|-1} \sum_{\beta=0}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot m_{j\beta}^{\Sigma(i)}$ .

**Proof:**

$$K_h(M^{(i)}) \stackrel{(4.11)}{=} \sum_{j=0}^{|V_i|-2} \sum_{\beta=1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot \left( \sum_{k=j+1}^{|V_i|-1} \sum_{\alpha=0}^{\beta-1} m_{k\alpha}^{(i)} \right) \stackrel{\text{def. 16}}{=} \sum_{j=0}^{|V_i|-2} \sum_{\beta=1}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot m_{j\beta}^{\Sigma^{(i)}} \stackrel{\text{cor. 3}}{=} \sum_{j=0}^{|V_i|-1} \sum_{\beta=0}^{|V_{i+1}|-1} m_{j\beta}^{(i)} \cdot m_{j\beta}^{\Sigma^{(i)}} \stackrel{\text{cor. 5}}{=} M^{(i)} \odot M^{\Sigma^{(i)}}$$

□

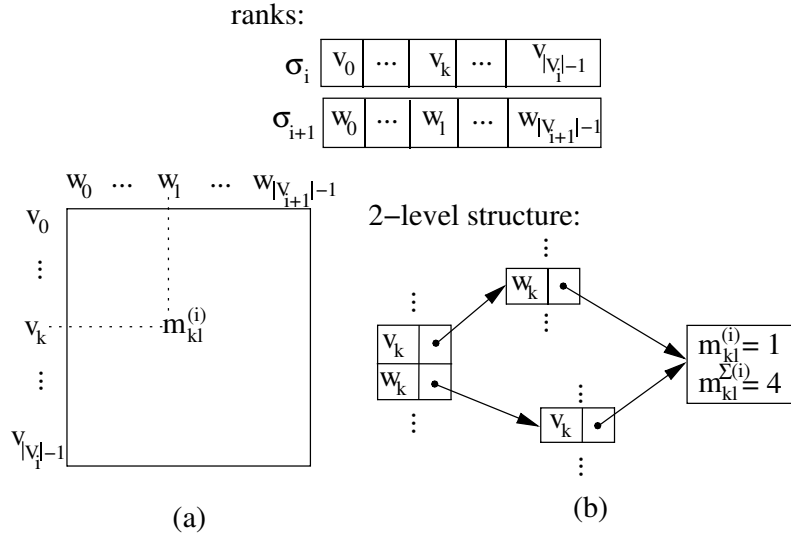


Figure 4.22: Realization of crossing matrices: (a) traditional integer array implementation, (b) 2-level hash map implementation.

Even if it is now obvious that calculation of the number of crossings can be done in  $O(|V|^2)$ , some algorithms like node sifting or incrementally building up the hierarchy run slightly faster, when the underlying data structures are changed incrementally. Traditionally, an integer array implementation might be selected for realizing the crossing implementation. It requires copying and moving row and columns as well as for incremental ordering strategies physical arrays, which are larger than the matrix itself. Alternatively, a 2-level hash table implementation as shown in Figure 4.22 can be considered. It supports sparse matrices and  $O(1)$  row or column exchange operations of adjacent rows and columns. Due to administrative overhead of the hashtables itself we use a combined data structure as shown in Figure 4.22 to store the entries of the interconnect matrix and the induced condensed interconnect matrix. Entries with value 0 in both matrices are not stored at all. Somehow, the sequences represented by the rank structure shown in Figure 4.22 have to be present in both variants as well as for following steps of the layout algorithm. Properties for calculating the contents of the condensed matrices incrementally will be discussed in the following.

**Corollary 6 (edge crossings after appending a node)**

Let us assume that node  $v$ , assigned to rank  $i$ , is inserted and  $v$  is appended at the end of  $\sigma_i$ . This causes appending a column to  $M^{(i-1)}$  and a row to  $M^{(i)}$ .

**Proof:**

$M^{(i-1)}$  and  $M^{(i)}$  are affected according to definition 15.

The update of  $M^{\Sigma(i-1)}$  can be done similarly to the calculation of a condensed matrix in corollary 4 by respecting the new row exclusively:  $m_{k,|V_i|-1}^{\Sigma(i)} = m_{k+1,|V_i|-1}^{\Sigma(i)} + \sum_{\alpha=1}^{|V_i|-2} m_{k+1,\alpha}^{(i)}$   $0 \leq k < |V_i|$ .

After adding the row to  $M^{(i)}$ , all entries in  $M^{\Sigma(i)}$  are affected according to  $m_{kl}^{\Sigma(i)} = m_{k+1,l}^{\Sigma(i)} + \sum_{\alpha=1}^{l-1} m_{k+1,\alpha}^{(i)}$   $0 \leq k < |V_i|$  from corollary 4. Propagating this change to the other matrix entries or recalculating the entire condensed matrix can be done as shown in corollary 4.  $\square$

Removing a node  $v$  assigned to rank  $i$  from  $\sigma_i$  or  $V_i$ , respectively, affects the same matrices as described in corollary 6. In general both matrices have to be recalculated in  $O(|V|^2)$  according to corollary 4, because at least in one matrix other entries depend on the entries to be removed.

**Lemma 2 (hierarchical crossings after an exchange with the preceding neighbor)**

Let  $v = \sigma_i[p]$  be the node at index  $1 \leq p < |\sigma_i|$  to be exchanged with its immediate left neighbor at index  $p-1$  in  $\sigma_i$ . Let  $M^{*(i-1)}$  and  $M^{*(i)}$  be the interconnect matrices,  $M^{*\Sigma(i-1)}$  and  $M^{*\Sigma(i)}$  the condensed interconnect matrices after the exchange. The following properties hold:

1. Column  $p-1$  and  $p$  are the only columns in  $M^{*(i-1)}$ , rows  $p-1$  and  $p$  the only rows in  $M^{*(i)}$  affected by the exchange.
2. Column  $p$  is the only column in  $M^{*\Sigma(i-1)}$ , row  $p-1$  is the only row in  $M^{*\Sigma(i)}$  affected by the exchange and can be calculated by:

$$m_{kp}^{*\Sigma(i-1)} = m_{k,p-1}^{\Sigma(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)}$$

$$m_{p-1,l}^{*\Sigma(i)} = \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} + m_{pl}^{\Sigma(i)}$$

3. The difference in edge crossings before the exchange and without calculating  $M^{*(i-1)}$ ,  $M^{*(i)}$ ,  $M^{*\Sigma(i-1)}$  or  $M^{*\Sigma(i)}$  can be written as

$$\delta_l^{(i-1)} = \sum_{k=0}^{|V_{i-1}|-1} \left( m_{k,p-1}^{(i-1)} \cdot \left( m_{kp}^{\Sigma(i-1)} - m_{k,p-1}^{\Sigma(i-1)} \right) + m_{kp}^{(i-1)} \cdot \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)} \right) \quad (4.13)$$

$$\delta_l^{(i)} = \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p-1,l}^{(i)} \cdot \left( m_{pl}^{\Sigma(i)} - m_{p-1,l}^{\Sigma(i)} \right) + m_{pl}^{(i)} \cdot \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} \right) \quad (4.14)$$

With

$$\delta^{*(i)}(k, l) = m_{kl}^{*(i)} \cdot m_{kl}^{*\Sigma(i)} - m_{kl}^{(i)} \cdot m_{kl}^{\Sigma(i)} \quad (4.15)$$

defined on all matrices the difference in edge crossings after the exchange can be expressed by

$$\delta_l^{*(i-1)} = \sum_{k=0}^{|V_{i-1}|-1} \left( \delta^{*(i-1)}(k, p-1) + \delta^{*(i-1)}(k, p) \right) \quad (4.16)$$

$$\delta_l^{*(i)} = \sum_{l=0}^{|V_{i+1}|-1} \left( \delta^{*(i)}(p-1, l) + \delta^{*(i)}(p, l) \right) \quad (4.17)$$

with  $\delta_l^{*(i-1)} = \delta_l^{(i-1)}$  and  $\delta_l^{*(i)} = \delta_l^{(i)}$ .

4. The number of crossings after the exchange is now

$$K_h(\bar{\mathbf{g}}^*) = K_h(\bar{\mathbf{g}}) + \delta_l^{*(i-1)} + \delta_l^{*(i)} = K_h(\bar{\mathbf{g}}) + \delta_l^{(i-1)} + \delta_l^{(i)} \quad (4.18)$$

**Proof:**

1. The exchange operation and the resulting interconnect matrices can be described by

$$m_{kl}^{*(i-1)} = \begin{cases} m_{k,p-1}^{(i-1)} & : \text{ if } l = p \\ m_{kp}^{(i-1)} & : \text{ if } l = p-1 \\ m_{kl}^{(i-1)} & : \text{ otherwise} \end{cases} \quad (4.19)$$

and

$$m_{kl}^{*(i)} = \begin{cases} m_{p-1,l}^{(i)} & : \text{ if } k = p \\ m_{pl}^{(i)} & : \text{ if } k = p-1 \\ m_{kl}^{(i)} & : \text{ otherwise} \end{cases} \quad (4.20)$$

Therefore  $p-1$  and  $p$  are the only columns in  $M^{*(i-1)}$ , rows  $p-1$  and  $p$  the only rows in  $M^{*(i)}$  that are affected by the exchange.

2. on  $M^{*\Sigma(i-1)}$ :

- column  $l < p$ :  $m_{kl}^{*\Sigma(i-1)} \stackrel{\text{def. 16}}{=} \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i-1)} \stackrel{(4.19)}{=} \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i-1)} \stackrel{\text{def. 16}}{=} m_{kl}^{\Sigma(i-1)}$
- column  $l = p$ :  $m_{kp}^{*\Sigma(i-1)} \stackrel{\text{def. 16}}{=} \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{p-1} m_{\beta\alpha}^{*(i-1)} = \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{p-2} m_{\beta\alpha}^{*(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta,p-1}^{*(i-1)} \stackrel{(4.19)}{=} \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{p-2} m_{\beta\alpha}^{(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)} \stackrel{\text{def. 16}}{=} m_{k,p-1}^{\Sigma(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)}$

- column  $l > p$ :  $m_{kl}^{*\Sigma(i-1)} \stackrel{\text{def. 16}}{=} \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i-1)} = \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{p-2} m_{\beta\alpha}^{*(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta,p-1}^{*(i-1)} +$   
 $\sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{*(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=p+1}^{l-1} m_{\beta\alpha}^{*(i-1)} \stackrel{(4.19)}{=} \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{p-2} m_{\beta\alpha}^{*(i-1)} +$   
 $\sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{*(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta,p-1}^{*(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=p+1}^{l-1} m_{\beta\alpha}^{*(i-1)} =$   
 $\sum_{\beta=k+1}^{|V_{i-1}|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i-1)} \stackrel{\text{def. 16}}{=} m_{kl}^{\Sigma(i-1)}$

on  $M^{*\Sigma(i)}$ :

- row  $k \geq p$ :  $m_{kl}^{*\Sigma(i)} \stackrel{\text{def. 16}}{=} \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i)} \stackrel{(4.20)}{=} \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} \stackrel{\text{def. 16}}{=} m_{kl}^{\Sigma(i)}$
- row  $k = p-1$ :  $m_{p-1,l}^{*\Sigma(i)} \stackrel{\text{def. 16}}{=} \sum_{\beta=p}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i)} = \sum_{\alpha=0}^{l-1} m_{p\alpha}^{*(i)} + \sum_{\beta=p+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i)} \stackrel{(4.20)}{=}$   
 $\sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} + \sum_{\beta=p+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} \stackrel{\text{def. 16}}{=} \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} + m_{pl}^{\Sigma(i)}$
- row  $k < p-1$ :  $m_{kl}^{*\Sigma(i)} \stackrel{\text{def. 16}}{=} \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i)} = \sum_{\beta=k+1}^{p-2} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i)} + \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{*(i)} + \sum_{\alpha=0}^{l-1} m_{p,\alpha}^{*(i)} +$   
 $\sum_{\beta=p+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{*(i)} \stackrel{(4.20)}{=} \sum_{\beta=k+1}^{p-2} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} + \sum_{\alpha=0}^{l-1} m_{p,\alpha}^{(i)} + \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} +$   
 $\sum_{\beta=p+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} = \sum_{\beta=k+1}^{|V_i|-1} \sum_{\alpha=0}^{l-1} m_{\beta\alpha}^{(i)} \stackrel{\text{def. 16}}{=} m_{kl}^{\Sigma(i)}$

3. In both interconnect matrices only columns  $p-1$  and  $p$  or rows  $p-1$  and  $p$ , respectively, and in both condensed interconnect matrix only column  $p$  or  $p-1$ , respectively, are affected by exchanging adjacent nodes. Therefore (4.16) as well as (4.17) contain the sums over the differences between before and after exchange (in (4.15)).

$$\delta_l^{*(i-1)} \stackrel{(4.16)}{=} \sum_{k=0}^{|V_{i-1}|-1} \left( \delta_l^{*(i-1)}(k, p-1) + \delta_l^{*(i-1)}(k, p) \right) \stackrel{(4.15)}{=}$$

$$\sum_{k=0}^{|V_{i-1}|-1} \left( m_{k,p-1}^{*(i-1)} \cdot m_{k,p-1}^{*\Sigma(i-1)} - m_{k,p-1}^{(i-1)} \cdot m_{k,p-1}^{\Sigma(i-1)} + m_{kp}^{*(i-1)} \cdot m_{kp}^{*\Sigma(i-1)} - m_{kp}^{(i-1)} \cdot m_{kp}^{\Sigma(i-1)} \right) \stackrel{(4.19)}{=}$$

$$\sum_{k=0}^{|V_{i-1}|-1} \left( m_{kp}^{(i-1)} \cdot m_{k,p-1}^{*\Sigma(i-1)} - m_{k,p-1}^{(i-1)} \cdot m_{k,p-1}^{\Sigma(i-1)} + m_{k,p-1}^{(i-1)} \cdot m_{kp}^{*\Sigma(i-1)} - m_{kp}^{(i-1)} \cdot m_{kp}^{\Sigma(i-1)} \right) \stackrel{\text{lemma 2.2}}{=}$$

$$\sum_{k=0}^{|V_{i-1}|-1} \left( m_{kp}^{(i-1)} \cdot m_{k,p-1}^{*\Sigma(i-1)} - m_{k,p-1}^{(i-1)} \cdot m_{k,p-1}^{\Sigma(i-1)} + (m_{k,p-1}^{(i-1)} - m_{kp}^{(i-1)}) \cdot m_{kp}^{\Sigma(i-1)} \right) \stackrel{\text{lemma 2.2}}{=}$$

$$\sum_{k=0}^{|V_{i-1}|-1} \left( m_{kp}^{(i-1)} \cdot (m_{k,p-1}^{\Sigma(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)}) - m_{k,p-1}^{(i-1)} \cdot m_{k,p-1}^{\Sigma(i-1)} + (m_{k,p-1}^{(i-1)} - m_{kp}^{(i-1)}) \cdot m_{kp}^{\Sigma(i-1)} \right) =$$



$$\begin{aligned}
& \sum_{k=0}^{|V_{i-1}|-1} \left( m_{kp}^{(i-1)} \cdot \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)} - m_{k,p-1}^{(i-1)} \cdot m_{k,p-1}^{\Sigma(i-1)} + m_{k,p-1}^{(i-1)} \cdot m_{kp}^{\Sigma(i-1)} \right) = \\
& \sum_{k=0}^{|V_{i-1}|-1} \left( m_{k,p-1}^{(i-1)} \cdot (m_{kp}^{\Sigma(i-1)} - m_{k,p-1}^{\Sigma(i-1)}) + m_{kp}^{(i-1)} \cdot \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta p}^{(i-1)} \right) \\
& \delta_l^{*(i)} \stackrel{(4.17)}{=} \sum_{l=0}^{|V_{i+1}|-1} \left( \delta_l^{*(i)}(p-1, l) + \delta_l^{*(i)}(p, l) \right) \stackrel{(4.15)}{=} \\
& \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p-1,l}^{*(i)} \cdot m_{p-1,l}^{*\Sigma(i)} - m_{p-1,l}^{(i)} \cdot m_{p-1,l}^{\Sigma(i)} + m_{pl}^{*(i)} \cdot m_{pl}^{*\Sigma(i)} - m_{pl}^{(i)} \cdot m_{pl}^{\Sigma(i)} \right) \stackrel{(4.20)}{=} \\
& \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p,l}^{(i)} \cdot m_{p-1,l}^{*\Sigma(i)} - m_{p-1,l}^{(i)} \cdot m_{p-1,l}^{\Sigma(i)} + m_{p-1,l}^{(i)} \cdot m_{pl}^{*\Sigma(i)} - m_{pl}^{(i)} \cdot m_{pl}^{\Sigma(i)} \right) \stackrel{\text{lemma 2 2}}{=} \\
& \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p,l}^{(i)} \cdot m_{p-1,l}^{*\Sigma(i)} - m_{p-1,l}^{(i)} \cdot m_{p-1,l}^{\Sigma(i)} + (m_{p-1,l}^{(i)} - m_{pl}^{(i)}) \cdot m_{pl}^{\Sigma(i)} \right) \stackrel{\text{lemma 2 2}}{=} \\
& \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p,l}^{(i)} \cdot \left( \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} + m_{pl}^{\Sigma(i)} \right) - m_{p-1,l}^{(i)} \cdot m_{p-1,l}^{\Sigma(i)} + (m_{p-1,l}^{(i)} - m_{pl}^{(i)}) \cdot m_{pl}^{\Sigma(i)} \right) = \\
& \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p,l}^{(i)} \cdot \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} - m_{p-1,l}^{(i)} \cdot m_{p-1,l}^{\Sigma(i)} + m_{p-1,l}^{(i)} \cdot m_{pl}^{\Sigma(i)} \right) = \\
& \sum_{l=0}^{|V_{i+1}|-1} \left( m_{p-1,l}^{(i)} \cdot (m_{pl}^{\Sigma(i)} - m_{p-1,l}^{\Sigma(i)}) + m_{p,l}^{(i)} \cdot \sum_{\alpha=0}^{l-1} m_{p-1,\alpha}^{(i)} \right)
\end{aligned}$$

4. With lemma 2 2), (4.18) follows immediately.  $\square$

**Lemma 3 (hierarchical crossings after an exchange with the succeeding neighbor)**

Let  $v = \sigma_i[p]$  be the node at index  $p = \sigma_i[v]$  with  $0 \leq p < |\sigma_i| - 1$  to be exchanged with its immediate right neighbor at index  $p + 1$  in  $\sigma_i$ . Let  $M^{*(i-1)}$  and  $M^{*(i)}$  be the interconnect matrices,  $M^{*\Sigma(i-1)}$  and  $M^{*\Sigma(i)}$  the condensed interconnect matrices after the exchange. The following properties hold:

1. Column  $p$  and  $p + 1$  are the only columns in  $M^{*(i-1)}$ , rows  $p$  and  $p + 1$  the only rows in  $M^{*(i)}$  affected by the exchange.
2. Column  $p + 1$  is the only column in  $M^{*\Sigma(i-1)}$ , row  $p$  is the only row in  $M^{*\Sigma(i)}$  affected by the exchange and can be calculated by:

$$m_{k,p+1}^{*\Sigma(i-1)} = m_{kp}^{\Sigma(i-1)} + \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta,p+1}^{(i-1)}$$

$$m_{pl}^{*\Sigma(i)} = \sum_{\alpha=0}^{l-1} m_{p\alpha}^{(i)} + m_{p+1,l}^{\Sigma(i)}$$

3. The difference in edge crossings before the exchange and without calculating  $M^{*(i-1)}$ ,  $M^{*(i)}$ ,  $M^{*\Sigma(i-1)}$  or  $M^{*\Sigma(i)}$  can be written as

$$\delta_r^{(i-1)} = \sum_{k=0}^{|V_{i-1}|-1} \left( m_{k,p+1}^{(i-1)} \cdot \left( m_{kp}^{\Sigma(i-1)} - m_{k,p+1}^{\Sigma(i-1)} \right) + m_{kp}^{(i-1)} \cdot \sum_{\beta=k+1}^{|V_{i-1}|-1} m_{\beta,p+1}^{(i-1)} \right)$$

$$\delta_r^{(i)} = \sum_{l=0}^{|V_{i+1}|-1} \left( m_{pl}^{(i)} \cdot \left( m_{p+1,l}^{\Sigma(i)} - m_{pl}^{\Sigma(i)} \right) + m_{p+1,l}^{(i)} \cdot \sum_{\alpha=0}^{l-1} m_{p\alpha}^{(i)} \right)$$

The difference in edge crossings after the exchange can be expressed as

$$\delta_r^{*(i-1)} = \sum_{k=0}^{|V_{i-1}|-1} \left( \delta^{*(i-1)}(k,p) + \delta^{*(i-1)}(k,p+1) \right)$$

$$\delta_r^{*(i)} = \sum_{l=0}^{|V_{i+1}|-1} \left( \delta^{*(i)}(p,l) + \delta^{*(i)}(p+1,l) \right)$$

with  $\delta_r^{*(i-1)} = \delta_r^{(i-1)}$  and  $\delta_r^{*(i)} = \delta_r^{(i)}$ .

4. The number of crossings after the exchange

$$K_h(\bar{\mathbf{g}}^*) = K_h(\bar{\mathbf{g}}) + \delta_r^{*(i-1)} + \delta_r^{*(i)} = K_h(\bar{\mathbf{g}}) + \delta_r^{(i-1)} + \delta_r^{(i)}$$

**Proof:**

The proof can be conducted similarly to the proof of lemma 2 except for modified exchange operations in part 1 and  $p+1$  or  $p$ , respectively, as splitting points in the distinction of cases in part 2.  $\square$

Hence, the number of edge crossings in bipartite graphs (hierarchical edges in our problem) can be calculated in  $O(|V|^2)$  using crossing matrices. Additional speed improvements can be gained by alternative implementations or by considering incremental changes to the graph, the hierarchy and the interconnect matrices.

In principle some memory can be saved if the type of the matrix according to corollary 3 is respected:  $M^{\Sigma(i)}$  can be stored as a  $(|V_i|-1) \times (|V_{i+1}|-1)$  matrix omitting the first column and the last row and adding appropriate conditions to the algorithms<sup>11</sup>.

### Non-Hierarchical Edges

So far non-hierarchical edges are not respected at all when calculating the number of crossings in a graph. Two general types of non-hierarchical edges have to be respected:

<sup>11</sup>Because of historical reasons the implementation uses transposed matrices. Starting with an alternative version of definition 15 which results in transposed matrices, all the results can be shown similarly.

**Definition 17 (non-hierarchical edges)**

Let  $\bar{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy according to definition 9. Then  $V_i$  is the partition of nodes representing rank  $i$  and  $\sigma_i$  the order of  $V_i$ . An edge  $e = \{u, v\} \in E_N, u \in V_i, v \in V_j$  and therefore  $r(u) = i$  and  $r(v) = j$  is called a

1. **multi-level non-hierarchical edge** if  $i \neq j$ . Regarding  $e$  from the viewpoint  $u$ ,  $e$  is a top multi-level non-hierarchical edge if  $j < i$ . Otherwise, if  $i < j$ ,  $e$  is a bottom multi-level non-hierarchical edge.<sup>12</sup>
2. **flat edge** if  $i = j$ . If  $e$  crosses multiple nodes,  $e$  (regarding  $e$  from the viewpoint  $u$ ) might be supposed to be drawn towards  $V_k, k < i$  (**top flat edge**) or  $V_k, k > i$  (**bottom flat edge**), respectively, to gain the minimum number of crossings. A **flat flat edge** connects neighbored nodes.

**Example:**

Edge  $e$  in Figure 4.23 is a top multi-level non-hierarchical edge seen from node  $w$  but a bottom

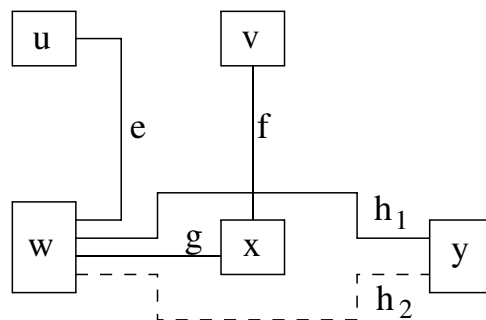


Figure 4.23: Different types of non-hierarchical edges and a hierarchical edge  $f$ .

multi-level non-hierarchical edge seen from node  $u$ . Edge  $g$  is a flat flat edge, because it connects direct neighbors within the same rank. Edge  $h_1$  is a top flat edge while edge  $h_2$  is a bottom flat edge. While edge crossing reduction at least pseudo coordinates arising from the rank assignment and the sequence of nodes within the individual ranks are present. Therefore Figure 4.23) is a drawing induced by the decisions of the edge crossing minimization. Edge  $h_1$  and edge  $h_2$  are alternative ways of routing edge  $d = \{w, y\}$ , which is not shown in Figure 4.23. Because  $h_2$  minimizes the number of edge crossings, edge  $h$  should be drawn as a bottom flat edge instead of a top flat edge. Edge  $g$  and edge  $h$ , drawn as one of the two possibilities  $h_1$  and  $h_2$ , are flat edges. As a convention introduced in [Seemann 1997; Noguchi and Tanaka 1998; Noguchi and Tanaka 1999] and manifested in UML\_HIERARCHY, flat edges are connected to the vertical sides of the nodes, hierarchical edges like  $f$  to the horizontal sides.

<sup>12</sup>Top/bottom arise from the direction to the (virtual) root of the graph in rank 0.

When calculating the number of edge crossings, a multi-level non-hierarchical edge can be treated as a hierarchical edge. One exception has to be considered for non-hierarchical edges connecting adjacent ranks: These edges have to be split temporarily by a dummy node located in the upper rank because otherwise crossings between hierarchical and such non-hierarchical edges cannot be detected. Therefore edge  $e$  in Figure 4.23 can be treated similar to edge  $f$  after splitting it by a dummy node located at the left or right side of  $u$ . This is also consistent with definition 9 and definition 15.

Flat edges are not covered by definition 15 because the interconnect matrix takes edges between different levels into account only.

To simplify the notation, in the remaining part of this section we will assume that  $\bar{G} = (V, E_H, E_N, \sigma)$  is a  $n$ -level hierarchy and due to inserting, moving or removing a node  $\bar{G}^* = (V^*, E_H^*, E_N^*, \sigma^*)$  is the modified  $n$ -level hierarchy.

**Definition 18 (upper and lower multi-level edges)**

Let  $mle^\downarrow(u) := \{e : e = \{u, v\} \in E_H \cup E_N, r(u) < r(v)\}$  and  $mle^\uparrow(u) := \{e : e = \{u, v\} \in E_H \cup E_N, r(u) > r(v)\}$ . For a node  $u$

$$K_f^\uparrow(\bar{G}, u) := \begin{cases} 0 & : \text{ if } r(u) \leq 0 \\ |mle^\uparrow(u)| & : \text{ otherwise} \end{cases}$$

$$K_f^\downarrow(\bar{G}, u) := \begin{cases} 0 & : \text{ if } r(u) \geq n - 1 \\ |mle^\downarrow(u)| & : \text{ otherwise} \end{cases}$$

denote the number of upper (lower) multi-level edges..

A dummy root would need a more detailed condition in  $K_f^\uparrow(\bar{G}, u)$  to consider the first rank, which contains visible nodes.

**Corollary 7 (upper and lower multi-level edges crossed by a flat edge)**

For a flat edge  $e = \{u, v\}$  with  $r(u) = r(v)$  let

$$s_f(u, v) := \min\{\sigma_{r(u)}[u], \sigma_{r(v)}[v]\} + 1$$

and

$$e_f(u, v) := \max\{\sigma_{r(u)}[u], \sigma_{r(v)}[v]\} - 1$$

be the exclusive start (end) 0-based index position of  $e$  in rank  $r(u)$ . As a conclusion from definition 18

$$K_f^\uparrow(\bar{G}, e) := K_f^\uparrow(\bar{G}, u, v) := \sum_{\alpha=s_f(u,v)}^{e_f(u,v)} K_f^\uparrow(\bar{G}, \sigma_{r(u)}[\alpha]) \quad (4.21)$$

$$K_f^\downarrow(\bar{G}, e) := K_f^\downarrow(\bar{G}, u, v) := \sum_{\alpha=s_f(u,v)}^{e_f(u,v)} K_f^\downarrow(\bar{G}, \sigma_{r(u)}[\alpha])$$

denote the number of upper (lower) multi-level edges touched by  $e$ .

**Proof:**

Definition 18 defines the number of visible upper and lower level edges of a given node. A flat edge is intended to be drawn according to GS\_POLYLINE and the shortest route between the connected nodes is assumed. Hence,  $s_f$  and  $e_f$  denote the index positions between the connected nodes excluding them and therefore  $K_f^\uparrow$  as well as  $K_f^\downarrow$  denote the number of upper (lower) multi-level edges touched by a specified edge.  $\square$

**Definition 19 (length of a flat edge)**

For a flat edge  $e = \{u, v\}$  with  $r(u) = r(v)$

$$\delta_f(\bar{G}, u, v) := |\sigma_{r(u)}(u) - \sigma_{r(v)}(v)| - 1$$

denotes the length of the flat edge  $e$ .

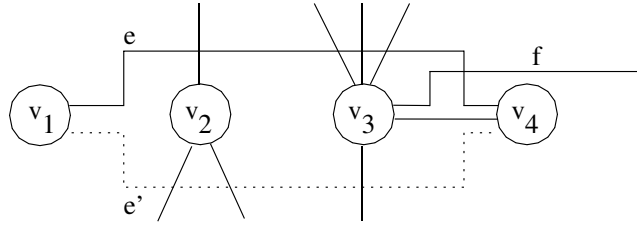


Figure 4.24: Example for the calculation of the number of flat crossings in theorem 2. The alternative route of  $e$  is drawn in dotted line style.

Calculating the exact number of flat crossings is illustrated in Figure 4.24. If  $e$  is drawn above the nodes, it obviously induces  $K_f^\uparrow(\bar{G}, v_1, v_4) = K_f^\uparrow(\bar{G}, v_2) + K_f^\uparrow(\bar{G}, v_3) = 4$  crossings with upper multi-level edges. If  $e$  is drawn below  $K_f^\downarrow(\bar{G}, v_1, v_4) = K_f^\downarrow(\bar{G}, v_2) + K_f^\downarrow(\bar{G}, v_3) = 3$  crossings occur.  $\min \{K_f^\uparrow(\bar{G}, v_1, v_4), K_f^\downarrow(\bar{G}, v_1, v_4)\} = 3$  denotes the expected number of crossings with multi-level edges, because we can assume that a routing mechanism in the coordinates assignment will consider this information, too.

The current number of crossings between flat edges and  $e$  can be calculated by considering the number of edges from  $v_2$  and  $v_3$  to nodes at the left of  $v_1$  or at the right of  $v_4$ .

$$K^{\leftrightarrow}(\bar{G}, v, s_f, e_f) := |\{e : e = \{v, w\} \in E_N, r(v) = r(w) \wedge (\sigma_w(w) < s_f \vee \sigma_w(w) > e_f)\}| \quad (4.22)$$

Both possible routes of  $e$  have to be taken into account. Thereby, the concrete number of flat edge crossings depends on the final route (top or bottom) of each involved edge. In fact, the static situation can be calculated in  $O(|E_N|)$  per node if a flag denoting the vertical direction of

the route is maintained for each edge. To influence the edge crossing reduction correctly, we are interested in the optimal configuration for all edges. Hence, (virtually) changing the route of one edge may require (virtual) changes to the flags for the edges inducing flat crossings to reduce the total number of crossings. To be more precise, the subgraph  $G_o$ , which consists of the nodes of a rank and all their flat edges, admits a drawing according to GS\_ORTHOGONAL, should be (virtually) laid out with a minimum number of crossings. Furthermore, the number of crossings with hierarchical edges depending on the route of an individual edge, have to be taken into account, e.g., by route-dependent weights.

Minimizing the number of edge crossings is in general NP-hard [Garey and Johnson 1983]. Unfortunately, the calculation of the (minimum) rectilinear crossing number appears to be exponential in runtime<sup>13</sup>.

As a basic approximation, we can denote the current number of crossings with hierarchical edges as

$$\begin{aligned} K_f(\bar{G}, e) &:= K_f(\bar{G}, \{u, v\}) \\ &:= w(\{u, v\}) \cdot \begin{cases} \min \left\{ K_f^\uparrow(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v) \right\} & : \text{ if } r(u) = r(v) \\ 0 & : \text{ otherwise} \end{cases} \end{aligned} \quad (4.23)$$

A more precise but also not exact value can be obtained by considering (4.22) or a similar formula, which also takes the direction of the route of a flat edge into account. Let

$$K_f^\uparrow(\bar{G}, e) := K_f^\uparrow(\bar{G}, u, v) := \sum_{\alpha=s_f(u,v)}^{e_f(u,v)} K_f^\uparrow(\bar{G}, \sigma_{r(u)}[\alpha]) + K^{\leftrightarrow}(\bar{G}, \sigma_{r(u)}[\alpha], s_f(u, v), e_f(u, v))$$

and  $K_f^\downarrow$  be defined similarly. Now  $K_f^\uparrow(\bar{G}, v_1, v_4) = 5$  and  $K_f^\downarrow(\bar{G}, v_1, v_4) = 3$  are valid crossing numbers for Figure 4.24. Let

$$\begin{aligned} K_f^{\leftrightarrow}(\bar{G}, e) &:= K_f^{\leftrightarrow}(\bar{G}, \{u, v\}) \\ &:= w(\{u, v\}) \cdot \begin{cases} \min \left\{ K_f^\uparrow(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v) \right\} & : \text{ if } r(u) = r(v) \\ 0 & : \text{ otherwise} \end{cases} \end{aligned} \quad (4.24)$$

Obviously,  $\sum_{e \in \bar{E}_N} K_f(\bar{G}, e) \leq \sum_{e \in \bar{E}_N} K_f^{\leftrightarrow}(\bar{G}, e)$  holds. Let  $K_f^*(\bar{G})$  be the minimum number of edge crossings induced by flat edges on a fixed sequence of nodes. In the case that no crossings between flat edges occur,  $K_f^*(\bar{G}) = \sum_{e \in \bar{E}_N} K_f(\bar{G}, e)$  holds.  $K_f^*(\bar{G}) = \sum_{e \in \bar{E}_N} K_f^{\leftrightarrow}(\bar{G}, e)$  may occur when no subsequent changes of edge routes for flat-flat edges are required to obtain the minimum number of crossings. Otherwise,  $K_f^*(\bar{G})$  lies somewhere between the basic approximation via (4.23) and the approximation in (4.24), which also takes the static flat-flat crossings into account.  $K_f^*(\bar{G})$  can be obtained by solving the combinatorial problem of subsequent changes to flat edge

<sup>13</sup>Currently, the bipartite crossing number [Garey and Johnson 1983] and the odd-crossing number [Pach and Tóth 2000] were proven to be NP-complete, but not the rectilinear crossing number [Pach 1998].

routes or by counting the crossings in the (virtual) drawing of  $G_o$  calculated by a layout algorithm for rectilinear drawings on fixed visible node sequences, which also considers the alternating edge weights arising from the crossings with hierarchical edges.

Due to the computational complexity for calculating  $K^{\leftrightarrow}(\bar{G}, \sigma_{r(u)}[\alpha], s_f(u, v), e_f(u, v))$  or  $K_f^*(\bar{G})$ , we will consider the basic approximation via (4.23) only. To avoid that longer edges are chosen as best configuration in situations of equal number of flat crossings, we also consider the length of flat edges in the following theorem:

**Theorem 2 (pseudo number of flat crossings)**

Let  $G = (V, E_H, E_N)$  be a graph,  $\bar{G} = (V, E_H, E_N, n, \sigma)$  the  $n$ -level hierarchy of  $G$  according to definition 9. Then

$$K_f(\bar{G}, e) := K_f(\bar{G}, \{u, v\}) \\ := w(\{u, v\}) \cdot \begin{cases} \min \left\{ K_f^\uparrow(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v) \right\} + \delta_f(\bar{G}, u, v) & : \text{ if } r(u) = r(v) \\ 0 & : \text{ otherwise} \end{cases}$$

with  $w(e) = w(\{u, v\}) > 0$  as the (externally given) weight of the edge  $e$  is the number of edge crossings induced by  $e$ .

$$K_f(\bar{G}, v) := \sum_{e \in e_1 : e_1 = \{v, w\} \in E_N, r(v) = r(w)} K_f(\bar{G}, e)$$

denotes the number of flat crossings respecting all flat edges connected to  $n$  and

$$K_f(\bar{G}) := \sum_{e \in E_H \cup E_N} K_f(\bar{G}, e) := \sum_{e \in E_N} K_f(\bar{G}, e)$$

the number of flat crossings of the graph  $G$ .

Similar to the binary version of the crossing matrix in definition definition 15, a variant of theorem 2 without respecting  $w(e)$  might be used in the implementation for example according to a certain configuration flag.

Theorem 2 is more appropriate for implementation purpose, (4.25) when reasoning about changes to the number of flat edge crossings after incremental changes, like exchanging neighbors or inserting nodes.

**Corollary 8 (alternative form of  $K_f(\bar{G}, e)$ )**

$K_f(\bar{G}, e)$  can also be written as

$$K_f(\bar{G}, \{u, v\}) = \\ w(\{u, v\}) \cdot \begin{cases} \min \left\{ K_f^\uparrow(\bar{G}, u, v) + \delta_f(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v) + \delta_f(\bar{G}, u, v) \right\} & : \text{ if } r(u) = r(v) \\ 0 & : \text{ otherwise} \end{cases} \quad (4.25)$$

**Proof:**

In preparation step S6 of the layout algorithm, self loops have been removed. Hence, with  $u \neq v$   $|\sigma_{r(u)}(u) - \sigma_{r(v)}(v)| \geq 1$  and therefore  $\delta_f(\bar{G}, u, v) = |\sigma_{r(u)}(u) - \sigma_{r(v)}(v)| - 1 \geq 0$  and  $\min \left\{ K_f^\uparrow(\bar{G}, u, v) + \delta_f(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v) + \delta_f(\bar{G}, u, v) \right\} = \min \left\{ K_f^\uparrow(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v) \right\} + \delta_f(\bar{G}, u, v)$ .

□

**Corollary 9** ( $K_f^\uparrow(\bar{G}, u, v)$ ,  $K_f^\downarrow(\bar{G}, u, v)$  and exchanging neighbors)

When  $u \in \sigma_i$  in  $G$  is exchanged with its immediate left neighbor  $l$  at  $\sigma_i(l) = \sigma_i(u) - 1$ , the following properties hold:

1.  $K_f^\uparrow(\bar{G}^*, u, l) = K_f^\uparrow(\bar{G}, u, l)$
2.  $K_f^\uparrow(\bar{G}^*, u, v) = \begin{cases} K_f^\uparrow(\bar{G}, u, v) - K_f^\uparrow(\bar{G}, l) & : \sigma_i(v) < \sigma_i(l) \\ K_f^\uparrow(\bar{G}, u, v) + K_f^\uparrow(\bar{G}, l) & : \sigma_i(v) > \sigma_i(l) \end{cases}$   
if  $\{u, v\} \in E_N \wedge v \neq l \wedge r(v) = i$
3.  $K_f^\uparrow(\bar{G}^*, l, v) = \begin{cases} K_f^\uparrow(\bar{G}, l, v) + K_f^\uparrow(\bar{G}, u) & : \sigma_i(v) < \sigma_i(u) \\ K_f^\uparrow(\bar{G}, l, v) - K_f^\uparrow(\bar{G}, u) & : \sigma_i(v) > \sigma_i(u) \end{cases}$   
if  $\{l, v\} \in E_N \wedge v \neq u \wedge r(v) = i$
4.  $K_f^\uparrow(\bar{G}^*, v, w) = K_f^\uparrow(\bar{G}, v, w)$  if  $\{v, w\} \in E_N \wedge v, w \notin \{l, u\}$

Similar properties hold, when  $u$  is exchanged with its immediate right neighbor  $r$  or if  $K_f^\downarrow$  is used instead of  $K_f^\uparrow$ .

**Proof:**

$K_f^\uparrow(\bar{G}^*, n) = K_f^\uparrow(\bar{G}, n)$  and  $K_f^\downarrow(\bar{G}^*, n) = K_f^\downarrow(\bar{G}, n)$ , because definition 18 is independent from node positions within ranks and no nodes are inserted or removed.

$u$  is exchanged with  $l$ . Let  $E_1 = \{e : e = \{u, v\} \in E_N, r(u) = r(v)\}$ . Let  $i = r(u)$  define a shortcut. Then for all  $e \in E_1$  the following equations are implicitly given by exchanging  $u$  and  $l$ :

$$\sigma_i^*(l) = \sigma_i^*(u) + 1 = \sigma_i(u) = \sigma_i(l) + 1 \quad (4.26)$$

and

$$\sigma_i^*(u) = \sigma_i^*(l) - 1 = \sigma_i(l) = \sigma_i(u) - 1 \quad (4.27)$$

Let  $x \in \sigma_{r(x)}$   $x \notin \{u, l\}$ , then

$$\sigma_{r(x)}^*(x) = \sigma_{r(x)}(x) \quad (4.28)$$

because  $x$  is not affected by the exchange operation. Let  $v, w \in \sigma_i$   $v, w \notin \{u, l\}$



$$1. K_f^\uparrow(\bar{G}^*, u, l) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.26)}{=} \sum_{\alpha=\sigma_i^*(u)+2}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) = 0 =$$

$$\sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(l)} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) \stackrel{(4.26)}{=} \sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, u, l)$$

via the definition of the empty sum.

$$2. \sigma_i^*(v) < \sigma_i^*(u) \leftrightarrow \sigma_i(v) < \sigma_i(l):$$

$$K_f^\uparrow(\bar{G}^*, u, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(v)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.27),(4.28)}{=} \sum_{\alpha=\sigma_i^*(v)+1}^{\sigma_i^*(l)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) =$$

$$\sum_{\alpha=\sigma_i^*(v)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) - K_f^\uparrow(\bar{G}, l) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, u, v) - K_f^\uparrow(\bar{G}, l)$$

$$\sigma_i^*(v) > \sigma_i^*(u) \leftrightarrow \sigma_i(v) > \sigma_i(l):$$

$$K_f^\uparrow(\bar{G}^*, u, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(u)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.27),(4.28)}{=} \sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) + K_f^\uparrow(\bar{G}^*, l) =$$

$$\sum_{\alpha=\sigma_i^*(u)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) + K_f^\uparrow(\bar{G}, l) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, u, v) + K_f^\uparrow(\bar{G}, l)$$

$$3. \sigma_i^*(v) < \sigma_i^*(l) \leftrightarrow \sigma_i(v) < \sigma_i(u):$$

$$K_f^\uparrow(\bar{G}^*, l, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(v)+1}^{\sigma_i^*(l)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.26),(4.28)}{=} \sum_{\alpha=\sigma_i^*(v)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) + K_f^\uparrow(\bar{G}^*, u) =$$

$$\sum_{\alpha=\sigma_i^*(v)+1}^{\sigma_i^*(l)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) + K_f^\uparrow(\bar{G}, u) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, l, v) + K_f^\uparrow(\bar{G}, u)$$

$$\sigma_i^*(v) > \sigma_i^*(l) \leftrightarrow \sigma_i(v) > \sigma_i(u):$$

$$K_f^\uparrow(\bar{G}^*, l, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.26),(4.28)}{=} \sum_{\alpha=\sigma_i^*(u)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) =$$

$$\sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) - K_f^\uparrow(\bar{G}, u) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, l, v) - K_f^\uparrow(\bar{G}, u)$$

$$4. \sigma_i^*(w) < \sigma_i^*(u) < \sigma_i^*(l) < \sigma_i^*(v) \leftrightarrow \sigma_i(w) < \sigma_i(l) < \sigma_i(u) < \sigma_i(v):$$

$$K_f^\uparrow(\bar{G}^*, w, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(w)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) = \sum_{\alpha=\sigma_i^*(w)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) + K_f^\uparrow(\bar{G}^*, u) +$$

$$K_f^\uparrow(\bar{G}^*, l) + \sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.27),(4.26),(4.28)}{=} \sum_{\alpha=\sigma_i^*(w)+1}^{\sigma_i^*(u)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) + K_f^\uparrow(\bar{G}, l) +$$

$$K_f^\uparrow(\bar{G}, u) + \sum_{\alpha=\sigma_i^*(l)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) = \sum_{\alpha=\sigma_i^*(w)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, w, v)$$

similarly for  $K_f^\uparrow(\bar{G}^*, v, w) = \dots = K_f^\uparrow(\bar{G}, w, v)$

$$\sigma_i^*(w) < \sigma_i^*(v) < \sigma_i^*(u) \vee \sigma_i^*(l) < \sigma_i^*(w) < \sigma_i^*(v) \leftrightarrow \sigma_i(w) < \sigma_i(v) < \sigma_i(l) \vee \sigma_i(u) < \sigma_i(w) < \sigma_i(v):$$

$$K_f^\uparrow(\bar{G}^*, w, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_i^*(w)+1}^{\sigma_i^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_i^*[\alpha]) \stackrel{(4.28)}{=} \sum_{\alpha=\sigma_i(w)+1}^{\sigma_i(v)-1} K_f^\uparrow(\bar{G}, \sigma_i[\alpha]) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, w, v)$$

similarly for  $K_f^\uparrow(\bar{G}^*, v, w) = \dots = K_f^\uparrow(\bar{G}, w, v)$

Now let  $E_2 = \{e : e = \{w, v\} \in E_N, r(u) \neq r(w) \wedge r(w) = r(v)\}$ . Then for all  $e \in E_2$ :

$$K_f^\uparrow(\bar{G}^*, w, v) \stackrel{(4.21)}{=} \sum_{\alpha=\sigma_{r(w)}^*(w)+1}^{\sigma_{r(w)}^*(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_{r(w)}^*[\alpha]) = \sum_{\alpha=\sigma_{r(w)}(w)+1}^{\sigma_{r(w)}(v)-1} K_f^\uparrow(\bar{G}^*, \sigma_{r(w)}^*[\alpha]) \stackrel{(4.21)}{=} K_f^\uparrow(\bar{G}, w, v)$$

because the exchange between  $u$  and  $v$  did not affect any other node in any other rank. The case when  $u$  is exchanged with  $r$  or for  $K_f^\downarrow$  can be handled similarly.  $\square$

### Corollary 10 ( $\delta_f(\bar{G}, u, v)$ and exchanging neighbors)

When  $u \in \sigma_i$  in  $G$  is exchanged with its immediate left neighbor  $l$  at  $\sigma_i(l) = \sigma_i(u) - 1$ , the following properties hold:

1.  $\delta_f(\bar{G}^*, u, l) = \delta_f(\bar{G}, u, l)$
2.  $\delta_f(\bar{G}^*, u, v) = \begin{cases} \delta_f(\bar{G}, u, v) - 1 & : \sigma_i(v) < \sigma_i(l) \\ \delta_f(\bar{G}, u, v) + 1 & : \sigma_i(v) > \sigma_i(l) \end{cases}$  if  $\{u, v\} \in E_N \wedge v \neq l \wedge r(v) = i$
3.  $\delta_f(\bar{G}^*, l, v) = \begin{cases} \delta_f(\bar{G}, l, v) + 1 & : \sigma_i(v) < \sigma_i(u) \\ \delta_f(\bar{G}, l, v) - 1 & : \sigma_i(v) > \sigma_i(u) \end{cases}$  if  $\{l, v\} \in E_N \wedge v \neq u \wedge r(v) = i$
4.  $\delta_f(\bar{G}^*, v, w) = \delta_f(\bar{G}, v, w)$  if  $\{v, w\} \in E_N \wedge v, w \notin \{l, u\}$

Similar properties hold, when  $u$  is exchanged with its immediate right neighbor  $r$ .

### Proof:

Using the conventions defined in the proof of corollary 9:

1.  $\delta_f(\bar{G}^*, u, l) = |\sigma_i^*(u) - \sigma_i^*(l)| - 1 \stackrel{(4.27), (4.26)}{=} |\sigma_i(u) - 1 - (\sigma_i(l) + 1)| - 1 = |\sigma_i(u) - \sigma_i(l)| - 1 = \delta_f(\bar{G}, u, l)$  Because  $u$  and  $l$  are neighbors,  $|\sigma_i^*(u) - \sigma_i^*(v)| = |\sigma_i(u) - \sigma_i(v)| = 1$  and  $\delta_f(\bar{G}^*, u, l) = 0 = \delta_f(\bar{G}, u, l)$
2.  $\sigma_i^*(v) < \sigma_i^*(u) \leftrightarrow \sigma_i(v) < \sigma_i(l)$ :  
 $\delta_f(\bar{G}^*, u, v) = |\sigma_i^*(u) - \sigma_i^*(v)| - 1 = \sigma_i^*(u) - \sigma_i^*(v) - 1 \stackrel{(4.27), (4.28)}{=} \sigma_i(u) - 1 - \sigma_i(v) - 1 = |\sigma_i(u) - \sigma_i(v)| - 2 = \delta_f(\bar{G}, u, v) - 1$   
 $\sigma_i^*(v) > \sigma_i^*(u) \leftrightarrow \sigma_i(v) > \sigma_i(l)$ :  
 $\delta_f(\bar{G}^*, u, v) = |\sigma_i^*(u) - \sigma_i^*(v)| - 1 = \sigma_i^*(v) - \sigma_i^*(u) - 1 \stackrel{(4.27), (4.28)}{=} \sigma_i(v) - (\sigma_i(u) + 1) - 1 = |\sigma_i(u) - \sigma_i(v)| = \delta_f(\bar{G}, u, v) + 1$
3.  $\sigma_i^*(v) < \sigma_i^*(l) \leftrightarrow \sigma_i(v) < \sigma_i(u)$ :  
 $\delta_f(\bar{G}^*, l, v) = |\sigma_i^*(l) - \sigma_i^*(v)| - 1 = \sigma_i^*(l) - \sigma_i^*(v) - 1 \stackrel{(4.26), (4.28)}{=} \sigma_i(l) + 1 - \sigma_i(v) - 1 = |\sigma_i(l) - \sigma_i(v)| = \delta_f(\bar{G}, l, v) + 1$

$$\sigma_i^*(v) > \sigma_i^*(l) \leftrightarrow \sigma_i(v) > \sigma_i(u):$$

$$\begin{aligned} \delta_f(\bar{G}^*, l, v) &= |\sigma_i^*(l) - \sigma_i^*(v)| - 1 = \sigma_i^*(v) - \sigma_i^*(l) - 1 \stackrel{(4.26), (4.28)}{=} \sigma_i(v) - (\sigma_i(l) + 1) - 1 = \\ &= |\sigma_i(l) - \sigma_i(v)| - 2 = \delta_f(\bar{G}, l, v) - 1 \end{aligned}$$

4. let  $\{e : e = \{v, w\} \in E_N, r(v) = r(w)\}$   $\sigma_{r(v)}^*(v) < \sigma_{r(w)}^*(w) \leftrightarrow \sigma_{r(v)}(v) < \sigma_{r(w)}(w)$ :

$$\begin{aligned} \delta_f(\bar{G}^*, w, v) &= |\sigma_{r(w)}^*(w) - \sigma_{r(v)}^*(v)| - 1 = \sigma_{r(w)}^*(w) - \sigma_{r(v)}^*(v) - 1 \stackrel{(4.28)}{=} \sigma_{r(w)}(w) - \sigma_{r(v)}(v) - \\ &= 1 = |\sigma_{r(w)}(w) - \sigma_{r(v)}(v)| - 1 = \delta_f(\bar{G}, w, v) \end{aligned}$$

The case when  $u$  is exchanged with  $r$  can be handled similarly. □

**Lemma 4 (number of flat crossings when exchanging a node and its neighbor)**

If  $u \in \sigma_i$  in  $G$  is exchanged with its immediate left neighbor  $l$  at  $\sigma_i(l) = \sigma_i(u) - 1$  or its immediate right neighbor  $r$  at  $\sigma_i(r) = \sigma_i(u) + 1$ , the number of crossings change as follows:

$$K_f(\bar{G}^*) = K_f(\bar{G}) - K_f(\bar{G}, u) - K_f(\bar{G}, l) + K_f(\bar{G}^*, u) + K_f(\bar{G}^*, l)$$

$$K_f(\bar{G}^*) = K_f(\bar{G}) - K_f(\bar{G}, u) - K_f(\bar{G}, r) + K_f(\bar{G}^*, u) + K_f(\bar{G}^*, r)$$

**Proof:**

Let  $E_1 = \{e : e = \{v, w\} \in E_N, r(v) = r(w)\}$  be the set of flat edges,

$E_2 = \{e : e = \{v, w\} \in E_1, v, w \notin \{l, u\}\}$  the set of flat edges which do neither connect to  $l$  nor to  $u$  and  $E_3 = \{e : e = \{v, w\} \in E_1, v \in \{l, u\} \vee w \in \{l, u\}\}$  the set of flat edges which connect to  $l$  or  $u$ .  $E_1 = E_2 \cup E_3$  and  $E_2 \cap E_3 = \emptyset$ .

Let  $E_4 = \{e : e = \{l, w\} \in E_3, w \neq u\}$ ,  $E_5 = \{e : e = \{l, u\} \in E_3\}$  and  $E_6 = \{e : e = \{u, w\} \in E_3, w \neq l\}$  so that  $E_3 = E_4 \cup E_5 \cup E_6$  and  $E_i \cap E_j = \emptyset, i \neq j, i, j \in \{4, 5, 6\}$ . Because  $E_5$  only contains the edges connecting the neighbors  $u$  and  $l$ ,  $\sum_{e \in E_5} K_f(\bar{G}^*, e) = 0$  according to corollary 9

and corollary 10.  $E_4 \cup E_5$  is the set of flat edges connected to  $l$  and  $E_5 \cup E_6$  the set of flat edges connected to  $u$ . Therefore

$$\begin{aligned} \sum_{e \in E_3} K_f(\bar{G}^*, e) &= \sum_{e \in E_4} K_f(\bar{G}^*, e) + \sum_{e \in E_5} K_f(\bar{G}^*, e) + \sum_{e \in E_6} K_f(\bar{G}^*, e) \\ &= \left( \sum_{e \in E_4} K_f(\bar{G}^*, e) + \sum_{e \in E_5} K_f(\bar{G}^*, e) \right) + \left( \sum_{e \in E_5} K_f(\bar{G}^*, e) + \sum_{e \in E_6} K_f(\bar{G}^*, e) \right) \\ &\stackrel{\text{th. 2}}{=} K_f(\bar{G}^*, u) + K_f(\bar{G}^*, l) \end{aligned} \tag{4.29}$$

and similarly for  $\bar{G}$ .

$$\begin{aligned} K_f(\bar{G}^*) - (K_f(\bar{G}^*, u) + K_f(\bar{G}^*, l)) &\stackrel{(4.29)}{=} K_f(\bar{G}^*) - \sum_{e \in E_3} K_f(\bar{G}^*, e) \stackrel{\text{th. 2}}{=} \sum_{e \in E} K_f(\bar{G}^*, e) - \\ &\sum_{e \in E_3} K_f(\bar{G}^*, e) \stackrel{\text{th. 2}}{=} \sum_{e \in E_N} K_f(\bar{G}^*, e) - \sum_{e \in E_3} K_f(\bar{G}^*, e) \stackrel{\text{th. 2}}{=} \sum_{e \in E_1} K_f(\bar{G}^*, e) - \sum_{e \in E_3} K_f(\bar{G}^*, e) = \\ &\sum_{e \in E_2} K_f(\bar{G}^*, e) + \sum_{e \in E_3} K_f(\bar{G}^*, e) - \sum_{e \in E_3} K_f(\bar{G}^*, e) = \sum_{e \in E_2} K_f(\bar{G}^*, e) = \sum_{e \in E_2} K_f(\bar{G}, e) = \end{aligned}$$

$$\begin{aligned} \sum_{e \in E_2} K_f(\bar{G}, e) + \sum_{e \in E_3} K_f(\bar{G}, e) - \sum_{e \in E_3} K_f(\bar{G}, e) &= \sum_{e \in E_1} K_f(\bar{G}, e) - \sum_{e \in E_3} K_f(\bar{G}, e) \stackrel{\text{th.2}}{=} \sum_{e \in E_N} K_f(\bar{G}, e) - \\ \sum_{e \in E_3} K_f(\bar{G}, e) &\stackrel{\text{th.2}}{=} \sum_{e \in E} K_f(\bar{G}, e) - \sum_{e \in E_3} K_f(\bar{G}, e) \stackrel{\text{th.2}}{=} K_f(\bar{G}) - \sum_{e \in E_3} K_f(\bar{G}, e) \stackrel{(4.29)}{=} K_f(\bar{G}) - \\ &(K_f(\bar{G}, u) + K_f(\bar{G}, l)) \end{aligned}$$

As discussed above, for a node  $u \in V$ ,  $K_f^\uparrow(\bar{G}, u)$  and  $K_f^\downarrow(\bar{G}, u)$  change only, if nodes are added or removed. □

**Lemma 5 (number of flat crossings when inserting or removing a node)**

When node  $u$  is added to graph  $G$ ,  $u$  is assigned to rank  $i$  so that  $i = r(u)$ , then  $u$  and the edges connected to  $u$  are inserted into  $\sigma_i$  at an arbitrary position.

Let  $G^*$  be the graph after inserting (removing)  $u$ . Then the number of flat crossings changes according to

$$K_f(\bar{G}^*) = K_f(\bar{G}) + K_f(\bar{G}^*, u)$$

if  $u$  is (temporarily) inserted into  $G$  and

$$K_f(\bar{G}^*) = K_f(\bar{G}) - K_f(\bar{G}^*, u)$$

if  $u$  is (temporarily) removed from  $G$ .

It makes sense to store the the flat edge direction as a rendering hint for the coordinates assignment. Furthermore, it implicitly considers that all flat edges in the first visible rank can be directed towards the root, in the rank at the bottom of the drawing towards the bottom due to the absence of hierarchical edges.

**Proof:**

When  $u$  is inserted,  $u$  and the edges connected to  $u$  are inserted into  $G^*$ :

$$\begin{aligned} K_f(\bar{G}^*) &\stackrel{\text{th.2}}{=} \sum_{e \in E} K_f(\bar{G}^*, e) = \sum_{e \in E \setminus (E, u)} K_f(\bar{G}^*, e) + \sum_{e \in E(E, u)} K_f(\bar{G}^*, e) \stackrel{\text{cor. 9, cor. 10}}{=} \sum_{e \in E \setminus (E, u)} K_f(\bar{G}, e) + \\ &\sum_{e \in E(E, u)} K_f(\bar{G}^*, e) \stackrel{\text{th.2}}{=} K_f(\bar{G}^*) + K_f(\bar{G}^*, u) \end{aligned}$$

It can be shown similarly that  $K_f(\bar{G}^*) = K_f(\bar{G}) - K_f(\bar{G}^*, u)$  holds when  $u$  is (temporarily) removed. □

As conclusion, the number of crossings respecting flat edges only can be calculated in  $O(|E|^2 \cdot |V|)$  or, if  $K^\uparrow$  and  $K^\downarrow$  are precalculated and cached,  $O(|E| \cdot |V|)$ . Incrementally calculating flat edge crossings reduces the runtime but not the complexity.

On highly connected graphs with a large number of flat edges, the calculation of flat edge crossings runs too slow. Therefore we discuss now an algorithm which runs in  $O(|V|^2)$  but partly allows incremental updates in  $O(|V|)$ .

**Definition 20 (flat edges connection matrices)**

Let  $F^{(i)}$  be a  $|V_i| \times |V_i|$  matrix representing the number of flat edges between nodes in rank  $i$  defined by

$$f_{kl}^{(i)} := |e = \{\sigma_i[k], \sigma_i[l]\}|$$

Let  $F^{\Sigma(i)}$  be a  $|V_i| \times |V_i|$  matrix containing the aggregated number of upper and lower level multi-edges given by

$$f_{kl}^{\Sigma(i)} := \sum_{l < j < k} K_f^\uparrow(\bar{G}, \sigma_i[j]) + \sum_{k < j < l} K_f^\downarrow(\bar{G}, \sigma_i[j]) \quad (4.30)$$

via definition 18.

**Corollary 11 (pseudo number of flat crossings by flat edges connection matrices)**

The pseudo number of flat crossings in a rank specified by theorem 2 can be expressed as

$$K_f(\bar{G}, i) := \sum_{0 \leq k < l < |\sigma_i|} w(\{\sigma_i[k], \sigma_i[l]\}) \cdot f_{kl}^{(i)} \cdot (\min\{f_{kl}^{\Sigma(i)}, f_{lk}^{\Sigma(i)}\} + |k - l| - 1) \quad (4.31)$$

and the number of flat crossings in  $G$  is

$$K_f(\bar{G}) := \sum_{0 \leq i < n} K_f(\bar{G}, i)$$

**Proof:**

Because self loops were eliminated in S6,  $F^{(i)}$  is a matrix as shown below

$$\begin{pmatrix} 0 & & * \\ & \ddots & \\ * & & 0 \end{pmatrix}$$

$F^{\Sigma(i)}$  is a matrix of the following form

$$\begin{pmatrix} 0 & 0 & & \boxed{K^\uparrow} \\ & \ddots & & \\ & 0 & 0 & 0 \\ \boxed{K^\downarrow} & & & \ddots \\ & & 0 & 0 \end{pmatrix}$$

because of the exclusive ranges and the mutual exclusive addenda in (4.30).

---

**Algorithm 4.7** calculate flat matrices
 

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), i$ 
**output:**  $(F^{(i)}, F^{\Sigma(i)})$ 
 $\sigma := \sigma_i$ 
 $F^{(i)} := \text{new } |V_i| \times |V_i|$ 
**for all**  $\{e : e = \{u, v\} \in E_N, r(u) = r(v)\}$  **do**
 $F_{\sigma(u), \sigma(v)}^{(i)} = F_{\sigma_i(u), \sigma_i(v)}^{(i)} + 1$ 
 $F_{\sigma(v), \sigma(u)}^{(i)} = F_{\sigma_i(v), \sigma_i(u)}^{(i)} + 1$ 
**end for**
 $F^{\Sigma(i)} := \text{new } |V_i| \times |V_i|$ 
**if**  $|\sigma_i| > 2$  **then**
**for**  $k := 0$  **to**  $|\sigma_i|$  **do**
 $v := \perp$ 
 $s^\uparrow := 0$ 
 $s^\downarrow := 0$ 
**for**  $l := k + 1$  **to**  $|\sigma_i|$  **do**
**if**  $l < k - 1 \wedge v \neq \perp$  **then**
 $s^\uparrow := s^\uparrow + K_f^\uparrow(\bar{G}, v)$ 
 $F_{kl}^{(i)} := F_{kl}^{(i)} + s^\uparrow$ 
**else if**  $l > k + 1 \wedge v \neq \perp$  **then**
 $s^\downarrow := s^\downarrow + K_f^\downarrow(\bar{G}, v)$ 
 $F_{kl}^{(i)} := F_{kl}^{(i)} + s^\downarrow$ 
**end if**
 $v := \sigma_i[l];$ 
**end for**
**end for**
**end if**
**return**  $(F^{(i)}, F^{\Sigma(i)})$ 


---

Let  $K_f(\bar{G}, u, v)$  be the number of flat crossings induced by all flat edges which connect  $u$  and  $v$ .  $K_f(\bar{G}, e)$  from theorem 2 denotes the number of flat crossings induced by  $e$ . Let  $f(i, k, l) =$

$w(\{\sigma_i[k], \sigma_i[l]\}) \cdot f_{kl}^{(i)} \cdot (\min\{f_{kl}^{\Sigma(i)}, f_{lk}^{\Sigma(i)}\} + |k - l| - 1)$  as a part of (4.31). Hence,

$$\begin{aligned}
K_f(\bar{G}, u, v) &= |\{e : e = \{u, v\}, r(u) = r(v)\}| \cdot K_f(\bar{G}, \{u, v\}) \\
&\stackrel{\text{cor. 11}}{=} f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{(r(u))} \cdot K_f(\bar{G}, \{u, v\}) \\
&\stackrel{\text{th. 2}}{=} f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{(r(u))} \cdot w(\{u, v\}) \cdot (\min\{K_f^\uparrow(\bar{G}, u, v), K_f^\downarrow(\bar{G}, u, v)\} + \delta_f(\bar{G}, u, v)) \\
&\stackrel{\text{cor. 7}}{=} f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{(r(u))} \cdot w(\{u, v\}) \cdot (\min\{\sum_{\alpha=s_f(u,v)}^{e_f(u,v)} K_f^\uparrow(\bar{G}, \sigma_{r(u)}[\alpha]), \sum_{\alpha=s_f(u,v)}^{e_f(u,v)} K_f^\downarrow(\bar{G}, \sigma_{r(u)}[\alpha])\} + \delta_f(\bar{G}, u, v)) \\
&\stackrel{\text{cor. 11}}{=} f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{(r(u))} \cdot w(\{u, v\}) \cdot (\min\{f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{\Sigma(r(u))}, f_{\sigma_{r(v)}(v), \sigma_{r(u)}(u)}^{\Sigma(r(u))}\} + \delta_f(\bar{G}, u, v)) \\
&\stackrel{\text{def. 19}}{=} f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{(r(u))} \cdot w(\{u, v\}) \cdot (\min\{f_{\sigma_{r(u)}(u), \sigma_{r(v)}(v)}^{\Sigma(r(u))}, f_{\sigma_{r(v)}(v), \sigma_{r(u)}(u)}^{\Sigma(r(u))}\} + |\sigma_{r(u)}(u) - \sigma_{r(v)}(v)| - 1)) \\
&= f(r(u), \sigma_{r(u)}(u), \sigma_{r(v)}(v))
\end{aligned}$$

Then (4.31) denotes the number of flat crossings in rank  $i$  and  $K_f(\bar{G})$  follows obviously. Algorithm 4.7 shows the calculation of both flat matrices.  $\square$

For  $F^{(i)}$  and  $F^{\Sigma(i)}$  we assume a matrix implemented according to the technique depicted in Figure 4.22.

**Corollary 12 (incremental operations on flat matrices when adding or exchanging a node)**

Let  $\sigma_i$  be a rank and  $v$  a node to be

- added at the end of  $\sigma_i$ . All entries in  $F^{(i)}$  concerning edge connections with  $v$  have to be adjusted. In  $F^{\Sigma(i)}$  the last column has to be adjusted only.
- exchanged with its immediate left or right neighbor in  $\sigma_i$ . In  $F^{(i)}$  the affected columns have to be exchanged. In  $F^{\Sigma(i)}$  the entries of the adjacent columns have to be adjusted by adding or subtracting the number of connected multi-level edges.

**Proof:**

- According to the definition of  $F^{(i)}$  in corollary 11 it is obvious that when adding  $v$  to  $\sigma_i$  all matrix entries connected to  $v$  have to be adjusted. None of the edges considered for  $F^{(i)}$  and  $F^{\Sigma(i)}$  is affected by inserting  $v$ . All edges connected from  $v$  to other nodes in  $\sigma_i$  would receive the correct crossing number if  $K^\uparrow(\bar{G}, v)$  and  $K^\downarrow(\bar{G}, v)$  would be present in  $F^{\Sigma(i)}$ . Due to the exclusive ranges in the definition of  $F^{\Sigma(i)}$ , the last existing column is not considered in  $F^{\Sigma(i)}$ . Hence, algorithm 4.8 shows the required changes to existing matrices.

---

**Algorithm 4.8** update flat matrices after adding  $v$

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), v, F^{(r(v))}, F^{\Sigma(r(v))}$

**output:**  $(F^{(r(v))}, F^{\Sigma(r(v))})$

```

 $\sigma := \sigma_{r(v)}$ 
for all  $\{e : e = \{v, w\} \in E_N, r(v) = r(w)\}$  do
  if notAdded( $e$ ) then
     $F_{\sigma(v), \sigma(w)}^{(r(v))} := F_{\sigma(v), \sigma(w)}^{(r(v))} + 1$ 
     $F_{\sigma(w), \sigma(v)}^{(r(v))} := F_{\sigma(w), \sigma(v)}^{(r(v))} + 1$ 
  end if
end for
for  $i := |\sigma| - 3$  downto 0 do
  if  $i > |\sigma| \vee i < |\sigma| - 2$  then
     $F_{|\sigma|-1, i}^{\Sigma(r(v))} := F_{|\sigma|-2, i}^{\Sigma(r(v))} + K_f^\uparrow(\bar{G}, v)$ 
     $F_{i, |\sigma|-1}^{\Sigma(r(v))} := F_{i, |\sigma|-2}^{\Sigma(r(v))} + K_f^\downarrow(\bar{G}, v)$ 
  end if
end for
return  $(F^{(r(v))}, F^{\Sigma(r(v))})$ 

```

---

- Exchanging nodes does not affect the number of edges which connect two nodes. Therefore, in  $F^{(i)}$  the appropriate columns have to be exchanged only. Nothing remains to be done for  $F^{(i)}$ , because we assume the implementation suggested in Figure 4.22. Let  $j := \sigma_i(v)$  and without loss of generality  $w := \sigma_i[j - 1]$  be the node to be exchanged with  $v$ . After the exchange and the automatic update of  $F^{\Sigma(i)}$  due to the implementation in Figure 4.22 the values in all rows except for  $j$  and  $j - 1$  are correct, because the sequence of the addenda in (4.31) does not change the result. The values of  $v$  in row  $j - 1$  have to be decremented by  $K^\uparrow(\bar{G}, w)$  or  $K^\downarrow(\bar{G}, w)$ , respectively, and symmetrically values of  $w$  in row  $j$  have to be incremented by  $K^\uparrow(\bar{G}, v)$  or  $K^\downarrow(\bar{G}, v)$ , respectively.

□

When implementing the calculation of the number of flat crossings, for non-iterative crossing reduction algorithms we now have two alternatives at hands. the non-matrix based algorithm runs in  $O(|E| \cdot |V|)$  and the matrix based in  $O(|V|^2)$ . Obviously there is a break even point when we may switch between the algorithms. If  $|E| \leq |V|$  we may use the direct implementation without matrices, if  $|E| \leq |V|$  the matrix variant seems to be appropriate. Of course this decision also depends on the type of the crossing reduction algorithm, which may be incremental, partly incremental or always requires a full recalculation. Furthermore, the concrete implementation also affects the practical break even point as well as the effective runtime.

As a summary we can conclude:



**Theorem 3 (total pseudo number of crossings)**

The total pseudo number of crossings on a given graph  $G$ , its  $n$ -level hierarchy  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  and its induced matrix realization  $\tilde{\mathbf{g}}$  is

$$K(G) = K_h(\tilde{\mathbf{g}}) + K_f(\tilde{G})$$

An incremental edge crossing reduction algorithm can also be implemented in  $O(\min\{|V|^2, |V| \cdot |E|\})$ , too, but effectively the runtime performance is better. Even if we have improved the effective runtime of the calculation of the number of edge crossings, this does neither touch the complexity of a concrete edge crossing reduction algorithm in Section 4.6.5 nor the fact that the minimization problem is NP-complete.

**Forbidden Edges – A Tribute to UML**

In usual graphs an edge can be connected to every side of a node. Application-domain specific graphs may restrict edges to be connected to only a subset of the available areas at the sides of a node. For example, in UML this restriction occurs when subsystems are used. Some basic shapes of subsystems are given in Figure 4.25.

The area of a subsystem can be partitioned into different compartments depending on the situation being modeled. In Figure 4.25 (a), the unnamed compartment usually contains the textual interface specification which allows the subsystem to be accessed from outside.

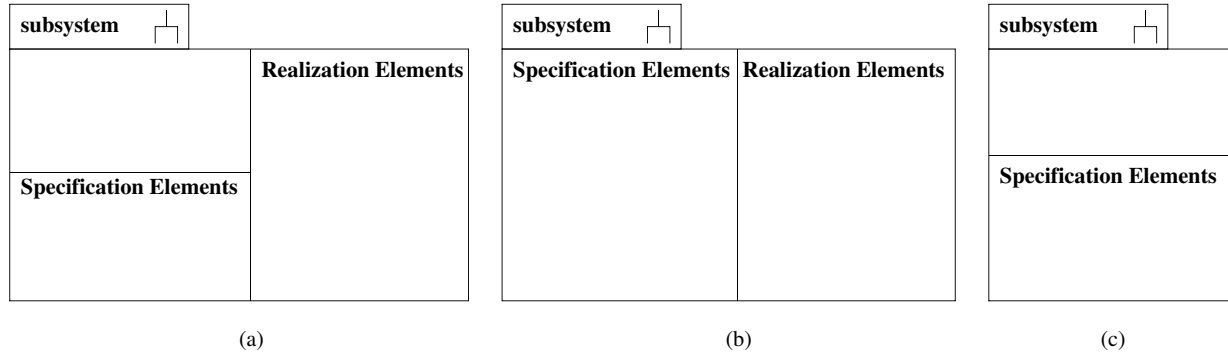


Figure 4.25: Different possibilities to specify a subsystem.

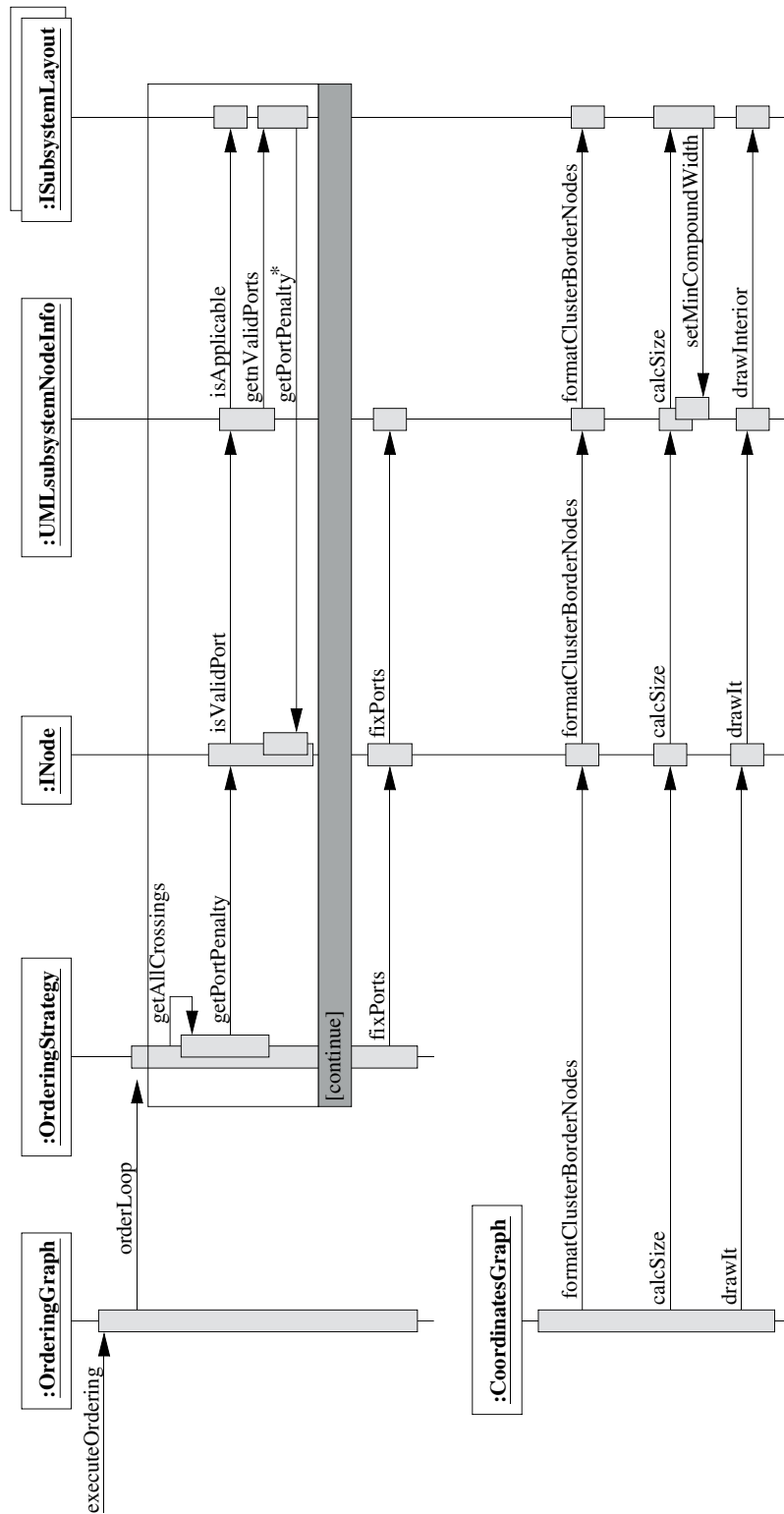


Figure 4.26: A UML sequence diagram depicting the basic mechanism for dynamically deriving the concrete polymorphic shape of a graph element representing a UML subsystem.

Often the “Specification Elements” compartment contains use cases, but also classes (interfaces) are possible. “Realization Elements” specify the relevant parts of the realization of the subsystem by use cases or nested class diagrams. Graphical model elements in both named compartments may be connected to elements outside the subsystem. Therefore, edges connecting nodes outside the subsystem with nodes inside a compartment should not cross the other compartments. Figure 4.25 shows that the decision, which side of the node is forbidden for particular edges, depends on the shape of the subsystem. Additionally, the decision on the concrete shape to be used can be made dynamically while performing the edge crossing reduction depending on the connectivity of the subsystem. This dynamic mechanism is depicted in in Figure 4.26: While calculating the current number of crossings, the port<sup>14</sup> penalty of the affected edges is calculated in the information object of the connected nodes. An example for the use of port penalties is depicted in Figure 4.27. A layout is applicable, if it is provided by the application as a possible

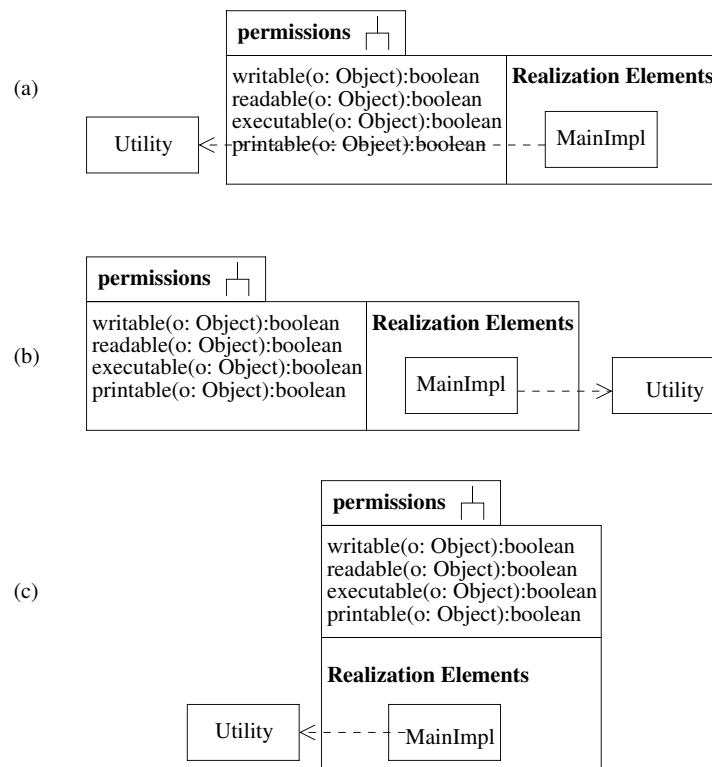


Figure 4.27: (a) Illegal result due to the dependency overlapping the interface compartment, (b) result after the dependency received an appropriate port penalty, (c) result after choosing another polymorphic shape.

layout and in the concrete case of subsystems, if the number of required compartments matches the number of provided compartments of the layout instance. As a side effect, the information object might select the most appropriate one. After the edge crossing reduction is done, all in-

<sup>14</sup>A port specifies the connection point of an edge to the connected node.

formation objects are informed to store their layout as an immutable reference. In coordinates calculation, after inserting all cluster border nodes, the information object and, in particular, the layout object is requested to format the extents of the connected cluster border nodes. Thereby, additional hidden nodes may be inserted to ensure the area for the interface specification and the contained nodes in the "Specification Elements" compartment as shown in Figure 4.25 (a). The layout plug-in object is also considered when calculating the size of the node or the information object, respectively. Finally, the layout plug-in is considered when drawing the node by delegation to its information object.

To keep the discussion on hierarchical and flat edge crossings in Section 4.6.3 simple, we delayed the handling of forbidden edges and extend now the edge crossing formulae.

**Definition 21 (port penalty function)**

Let

$$\phi : E_H \cup E_N \times V \rightarrow \{0, 1\}$$

be the port penalty function which returns, if a given edge illegally overlays a part of the specified node.

**Example:**

Let  $v$  be the class `MainImpl`,  $u$  the class `Utility`,  $\vec{e} = (v, u)$  and  $p$  be the subsystem permissions in Figure 4.27. According to the rank sequences in Figure 4.27 (a),  $\phi(e, p) = 1$  but in Figure 4.27 (b)  $\phi(e, p) = 0$ .

**Corollary 13 (aggregated penalty functions)**

Let  $\phi : E_H \cup E_N \times V \times V \rightarrow \{0, 1\}$  be a penalty function as given in definition 21. With

$$\begin{aligned} \phi^\Sigma : E_H \cup E_N \times V &\rightarrow \mathbb{N}_0 \\ \phi^\Sigma(e = \{v, w\}, x) &:= (|E_H \cup E_N| - 1) \cdot \sum_{p \in \text{ulc}(x)} \phi(e, p) \end{aligned}$$

the aggregated penalty function on all cluster parents of a specified edge can be written as

$$\begin{aligned} \phi : E_H \cup E_N &\rightarrow \mathbb{N}_0 \\ \phi(e) &:= \phi^\Sigma(e = \{v, w\}, v) + \phi^\Sigma(e = \{v, w\}, w) \end{aligned}$$

The aggregated penalty function in corollary 13 multiplies the sum of all parent clusters of the specified with the maximum number of edges. If only node position configurations with forbidden edges exist, the algorithm selects one of the better ones.

Now we can extend the crossing formulae from Section 4.6.3 to respect the edge penalty function:

$$K_h^P(M^{(i)}) = \sum_{j=0}^{|V_i|-2} \sum_{k=j+1}^{|V_i|-1} \left( \sum_{\alpha=0}^{|V_{i+1}|-2} \sum_{\beta=\alpha+1}^{|V_{i+1}|-1} \left\{ \begin{array}{ll} (\phi(e = \{v_j, w_\beta\}) + 1) \cdot m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)} & : \text{if } m_{j\beta}^{(i)} \cdot m_{k\alpha}^{(i)} > 0 \\ \phi(e = \{v_j, w_\beta\}) & : \text{otherwise} \end{array} \right. \right)$$

from (4.7) or

$$K_h^p(M^{(i)}) = \sum_{j=0}^{|V_i|-2} \sum_{\beta=1}^{|V_{i+1}|-1} \begin{cases} (\phi(e = \{v_j, w_\beta\}) + 1) \cdot m_{j\beta}^{(i)} \cdot m_{j\beta}^{\Sigma(i)} & : \text{ if } m_{j\beta}^{(i)} \cdot m_{j\beta}^{\Sigma(i)} > 0 \\ \phi(e = \{v_j, w_\beta\}) & : \text{ otherwise} \end{cases}$$

from corollary 5 with  $v_j \in \sigma_i$  and  $w_\beta \in \sigma_{i+1}$ . Similarly,  $K_f(\bar{G})$  given in theorem 2 can be written as

$$K_f^p(\bar{G}) = \sum_{e \in E_N} \begin{cases} (\phi(e) + 1) \cdot K_f(\bar{G}, e) & : \text{ if } K_f(\bar{G}, e) > 0 \\ \phi(e) & : \text{ otherwise} \end{cases}$$

Finally, as in theorem 3:

**Theorem 4 (total number of crossings respecting penalties)**

Let

$$\phi : E_H \cup E_N \times V \rightarrow \{0, 1\}$$

be an edge penalty function,  $G$  a graph,  $\bar{G} = (V, E_H, E_N, n, \sigma)$  its  $n$ -level hierarchy and  $\bar{\mathbf{g}}$  its induced matrix realization. The total number of crossings of  $G$  respecting edge penalties is

$$K^p(G) = K_h^p(\bar{\mathbf{g}}) + K_f^p(\bar{G})$$

Currently, port penalties are used for subsystems only. Thereby overlays of edges according to the current sequences of nodes in ranks are considered as illustrated in Figure 4.27. Port penalty functions may also be helpful for comments or in conjunction with alternative UML notations like compositions as depicted in Figure 2.10.

#### 4.6.4 Cluster Handling

For clustered graphs, we only ensure that clusters are drawn as convex polygons, while it is desirable to represent clusters as more regular bodies such as circles and rectangles.

[Eades et al. 1997]

Most of the edge crossing reduction techniques briefly described in Section 4.6.1 do not respect clusters at all. As a global invariant, all steps to be executed after S10 have to produce a cluster-valid output graph as defined in Section 4.3.4. Hence, the edge crossing reduction is responsible for reordering the sequences of nodes to reduce the number of crossings (GDR\_EDGE\_CROSS) with respect to cluster-validity (UML\_SEMANTIC\_CLUSTERS) according to definition 13.

In the first step, we are interested how to calculate individual positions for a node  $v$  to be placed in  $\sigma_{r(v)} \setminus \{v\}$ . Minimization techniques, like barycentric ordering, which are able to assign new positions to all nodes of one rank in one step, need a mechanism to ensure that the result is also cluster-valid. The algorithms designed to determine valid positions for individual nodes can be reused to force an arbitrary  $n$ -level hierarchy to be cluster valid.

### Cluster-Valid Positions for Individual Nodes

Let  $G$  be a graph having a cluster-valid  $n$ -level hierarchy, i.e. it is inter-rank-valid and intra-rank-valid according to definition 13 and  $v$  be a node (temporarily removed from)  $G$ . Let

$$\textit{insertPosResult} := \{NO, GLOBAL, EQUAL, CLUSTER, INBETWEEN, EOR, BETWEENRELATED\}$$

be the set of result states of algorithm 4.9. The additional information, which arise from  $\textit{insertPosResult}$ , may be used by the caller to realize certain insertion priorities.

---

#### Algorithm 4.9 calculateInsertPosCluster

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), r, v, d \in \{-1, 1\}$

**output:** valid cluster positions  $\{0 \dots |\sigma_r|\} \rightarrow \textit{insertPosResult}$

```

if  $|\sigma_r| = 0$  then
   $result : \{0\} \rightarrow \textit{insertPosResult}$ 
   $result_0 \rightarrow EQUAL$ 
  return  $result$ 
else
  if  $global(v)$  then
    return  $ip\_handleGlobal(\bar{G}, r)$ 
  else
    return  $ip\_handleNonGlobal(\bar{G}, r, v, d)$ 
  end if
end if

```

---

Algorithm 4.9 handles the case of an empty rank by returning a result other than *NO* and delegates further calculation to sub algorithms depending on the membership of  $v$  in the global cluster.

Algorithm 4.10 is called, if  $v$  is member of the global cluster. Obviously,  $v$  can then be positioned at both ends of the rank, next to a node being also member of the global cluster or between two nodes, which are not member of a common cluster.

Algorithm 4.11 handles the case that  $v$  is not member of the global cluster. First the augmented cluster sequence of the reference rank  $\sigma_{r+d}$  and the positions, which probably restrict the candidate positions, are calculated. Minimum (*minSpec*) and maximum (*maxSpec*) positions denoting the most specific rank members containing  $v$  as well as the minimum (*minAlign*) and maximum (*maxAlign*) positions pointing to the allowed inter-rank area in  $\sigma_r$  are calculated before determining valid positions for  $v$ .

**Algorithm 4.10** *ip\_handleGlobal***input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), r$ **output:** cluster-valid positions if  $global(v) \{0 \dots |\sigma_r|\} \rightarrow insertPosResult$  $\sigma := \sigma_r$  $result : \{0 \dots |\sigma|\} \rightarrow insertPosResult$ **for**  $i := 0$  **to**  $|\sigma|$  **do****if**  $i - 1 < 0 \vee i \geq |\sigma| \vee global(\sigma[i - 1]) \vee global(\sigma[i]) \vee global(LCC(\sigma[i - 1], \sigma[i]))$  **then** $listPos(result, i) := GLOBAL;$ **else** $listPos(result, i) := NO;$ **end if****end for****return**  $result$ **Algorithm 4.11** *ip\_handleNonGlobal***input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), r, v, d \in \{-1, 1\}$ **output:** cluster-valid positions if  $\neg global(v) \{0 \dots |\sigma_r|\} \rightarrow insertPosResult$  $sAlign := augmentClusters(\sigma_{r+d})$  $result : \{0 \dots |\sigma_r|\} \rightarrow insertPosResult$  $S := \{i : 0 \leq i < |\sigma_r|, v \preceq \sigma_r(i) \wedge \nexists w \in \sigma_r, v \prec w \preceq \sigma_r[i]\}$  $minSpec := \min_{-1} S$  $maxSpec := \max_{-1} S$ **for**  $i := 0$  **to**  $|\sigma_r|$  **do****if**  $insideClusterBorderNodes(\sigma_r, v, i)$  **then** $listPos(result, i) := ip\_intraRankPosition(\sigma_r, i, v, minSpec, maxSpec)$ **else** $listPos(result, i) := NO$ **end if****if**  $listPos(result, i) \neq NO \wedge listPos(result, i) \neq EQUAL \wedge listPos(result, i) \neq GLOBAL$  **then****if**  $\neg interRankValidityTest(sAlign, augmentClusters(\sigma_r, v, i))$  **then** $listPos(result, i) := NO$ **end if****end if****end for****return**  $result$ 

Then all positions in  $\sigma_r$  are regarded. If  $i$  is within the cluster border nodes of the containing cluster if present and  $i$  is considered to be intra-rank valid, the inter-rank validity is ensured by calculating the augmented cluster sequence of  $\sigma_r$  with  $v$  virtually inserted at  $i$  and finally testing the augmented cluster sequences for equality.

The method `ip_intraRankPosition` classifies a candidate insertion position according to the intra-rank validity. If  $v$  is inserted at  $i$

- at least in the direct vicinity of a node of the same cluster, *EQUAL*
- at least in the direct vicinity of a node of a containing cluster, *CLUSTER*
- between two not-related clusters, *INBETWEEN*
- between two related but not equal clusters, *BETWEENRELATED*
- at the left or the right end of the rank, *EOR* (end of rank)

is returned.

**Corollary 14 (runtime and correctness of algorithm 4.9)**

Algorithm 4.9 runs in  $O(|V|^2)$ <sup>15</sup> and returns (intra-rank and) inter-rank valid positions only.

**Proof:**

Algorithm 4.9 directly applies definition 11 and definition 12. Therefore we have to prove that the insertion positions for  $v$  suggested by algorithm 4.9 do not taint the validity at the graph when  $v$  is inserted at one of these positions.

- if  $|\sigma_r| = 0$ : if  $|\sigma_r| < 2$ , (4.1) always returns an empty list and, hence,  $\sigma_r$  is still intra-rank valid when inserting  $v$  at  $i = 0$ . If  $v$  is not related to  $\sigma_{r+d}$ , both augmented cluster sequences are empty. If  $v$  is related to  $\sigma_{r+d}$ , both augmented cluster sequences are  $stepUpSet^g(v) \cup \{v\}$  and, therefore,  $\sigma_r$  is still inter-rank valid. In this case Algorithm 4.10 and algorithm 4.9 run in  $O(1)$ .
- if  $|\sigma_r| > 0$ :
  - If  $global(v)$ , algorithm 4.10 is called.  $v$  can be inserted at either the ends of the rank (*EQUAL*) or between two not related clusters (*INBETWEEN*). Hence, these positions are intra-rank valid. The positions are valid, because members of the global cluster do not taint the inter-rank validity.
  - According to definition 11, the following exhaustive enumeration of cases has to be considered to avoid that existing clusters are illegally split:
    - \*  $\nexists_{0 \leq j < |\sigma_r|} v \preceq \sigma_r[j] \Rightarrow minSpec, maxSpec < 0$   
If no related clusters exist in  $\sigma_r$ ,  $v$  can be inserted at either the ends of the rank or between two not related clusters. Hence, these positions are intra-rank valid but may be not inter-rank valid, because there is no related cluster in  $\sigma_r$ , which restricts the insertion position. If  $minAlign, maxAlign \geq 0$ , a related surrounding cluster in  $\sigma_{r+d}$  and  $\sigma_r$  exists. In this case, inter-rank validity has to be ensured.
    - \*  $\exists_{0 \leq j < |\sigma_r|} v \preceq \sigma_r[j] \Rightarrow 0 \leq minSpec \leq maxSpec < |\sigma_r|$

<sup>15</sup>Refer to Chapter A3 for a linear time algorithm, which has been discovered shortly before publishing the final version of this thesis.



- If  $\exists_{0 \leq j < |\sigma_r|} v =_N \sigma_r[j]$ , then  $minSpec$  and  $maxSpec$  denote exactly the area of cluster-equal nodes. By inserting  $v$  next to a cluster-equal node (*EQUAL*), the cluster sequence is not changed significantly: No existing clusters are split and therefore  $\sigma_r$  is still intra-rank valid as well as inter-rank valid regarding  $\sigma_{r+d}$ .
- Otherwise  $minSpec$  and  $maxSpec$  point to the most specific cluster. The intra-rank valid positions can now be determined by testing for (4.2) on each  $minSpec < i \leq maxSpec$  (*BETWEENRELATED*). As long as the cluster of  $v$  was not inserted so far,  $v$  may also be added at the outer border of  $minSpec$  or  $maxSpec$ , respectively (*CLUSTER*).

Because intra-rank validity was defined on pairs of cluster-related nodes in definition 11, we have to ensure, that inserting  $v$  does not change the validity of all pairs. Let  $v_l$  be a node at the left side of  $v$  in  $\sigma_r$ . If  $v_l =_N v$  and no node of a non-contained cluster lies between  $v_l$  and  $v_r$ , then (4.2) holds. If a non-contained cluster would occur between  $v_l$  and  $v$ , then  $\sigma_r$  was not intra-rank valid as a contradiction. The restriction to the most specific cluster does not allow that other members in the cluster of  $v$  occur outside  $minSpec < i \leq maxSpec$ . Hence,  $\sigma_r$  remains valid if  $v \prec v_l$  or  $v_l \prec v$ . If  $v_l \not\prec v$ , no further node to the left of  $v_l$  must be regarded, because these pairs do not taint the intra-rank validity and, therefore, are not considered by definition 11. The right nodes  $v_r$  of  $v$  can be handled similarly.

Inter-rank validity has to be ensured by an additional test for definition 12.

`ip_intraRankPosition`, as implicitly described in this proof, runs in  $O(1)$  and, therefore, the for-loop in Algorithm 4.11 in  $O(|V^2|)$ . The calculation of  $minSpec$ ,  $maxSpec$  as well as augmenting clusters or comparing two augmented cluster sequences for equality can be realized in  $O(|V|)$ . Therefore Algorithm 4.11 is in  $O(|V^2|)$ . Algorithm 4.10 lies in  $O(|V|)$ . Hence, algorithm 4.9 runs in  $O(|V^2|)$ . □

A similar mechanism can be applied in non-compound or mixed-compound graphs, when hierarchical and non-hierarchical relations are taken into account. Thereby, the hierarchical subtree or, if not present, the subgraph of non-hierarchical relations may be used to restrict the available positions and to improve the runtime of the crossing reduction algorithms.

### Enforcing Cluster Alignment

Traditional edge crossing reduction strategies do not respect cluster validity rules. To simply reuse and experiment with edge crossing reduction algorithms, which rely on a certain kind of node sorting, a combination of sorting the nodes of a rank and aligning the nodes to the adjacent upper or lower rank, respectively, is required. Therefore, let

$$sortValue : V \rightarrow \mathbb{R}$$

be a function, which returns the value according to which the nodes within a rank should be ordered. For example, in the barycentric edge crossing minimization algorithm, this value would be the barycenter of the connected nodes. The sorting values determined by the edge crossing reduction algorithm may be adjusted by algorithm 4.12, because the validity of ranks according to definition 13 has a higher priority than the sorting value.

In [Sander 1996b], a barycenter implementation was used for basic edge crossing reduction and then the nodes were carefully repositioned to gain cluster validity. In [Forster 2002], a sorting mechanism, which respects cluster rules, is suggested.

To describe such an algorithm we need the following functions:

$$\text{allSubNodes}(v, \sigma) = \{w : w \in \sigma, w \preceq v\}$$

is the set of all nodes in rank  $\sigma$ , which are somehow contained in  $v$  and

$$\text{directSubNodes}(v, \sigma) = \{w : w \in \sigma, \uparrow(w) =_N v\}$$

is the set of all nodes in rank  $\sigma$  which are directly contained in  $v$ .

$$\text{sortValueAsc}(v_1, v_2) := \text{sortValue}(v_1) < \text{sortValue}(v_2)$$

$$\text{directSubNodes\_sorted}(v, \sigma) = \text{sort}(\text{directSubNodes}(\text{crit}(v, \sigma)), \text{sortValueAsc}(.))$$

returns all nodes directly contained in  $v$  ordered by their current *sortValue* and

$$\text{topNodes}(\sigma) = \{v : v \in \sigma, \text{global}(\uparrow(v))\}$$

is the set of nodes, which are not contained in a cluster and, therefore, member of the global cluster.

---

**Algorithm 4.12** sortAndAlignRank
 

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), r \in \{0 \dots n\}, d \in \{-1, 1\}$

**output:**  $\bar{G}$

```

 $\sigma_r := \text{sort}(\sigma_r, \text{sortValueAsc}(.))$ 
for all  $v \in \text{topNodes}(\sigma_r)$  do
   $\text{updateSortValues}(v)$ 
end for
 $\sigma := \sigma_r$ 
 $\sigma_r := \emptyset$ 
for all  $v \in \sigma_{r+d}$  do
   $\sigma_r := \text{unrollAlongAlign}(\sigma, \sigma_r, v)$ 
end for
 $\sigma_r := \text{unrollNodes}(\sigma, \bar{G}, r, d)$ 
 $\text{enforceClusterBorderPositions}(\sigma_r)$ 
 $\text{calculateCondensedMatrix}(\sigma_r)$ 
return  $\bar{G}$ 

```

---

First, in Algorithm 4.12  $\sigma_r$  is sorted according to the individual *sortValue* as required by the calling edge crossing reduction algorithm. Then, the sort values are adjusted recursively.  $\sigma_r$  is cleared and the clusters, which can directly be identified from the alignment rank  $\sigma_{r+d}$ , are inserted into  $\sigma_r$  as a basic sequence. Then, all nodes which were not inserted so far, are now inserted with respect to cluster-valid positions and *sortValue*. Thereby, for each node in the rank the cluster valid positions are calculated and the best is selected considering the individual types of positions returned by algorithm 4.9. Finally, the positions of the left or right cluster border nodes of the clusters in  $\sigma_r$  must be guaranteed and the condensed interconnect matrix is recalculated as described in Section 4.6.3.

---

**Algorithm 4.13** updateSortValues
 

---

**input:**  $v$ 
**output:** the sort value of  $v$  after the update

$$\text{sortValue}(v) := \frac{\text{sortValue}(v) + \sum_{w \in \text{allSubNodes}(w)} \text{updateSortValues}(w)}{1 + |\text{allSubNodes}(w)|}$$

**return**  $\text{sortValue}(v)$ 


---

Algorithm 4.13 adjusts recursively the individual *sortValue* to keep the nodes of the clusters in a close vicinity.

**Corollary 15 (runtime of algorithm 4.12)**

Algorithm 4.12 runs in  $O(|V|^3)$ .

**Proof:**

A concrete implementation may precalculate the sets defined by *directSubNodes* and may adjust *sortValue* according to algorithm 4.13 in  $O(|V|)$ . Sorting the sets as defined in *directSubNodes\_sorted* for example can be done by one call of quicksort and takes  $O(|V| \log |V|)$ . Only one call is necessary, because *sortValue* can be used to implicitly partition the rank and keep the subsets in one flat set. The individual subsets may be stored in a hash table to gain fast access.

*unrollAlongAlign* runs at most over all nodes in a rank and therefore works in  $O(|V|)$ .

*unrollNodes* also runs at most over all nodes but needs to calculate the valid insertion positions in  $O(|V|^2)$  as shown in Algorithm 4.9 and the best position which runs over the all nodes in  $\sigma_r$  and lies in  $O(|V|)$ . Therefore, *unrollNodes* lies in  $O(|V|^3)$ .

In algorithm 4.12 the initial sorting takes  $O(|V| \cdot \log |V|)$  and updating the condensed matrix  $O(|V|^2)$  as shown in corollary 4. Therefore algorithm 4.12 runs in  $O(|V|^3)$ .  $\square$

In principle, a traditional edge crossing reduction algorithm can be forced to cluster-valid rank sequences

- directly after modifying a rank by calling algorithm 4.12. We will call this type of algorithms *intertwined edge crossing reduction*.

- at the end of the crossing reduction algorithm by executing algorithm 4.12 on adjacent ranks in a top-down loop. We will call this type of algorithms *postprocessing edge crossing reduction*.

### 4.6.5 Extended Crossing Algorithms

A critic is a gong at a railroad crossing clanging loudly and vainly as the train goes by.

Christopher Morley (1890 – 1957)

In this section, the results of Section 4.6.3 on calculating the number of crossings and Section 4.6.4 on handling clusters will be combined to concrete crossing reduction algorithms. First, we will modify and extend the barycentric and the median method. Both algorithms were part of the first version of *SugiBib*, but at that time the implementation did not respect clusters at all. Then we will discuss the hierarchical method, an algorithm which was specifically designed for our application domain.

Main requirement to all edge crossing reduction algorithms is to improve `UML_GRAPHDRAWING GDR_EDGE_CROSS` on all edges of a UML class diagram with respect to `UML_SEMANTIC_CLUSTERS`. This also taints aspects of `GDR_MIN_EDGES`, because the sequence of nodes which form the skeleton of the final drawing, is determined in this step.

So far, we did not discuss the problem of edges crossing a cluster. Edges connected to nodes inside a cluster are allowed to cross the cluster borders, but also edges, which are not connected to nodes inside a cluster, may accidentally cross a cluster. Several mechanisms can be considered to avoid these unpleasing situations, which violate `UML_EDGES (GDR_OVERLAP)`, influence the perception of clusters described by `UML_SEMANTIC_CLUSTERS` and usually imply long edge chains:

- The weight of the hidden edges, which were inserted in Section 4.5.5 to ensure the inner connectivity of clusters, can be increased so that implicit penalties are introduced on other edges crossing these edges. Unfortunately, crossings with hidden edges may lead to other avoidable visible edge crossings, which appear as erroneous visual artifacts to a user, who is not familiar with this mechanism.
- The number of edges, which illegally cross a cluster can be calculated with similar techniques as described in Section 4.6.3. The condensed matrices for the hierarchical cluster crossing number are calculated on edge partitions, one for each cluster of the graph. The crossing number can then be considered with a higher priority than the number of hierarchical and non-hierarchical edge crossings. Even if this approach implies the extendibility to arbitrary partitions, e.g., for new types of edges introduced by future versions of UML or other application domains, calculating the condensed matrices of the partitions is extremely time consuming.

- As described in Section 4.5.5, cluster border nodes have been inserted at the end of the rank assignment. Hence, two edge chains surround each cluster. To reduce cluster crossings, also cluster border nodes in the rank of the cluster base node are required. By assigning appropriate weights dependent on the edges of the surrounding clusters and applying the weighted variant of the edge crossing calculation introduced in Section 4.6.3, a fast mechanism to avoid cluster crossings can be realized.

We decided to implement the latter version based on cluster border nodes. As an alternative to improve the execution speed for larger graphs (more than 200 nodes initially), the first alternative, which implies lower layout quality was also realized.

Depending on the method and sequence of improvements to the number of edge crossings, there is no guarantee that cluster crossings will not occur in the final drawing. Therefore, individual edge crossing methods may apply a postprocessing, e.g., a variant of the transpose heuristic, which will be mentioned along with the median and our hierarchical heuristic.

### Barycentric Method

We took the description of the barycentric method in [Sugiyama et al. 1981] as a prototype and modified it using the algorithms in Section 4.6.4. In principle, the barycentric method is a one sided fixed 2-level algorithm, which is applied to  $n$ -level hierarchies according to the layer-by-layer sweep paradigm.

In [Sugiyama et al. 1981] the algorithm was described in three phases: Phase 1, sorts adjacent ranks until there is no improvement or a maximum number of iterations is reached. Thereby, sorting is done by calling algorithm 4.12 on

$$\text{sortValue}(v) := \frac{1}{|\text{edges}_b(v)|} \cdot \sum_{e=\{v,w\} \in \text{edges}_b(v)} \sigma_{r(w)}(w)$$

with  $\text{edges}_b(v) := \{e : e = \{v,w\} \in E_H \cup E_N, r(v) \neq r(w)\}$ . Phase 2 executes phase 1 and reverses the order of nodes having equal barycenter values in the given rank. In [Sugiyama et al. 1981] it was mentioned that modifying the position of certain nodes by Phase 2 has been empirically found to be effective. In [Fröhlich and Werner 1994], a the nodes were randomly distributed, but this is not appropriate to our application domain due to REQ\_DETERMINISTIC\_ALGORITHM. Phase 2 runs until a maximum number of iterations is exceeded or the graph remains unchanged. Phase 3 realizes the layer-by-layer sweep in alternating directions as long as node positions are changed or a maximum number of iterations was carried out. Finally, cluster validity may be enforced as a postprocessing step.

In phase 1 the node sequences determined by the sorting and the previous node sequences have to be compared. Using theorem 4 or theorem 3, we found the following criterion to be effective:

$$\text{isBetter}(\bar{G}_{old}, \bar{G}) := \begin{cases} \text{true} & : \text{ if } K^P(G) < K^P(G_{old}) \\ \text{true} & : \text{ if } K^P(G) = K^P(G_{old}) \wedge K_h(\bar{\mathbf{g}}) < K_h(\bar{\mathbf{g}}_{old}) \\ \text{false} & : \text{ otherwise} \end{cases} \quad (4.32)$$

To realize `UML_CONSTRAINT_SEQUENCE`, e.g., as a secondary ordering criterion, it makes sense to hide the comparison within a function or a criterion class, which is responsible for efficient comparison of changes induced by successive iterations and for additional secondary ordering criteria.

### Median Method

For the median method, the description in [Gansner et al. 1993] was taken as prototype. In principle, the median method is another one sided fixed 2-level algorithm, which is applied to  $n$ -level hierarchies according to the layer-by-layer sweep paradigm.

In phase 1, all ranks are considered. For each rank, the median values are calculated as follows:

The median value of a vertex is defined as the median position of the adjacent vertices if that is uniquely defined. Otherwise, it is interpolated between the two median positions using a measure of tightness. Generally, the weighted median is biased toward the side where vertices are more closely packed.

[Gansner et al. 1993]

Each rank is sorted by algorithm 4.12, which implicitly handles enforcing cluster validity according to the intertwined fashion. Then, the transpose heuristic, which was described in [Gansner et al. 1993], is applied: It exchanges neighbored nodes if the number of crossings can be reduced. If clusters are present, only nodes of the same cluster may be swapped. Depending on *isBetter*, the more appropriate graph is considered for further processing. Phase 2 realizes the layer-by-layer sweep by executing phase 1 in alternating direction of rank traversal. Finally, cluster validity may be enforced as a postprocessing step.

### Hierarchical Method

After experiments with the approaches described above, we searched for an edge crossing method which takes the hierarchical structure required by `UML_HIERARCHY` into account. Barycenter and median method usually have problems avoiding edge crossings at edge chains in dependent layers due to the layer-by-layer sweep paradigm.

The implementation of our first trial, the  $n$ -level backtracking method, which was briefly described in [Eichelberger and von Gudenberg 2003a], appeared to be exponential in runtime. When comparing the measured runtimes and numbers of edge crossings in Section 5.3 we will have a closer look on the backtracking method.

Instead of sorting ranks according to some heuristics, we will start with an empty rank structure and incrementally reinsert the nodes according to a certain sequence. When reinserting an individual node, the cluster-valid positions are determined, for each position the crossing number  $K^P(G)$  is incrementally calculated and the best position is selected. Thereby the connected edge chains can be considered.

**Algorithm 4.14** hierarchical**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$ **output:**  $\bar{G}^*$  $\bar{G}^* = (\{\}, E_H, E_N, n, \sigma^*)$  $done := \{\}$  $cluster := \{\}$ **while**  $|done| < |V|$  **do** $M := \{w : w \in V, hashGet(done, w) = \perp \wedge c_H(w) = \max_{x \in V} c_H(x)\}$  $v := listGet(M, 0)$  $\bar{G}^* := h\_insertNodeOrCluster(\bar{G}, \bar{G}^*, v, done, cluster, true)$ **end while** $\bar{G}^* := transpose(\bar{G}^*)$ **return**  $\bar{G}^*$ **Algorithm 4.15**  $h\_insertNodeOrCluster$ **input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), \bar{G}^* = (V^*, E_H^*, E_N^*, n, \sigma^*), v \in V, done, cluster, alignToRoot \in bool$ **output:**  $\bar{G}^*$  $C := compoundChildren(v) \cup compoundParents(v)$  $h\_insertChains(\bar{G}, \bar{G}^*, done, cluster, v)$  $clusterInsert := |C| > 0 \wedge hashGet(cluster, crit(v)) \neq \perp$  $V^* := V^* \cup \{v\}$  $\sigma_{r(v)}^* := \sigma_{r(v)}^* \cup \{v\}$  $h\_orderLoop(\bar{G}, \bar{G}^*, v, done)$  $hashPut(done, v, v)$ **if**  $clusterInsert$  **then** $hashPut(cluster, crit(v), v)$  $h\_insertCluster(\bar{G}, \bar{G}^*, done, cluster, v, true)$  $h\_insertCluster(\bar{G}, \bar{G}^*, done, cluster, v, false)$ **end if****return**  $\bar{G}^*$ 

To each node  $v$  an individual complexity  $c_H(v)$  is assigned. Basically, it is sufficient to multiply the number of the currently connected hierarchical edges by two and to add the number of the currently connected non-hierarchical edges. This emphasizes the edges partitions required by UML\_HIERARCHY. When reinserting the first node, all edges are regarded as connected, then only edges to nodes reinserted so far are considered. Dummy nodes in edge chains should receive a complexity of maximum 2.

If no node naming function is present, the nodes can be inserted according to decreasing complexities. Otherwise, when a cluster member is inserted, the basic structure of that cluster and the containing clusters has to be ensured. For each node, the cluster-valid positions are determined and it is inserted at the position which locally admits the best position.

**Algorithm 4.16** *h\_insertCluster***input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$ ,  $\bar{G}^* = (V^*, E_H^*, E_N^*, n, \sigma^*)$ , *done*, *cluster*,  $v \in V$ , *rootDir*  $\in$  *bool***output:**  $\bar{G}^*$ **if** *rootDir* **then** $r := r(v) - 1$  $r_2 := 0$  $\delta := -1$ **else** $r := r(v) + 1$  $r_2 := n$  $\delta := 1$ **end if** $c := \text{crit}(v)$ **while**  $((\text{rootDir} \wedge r \geq r_2) \vee (\neg \text{rootDir} \wedge r \leq r_2))$  **do****if**  $\neg \text{clusterBorderNodeInserted}(v)$  **then** $\text{insertClusterBorderNodes}(v, r)$ **else****if** *rootDir* **then** $c := \text{checkStepUp}(c, r, v)$  {adjust current criterion to parent if necessary}**end if** $v := \text{mostSpecificAndComplexNode}(\text{done}, r, \text{rootDir})$  $\text{h\_insertNodeOrCluster}(\bar{G}, \bar{G}^*, v, \text{done}, \text{cluster}, \neg \text{rootDir})$  $\text{hashPut}(\text{cluster}, \text{crit}(w), w)$ **end if** $r := r + \delta$ **end while** $\text{considerChainsOfLatestInserted}(\bar{G}, \bar{G}^*)$ **return**  $\bar{G}^*$ 

Algorithm 4.14 shows the main steps of the hierarchical edge crossing reduction algorithm. As long as not all nodes have been processed, the locally most complex node is selected and inserted. In the case of a cluster member, for which the surrounding cluster skeleton was not inserted so far, algorithm 4.16 is called for both vertical directions. Finally, after all nodes of the graph have been processed, an implementation of the transpose heuristic similar to the one described in [Gansner et al. 1993] is executed. As an extension, our transpose heuristic is able to eliminate certain crossings at edge chains. transpose may reduce the number of hierarchical edge crossings but also introduce some flat edge crossings or lengthy flat edges, because flat crossings are considered with a lower priority in (4.32). Further experiments adjusting the judgment of the edge crossings in (4.32) are left for future work.

Algorithm 4.16 sweeps over the specified range of ranks and determines the most cluster specific and complex node per rank respecting the containing cluster. With a preference, nodes which are directly member of the surrounding cluster are inserted (not shown in Algorithm 4.16). When



handling a node, also edge chains to nodes processed so far are considered. Processing chains while inserting the nodes of the cluster skeleton has a negative impact on the result. Therefore, while building up the skeleton, chains are not considered but the chains of all nodes of the skeleton are handled afterwards by `considerChainsOfLatestInserted`. This is not explicitly shown here.

For each node, the ordering loop in `h_orderLoop`, which determines a hopefully pleasing position depending on cluster-valid insertion positions and incremental edge crossing number calculation, is called.

## Problems

Unfortunately, some problems arise, which are not sufficiently handled by the described edge crossing mechanisms. In principle, the situations shown in Figure 4.28 can be addressed by also respecting the length of edges in edge crossing calculation. Currently, only pseudo-coordinates induced by the position of the nodes in their rank can be taken into account, because no coordinates are available. Furthermore, these pseudo-coordinates are not always cluster-valid even if the graph is cluster-valid, because a concrete alignment of the coordinates to clusters is not performed at this point of time. Additionally, in *SugiBib*, the extents of the elements of a graph are not available before entering the coordinates assignment phase. Therefore, Figure 4.28 (a) might be handled by considering edge lengths as a secondary criterion in *isBetter* defined by (4.32). The situation in Figure 4.28 (b), which cannot be handled here due to conventions of the implementation, might be algorithmically solved, when the required data and alignment is present. In particular, for flat edges, the edge length criterion can help to reduce lengthy edges in the first and last visible rank, because no crossings at flat edges can occur due to missing visible hierarchical edges.

Then, if a coordinates assignment feature, which we will call “node hopping”, is applied, selected nodes like dummy nodes may be moved inside the cluster. In this case, the separation of tasks between edge crossing reduction and coordinates assignment, as usual in hierarchical drawing algorithms, is not kept anymore, but a better drawing may be produced.

Composite nodes, which are not connected by hierarchical edges and which contain comments or association classes, might be placed anywhere in a rank as depicted in Figure 4.28 (c). This problem does not occur at other non-hierarchical edges, because these edges were not split before. In particular, it occurs when applying the median and the barycenter method, because flat edges are not considered at all. An appropriate postprocessing method can prevent this structural inconsistency.

A discussion, which concrete edge crossing reduction algorithm should be applied on which type of graph will be given in Section 5.3, where various measurements on the concrete implementation will be presented.



methods described in this section. As discussed in [Stallmann et al. 2001], the barycentric as well as the median method admit fast implementations on non-clustered graphs. Due to corollary 15, the runtimes increase by considering validity for individual node positions in intertwined or postprocessing fashion.

The hierarchical method relies on searching positions for individual nodes. Thereby, in the inner loop, first the valid positions are calculated and then for each existing positions the number of edge crossings is calculated. The outer loop determines the hierarchical sequence for inserting the nodes into an initial empty rank structure. This leads to a higher complexity on compound as well as on non-compound graphs<sup>16</sup>.

## 4.7 Intermediary Processing

Human history becomes more and more a race between education and catastrophe.

H. G. Wells (1866 – 1946)

As described in Section 4.1, the intermediary processing steps S12-S14 have to be executed, before executing the coordinates assignment. In this section, we will discuss these steps and the individual actions to be processed.

### 4.7.1 Expand Composite Nodes for Association Classes or Hyperedges

Life shrinks or expands in proportion to one's courage.

Anais Nin (1903 – 1977)

Association classes have been compiled into (existing) composite nodes in S7. In principle, the composite nodes might be kept while coordinates assignment and could be expanded later in S16. This may require that several nodes in the same rank as the association class have to be repositioned. The positions of edges, which have been arranged as horizontal or vertical segments, as well as nodes, which have been positioned next to their children by the coordinates assignment, might thereby be disturbed.

Therefore and from personal experience, we have found the following rule:

*Solve a layout problem at the point of time when it occurs and try to avoid postprocessing.*

This sounds like one of several basic programming rules, e.g., as in [Maguire 1993], but it helps to prevent detours and problems while implementing parts of the layout algorithm.

Due to algorithmic and programming concerns like complexity, small classes, coupling, locality

---

<sup>16</sup>Refer to Chapter A3 for improved versions of the hierarchical sorting algorithm running in quadratic time, which have been discovered shortly before publishing the final version of this thesis.

and separation of general and application domain specific processing we can not avoid postprocessing at different levels at all.

Especially in the case of association classifiers, these classes could better be positioned in the same step with all the other nodes instead of implementing a complex repositioning algorithm. Therefore, we have to ensure at least the space required for the association class and the hidden node simulating the hyperedge.

The same arguments apply to the connection nodes for hyperedges, in particular to avoid overlapping with adornments (UML\_ADORNMENTS).

### 4.7.2 Remove Nesting Edges

The bird a nest, the spider a web, man  
friendship.

William Blake (1757 – 1827)

In S4, nesting relations were represented as hidden edges to influence the rank assignment algorithm S10 and the edge crossing reduction S11. In the coordinates calculation S15, these hidden edges will not be required anymore and would have a negative impact on the port assignment. We will not discuss the influence of these edges on the coordinates assignment here in detail, because the concrete impact arises from the applied coordinates assignment technique. In our case, median positioning of nodes (UML\_MEDIAN) will be a part of the coordinates assignment and therefore the nesting edges would help to align the cluster parent above its children and v.v. This can also be done by considering the positions of the cluster borders. A more important influence arises from invisible cluster dependencies to properly align not connected clusters. As discussed along with algorithm 4.5, this information will also be present implicitly. Therefore, the nesting edges will not be required for coordinates assignment in our case and are removed in this step.

### 4.7.3 Conclusions

When I examine myself and my methods  
of thought, I come to the conclusion that  
the gift of fantasy has meant more to me  
than any talent for abstract, positive think-  
ing.

Albert Einstein (1879 – 1955)

The intermediary processing steps S12, S13 and S14 do neither realize aesthetic criteria nor directly contribute to the layout result. As described above, these steps change the structure of the graph to ensure the presence of certain graph elements and to prevent negative impacts. Therefore, the intermediary processing steps help realizing UML\_ASSOCIATIONCLASSES, UML\_HYPEREDGES and implicitly induce information to prevent various kinds of overlappings according to UML\_NODES, UML\_EDGES and UML\_ADORNMENTS.

Table 4.5 shows the runtime complexities of the intermediary processing steps. Both parts deal-

algorithmic step	runtime complexity
S12	$O( V  +  E_H \cup E_N )$
S13	$O( V  +  E_H \cup E_N )$
S14	$O( E_H )$
intermediary processing macro phase	$O( V  +  E_H \cup E_N )$

Table 4.5: Runtime complexities of the intermediary processing macro phase.

ing with unpacking from composite node have to consider the contained nodes as well as the connected edges. The position of the composite node thereby acts as anchor when reinserting the contained elements. When removing the hidden hierarchical edges in S14, simply all edges are considered.

## 4.8 Coordinates Assignment

A metrical layout consists of *vertex positioning* (ie., determining horizontal and vertical positions, widths and heights of rectangles), and *edge routing* (ie., determining metrical layouts of adjacency edges).

[Sugiyama and Misue 1991]

The final step of the classical hierarchical graph layout algorithm assigns coordinates to a layered graph, which was optimized with respect to edge crossings. At a first glance, this step might appear to be an easy task. In principle, the vertical positions are given by the rank assignment. While sweeping from top to bottom and from left to right over all ranks, the nodes can be positioned successively via accumulating their size in rank. The ranks can be placed by considering the maximum height of the nodes in their rank.

Without further consideration of virtual roots or leafs, edge chains, area requirements of edges or advanced vertical formatting, the simple assignment is illustrated in algorithm 4.17.  $nodeSep(\bar{G})$  denotes the minimum distance between two adjacent nodes in the same rank and  $rankSep(\bar{G})$  the minimum distance between two adjacent ranks.

Some algorithms, as the one which produced Figure 2.15, seem to apply this simple, initial coordinates assignment only.

Handling clusters, area requirements at nodes and edges like non-overlapping adornments, routing hierarchical and non-hierarchical edges and, however, keeping the drawing small and producing a pleasing and readable layout requires much more effort. Unfortunately, this critical task of the layout algorithm is seldom described in detail for hierarchical algorithms or not available for various reasons. Often only point-size nodes and edges without adornments are discussed in the literature.

**Algorithm 4.17** basicInitializeXYCoordinates**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$ **output:**  $\bar{G}$ 


---

```

y := 0
for r := 0 to n - 1 do
  x := 0
  for i := 0 to  $|\sigma_r| - 1$  do
    left( $\sigma_r[i]$ ) := x
    top( $\sigma_r[i]$ ) := y
    x := x + width( $\sigma_r[i]$ ) + nodeSep( $\bar{G}$ )
  end for
  y := y + rankSep( $\bar{G}$ ) + maxv ∈  $\sigma_r$  height(v)
end for
return c_postprocessing( $\bar{G}$ )

```

---

In this section, we will discuss the basic coordinates assignment, which can also be applied to general graphs. For certain parts, adaptations for UML class diagrams will also be discussed in this section. UML specific postprocessing algorithms will be described in Section 4.9.

First, we will briefly mention other work on assigning coordinates for layered graphs. Then, basic issues, e.g., area requirements for nodes and edges, will be discussed and the overall coordinates assignment algorithm will be introduced. Details on that algorithm will be given in the following sections.

### 4.8.1 Previous Work

A well cultivated mind is made up of all the minds of preceding ages; it is only the one single mind educated by all previous time.

Bernard de Fontenelle (1657 – 1757)

Two different coordinates assignment methods for hierarchical graphs have been proposed in [Sugiyama et al. 1981]: Quadratic programming and the priority layout method. For the first one, formulae to consider close and balanced layout as well as fixed orders and straightness of the edges were given. An objective function was constructed which leads to horizontal positions. As mentioned above, the vertical positions can simply be obtained from the rank assignment and the maximum height of the individual nodes in their rank. The priority layout method starts with an initial coordinates assignment, e.g., as that shown in algorithm 4.17 and sweeps then in alternating direction up and down over the ranks. Thereby, the nodes are processed in sequence of priorities and moved as close as possible to barycenter positions of their parents or children, respectively. Even if the nodes are seldom processed in the sequence determined by the edge crossing reduction, that sequence must not be changed while calculating coordinates.

Both methods, quadratic programming as well as priority layout can be reused for the layout of compound graphs [Sugiyama and Misue 1991]. Thereby, a local layout procedure processes the compounds inside-out applying the priority layout method and the global layout recursively moves parent nodes to a median position. An improved variant requires the insertion of artificial sequences of dummy nodes to the left and right of each compound was mentioned but not described in [Sander 1996b].

According to [Gansner et al. 1993; Ellson et al. 2003], the coordinates assignment can be described as the following integer optimization problem, which respects GDR\_MIN\_EDGES by preferring straight lines:

$$\begin{aligned} \min \quad & \sum_{e=\{u,v\} \in E_H} \Omega(e) \cdot w(e) \cdot |x(u) - x(v)| \\ \text{subject to: } & x(z_l) - x(z_r) \geq \rho(z_l, z_r) \end{aligned}$$

where  $z_l$  is the left neighbor of  $z_r$  on the same rank and

$$\rho(z_l, z_r) = \frac{\text{width}(z_l) + \text{width}(z_r)}{2} + \text{nodeSep}(G)$$

is the minimum horizontal separation of  $z_l$  and  $z_r$ . Furthermore,  $\Omega$  is an additional weight function favoring the straightness of long edges and  $w(\vec{e})$  denotes the edge weight function introduced in definition 10. As mentioned in [Gansner et al. 1993], this optimization problem can be solved by the simplex method. Beside the fact that compounds and further properties like UML\_MEDIAN are not considered, applying the simplex method leads to a matrix of  $|V| \cdot |E_H| + |E_H|^2$  entries and runtime does not seem to be satisfactory.

Therefore, as a heuristical approach, the following algorithm was proposed for non-compound graphs in [Gansner et al. 1993]:

---

**Algorithm 4.18** xcoordinate
 

---

```

xcoord := init_xcoord()
xbest := xcoord
for i := 0 to max_iterations do
  medianpos(i, xcoord)
  minedge(i, xcoord)
  minnode(i, xcoord)
  minpath(i, xcoord)
  packcut(i, xcoord)
  if xlength(xcoord) < xlength(xbest) then
    xbest := xcoord
  end if
end for
return xbest

```

---

The coordinates are basically initialized by `init_xcoord`, e.g. by calling algorithm 4.17. Similar to the priority method in [Sugiyama et al. 1981], `medianpos` assigns coordinates with respect

to the median position of the parents or children. `minedge` works similar to `medianpos`, but considers only edges between two real nodes. `minnode` performs local optimizations to reach the median position of all parents and children. `minpath` straightens chains of virtual nodes and `packcut` searches for blocks that can be compacted. Obviously, this algorithm directly supports `UML_HIERARCHY` and `UML_MEDIAN`.

While the simplex method mentioned appears to be realized easily, the heuristic approach implies its own difficulties:

These heuristics make good layouts quickly, but they are complicated to program and the results are sometimes noticeably imperfect. Further fine tuning is difficult because the heuristics begin to interfere with each other.

[Gansner et al. 1993]

Despite that warning, we have implemented algorithm 4.18 in the first version of *SugiBib* and it is still an important part of the coordinates assignment phase. We tried to change the implementation of appropriate algorithms as few as possible, because we are aware of disturbing the results produced so far, and we have extended the algorithm over the time to handle compound graphs, mixed compound graphs as well as non-compound graphs. Furthermore, we handle all nodes and edges at once instead of realizing a separation of local and global layout, which was mentioned above and criticized in [Sander 1996b].

Assignment of coordinates to hierarchical edges, flat edges and self edges was also discussed in [Gansner et al. 1993]: Polygonal regions for the edges were identified and the edges were routed within these regions as splines. A discussion for using funnels [Hershberger and Snoeyink 1994] as edge regions was given in [Dobkin et al. 1997].

To meet `GDR_MANHATTAN`, in [Sander 1996a] a segment ordering graph was constructed, a topological sort of that graph was performed and coordinates were assigned by choosing the locally leftmost position. Balance was reached by applying a variant of the pendulum method. Edge chain segments were simulated as balls and strings of a pendulum and coordinates were assigned by a gravity-driven algorithm. Furthermore, adjustments to fit a horizontal grid were given. Multi-layered lines were drawn by introducing line rows between ranks.

A fast non-iterative heuristic was explained in [Buchheim et al. 2001]: The dummy nodes of each long edge were grouped and leftmost and rightmost top-to-bottom placements were considered to gain a kind of funnel. The dummy vertices were assigned to the mean of the left and right positions obtained from the grouping. Without changing the positions of the dummy vertices, the other vertices were placed to minimize the length of some edges. A similar method can be used to implement `minEdge` in algorithm 4.18.

For hierarchical graphs, in [Eades and Sugiyama 1990] another algorithm based on [Tutte 1963] was given. After initial coordinates for each nonboundary node were chosen, coordinates were assigned according to the following formula

$$x(u) = \frac{1}{2 \cdot d^-(u)} \sum_{v \in V^-(u)} x(v) + \frac{1}{2 \cdot d^+(u)} \sum_{w \in V^+(u)} x(w)$$

and the assignment was repeated until the horizontal positions converge. Variants to consider `GDR_SYMMETRY` were also discussed.



A fast and simple horizontal coordinates assignment algorithm, which runs in  $O(|V|\log^2|V|)$ , was given in [Brandes and Köpf 2002]. In a vertical alignment step, all vertices were positioned next to the median of either the upper or lower neighbors. Thereby, certain conflicts arising from edge crossings or sharing vertices were detected and solved. Then, the horizontal compaction placed all vertices as close as possible to the next vertex in preferred direction. Each of both steps was carried out four times. Finally, a balancing step tried to compensate horizontal and vertical tendencies induced by the preceding steps.

An extension to the Sugiyama algorithm to handle compounds for the visualization of biochemical pathways was given in [Schreiber 2002; Brandenburg et al. 2003]. Further discussions on hierarchical node positioning algorithms can be found in [Battista et al. 1999; Bastert and Matuszewski 2001; Brandes and Köpf 2002].

## 4.8.2 Basics

Basic research is what I am doing when I don't know what I am doing.

Wernher von Braun (1912 – 1977)

For UML class diagrams, nodes and edges cannot be simplified to points or lines as usual in abstract graph drawing. Different types of nodes require various shapes, edges may have graphical and textual adornments. Therefore, in general, nodes and edges must be able to express the minimum required area. On the one side, as discussed in Section 4.2, in our graph model nodes and edges do not know this application domain specific information. Specialized information objects determine the shape, and, therefore, the minimum area requirements. On the other side, these information objects do not know the edges attached to the node and therefore are not able to handle additional area to place graphical adornments.

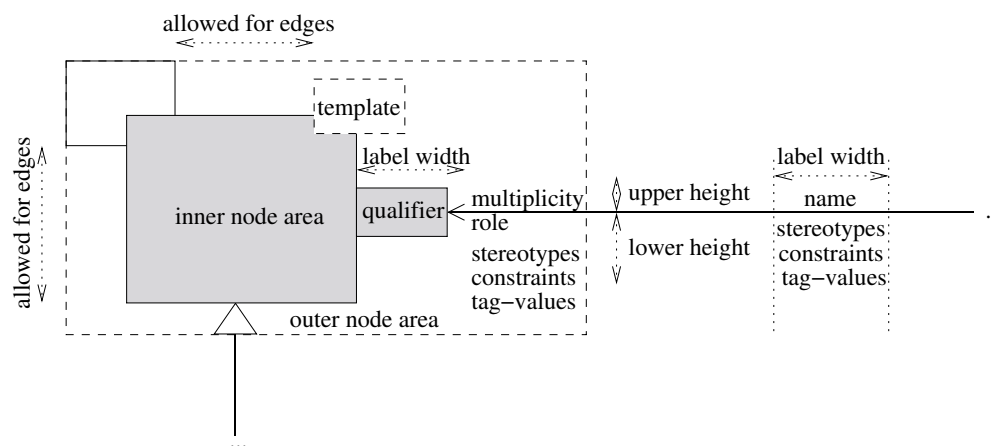


Figure 4.29: Area requirements for nodes and edges. Edge extents are reinterpreted according to the side at which the edge connects to the node.

Figure 4.29 shows the basic areas for nodes and edges. A node consists of an inner area, which

is primarily determined by the information object. For example, the area of UML classes is calculated from the information on compartments and the basic extents of an n-ary rhomb are calculated from default values and default font sizes (UML\_NODES and UML\_CLASS).

The simple separation of a node into an inner node area, the information object is exclusively responsible for, and an outer node area, the node is responsible for, becomes complicated, when non-rectangular shapes are required. In Figure 4.29, a template is shown which partly belongs to both areas. A concrete handling depends on the implementation of the information object, but it becomes obvious that further information has to be provided for the general case. Restrictions and hints, where edges may be attached to, are required. As reflective edges are handled internally by their connected nodes as a kind of “private” edges, they are not part of the area which is reserved for “public” edges.

On the one side, the area of a node might be minimized to save drawing space (GDR\_DRAWING\_SIZE). On the other side, also nodes with less area might be highly connected and the adornments to the edges might not be clearly visible in that case. Therefore, the nodes or edges may be internally scaled to meet either UML\_GRAPHDRAWING via GDR\_DRAWING\_SIZE or UML\_ADORNMENTS and UML\_REFLECTIVE. Another mechanism may also gain influence on the inner node area: When the implementation of the layout algorithm should be used as a layout plug-in to other (graphical) tools providing their own editors, the sizes and relative positions may be externally predefined. This was also mentioned in Section 4.2 when the information object life cycle was introduced.

Similar arguments have to be considered for edges as well. The basic area requirement, separated for start and end node, can be calculated from the attached adornments. This information can directly be taken into account when calculating the outer area of the nodes. By incorporating this information and ensuring that nodes do not overlap, we implicitly can realize UML\_ADORNMENTS. Other adornments neither attached to start nor to end node but directly to the edge itself like a discriminator, an association name, stereotypes, constraints or tag-value lists also have to be taken into account when determining the minimum distance between two nodes. The first implementation of *SugiBib* only provided the global minimum distance between two adjacent nodes via  $nodeSep(G)$ . Thereby, the area of the edge label was simply merged into the outer node area of the start or the end node to ensure its area requirements. The current implementation provides a node-individual distance mechanism which can take this edge specific area into account.

As described in [Gansner et al. 1993], the algorithm will respect a minimum rank separation value as well as a minimum node distance for adjacent ranks or nodes, respectively. But to be more general, we will operate on node-individual distance function  $nodeSep(\bar{G}, \sigma_r[i], \sigma_r[i+1])$  for adjacent nodes  $\sigma_r[i]$  and  $\sigma_r[i+1]$ . This function will also be used to provide a cluster-individual distance between the cluster border and the contained nodes. Implicitly, individual distances for cluster separator nodes can be realized using that function. Similarly, a rank-individual distance function  $rankSep(\bar{G}, \sigma_r, \sigma_{r+1})$  can be defined. As initialization we can assume the functions from [Gansner et al. 1993] as default values, so that initially  $nodeSep(\bar{G}, \sigma_r[i], \sigma_r[i+1]) := nodeSep(\bar{G})$  and  $rankSep(\bar{G}, \sigma_r, \sigma_{r+1}) := rankSep(\bar{G})$  for all adjacent nodes or ranks, respectively.

As input to the coordinates assignment, a layered, cluster-valid graph  $G$  according to definition 9 and definition 13 is required. The sequence of nodes in the individual ranks is maintained while coordinates assignment except for hidden nodes, which might be repositioned to obtain a better drawing.

Depending on the presence of clusters in the input graph, the algorithm will produce a coordinates or a cluster and coordinates valid graph. To describe these properties, we will introduce basic operations on nodes, cluster border chains and cluster as well as coordinates validity in the next definitions.

### Definition 22 (coordinates operations on nodes and edges)

Let  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  be a graph and  $u, v \in V$  be nodes of  $G$ . The following operations are defined:

- $x(v) \rightarrow \mathbb{Z}$  and  $y(v) \rightarrow \mathbb{Z}$  denote the central position of the node  $v$ ,  $width(v) \rightarrow \mathbb{N}_0$  and  $height(v) \rightarrow \mathbb{N}_0$  the extents. Let  $left(v) := \lfloor x(v) - \frac{1}{2} \cdot width(v) \rfloor$  be the left position of  $v$ .  $right(v)$ ,  $top(v)$  and  $bottom(v)$  are defined similarly.
- $topPorts(v) := in(v) \cap E_H$  is the set of edges (ports) at the upper horizontal side of  $v$  and  $bottomPorts(v) := out(v) \cap E_H$  is the set of ports at the lower horizontal side.  $leftPorts(v) \rightarrow edges(v) \cap E_H$  and  $rightPorts(v) \rightarrow edges(v) \cap E_H$  with  $leftPorts(v) \cap rightPorts(v) = \emptyset$  denote the ports at the vertical sides of  $v$ .
- Let  $\vec{e} = (u, v) \in E_H \cup E_N$  be an edge of  $G$ . Then  $startX(\vec{e}) \rightarrow \mathbb{Z}$  is the horizontal position of  $\vec{e}$  at  $u$ .  $startY(\vec{e} = (u, v))$ ,  $endX(\vec{e} = (u, v))$ ,  $endY(\vec{e} = (u, v))$  are defined similarly.

We did not use  $\mathbb{N}$  or  $\mathbb{N}_0$  in definition 22, because some operations may temporarily produce negative coordinates. According to definition 4, an edge can be seen as directed or undirected dependent on the context the edge is used in. Because all underlying edges are directed, we can assume that the coordinates operations on edges are similarly defined on undirected edges, too.

### Definition 23 (coordinates-valid graph)

Let  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy.  $\tilde{G}$  is called coordinates-valid, if

- $\forall_{1 \leq r < n} \forall_{0 \leq i < |\sigma_r|} left(\sigma_r[i]) - right(\sigma_r[i-1]) \geq nodeSep(\tilde{G}, \sigma_r[i-1], \sigma_r[i])$
- $\forall_{1 \leq r < n} \min_{v \in \sigma_r} y(v) - \max_{v \in \sigma_{r-1}} y(v) \geq rankSep(\tilde{G}, \sigma_{r-1}, \sigma_r)$  for global rank assignment
- $\forall_{\vec{e}=(u,v) \in E_H} hValid(u, \vec{e}, startX(\vec{e}), startY(\vec{e})) \wedge hValid(v, \vec{e}, endX(\vec{e}), endY(\vec{e}))$   
with

$$hValid(u, \vec{e}, x_e, y_e) := left(u) \leq x_e \leq right(u) \wedge ((\vec{e} \in in(u) \wedge y_e = top(u)) \vee (\vec{e} \in out(u) \wedge y_e = bottom(u)))$$

- $\forall_{\vec{e}=(u,v) \in E_N} nValid(u, startX(\vec{e}), startY(\vec{e})) \wedge nValid(v, endX(\vec{e}), endY(\vec{e}))$   
with

$$nValid(u, x_e, y_e) := top(u) \leq y_e \leq bottom(u) \wedge (x_e = left(u) \vee x_e = right(u))$$

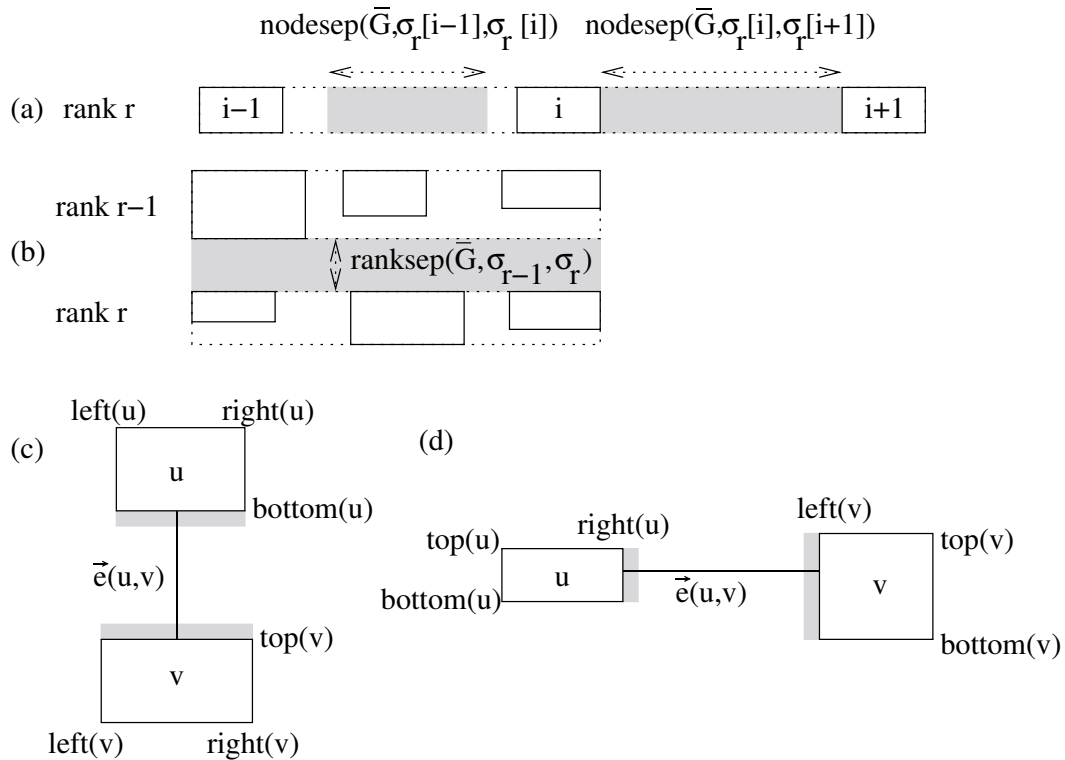


Figure 4.30: The four rules of coordinates validity in definition 23: (a) minimum distance between nodes in one rank, (b) minimum distance between two ranks and restricted positions for hierarchical (c) and non-hierarchical (d) edges.

The rules of coordinates-validity given by definition 23 are illustrated in Figure 4.30.

We do not go into more details here, because we did not define the inner and outer positions of a node. The formulae in definition 23 can easily be adapted to also respect valid edge areas, etc.

If  $G$  is a compound graph, the coordinates assignment will produce a (temporarily) cluster-augmented graph by inserting cluster separator nodes. Hence, further restrictions have to be fulfilled to produce a coordinates and cluster-valid graph.

As described in Section 4.2, each cluster is bounded vertically by a chain of cluster border nodes. These chains have the following properties:

**Definition 24 (anchored cluster border nodes and cluster border chain)**

Let  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  be a graph. The upper endpoint of the left chain attached to  $v$  will be denoted by  $topLeftBorder(v) \in V \setminus \{v\} \cup \{\perp\}$ .  $bottomLeftBorder(v)$ ,  $topRightBorder(v)$ ,  $bottomRightBorder(v)$  are defined similarly.

Let  $v \in V$  be a cluster parent so that  $|compoundChildren(v)| > 0$ .  $chain^C(v, v_s, v_e) := v_0 \dots v_k$  with  $k \in \mathbb{N}_0$ ,  $v_0 = v_s$ ,  $v_k = v_e$  is called a cluster border chain if

$$\begin{aligned} \forall_{0 \leq i \leq k} type(v_i) &= CLUSTERBORDER \wedge compoundParents(v_i) = \{v\} \wedge \\ \forall_{0 \leq i \leq k-1} type(e = \{v_i, v_{i+1}\}) &= CLUSTERBORDER \end{aligned}$$

A similar definition can be used to introduce chains of cluster separators. We omit a definition, because we do not deal with cluster separators in detail in this thesis.

**Definition 25 (cluster- and coordinates valid hierarchy)**

Let  $\tilde{G} = (V, E_H, E_N, n, \sigma)$  be a  $n$ -level hierarchy.  $\tilde{G}$  is called coordinates and cluster-valid, if

- $\tilde{G}$  is coordinates-valid according to definition 23.
- $\tilde{G}$  is cluster-valid according to definition 13.
- for all cluster base nodes  $\{v : v \in V, |compoundChildren(v)| > 0\}$  the encapsulating cluster border chains  $chain^C(v, topLeftBorder(v), bottomLeftBorder(v)) = u_0, \dots, u_k$  and  $chain^C(v, topRightBorder(v), bottomRightBorder(v)) = w_0, \dots, w_k$  exist and
  - $\forall_{0 \leq i \leq k-1} x(u_i) = x(u_{i+1})$
  - $\forall_{0 \leq i \leq k-1} x(w_i) = x(w_{i+1})$
  - $x(v) = \lfloor \frac{1}{2} \cdot (x(u_0) + x(w_0)) \rfloor$  as a convention
  - $\forall_{w \in llc(v) \setminus \{u_0, \dots, u_k, w_0, \dots, w_k\}} x(u_i) + \delta_l(v) < x(w) < x(u_i) - \delta_r(v)$ , where  $\delta_l(v)$  and  $\delta_r(v)$  denote the left or right inner distance between the cluster border and the contained nodes, respectively.

Similar constraints must hold when cluster separators are used.

The rules of coordinates-validity given by definition 25 are illustrated in Figure 4.31.

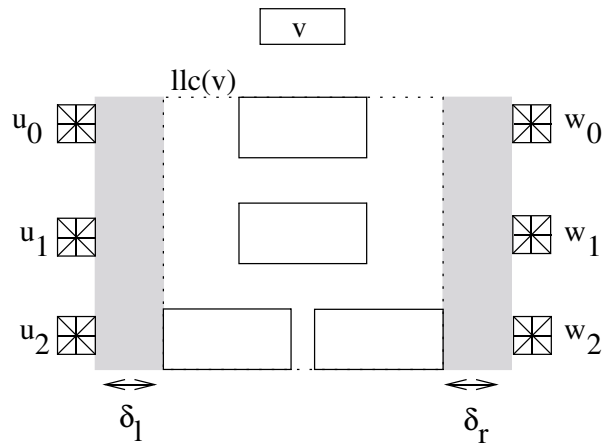


Figure 4.31: Additional rules for cluster validity introduced by definition 25.

### 4.8.3 The Coordinates Assignment Algorithm

There is no procedure for learning to write. What you must do, is learn to think.

S. Leonard Rubenstein

As mentioned in Section 4.8.1, we decided to adapt algorithm 4.18 which was implemented in the first version of *SugiBib*. In this section, the core iteration of the coordinates assignment algorithm will be discussed. Tasks to be executed before, while or after the main loop will be described in the next sections.

Regarding the types of graphs acting as input to the coordinates assignment, it is not required that every input graph is a compound graph. Class diagrams as that in Figure 4.9 belong to the class of graphs, which, as discussed in Section 2.2.3, is currently handled by most of the layout algorithms for UML class diagrams. On the one side, such a graph might be mapped into a compound graph containing the global package as invisible cluster. Therefore, only a coordinates algorithm which is able to handle clusters is required. On the other side, such graphs as well as *mixed compound graphs*, which also have classes in the (invisible) global cluster, can be handled by an integrated approach. Hence, we prefer a coordinates algorithm which implicitly is capable of taking non-compound, compound and mixed compound graphs as input.

**Algorithm 4.19** coordinatesAssignment**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$ **output:**  $\bar{G}$  $\bar{G} := c\_preprocessing(\bar{G})$  $\bar{G}_{best} := \bar{G}$ **for**  $i := 0$  **to**  $max\_iterations$  **do** $\bar{G} := c\_iteration(\bar{G}, i)$ **if**  $isBetter(\bar{G}, \bar{G}_{best})$  **then** $\bar{G}_{best} := \bar{G}$ **end if****end for****return**  $c\_postprocessing(\bar{G}_{best})$ 

In principle, Algorithm 4.19 is a simple transcription of algorithm 4.18 providing generic hot spots for preprocessing, iteration and postprocessing. While all steps deal with the concrete positions of nodes, only the latter one will determine the coordinates of individual edges.

As in [Gansner et al. 1993], we define

$$isBetter(\bar{G}, \bar{G}_{best}) := xLength(\bar{G}) < xLength(\bar{G}_{best})$$

and

$$xLength(\bar{G}) := \sum_{e=\{u,v\} \in xEdges(\bar{G})} \Omega(e) \cdot w(e) \cdot |x(u) - x(v)|$$

with

$$xEdges(\bar{G}) := \{e : e = \{u, v\} \in E_H, u \neq virtualRoot(\bar{G})\} \cup \\ \{e : e = \{u, v\} \in E_N, r(u) \neq r(v)\}$$

to consider GDR\_MIN\_EDGES in general and to prevent negative impacts of a virtual root. Flat edges are excluded because certain specialized functions will keep them close to their connected nodes.  $\Omega$  was chosen as follows: 8 is returned for chain segments, 1 for edges connecting visible nodes and 2 for all other edges.

In the next sections, we will describe the most important algorithms for preprocessing, iterating and postprocessing the coordinates assignment in detail. Algorithmic descriptions will be given for the most relevant parts but not for steps introduced for some speed improvement or pure technical parts.

### 4.8.4 Coordinates Preprocessing

Life is just one damned thing after another.

Elbert Hubbard (1856 – 1915)

Before assigning coordinates to nodes and edges, various preprocessing steps must be executed:

---

#### Algorithm 4.20 *c\_preprocessing*

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$

**output:**  $\bar{G}$

```

 $\bar{G} := initializeGraphDistances(\bar{G})$ 
 $\bar{G} := insertClusterBorderNodes(\bar{G})$ 
 $\bar{G} := calculateSizes(\bar{G})$ 
 $\bar{G} := initializeXYCoordinates(\bar{G})$ 
 $\bar{G} := sortPorts(\bar{G})$ 
 $\bar{G} := determineStretchingFactor(\bar{G})$ 
 $\bar{G} := initializeNodePriorities(\bar{G})$ 
return  $fixGraphDistances(\bar{G})$ 

```

---

#### Initialize Graph Distances

`initializeGraphDistances` prepares the node and rank individual distance functions as described in Section 4.8.2. More specialized values for certain nodes, e.g., cluster separators are implicitly specified while executing algorithm 4.20.

#### Cluster Border and Cluster Separator Nodes

If not done at the end of the rank assignment due to the lower quality option for larger graphs, chains of border nodes are inserted at each side of a compound (`UML_SEMANTIC_CLUSTERS`). Furthermore, chains of cluster separator nodes to realize aspects of `UML_SPATIAL` or `UML_COUPLING` can also be inserted. This step was described in Section 4.5.5.

#### Calculate Sizes of Nodes and Edges

This step calculates the minimum area required required by the individual nodes and edges of the graph. This is done by sending an appropriate request to the individual information objects. For nodes, the information object determines the inner node area as depicted in Section 4.8.2. The outer node area is calculated by considering the edges connected to a node. The sizes of all nodes or a subset of the nodes might be initialized taking external information into account, e.g., when the algorithm is used as a layout plug-in or for incremental layout.

This step is directly responsible for `UML_NODES`, `UML_CLASS`, `UML_CONTAINER` (the tab or the signatures in the interface compartment of UML subsystems) and `UML_ADORNMENTS`.



## Calculate the Initial Cluster-Valid Coordinates Assignment

The following part of the coordinates assignment will move individual nodes based on initial coordinates always considering the minimum distance restrictions represented by the graph distances. Therefore, an initial coordinates assignment must admit non-overlapping nodes (UML\_NODES) with proper distances between the nodes and the ranks. Furthermore, in the case of a compound or mixed compound graph, the assignment must also be cluster valid to realize UML\_SEMANTIC\_CLUSTERS.

The basic coordinates initialization for non-cluster graphs was given in algorithm 4.17. According to the examples in [Seemann 1997], the nodes in the first rank can be aligned to the bottom side, the others to the top side. Furthermore, to respect the outer node area, the nodes should always be aligned to the top or bottom inner area, which marks the visible part of the nodes.

The initialization might be done similarly to the segment ordering graph mentioned in [Sander 1996a]. We have implemented a scanline algorithm for the coordinates initialization of compound and mixed compound graphs, because we also want to directly process mixed compound graphs.

First, we run the coordinates initialization for non-compound graphs to obtain the vertical coordinates assignment as described above. Algorithm 4.21 exclusively modifies the horizontal positions to determine cluster-valid positions. Thereby, `InitXYConstraint` represents a (nested) cluster and realizes a simple finite automaton for arbitrary partitioned clusters. An usual cluster has at least three states: border nodes, cluster contents and cluster completed. Further states for representing partitions, e.g., for UML subsystems, can be added dynamically depending on the type of the node. While assigning basic cluster-valid coordinates, aspects of UML\_SPATIAL can be considered by adjusting the node and rank individual distance functions.

First a stack, that stores the currently nested clusters, is initialized with the global cluster from the node naming function as top element. Then, the scanline and the concrete leftmost positions are initialized. As long as not all nodes are processed, the ranks are considered from bottom to top searching for nodes which match the (nested) cluster  $top(stack)$  and the current state of the cluster automata at the top of the nesting stack. If the current node  $v$  is the cluster parent of  $top(stack)$  and all nodes contained in  $top(stack)$  have been processed,  $v$  is moved to the median position above all contained nodes according to the convention in definition 25. Otherwise, if  $v$  is an usual node, it is placed in the current leftmost position. Then, the leftmost position of the current rank is set to the right position of  $v$  and the scanline array is adjusted.

After considering all ranks, the next cluster is identified. Therefore, all ranks are processed starting at  $\sigma_{n-1}$  and searching for the next appropriate cluster with respect to the scanline. If currently a cluster is processed and a new contained cluster was found, the leftmost positions are set to the maximum and the new one is pushed onto the stack. If no further nested cluster was found and the current cluster automata is in its final state, the rightmost cluster border nodes are placed to fulfill definition 25. Additional steps for cluster separator nodes are not shown in algorithm 4.21. The algorithm does not necessarily produce an area efficient assignment, but this the task of the iterative part of the coordinates assignment.

---

**Algorithm 4.21** initializeClusterX (as a part of initializeXYCoordinates)
 

---

**input:**  $\bar{G}^C = (V, E_H, E_N, n, \sigma)$ 
**output:**  $\bar{G}^C$ 

```

stack := new Stack
push(stack, new InitXYConstraint(g)) {global cluster}
 $\forall_{0 \leq r < n} \text{scanpos}[r] = \text{xpos}[r] = 0$ 
repeat
  for  $r := n - 1$  downto 0 do
    if  $\text{scanpos}[r] < 0$  then
      continue
    end if
    for  $i := \text{scanpos}[r]$  to  $|\sigma_r| - 1$  do
       $v := \sigma_r[\text{scanpos}[r]]$ 
      if  $\text{top}(\text{stack}) =_N v$  then
        if matchesState(top(stack), v) then
          break
        end if
        if  $v = \text{topNode}(\text{top}(\text{stack})) \wedge \forall_{w \in \text{llc}(v)} \text{processed}(w)$  then
           $x^* := \max_{w \in \text{llc}(v)} x(w)$ 
           $x(v) := \lfloor \frac{1}{2} \cdot (x_r + \min_{w \in \text{llc}(v)} x(w)) \rfloor$  {according to definition 25}
        else
           $\text{left}(v) := \text{xpos}[r] + \text{nodeSep}(\bar{G}^C, \sigma_r[\text{scanpos}[r] - 1], v)$ 
           $x^* := \text{right}(v)$ 
        end if
         $\text{xpos}[r] := x^*$ 
        if  $i < |\sigma_r|$  then
           $\text{scanpos}[r] := i$ 
        else
           $\text{scanpos}[r] := -1$ 
        end if
      end if
    end for
  end for
   $v := \text{nextCluster}(\bar{G}, \text{stack}, \text{scanpos})$ 
  if  $|\text{stack}| > 1$  then
    if  $v \neq \perp$  then
      maximize(xpos); push(stack, new InitXYConstraint(v))
    else
      if  $\neg \text{advanceState}(\text{top}(\text{stack}))$  then
        placeClusterBorders( $\bar{G}, \text{scanpos}$ )
        pop(stack)
      end if
    end if
  end if
until  $|\text{stack}| = 0 \vee \forall_{0 \leq i < n} \text{scanpos}[i] < 0$ 
return  $\bar{G}^C$ 

```

---

## Sort Ports

In addition to the initial sorting of the incoming and outgoing edges of a node, now the port sets are ordered. As denoted in algorithm 4.22, the vertical port sequence can simply be obtained by sorting the ports according to the coordinates (or rank positions) of the connected nodes.

---

### Algorithm 4.22 sortPorts

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$

**output:**  $\bar{G}^C$

```

for  $r := 0$  to  $n - 1$  do
  for all  $v \in \sigma_r$  do
     $horizontalSort(e = \{v, u\}, f = \{v, w\}) := x(u) < x(w)$ 
     $topPorts(w) := sort(topPorts(w), horizontalSort(.))$ 
     $bottomPorts(w) := sort(bottomPorts(w), horizontalSort(.))$ 
     $leftPorts(w) := sortVerticalPorts(leftPorts(w), true)$ 
     $rightPorts(w) := sortVerticalPorts(rightPorts(w), false)$ 
  end for
end for
return  $\bar{G}^C$ 

```

---

`sortVerticalPorts` partitions the given vertical port set of node  $v$  and sorts each individual partition:

- Direct connections to a rank with  $r < r(v)$  and at most 2 dummy nodes ( $P_l$ )
- Connections to a rank with  $r < r(v)$  and more than 2 dummy nodes which requires a flat connection in a row between two ranks ( $P_{tm}$ )
- Hyperedge connections to a rank with  $r < r(v)$  ( $P_{th}$ )
- Top flat edges according to definition 17 ( $P_{tf}$ )
- Flat flat edges according to definition 17 ( $P_f$ )
- Flat hyperedge connections ( $P_{fh}$ )

and similar sets at the bottom side. Figure 4.32 depicts examples for most of these sets and shows the intended ordering at the centered node. Finally, the partitions are merged back into the port set.

Sorting ports is responsible for aspects of `GDR_EDGE_CROSS`, `GDR_MIN_EDGES` and `UML_ADORNMENTS` on flat edges. The quality of the port sequences depends on the information provided to the sort algorithm. Obviously, stable coordinates positions are preferable over initial coordinates. Therefore, this step will be executed again later in the postprocessing phase of the coordinates assignment algorithm.

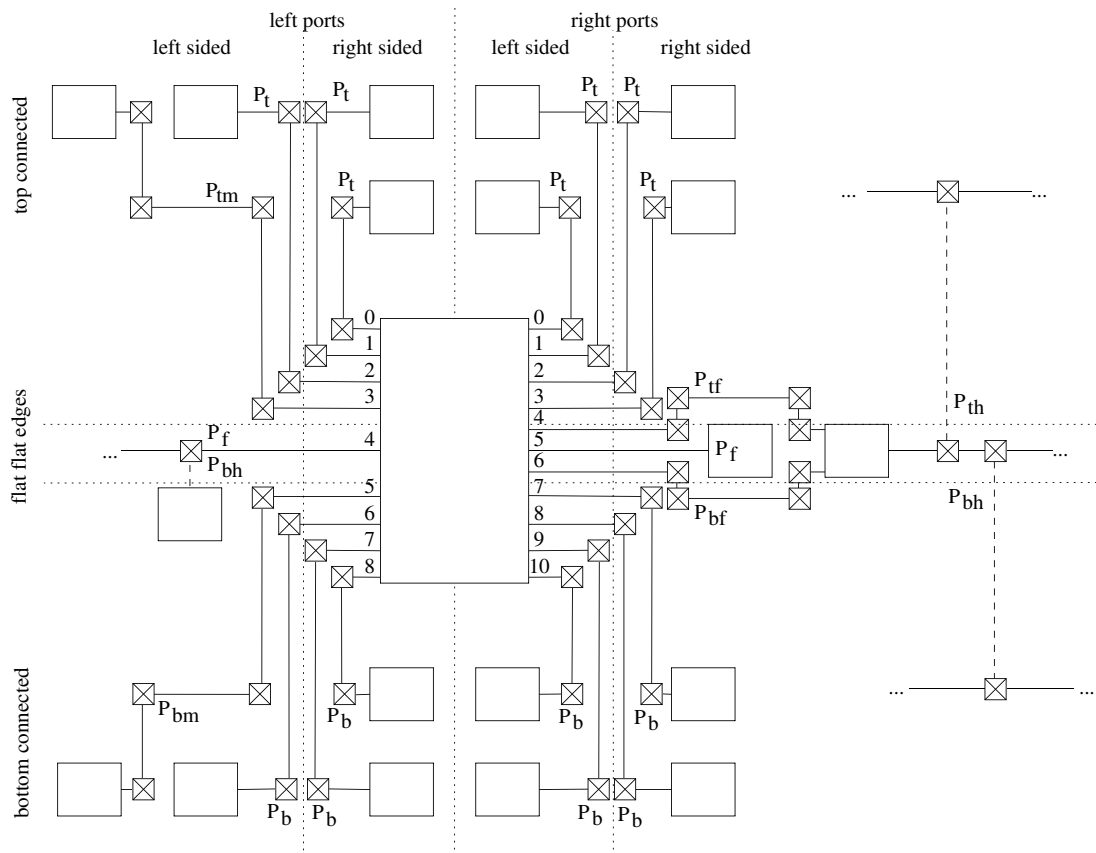


Figure 4.32: The vertical ports of a node are partitioned while port sorting into 10 subsets which are sorted as individual sets according to different criteria and finally merged.

### Determine Stretching Factor

Depending on global preferences, the contents of the graph may be scaled to emphasize certain information. As mentioned in Section 4.8.2, some basic scalings of the entire graph may be considered:

- **Rescale the nodes** to provide a clear visibility of all edge adornments according to `UML_ADORNMENTS` and `UML_REFLECTIVE`.
- **Rescale the edges** to keep minimum node sizes respecting the contents represented by the individual node information objects. It is not required to rescale each graph, because in many cases highly connected nodes also allocate a larger area due to the number of their compartment entries (`SE_COUPLING`). In contradiction to rescaling the nodes this method supports `UML_GRAPHDRAWING` via `GDR_DRAWING_SIZE`, but not always `UML_ADORNMENTS` or `UML_REFLECTIVE`.

- **Scale individual elements**, like font sizes for edges or individual nodes. This would introduce contradictions with `UML_NODES`, `UML_CLASS`, `UML_ADORNMENTS` and `UML_REFLECTIVE`, because we prefer to use general font preferences instead of locally changing ones (similar to `GDR_FONTS`).
- **No scaling**, if external sizes for the nodes are present, e.g., if the implementation is used as a layout plug-in or for incremental layout. In this mode, the framework is not responsible for the sizes.

Depending on the scaling factor, visually unpleasing artifacts may occur, especially when reducing the size of fonts. Below a certain scaling factor, the underlying operating system may not be able to display the fonts properly and even if the sizes have been calculated correctly, overlappings of text labels with other elements of a diagram may occur accidentally.

### Initialize Node Priorities

In `initializeNodePriorities`, the sequence of the nodes to be considered in the iterative positioning according to `UML_MEDIAN` is determined. Priorities are calculated and assigned for `medianpos` and `minedge` mentioned along with algorithm 4.18. Two kinds of priorities, upward and downward priorities, are considered in `medianpos` or `minedge`, dependent on the sweep direction, in which the individual ranks are processed. For `medianpos`, the number of visible edges not connected to the virtual root is relevant. For `minedge`, only edges between visible (not dummy) nodes is considered. Hidden nodes may receive the maximum priority to support straightness of edge chains as mentioned in [Sugiyama et al. 1981].

Along with algorithm 4.5, the invisible cluster dependencies were discussed. The lengths of the incoming/outgoing cluster dependencies can simply be added to the upward/downward priorities. Without considering cluster dependencies, visibly disconnected clusters would be arranged according to lowest priority, probably somewhere far away from the other nodes. Disconnected clusters without (invisible) dependencies can be handled by considering artificial dependencies to their cluster parent or the virtual root.

### Fix Graph Distances

Finally, the individual distance values are fixed in `fixGraphDistances`. In fact, initializing and fixing the graph distances has major impact on `UML_GRAPHDRAWING` (via `GDR_DENSITY` and `GDR_DRAWING_SIZE`) as well as on overlappings of `UML_NODES`, `UML_SPATIAL` and the realization of `UML_SEMANTIC_CLUSTERS`.

### 4.8.5 Iterative Coordinates Assignment

You don't just luck into things as much as you'd like to think you do. You build step by step, whether it's friendships or opportunities.

Barbara Bush

According to the experience described in [Gansner et al. 1993], some iterations of mutually contradictory heuristics lead to an appropriate coordinates assignment. Some heuristics may alternate the direction of processing the ranks per iteration. Therefore, Algorithm 4.23 also receives the current number of iteration.

---

#### Algorithm 4.23 *c\_iteration*

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), i \in \mathbb{N}_0$

**output:**  $\bar{G}$

$\bar{G} := \text{medianPos}(\bar{G}, i)$

$\bar{G} := \text{minEdge}(\bar{G}, i)$

$\bar{G} := \text{minNode}(\bar{G}, i)$

$\bar{G} := \text{minPath}(\bar{G}, i)$

$\bar{G} := \text{packcut}(\bar{G}, i)$

**return**  $\text{restrictVirtualRoot}(\bar{G})$

---

The heuristics *medianpos*, *minedge*, *minNode*, *minPath* and *packcut* are based on the description given in [Gansner et al. 1993]. All heuristics were modified to support cluster validity. Unfortunately, a virtual root, inserted in the preprocessing of the rank assignment S10, may appear as a dangling node, which has to be restricted to avoid unnecessary stretching of the width of the graph.

#### Moving Nodes

An important operation of the coordinates assignment is to move an individual node. In principle, a node  $v$  can simply be repositioned by  $x(v) := x(v) + \delta$ , but this does neither consider the coordinates validity given in definition 23 nor the cluster validity in definition 25. When moving a node, the extents and the type of the neighbor in direction of the move have to be respected. Figure 4.33 depicts the four basic situations for leftwards moves. The opposite direction can be handled similarly.

If a neighbor in Figure 4.33 (a) exists,

$$\text{left}(w) \geq \text{right}(v) + \text{nodeSep}(\bar{G}, v, w)$$

must hold according to definition 23. When moving a cluster base node, as depicted in Figure 4.33 (b), the entire cluster is moved, if no cluster border node collides with its neighbor. When a cluster border node should be moved, the connected border nodes must not collide with

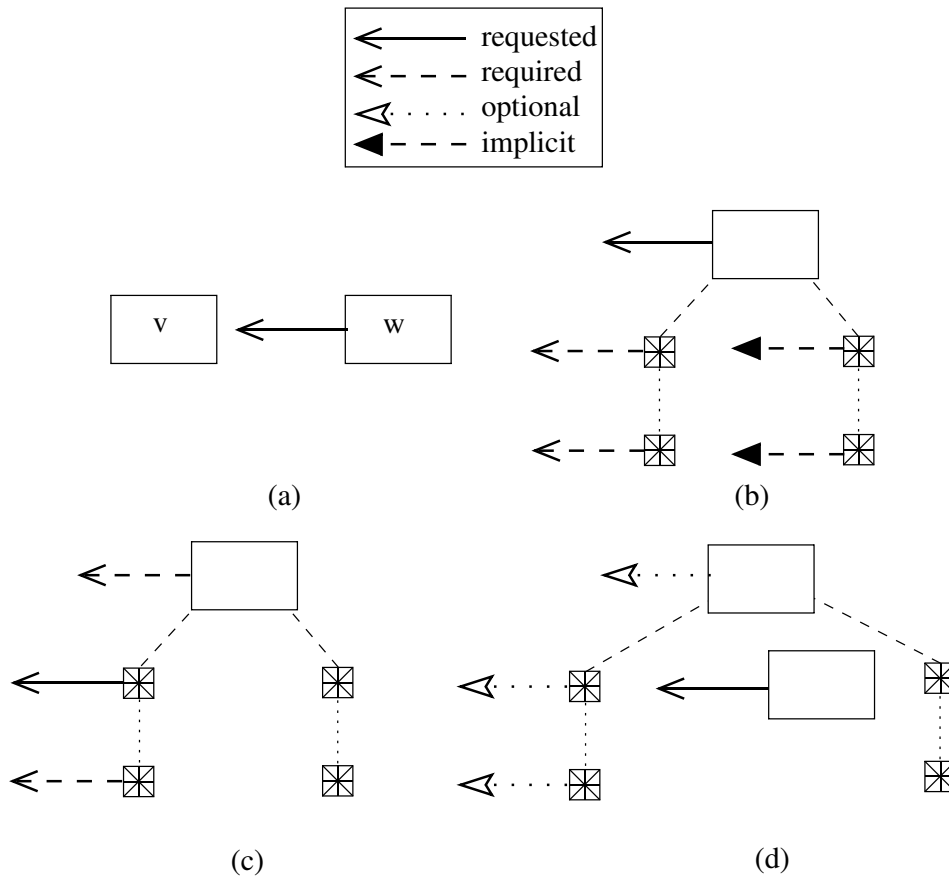


Figure 4.33: Situations to be respected when moving a node: (a) an usual node  $v$  is moved with respect to neighbor  $w$ , (b) the cluster parent as well as the entire cluster is moved, (c) a cluster border node is moved and the cluster is enlarged or (d) a contained node is moved and optionally the cluster may be enlarged.

any of their neighbors. Thereby, the cluster will be enlarged and, according to definition 25, the cluster base node has to be kept at the median position above the contained nodes. Finally, if a node inside a cluster should be moved and the neighbor is not a cluster border node, the situation is similar to Figure 4.33 (a). Otherwise, if the node would collide with a cluster border node, the cluster might optionally be enlarged by also considering the appropriate cluster border nodes and the cluster base for a move.

We will call this operation  $shiftRelXPos(\bar{G}, v, \delta)$ , which returns, if the move was executed successfully.

As an alternative, if moving a node is not possible due to a collision with neighbors, the node may be moved as close as possible to its neighbor. Considering similar cases as discussed for  $shiftRelXPos$  above,  $shiftCloseToNeighbor(\bar{G}, v, \delta)$  accomplishes this task, whereby  $\delta$  is relevant for the direction only.

Both node positioning functions are responsible for ensuring the basic proper-

ties of UML\_NODES. If clusters are present, the cluster validity invariant realizes UML\_SEMANTIC\_CLUSTERS. Furthermore, aspects of UML\_SPATIAL, which can be specified via the node individual distance function and UML\_COUPLING are guaranteed.

### Hidden Nodes Hopping

As discussed in Section 4.6.5, the edge crossing reduction may place dummy nodes in visually displeasing positions next to a cluster parent node. Furthermore, dependent on the concrete shape of a cluster, several restrictions for positions of visible nodes may exist. One of these restrictions is that the cluster base node appears to have the same width than the area occupied by its contained node to prevent visible nodes from running into the cluster. In the case of dummy nodes, this is the reason for displeasing edge bends as shown in Figure 4.34 (a).

According to the definition of clusters and nested relations, these dummy nodes cannot be member of the cluster, because they are located in the same rank as the cluster parent. In particular for this situation, but also for other dummy nodes, the node naming function is not able to automatically detect that these hidden nodes might be “member” of the cluster. Also the edge crossing reduction in S11 cannot be made responsible for this problem, because both drawings imply the same number of crossings and even edge length calculations, as discussed in Section 4.6.5, might not always solve the problem.

Therefore, the coordinates assignment may be allowed to support *hidden nodes hopping*. To get rid of the left sided nodes, the sequence of nodes in their ranks can be changed dynamically. The nodes at the right side may then be allowed to move into the area of the cluster. Therefore, the basic functionality of `shiftRelXPos` and `shiftCloseToNeighbor` can be extended to adjust the node sequences dependent on the desired positions of hidden nodes. More generally, to also support other application domains, hidden nodes hopping must consider the visual shape of the cluster. Nodes hopping may also be applied to selected visible nodes.

Node hopping helps reducing the length of certain edges (GDR\_MIN\_EDGES) and, therefore, may help emphasizing UML\_HIERARCHY via UML\_MEDIAN.

### MedianPos and MinEdge

As mentioned in Section 4.8.1, two heuristics called `medianpos` and `minedge` are executed at the beginning of the iterative assignment. Both heuristics support UML\_MEDIAN. To respect influences arising from parent levels and child levels to an individual node and to prevent an extremely widespread layout, the direction of processing the levels of the input graph is alternated per iteration. The basic heuristics as described in [Gansner et al. 1993] differ only in the handling of dummy nodes: `minedge` does not consider edge chains. When processing a rank, the nodes are sorted according to the priorities calculated by `initializeNodePriorities` in the coordinates preprocessing. The nodes are processed according to decreasing priorities. For each node  $v$ , the horizontal distance to the desired median position is calculated. In the case of upward processing  $V^-(v)$ , while downward processing  $V^+(v)$  are considered. Thereby, the distance to the connected (visible) node is considered to prevent a widespread layout, which might occur, if these nodes are not moved at all or accidentally moved in the wrong direction. Furthermore,



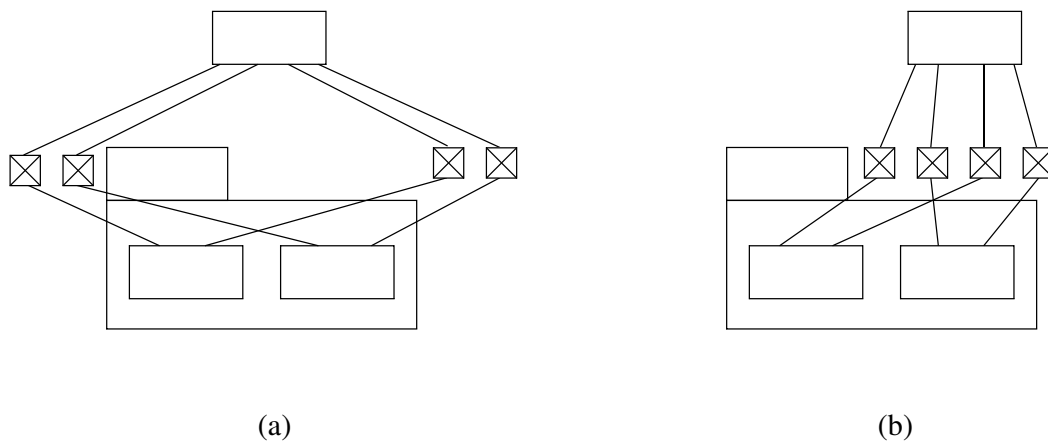


Figure 4.34: (a) conventional coordinates assignment, which respects strict borders of clusters even in the rank of the cluster base node. In (b), hidden nodes hopping was enabled.

invisible cluster dependencies are respected.

As mentioned above, in *minedge*, edge chains are ignored. Furthermore, the concrete implementation in *SugiBib* provides template methods to exclude certain edges depending on *minedge*. For example, *nNon-hierarchical* edges, edges to the virtual root and hidden edges are also excluded for UML class diagrams. If at least one node is not excluded by the filtering, the symmetrical median determines the desired horizontal distance  $v$ . Otherwise  $v$  is regarded as disconnected and aligned with respect to maximum number of flat connected neighbors.

#### Local corrections at nodes: *minNode*

Unfortunately, not all situations to reach *UML\_MEDIAN* are considered by the simple but effective combination of *medianpos* and *minedge*. Therefore, in [Gansner et al. 1993], local optimizations at nodes were used to perform additional corrections.

*minNode* also processes all ranks in alternating directions. On each node, which is not considered as a hidden node, i.e., it is not a dummy node and in UML class diagrams it is no hyperedge connection point, the following heuristics have found to be effective:

- **Correct parent nodes:** If the current node  $v$  has more than one bottom port,  $v$  should be replaced to the median position of its children. If  $v$  is part of a cluster, the cluster is considered for a movement by calling *shiftRelXPos* or *shiftCloseToNeighbor* on the top level cluster parent.

If  $v$  has exactly one bottom port to  $w$  and  $w$  also has this connection as top port, we simply align the nodes according to their central position. In fact, we consider the estimated position of the edge instead of the central position, because the edge position depends on application domain specific information and might not always be the central position of the node. If  $w$  has multiple top ports we consider an alignment only, if  $w$  has no top ports to hidden nodes. This condition is required to prevent accidental repositioning of edge chains.

- **Move to center:** To partly realize UML\_CENTER, certain nodes like an n-ary rhomb can be centered upon its connected children.
- **Correct children:** In this heuristic, the children are considered for a move. The minimum horizontal area for the children is calculated and then the child nodes are repositioned. Instead of terminating the loop when a collision is detected by `shiftRelXPos`, also all nodes might be requested to be moved instead.
- **Correct leaves:** Here a leaf is a node having exactly one top connection and no bottom edges. The optimal position of the leaf below its parent is calculated and a replacement is requested by `shiftRelXPos` or `shiftCloseToNeighbor`.

### Emphasizing vertical segments: `minPath`

As a side effect, gaining straight edges improves UML\_GRAPHDRAWING, in particular GDR\_MIN\_EDGES, GDR\_MIN\_BENDS, GDR\_UNIFORM\_LENGTHS and GDR\_DRAWING\_SIZE. Also in this heuristic, the ranks of the graph are processed in alternating directions. In principle, an edge chain is traversed and a corridor of the neighbored nodes is recorded. Depending on the type of the corridor (bounded at both sides, at one side or unbounded), the nodes of the chain are moved to reach the maximum number of correctly aligned subchains.

Furthermore, two different types of edge chains have to be considered. On the one side, chains arising from hierarchical connections and therefore starting and ending with a visible node may occur. For the corridor of such a chain, the available port positions at start and end node have also to be respected. On the other side, multilevel, non-hierarchical edges induce chains starting and ending with hidden nodes. At a first glance, considering this type of chains seems to be superfluous, because non-hierarchical edges have not been processed in coordinates assignment so far. As most of the heuristics for iterative assignment, `minPath` will be reused for recalculating coordinates after orthogonalizing non-hierarchical edges. The algorithm treats chains arising from hierarchical edges with a higher priority.

The shape of the chain, described by the corridors, can be considered to prepare the compaction of the drawing in the next step. If the wrong side (left) is chosen in Figure 4.35 (a), the length of the edge and probably the size of the drawing is increased successively.

### Packcut

Due to the iterative assignment and the alternating alignment to parents and children in the heuristics discussed in this section so far, larger unused or non-compact areas in the graph may occur. Additionally to the technique mentioned in [Gansner et al. 1993], also flat edges and edge chains induced by non-hierarchical edges have to be considered.

First, as long as flat edges can be shortened without increasing the lengths of other flat edges, nodes which are not connected by hierarchical edges are repositioned. For the general compaction we use a secondary data structure, which captures visually connected areas. To receive a

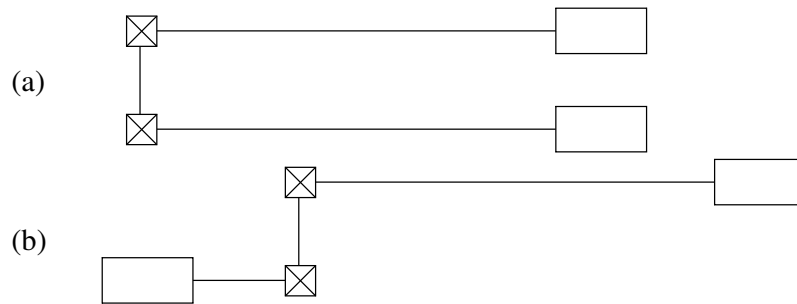


Figure 4.35: (a) U-turn shape which can be shortened, (b) S-turn shape at which one side has to be preferred. Similar situations may occur at hierarchical edge chains, too.

coordinates and cluster valid result, algorithm 4.24 processes each cluster by a depth first traversal of the nesting relations. Thereby, the area required by each cluster is minimized, because clusters may be enlarged, e.g., by moving certain nodes as described along with Figure 4.33. The cluster border nodes are adjusted inside-out and all cluster parents and cluster separators are processed once by `shiftRelXPos`.

Then, all nodes are considered and, thereby, no cluster is distributed to more than one area. For each node  $v$  to be considered by the current call of `packcutAreas`, all existing areas are investigated. The decision, if  $v$  belongs to an area, is made upon the minimum node distance function `nodeSep`. If no area is found, a new area containing only  $v$  is created. If exactly one area is found,  $v$  is added to that area. In the case of more areas,  $v$  induces a visual connection and therefore these areas are merged into one. Finally, the individual areas are sorted by their minimum left position and the distances between the areas are eliminated by moving all nodes contained in an area. .

### Restrict the Virtual Root

If a virtual root  $v_r$  exists, it is most times treated like other dummy nodes: edges connected to  $v_r$  are considered as (hidden) edges and therefore  $v_r$  often appears as a dangling node. On the one side, this situation cannot be perceived directly by a user, because  $v_r$  is not drawn and therefore does not occur in the result. On the other side,  $v_r$  influences the positioning of child nodes while executing the parts of the priority assignment method. Therefore, this processing step repositions  $v_r$  towards the median position of its children.

**Algorithm 4.24** packcutAreas**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), W \subseteq V$ **output:**  $(\bar{G}, A)$ 


---

```

A :=  $\perp$ 
for all  $v \in \{w : w \in W, |\text{compoundChildren}(w)|\} > 0$  do
   $(\bar{G}, A) := \text{packcutAreas}(\bar{G}, \text{compoundChildren}(w))$ 
   $\bar{G} := \text{adjustClusterWidthToMinimum}(\bar{G}, v)$ 
end for
if  $A = \perp$  then
   $A := \{\}$ 
end if
for all  $v \in W$  do
  if  $\text{unmarked}(v)$  then
     $\text{mark}(v)$ 
    for all  $a \in A$  do
      if  $\text{belongsTo}(v, a, W = V)$  then
         $A_v := A_v \cup \{a\}$ 
      end if
    end for
    if  $|A_v| = 0$  then
       $A := A \cup \{\text{new PackcutArea}(v)\}$ 
    else if  $|A_v| = 1$  then
       $\text{add}(\text{listGet}(A_v, 0), v)$ 
    else
       $a_1 := \text{listGet}(A_v, 0)$ 
      for  $i := 1$  to  $|A_v|$  do
         $a_2 := \text{listGet}(A_v, i)$ 
         $\text{merge}(a_1, a_2)$ 
         $A := A \setminus \{a_2\}$ 
      end for
    end if
  end if
end for
 $\text{sort}(A, \text{leftPositions}(\cdot))$ 
for  $i := 1$  to  $|A|$  do
   $\text{move}(\text{listGet}(A, i), \text{left}(\text{listGet}(A, i)) - \text{right}(\text{listGet}(A, i - 1)))$ 
end for
return  $(\bar{G}, A)$ 

```

---

### 4.8.6 Postprocessing

For graphs, we seek paths that are easy to follow and add meaning to the layout.

[Dobkin et al. 1997]

So far, we have calculated the coordinates of the nodes considering to hierarchical edges and clusters only and to realize the high priorities of UML\_HIERARCHY, UML\_SPATIAL, UML\_SEMANTIC\_CLUSTERS, UML\_MEDIAN, UML\_NODES and, if enabled, UML\_COUPLING. As discussed for the iterative coordinates assignment in Section 4.8.5, some heuristics are yet prepared for the layout of flat edges. But at this point of time in the execution of the coordinates assignment, no positions have been assigned to edges at all and flat edges still exist as relations but they have not been routed. Therefore, we have to postprocess the coordinates assignment to derive the missing data.

First, the ports of flat edges are prepared towards orthogonalization by successively inserting dummy nodes. Unfortunately, inserting these dummy nodes induces shifting several nodes including entire clusters and therefore produces an widespread and unbalanced layout which is not compliant to UML\_GRAPHDRAWING (GDR\_MIN\_EDGES, GDR\_DRAWING\_SIZE and GDR\_DENSITY). To circumvent this, we repeat the first two iterations by calling algorithm 4.23 again. Now, the routes of the edges can be determined by assigning individual port positions for each edge. On hierarchical edges, this is a simple task, because they can linearly be arranged at the horizontal sides of the connected nodes (UML\_HIERARCHY). For non-hierarchical edges, rows between ranks and additional dummy nodes may be required. Certain bends may be superfluous, other bends especially at hierarchical edges have to be inserted to avoid node-edge overlappings (UML\_NODES and UML\_EDGES). Due to the iterative assignment, several coordinates may be negative. Therefore, `correctCoordinates` normalizes the graph to the origin of the coordinate system.

---

#### Algorithm 4.25 `c_postprocessing`

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$

**output:**  $\bar{G}$

$\bar{G} := \text{prepareEdgePorts}(\bar{G})$

$\bar{G} := \text{c\_iteration}(\bar{G}, 0)$

$\bar{G} := \text{c\_iteration}(\bar{G}, 1)$

$\bar{G} := \text{routeEdges}(\bar{G})$

$\bar{G} := \text{correctBends}(\bar{G})$

**return**  $\text{correctCoordinates}(\bar{G})$

---

#### Prepare Edge Ports

At this point, the algorithm has calculated stable coordinates respecting hierarchical relations only. According to UML\_HIERARCHY, non-hierarchical edges should be drawn in orthogonal fashion. While preprocessing in algorithm 4.20, the sizes of nodes and edges have been calcu-

lated. Algorithm 4.26 is responsible for inserting dummy nodes for the orthogonalization of the non-hierarchical edges. Due to additional information arising from the coordinates assigned so far, the assignment of edges to the left or right side of connected nodes may change. Therefore, we first have to resize the outer area of the nodes, to initialize and sort the ports again. Then, for each non-hidden node an appropriate number of dummy nodes for the connected non-hierarchical is inserted. `insertFlatDummyNodes` splits all non-hierarchical edges which do not connect directly neighbored visible nodes by inserting two dummy nodes. Thereby, all nodes, which have a larger horizontal position, are moved to the right side to ensure space for the connections to the dummy node. This is illustrated in Figure 4.36 (b). To support rows between ranks, further dummy nodes are inserted at multi-level edges as depicted in Figure 4.36 (c). The second multi-level edge in Figure 4.36 (c) may be handled in the same way, because superfluous bends will be removed in `correctBends`. A similar approach of first successively inserting more bends then necessary and then removing superfluous ones in a postprocessing step was mentioned in [Di Battista et al. 2002]. As long as no stable coordinates are assigned to the newly inserted dummy nodes, they are simply kept in the appropriate ranks and not in line rows. Even these intermediary graphs are coordinates and cluster valid, because the nesting relations are adjusted while inserting the dummy nodes and the coordinates validity in definition 23 includes equal horizontal positions for neighbored nodes.

In a final step, flat edges, which cross visible nodes are routed according to the directional information collected while reducing edge crossings.

For UML class diagrams, hyper edge connection points, association classes and comments have to be treated by additional conditions to avoid that these nodes are handled like visible nodes and edges are wrapped around them.

---

**Algorithm 4.26** `prepareEdgePorts`


---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$

**output:**  $\bar{G}$

**for all**  $v \in V$  **do**

$\Theta := \text{getRelevantValues}(v)$  {outer positions, external area size}

`calculateSize`( $v$ )

`setRelevantValues`( $v, \Theta$ )

**end for**

$\bar{G} := \text{initializeNodePorts}(\bar{G}, \text{false})$

$\bar{G} := \text{sortPorts}(\bar{G})$

$\bar{G} := \text{insertFlatDummyNodes}(\bar{G})$

$\bar{G} := \text{cutEdgesBetweenVirtuals}(\bar{G})$

**return** `wrapFlatCrossingEdges`( $\bar{G}$ )

---

### Route Edges

After executing 2 iterations of algorithm 4.23, which now also consider non-hierarchical edges and the connected dummy nodes, the coordinates of all graph elements have been adjusted to the

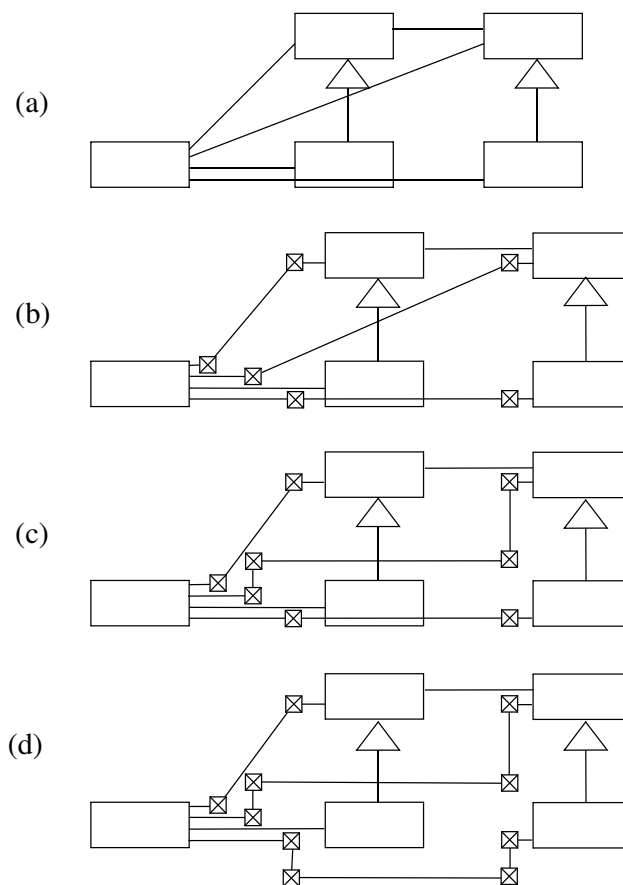


Figure 4.36: Preparation of the orthogonalization of non-hierarchical edges: The initial situation is shown in (a). In (b), the graph after adding two dummy nodes at each non-hierarchical edge is depicted. (c) shows the preparation of rows between ranks and in (d) a bottom flat edge is routed to avoid node-edge overlappings. Non-orthogonal edge segments will be considered by repeated iterations and the non-hierarchical edge routing algorithm.

presence of non-hierarchical edges.

This step assigns positions to each edge in two phases. First, coordinates are assigned to hierarchical edges and flat edges, which are directly connected to visible nodes. Secondly, additional rows between ranks are inserted and positions of the the multi-level non-hierarchical edges as well as the top flat and bottom flat edges are determined.

As mentioned above, the positions of the hierarchical edges can simply be calculated from the inner area of a node with additional consideration to allowed and forbidden edge areas. For UML class diagrams, the sizes of the various adornments must also be considered (UML\_ADORNMENTS). As far as possible, equal distances between ports at one side of a node are desired. Similarly, according to the separation of the flat edges in top, flat and bottom flat edges, the vertical sides can be partitioned per rank. Together with the grouping of edges of the same type in `sortPorts` this spatial separation supports the visual perception of edges of the

same type or (routing) direction, respectively. An illustration of the result produced by the *grid line layout* approach is depicted Figure 4.32 or Figure 4.37.

To layout the edges, which occur between two adjacent ranks  $\sigma_r$  and  $\sigma_{r+1}$ , the area between the ranks is separated for bottom flat edges of  $\sigma_r$ , middle rank edges and top flat edges of  $\sigma_{r+1}$ . This is depicted in Figure 4.37. If a middle rank edge connects  $\sigma_{r_1}$  and  $\sigma_{r_2}$  with  $r_2 - r_1 > 1$ , we currently choose the most simple way: the edge is routed through the middle rank area above  $\sigma_{r_2}$ , crosses  $\sigma_{r_1+1}$  to  $\sigma_{r_2-1}$  and occupies area in the middle rank area below  $\sigma_{r_1+1}$ . Thereby, row lines are shared if possible. A floorplanning algorithm similar to those in VLSI or [Messinger et al. 1991; Hershberger and Snoeyink 1994; Sander 1996a] may be considered for a future optimization of the results.

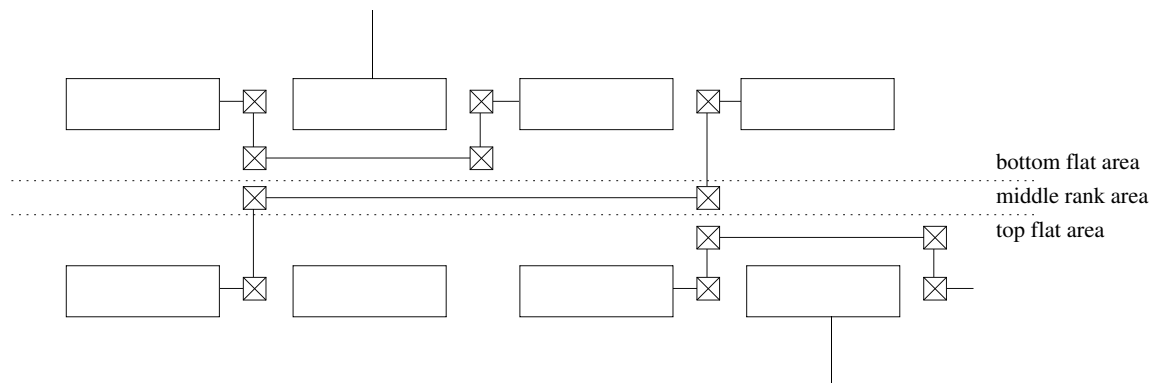


Figure 4.37: Separate areas for edges to be routed between two ranks.

As an extension for UML class diagrams, the offset values to move the ranks can be enlarged to ensure space for edge labels like discriminators, association names, stereotypes, constraints and tag-value lists.

### Correct Bends

Superfluous bends may arise from `prepareEdgePorts` or from erroneous implementation of some heuristics. Some dummy nodes, which connect edge segments of the same gradient, can be deleted from the graph. Furthermore, by comparing the coordinates assigned to the edges, some of the S-turn shapes in hidden chains can also be removed.

### Correct Coordinates

Due to the iterative positioning in Section 4.8.5, graph elements are moved in both horizontal directions and positions may become negative. To properly draw the result on an usual computer display or a printer, negative coordinates should be avoided. Therefore, the minimum left outer horizontal position  $x_l$  is determined and all nodes are moved so that  $x_l = 0$  is valid. If a virtual root or a virtual leaf was inserted, these nodes are not considered when calculating the distances of the nodes to  $x_l$  and are repositioned to the median position of all other nodes. Thereby, also



the central position of certain nodes like an n-ary rhomb according to UML\_CENTER can be realized.

### 4.8.7 Conclusions

I have come to the conclusion that politics are too serious a matter to be left to the politicians.

Charles De Gaulle (1890 – 1970)

In this section, the processing steps of the coordinates assignment have been described. A preprocessing phase initializes the node distance function, determines the minimum extents of nodes and edges respecting a certain scaling of the graph, calculates initial cluster valid coordinates and basically assigns ports. The iterative phase is responsible for realizing the high priorities of the hierarchical aesthetic rules like UML\_HIERARCHY, UML\_SEMANTIC\_CLUSTERS and UML\_MEDIAN. Via the node distance function, which provides individual minimum distances for adjacent nodes, UML\_SPATIAL can be ensured. UML\_COUPLING, as an aspect of UML\_SPATIAL, is implemented by cluster separator nodes. While assigning coordinates to nodes, the basic functions also ensure UML\_NODES to avoid overlappings and cluster invalidity. Via the packcut heuristic, GDR\_DRAWING\_SIZE is taken into account. UML\_CENTER is considered while incremental assignment as well as in the last step of the postprocessing phase. In that last phase, non-hierarchical edges are orthogonalized. Thereby, UML\_EDGES and UML\_ADORNMENTS are considered by the routing mechanism. In particular, UML\_ADORNMENTS may influence the extents of the nodes when (re)scaling due to the minimum area requirements of individual edges and adornments. While routing edges, also GDR\_MIN\_BENDS, GDR\_UNIFORM\_LENGTHS, GDR\_EDGE\_CROSS and GDR\_DRAWING\_SIZE can partially be respected. Hence, the coordinates assignment fulfills the priorities assigned to a subset of our aesthetic rules as claimed in Section 3.3.6.

Table 4.6 shows the runtime complexities of the individual phases of the coordinates assignment. The preprocessing of usual graphs and (mixed) compound graphs appear to be equal in complexity. Even if the calculation of the sizes of nodes and edges is highly dependent on the application domain, because this is done by the information objects, we can assume without loss of generality that it runs in  $O(1)$  per node or edge. Furthermore, the additional complexity introduced by moving compounds or realizing hidden nodes hopping does not influence the preprocessing, because even when determining the initial coordinates, the valid positions are derived from the scanline positions and not by regarding cluster border or cluster separator nodes.

The differences between usual graphs and (mixed) compound graphs in the iterative phase arise from ensuring cluster validity while moving individual nodes. The logarithmic factor in `medianPos` and `minEdge` arises from sorting the nodes according to their upward or downward priorities, respectively. The other heuristics run over nodes and edges and usually move individual graph elements, probably after calculating edge funnels or corridors.

coordinates assignment step	usual graph	(mixed) compound graph
initializeGraphDistances	$O( V )$	$O( V )$
insertClusterBorderNodes	-	$O( V )$
calculateSizes	$O( V  +  E_H \cup E_N )$	$O( V  +  E_H \cup E_N )$
initializeXYCoordinates	$O( V )$	$O( V )$
sortPorts	$O( V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$	$O( V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$
determineStretchingFactor	$O( V  +  E_H )$	$O( V  +  E_H )$
initializeNodePriorities	$O( V )$	$O( V )$
fixGraphDistances	$O(1)$	$O(1)$
preprocessing: algorithm 4.20	$O( V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$	$O( V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$
medianPos	$O( V  \cdot \log  V )$	$O( V ^2)$
minEdge	$O( V  \cdot \log  V )$	$O( V ^2)$
minNode	$O( V  +  E_H )$	$O( V ^2 +  E_H )$
minPath	$O( V  +  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N )$
packcut	$O( V  +  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N )$
restrictVirtualRoot	$O( E_H )$	$O( E_H )$
iteration: algorithm 4.23	$O( V  \cdot \log  V  +  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N )$
prepareEdgePorts	$O( V  +  E_N  \cdot \log  E_N )$	$O( V ^2 +  E_N  \cdot \log  E_N )$
c_iteration	$O( V  \cdot \log  V  +  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N )$
routeEdges	$O( V  +  E_N  \cdot \log  E_N )$	$O( V ^2 +  E_N  \cdot \log  E_N )$
correctBends	$O( V  +  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N )$
correctCoordinates	$O( V  +  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N )$
postprocessing: algorithm 4.25	$O( V  \cdot \log  V  +  E_H \cup E_N  \cdot \log  E_N )$	$O( V ^2 +  E_H \cup E_N  \cdot \log  E_N )$
coordinates assignment macro phase	$O( V  \cdot \log  V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$	$O( V ^2 +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$

Table 4.6: Runtime complexities of the coordinates assignment macro phase.

The postprocessing phase heavily relies on sorting of ports or non-hierarchical edges for rows between ranks as well as repositioning all nodes after inserting of dummy nodes. By right-to-left processing of the ranks and caching offsets for individual nodes, the linear parts for nodes (in usual graphs) can be realized.

## 4.9 Postprocessing

After all is said and done, a lot more will be said than done.

Unknown

The result provided by the coordinates assignment S15 is a coordinates and cluster valid graph, in which certain aspects of UML class diagrams like association classes, constraint hyperedges, comments or disconnected nodes may not be respected properly. The processing steps presented in the last section are primarily dedicated to general (compound) graphs with hot spots, which allow an adaption to application domain specific graphs. As shown in Figure 6.5, a specialization of the coordinates assignment was provided, which specifies additional rules for UML class diagrams. Therefore, the UML specific handling of the aspects mentioned above are missing and will be handled by the postprocessing macro phase.

In this section, the layout of association classes (S16), hyperedges (S17), annotations (S18), disconnected nodes (S19), the optional snap-to-grid step (S20) for graph elements and the creation of the result instances (S21) are described.

### 4.9.1 Association Classes

Man associates ideas not according to logic or verifiable exactitude, but according to his pleasure and interests. It is for this reason that most truths are nothing but prejudices.

Remy de Gourmont (1858 – 1915)

Two main situations for association classes have to be handled here:

- At non-hierarchical edges: At a flat flat edge, the association class can simply be shifted below the rank and the distances of multiple association classes between the related visible nodes may also be adjusted. This works, because the association classes have been unpacked from composite nodes in S12 and space was reserved in coordinates assignment. At a top or bottom flat edge the association class is moved into the top/bottom flat middle rank area, which then might have to be enlarged. Placing an association class below the association maps to the default UML layout style and to `UML_ASSOCIATIONCLASSES`.

- At hierarchical edges: In this case, the association class should be placed at the left or right side of a hierarchical association respecting the presence of textual adornments (UML\_ADORNMENTS). Formally, the association class might be assigned to a row line (middle rank area) between the adjacent ranks and placed next to the association to be compliant to UML\_ASSOCIATIONCLASSES.

Association classifiers, association classes with further relations, were considered implicitly by the coordinates assignment. These graph elements have to be repositioned with respect to the further relations.

## 4.9.2 Hyperedges and Constraints

The more constraints one imposes, the more one frees one's self. And the arbitrariness of the constraint serves only to obtain precision of execution.

Igor Stravinsky (1882 – 1971)

In S5, we have compressed hyperedges to composite nodes and in S13 these composite nodes have been expanded to ensure space for the hyperedge connection nodes in their ranks. Figure 4.38 shows a typical situation which may occur for hyperedges after coordinates assignment. Due to S5, dummy nodes in hierarchical edges are packed together with either the visible start or end node of the hierarchical edge and therefore are assigned to one of both ranks. Hence, these dummy nodes remain in the assigned ranks and must be postprocessed now. Furthermore, hyperedges between non-hierarchical edges may be assigned in rectilinear style in the iterative

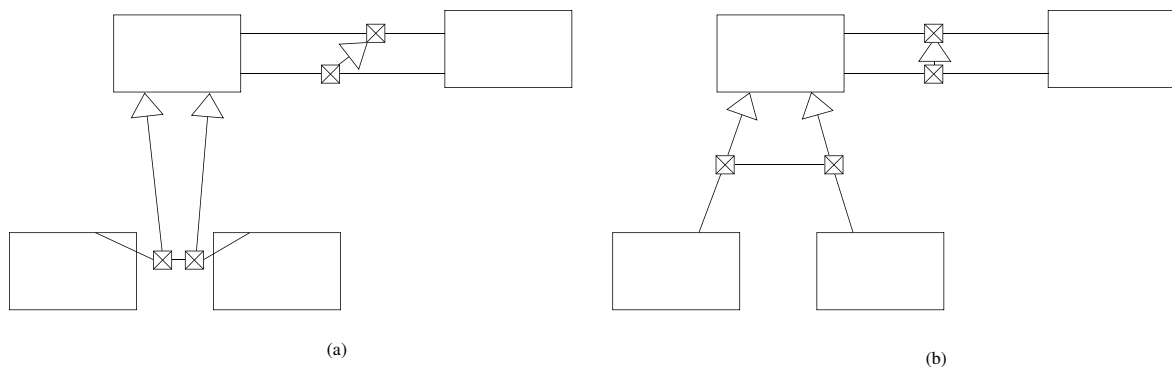


Figure 4.38: (a) typical situation for hyperedges after coordinates assignment, (b) after postprocessing the hyperedges.

coordinates process discussed in Section 4.8.5. Therefore, neither hyperedges at hierarchical edges nor the involved hierarchical edges themselves will probably not be drawn as straight lines according to UML\_HIERARCHY or in a proper placement for UML\_JOIN, because the

dummy nodes of these hyperedges are also positioned by the coordinates postprocessing for flat edges (Section 4.8.6). To fulfill UML\_HYPEREDGES, certain changes to the node and edge positions have to be done. The space wasted for hyperedges at non-hierarchical edges as depicted in Figure 4.38 (a) can be avoided by appropriate rules for the dummy nodes in the coordinates assignment macro phase.

Algorithm 4.27 shows the principle steps for postprocessing hyperedges in UML class diagrams.

---

**Algorithm 4.27** *postprocess\_hyperedges*


---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$

**output:**  $\bar{G}^C$

$\forall_{0 \leq r < n} \delta[r] := 0$

**for**  $r := 0$  **to**  $n - 1$  **do**

$H := \{e : e = \{v, w\} \in E_H \cup E_N, r(v) = r(w) \wedge \text{type}(e) = \text{HYPEREDGE}\}$

$\text{sort}(H, \text{edgeLengthSort}(\cdot, \cdot))$

$\delta := \text{formatAsMiddleRankEdges}(H)$

$\text{alignPositions}(H)$

**for**  $r_1 := r$  **to**  $n - 1$  **do**

$\delta[r] := \delta[r] + \delta$

**end for**

**end for**

**for**  $r := 0$  **to**  $n - 1$  **do**

$\text{moveY}(\sigma_r, \delta[r])$

$\text{repositionDiscriminators}(\sigma_r)$

**end for**

**return**  $\bar{G}^C$

---

### 4.9.3 Annotations

Don't get suckered in by the comments  
– they can be terribly misleading. Debug  
only code.

Dave Storer

In the current state of the layout algorithm, comment nodes are still members of composite nodes or removed from the graph in the case of disconnected comments. As described in Section 4.4.8, some comment, which are involved in “hierarchical” relations, have been processed implicitly by the coordinates assignment macro phase.

According to UML\_COMMENTS, disconnected comments are positioned at the corners of the drawing. The allowed corners or the priority of the target positions can be given by global user preferences.

Other comments must first be unpacked from their containing composite nodes. Even if the results may not be optimal, we try to avoid moving nodes placed so far. The remaining comments are processed in decreasing size. With priority, areas at the boundary of the graph with respect to

a short connection are searched. Especially, conformance to the layered structure is not required for comments due to `UML_COMMENTS`. For comments, which have to be placed at nodes apart from the boundary of the graph, the area around the target node is scanned for an appropriate placement. Then the space between ranks may be enlarged and, in the worst case, space has to be ensured by enlarging the area between nodes and thereby moving all nodes at the right of the target node to the right direction.

#### 4.9.4 Disconnected Nodes

Most graph layout algorithms assume that the graph is connected. Given a disconnected graph, one can either apply the basic algorithm to each connected component and then arrange the components, or make the graph connected.

[Ellson et al. 2003]

Disconnected nodes have been removed in `S9`, because these nodes would disturb the iterative coordinates assignment. In particular, in `medianPos` and `minEdge`, disconnected nodes would appear as dangling nodes which have to be considered by additional heuristics.

In general graph drawing it is appropriate to place disconnected nodes at positions, where minimum distance constraints are not violated. Therefore, an adaption of the both coordinates heuristics mentioned above may be appropriate. As a postprocessing step, data on unused space in the graph can be collected and the disconnected nodes can be reinserted greedily in decreasing size of extents or, like `GraphViz` [Ellson et al. 2003], the polyomino packing algorithm could be applied.

According to `UML_DISCONNECTED`, we are not in need of a complicated methods. Depending on the desired emphasize on disconnected nodes, the positions on the external boundary can be calculated from the minimum and maximum outer positions of all nodes and the extends of the ranks. Thereby, nodes are inserted in a way that the existing rank heights can be kept stable as far as possible.

#### 4.9.5 Alignment to a Specified Grid

New York... is a city of geometric heights, a petrified desert of grids and lattices, an inferno of greenish abstraction under a flat sky, a real Metropolis from which man is absent by his very accumulation.

Roland Barthes (1915 – 1980)

To realize `REQ_GRID`, we could have changed the basic functions in the coordinates assignment so that only coordinates, which are restricted to a specified grid, are used. The other alternative is to postprocess the layout result (`S20`) to meet `REQ_GRID`.

In algorithm 4.28 we apply a scanline algorithm. Vertical segments of edge chains are preserved, because global data is used while repositioning.

**Algorithm 4.28** snapToGrid**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), g_x, g_y$ **output:**  $\bar{G}$  $\delta_y := 0$  $\forall_{0 \leq r < n} \text{scanpos}[r] := \delta_x[r] := 0$ 

$$xpos(r) := \begin{cases} 0 & : \text{ if } \text{scanpos}[r] < 0 \\ \text{left}(\sigma_r[\text{scanpos}[r]]) & : \text{ otherwise} \end{cases}$$
**repeat** $max_x := \max_{0 \leq r < n} xpos(r)$ **for**  $r := 0$  **to**  $n - 1$  **do**  **if**  $\text{scanpos}[r] \geq 0$  **then**    **for**  $i := \text{scanpos}[r]$  **to**  $|\sigma_r| - 1$  **do**       $v := \sigma_r[i]$       **if**  $\text{left}(v) \geq max_x$  **then**        **break**      **end if**       $\text{left}(v) := \delta_x[r]$       **if**  $\text{right}(v) \bmod g_x \neq 0$  **then**         $\delta_x := \text{right}(v)$       **else**         $\delta_x := ((\text{right}(v) \bmod g_x) + 1) \cdot g_x$       **end if**      **if**  $i < |\sigma_r|$  **then**         $\text{scanpos}[r] := i$       **else**         $\text{scanpos}[r] := -1$       **end if**    **end for**  **end if**  **end for** $p := \max_{0 \leq r < n} \delta_x[r]$  $\forall_{0 \leq r < n} \delta_x[r] := p$ **until**  $\forall_{0 \leq r < n} \text{pos}[r] < 0$ 

{similarly for vertical positions but adjust node sizes}

**return**  $\bar{G}$

### 4.9.6 Create the Result Graph

Results! Why, man, I have gotten a lot of results. I know several thousand things that won't work.

Thomas A. Edison (1847 – 1931)

The last step of the entire layout algorithm is to create distinct instances which represent the result. In an application, temporary data of the layout algorithm, like the rank assignment or ports should not be accessible. .

Algorithm 4.29 shows the creation of the result graph instances. The first step destroys that additional data. This can implicitly be realized by the graph copy mechanism. For persistent storage of the layout results or further analysis, like calculation of metrics, implicit data has to be made explicit. In `replaceImplicitEdges`, e.g., reflective edges are transformed to nodes and edges. Finally, all nodes and edges are requested to place their compartments or adornments properly according to the coordinates calculated so far. Due to extensions for writable repository access or diagram interchange formats, also this data has to be made accessible from outside. Furthermore, all nodes and edges are locked to avoid accidental changes by applications working on the layout algorithm.

Algorithm 4.29 does not show further steps, which are important for a concrete application. The result may be rendered to a graphics context or a buffer to speed up scrolling the result. Thereby, debugging information, like the outer node area, the edge extents, the area of the graph or the closure of individual ranks, may be mapped into the result to appear when drawing the result.

---

#### Algorithm 4.29 createResult

---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma)$

**output:**  $G$

```

G := removeInternalData( $\bar{G}$ )
G := replaceImplicitEdges(G)
for all  $o \in V \cup E_H \cup E_N$  do
   $o := layout(o)$ 
   $o := lock(o)$ 
end for
return G

```

---

### 4.9.7 Conclusions

Finally, in conclusion, let me say just this.

Peter Sellers (1925 – 1980)

As discussed in this section, the postprocessing macro phase is responsible for the realization of `UML_HYPEREDGES` and `UML_ASSOCIATIONCLASSES`, both with respect to `UML_HIERARCHY`. Furthermore, `UML_COMMENTS`, `UML_DISCONNECTED` and



REQ\_GRID are provided by own algorithmic steps. The concrete compliance to the priorities for the aesthetic principles introduced in Section 3.3.6 cannot be determined here, because this heavily depends on the implementation. Obviously, an aggressive algorithm may easily destroy the results achieved by the preceding processing steps.

Table 4.7 shows the complexities of the individual steps of the postprocessing macro phase. Depending on the type of the graph, the nodes involved in association class, hyperedge or annotations structures must be repositioned with respect to cluster validity. Postprocessing hyperedges involves sorting the hyperedges for priorities to normalize the processing sequence. Disconnected nodes do not taint clusters and can be reinserted in linear time. Enforcing a grid placement of nodes and edges simply repositions all nodes according to a scanline. Replacing the internal structures, like reflective edges, to make them visible for external applications can be done in linear time.

## 4.10 Summary

Don't fear failure so much that you refuse to try new things. The saddest summary of a life contains three descriptions: could have, might have, and should have.

Louis E. Boone

In this chapter we have intensively discussed the issues of a standard implementation of the Sugiyama algorithm as well as extensions to apply that algorithm to UML class diagrams. We also have collected several information on the theoretical complexity of the individual (macro) steps. The aggregated formulae are summarized in Table 4.8. We can expect that the effective runtime will differ from the theoretical issues, because we did not take several impacts into account, which arise in a concrete (prototypical) implementation. Furthermore, not all conditions and application domain specific tests in the application have been discussed due to space limitations and readability.

coordinates assignment step	usual graph
S16	$O( V  +  E_H \cup E_N )$
S17	$O( V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$
S18	$O( V  +  E_H \cup E_N )$
S19	$O( V )$
S20	$O( V  +  E_H \cup E_N )$
S21	$O( V  +  E_H \cup E_N )$
postprocessing macro phase	$O( V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$

Table 4.7: Runtime complexities of the postprocessing macro phase. On (mixed) compound graphs we can expect  $O(|V|^2 + |E_H \cup E_N| \cdot \log |E_H \cup E_N|)$ .

To gain an overview of the real world runtime behavior of the prototypical implementation

preprocessing	S1-S9	$O( V  \cdot \log  V  +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$
rank assignment	S10	$O( V  \cdot \log  V  +  V  \cdot  E_H \cup E_N ^2)$
edge crossing reduction	S11	
	barycenter:	$O( E_H  +  V ^3)$
	median:	$O( V ^3)$
	hierarchical:	$O( V ^4)$
intermediary processing	S12-S14	$O( V  +  E_H \cup E_N )$
coordinates assignment	S15	$O( V ^2 +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$
postprocessing	S16-S21	$O( V ^2 +  E_H \cup E_N  \cdot \log  E_H \cup E_N )$

Table 4.8: Individual complexities of the macro steps of the *SugiBib* algorithm introduced in Section 4.1 on compound graphs.

*SugiBib*, we will discuss issues on optimizing Java programs in general and specific performance tuning for *SugiBib* as well as measurement various runtime issues in Chapter 5.

We can conclude that drawing application domain specific graphs increases the complexity of the applied algorithms compared with basic techniques known from graph drawing as mentioned in Section 2.2.2. Simple graphs with point-sized nodes denote one of the favorite types in graph drawing, on which most theoretical results have been proved. Obviously, the complexity of the algorithms will increase when areas for nodes are introduced, because node-node and node-edge overlappings have to be avoided. Changing the graph type from simple to non-simple, the storage mechanisms, the access paths within the graph and the algorithms working on the graph have to be adjusted. Introducing partitions of edges or nodes to be handled differently or considering clusters, compounds or mixed-compound contents also significantly increases the complexity. Due to our application domain, we are forced to work on structural complex graphs to be processed by runtime-intensive algorithms. Features like (different types of) edge labels, forbidden sides of nodes, as described along with port penalties in Section 4.6.3, fixed-wired or dynamic (external specified) constraints can be seen as orthogonal issues, which furthermore increase the complexity.

At a first glance, handling non-compound graphs, compound graphs as well as mixed-compound graphs by one algorithm seems to be a superfluous implementation effort. As shown in Section 4.8.5, the runtime complexity of the node movement algorithms, which are responsible for ensuring basic coordinates consistency, run in  $O(|V|)$  on non-compound graphs instead of  $O(|V|^2)$  on compound graphs. Hence, the execution speed will also implicitly increase for mixed-compound graphs compared with a compound graph in which the global cluster was mapped to an (invisible) package.



# 5 Measurements

---

In Chapter 3, we have introduced a functional specification for a layout algorithm for UML class diagrams. In particular, we have enumerated several general requirements and presented our unique set of layout criteria for UML class diagrams. In Chapter 4, a detailed discussion on our layout approach was given. In this chapter, we will complete our discussion on layout algorithms by methods to measure and compare the results produced by concrete layout algorithms. First, we will introduce formulae to realize the measurement of selected aesthetic criteria from Section 3.3.6. Layout metrics are capable of reflecting the priorities of aesthetic criteria and, therefore, consider UML specific as well as graph drawing relevant issues. Then, we will compare our layout algorithm to some of the other layout algorithms for UML class diagrams, which were briefly introduced in Section 2.2.3. Finally, we will review the theoretic runtime complexities collected in the last chapter in the context of results retrieved from running *SugiBib* on a set of test diagrams. Thereby, measurements more relevant from the viewpoint of algorithm theory and graph drawing will be collected.

As mentioned earlier, *SugiBib* is the realization of our layout algorithm as a graph drawing framework, which was specialized for the layout of UML class diagrams. An introduction to the architecture and the implementation of the framework will be given in the next chapter.

## 5.1 Measuring Aesthetics

Measure what is measurable, and what is not measurable, make it measurable.

Galileo Galilei (1564 – 1642)

In Section 3.3, we discussed general aesthetics for different types of diagrams and combined them to one set of rules for UML class diagrams in Section 3.3.6. Then we designed and implemented a layout algorithm to realize our set of rules. According to our set of rules, the results of the survey on layout facilities of UML tools in [Eichelberger 2002b] can be reinterpreted and refined. All results for that evaluation had been collected manually by looking on the result diagrams and estimating the values for different basic aesthetics. A more objective result can be obtained, if the values, e.g. as one metric for each aesthetic rule, are calculated by a program.

In this section, selected mandatory aesthetic principles from Section 3.3.6 will be introduced to gain a basic set of layout metrics for UML class diagrams. On the one side, these metrics will help comparing concrete results of other layout algorithm to those produced by *SugiBib* in the next section. On the other side, the obtained values can be used to realize a metric based testing approach for a layout algorithm. This aspect will be discussed in Section 6.4.

In [Purchase 2002], several metric formulae for general graphs have been described to formally analyze the result of a graph layout algorithm or to guide iterative methods like genetic algorithms or simulated annealing. Based on assumptions like point-sized nodes and connected, simple graphs, metrics for crossings, bends, symmetry, minimum angular resolution as well as edge and node orthogonality were defined. Some details of that work may also be reusable in our context.

Beside the distinction of aesthetic criteria in mandatory, optional and user defined issues as it was done in Section 3.3.6, we can distinguish the layout rules as well as the resulting metrics into

- structural rules, which reflect the validity of the diagram according to the underlying UML specification, e.g. ,UML\_SEMANTIC\_CLUSTERS or aspects of overlapping of classes, class interiors, relations or adornments. These rules summarize the UML requirements as a modification and extension of the general grammar of node-link diagrams and must hold on a valid diagram. Due to the layout results shown in Section 2.2.3 or [Eichelberger 2002b; Eichelberger and von Gudenberg 2003b], it is also important to capture these issues by layout metrics.

For these metrics, a binary output, e.g., in  $\{0, 1\}$  is sufficient, whereby 1 signals that the measured aspect is valid in a concrete diagram. We will call these metrics *decision metrics*.

- reading issues. A large number of aesthetic principles like UML\_HIERARCHY, UML\_MEDIAN or UML\_ASSOCIATIONCLASSES are intended to support the perception of a UML class diagram by human beings. As shown in Section 3.3.6, some arise from the default UML layout style, others from several disciplines involved in the task of drawing and recognizing diagrammatic structures. Due to the conflicts between structural complexity and perceptual simplicity, expressed by several priority levels, oftentimes the rules are realized in a concrete diagram as far as possible.

For these metrics, a continuous measurement is appropriated. We will call these metrics *quality metrics*.

The discussion above can be summarized as follows:

**Definition 26 (layout metric)**

A *layout decision metric* reflects the conformance of  $G^R$  to the structural rules of a UML class diagram. To reflect the severity of a basic diagramming rules  $p$ ,

$$metric_p : G^R \rightarrow \{0, 1\}$$

must hold.

A *layout quality metric* is a function

$$metric_p : G^R \rightarrow [0 \dots 1] \subset \mathbb{R}$$

that returns a value representing the conformance of a layout result to a certain aesthetic principle  $p$  or a combination of different principles in the case of a combined metric. The better  $p$  is realized by  $G^R$ , the higher  $metric_p(G^R)$  is required to be. If  $p$  considers a subset  $S$  of graph elements and  $|S| = 0$ ,  $metric_p(G^R) = 1$  must hold.

aesthetic	sub issue	measured by
UML_GRAPHDRAWING	GDR_EDGE_CROSS	number of edge crossings
UML_GRAPHDRAWING	GS_ORTHOGONAL	number of non-rectilinear edges
UML_GRAPHDRAWING	GDR_MIN_BENDS	illegal bend situations
UML_GRAPHDRAWING	GDR_MIN_EDGES	superfluous edge length
UML_MEDIAN		distance to median position
UML_ASSOCIATIONCLASSES		distance of an association class
UML_COMMENTS		distance of comments
UML_HIERARCHY		conformance to vertical flow
UML_SPATIAL		common shape of hierarchical layers
UML_NODES	GDR_OVERLAP	overlapping area
UML_SEMANTIC_CLUSTERS		not entirely contained elements

Table 5.1: Layout metrics currently realized in *SugiBib* described by relations between UML specific aesthetic principles and individual issues to be considered for a realization as metric. The upper part of the table contains quality metrics, the lower part decision metrics.

Table 5.1 shows the set of aesthetic criteria, which were considered to be discussed as metrics here. Furthermore, Table 5.1 displays basic ideas how to realize the individual metrics in the sequence they will be discussed in this section.

We will assume a result graph to be measured, i.e., the graph contains all relevant information to be drawn on a graphical context in terms of nodes and edges. Therefore, in this section, bends will be represented as dummy nodes and edge paths as edge segments. Without loss of generality we will assume normalized coordinates  $x \in \mathbb{N}_0$ .

For notational convenience, we introduce the following functions: Let

$$len(e) := \sqrt{(startX(e) - endX(e))^2 + (startY(e) - endY(e))^2}$$

be the euclidian length of the edge  $e$  and  $A(v) := width(v) \cdot height(v)$  be the area of the bounding box of node  $v$ . Let  $T$  be an arbitrary set of elements denoting types of nodes or edges. Similar to the type of a node as introduced in definition 5, we will assume that an edge also provides a function returning its type, i.e., when an edge  $e$  acts as the association class connecting line,  $type(e) = ASSOCCLASS$  holds. Then

$$filter(V, T) := \{v : v \in V, type(v) \notin T\}$$

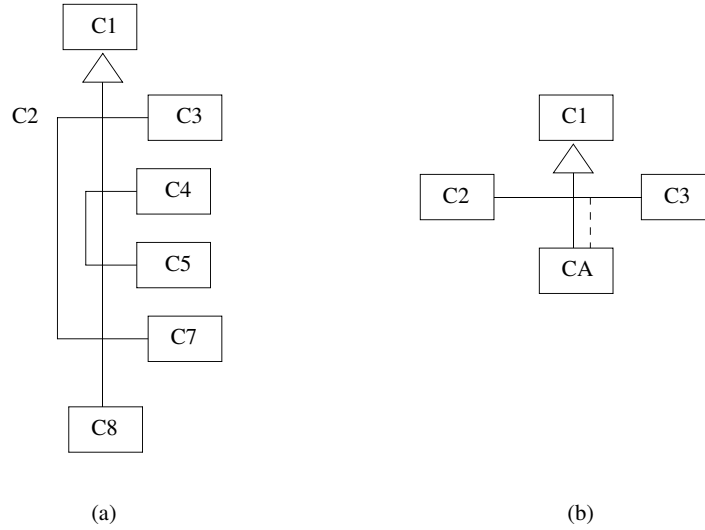


Figure 5.1: (a) an edge crossing situation, (b) an UML specific crossing situation.

$$\text{filter}(E, T) := \{e : e = \{v, w\} \in E, \text{type}(e) \notin T\}$$

be filter functions which return nodes or edges not being of a type specified in  $T$ , respectively. When giving the notation of the metric formulae, some external parameters on how to consider separate aspects of an individual measurement will implicitly be defined. These parameters will be given by Greek letters and must be defined before measuring concrete diagrams. At the end of this section, an example will demonstrate the usage of our formulae. In the context of that example, concrete values for the metric parameter will be specified.

## Edge Crossings

Using theorem 3 we have

$$m(G^R) = 1 - \begin{cases} \frac{K(G^R)}{|E|^2} & : \text{ if } |E| > 0 \\ 0 & : \text{ otherwise} \end{cases}$$

as an obvious proposal of a measurement for GDR\_EDGE\_CROSS in which the number of crossing  $K(G^R)$  is scaled against the maximum number of crossings. In [Purchase 2002] a more sensitive scaling was suggested in which  $\frac{|E| \cdot (|E| - 1)}{2}$  was decreased by the number of impossible crossings in straight-line drawings of connected graphs.

Due to our convention of considering a result graph, which contains edge segments and dummy nodes, we propose a scaling based on the number of edge chains instead of individual edges. In Figure 5.1 (a)  $m(G^R) = 1 - \frac{4}{49} = \frac{45}{49} \approx 0.92$  and the Purchase scaling would lead to  $1 - \frac{4}{21} = \frac{17}{21} \approx 0.82$ . A scaling based on the square number of edge chains would return  $1 - \min\{1, \frac{4}{9}\} = \frac{5}{9} \approx 0.56$ . Without application domain specific corrections to the number of

edge crossings, e.g., to consider the default UML layout style so that vertically connected association classes appear below the related association, Figure 5.1 (b) would be judged as follows:  $m(G^R) = 1 - \frac{1}{9} = \frac{8}{9} \approx 0.89$  and according to the Purchase scaling  $1 - \frac{1}{3} = \frac{2}{3} \approx 0.67$ . When the default UML layout style is considered, 1 would be the result.

Also because of practical experience in testing the implementation, we prefer the scaling according to edge chains with respect to UML specific corrections. Let

$$\text{chains}(G^R) := |\{(v, w) : \neg \text{isHidden}(v) \wedge \neg \text{isHidden}(w) \wedge \exists k \geq 0, \vec{e}_0=(v, v_1), \dots, \vec{e}_k=(v_k, w) \forall v_i \text{ isHidden}(v_i)\}|$$

be the number of edge chains in  $G^R$ , whereby an edge chain is a path, which connects two visible nodes via an arbitrary number of dummy nodes. Let  $K_a(G^R) \leq K(G^R)$  be the number of allowed crossings in  $G^R$  due to the application domain, e.g., at association classes. Then, from a practical viewpoint, the more appropriate metric is defined by

$$m_{\text{cross}}(G^R) := 1 - \begin{cases} \min \left\{ 1, \frac{K(G^R) - K_a(G^R)}{\text{chains}(G^R)^2} \right\} & : \text{ if } |\text{chains}(G^R)| > 0 \\ 0 & : \text{ otherwise} \end{cases}$$

The less edge crossings (per edge chain) occur, the more close to 1 is the value of  $m_{\text{cross}}(G^R)$ . Hence, the less edge crossings the better the aspect of layout quality measured by  $m_{\text{cross}}(G^R)$ .

### Rectilinear Edges

For non-hierarchical edges, which are drawn orthogonally in *SugiBib* according to UML\_HIERARCHY, this metric simply counts the number of non-conforming edges. Let

$$\text{fail}(E) := \{e : e \in E, \text{startX}(e) \neq \text{endX}(e) \wedge \text{startY}(e) \neq \text{endY}(e)\}$$

be the subset of edges of  $E$  which are not rectilinear and  $\text{joined}(E) \subset E$  the subset of edges of  $E$  which are joined according to UML\_JOIN. We will not take the connecting lines of comments and association classes into account here, because individual minimum length constraints will be considered by specialized metrics. With

$$\text{filter}(E) := \text{filter}(E, \{\text{COMMENT}, \text{ASSOCCLASS}\})$$

$$\text{notRectilin}(E) := \begin{cases} \frac{|\text{fail}(\text{filter}(E))|}{|\text{filter}(E)|} & : \text{ if } |\text{filter}(E)| > 0 \\ 0 & : \text{ otherwise} \end{cases}$$

is the judgment on rectilinear edges for a given set of edges. Then

$$m_{\text{rectilin}}(G^R) := 1 - \frac{\alpha \cdot \text{notRectilin}(\text{joined}(E_H)) + \beta \cdot \text{notRectilin}(E_N)}{\alpha + \beta}$$

is the appropriate metric with  $\alpha, \beta \geq 0$  and  $\alpha + \beta \neq 0$  as given scaling values, e.g., to emphasize failures on non-hierarchical edges. The more non-rectilinear edges for a given set of edges occur, the higher  $\text{notRectilin}(E)$  and the lower  $m_{\text{rectilin}}(G^R)$  is.



## Bends

From the viewpoint of graph drawing, the number of bends is an indicator of quality to compare different graph drawing algorithms. In [Purchase 2002], two views on a graph are used to define a metric which judges the number of bends (GDR\_MIN\_BENDS). Beside the polyline drawing of a graph, an auxiliary straightline graph is derived by bends promotion. The number of bends, the difference between the numbers of edges of the two graphs, is then normalized against the number of edges in the straightline drawing.

We analyze the graph for avoidable bend situations as follows: Figure 5.2 shows the situations

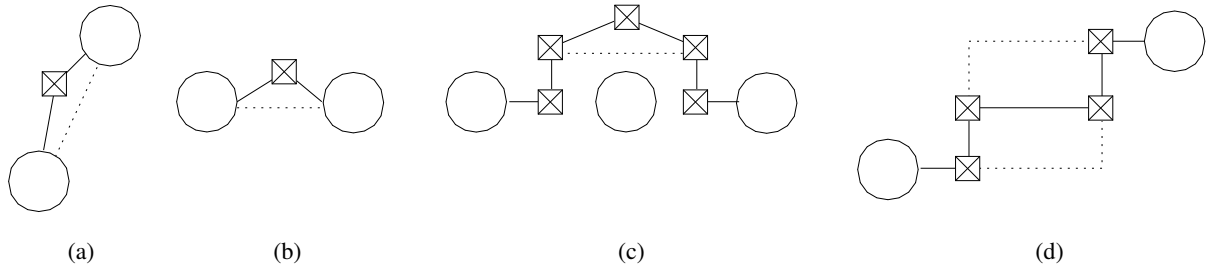


Figure 5.2: Avoidable bend situations: (a) hierarchical edge chain, (b) flat flat edge chain between neighbors, (c) top flat edge chain, (d) Z-shaped edge chain

which lead to avoidable, probably visually unpleasing layout situations. While in (b) and (c) a bend must not occur, because straight lines can be used instead, the bend in (a) and (d) can be avoided, if no other node is overlapped by using an alternative route depicted as dotted lines.

Let  $\#bends_H$  the number of hierarchical bends,  $\#avoidable_H$  the number of avoidable situations at hierarchical relations according to Figure 5.2 (a), then  $bends_H(G^R)$

$$bends_H(G^R) := \begin{cases} 0 & : \text{ if } \#bends_H = 0 \\ \frac{\#avoidable_H}{\#bends_H} & : \text{ otherwise} \end{cases}$$

is the ratio between avoidable situations and number of bends at hierarchical edges. The higher the number of avoidable bends is, the more close to 1  $bends_H(G^R)$  is. Let  $\#bends_N$ ,  $\#avoidable_N$  and  $bends_N(G^R)$  be defined similarly according to Figure 5.2 (b) to (d).

Finally, with  $\alpha, \beta \geq 0$  and  $\alpha + \beta \neq 0$  as given scaling values, e.g., to emphasize failures on bends at non-hierarchical edges,

$$m_{bends}(G^R) := 1 - \frac{\alpha \cdot bends_N(G^R) + \beta \cdot bends_H(G^R)}{\alpha + \beta}$$

judges the bends situation in  $G^R$ .

## Edge Lengths

Let  $reqLen(e)$  be the required edge length of  $e$ , which is defined as the space allocated by the textual and graphical adornments, the nodes next to the edge segment and the minimum node dis-

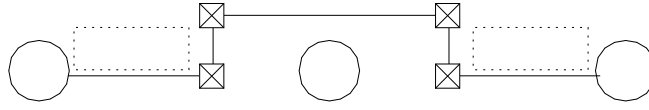


Figure 5.3: Calculating the required size of a horizontal edge.

tance between neighbored nodes, if present. Figure 5.3 illustrates the influences of adornments and neighbored nodes for a non-hierarchical edge. Similar considerations can be made for hierarchical edges.

Let  $\min(e) := \min\{reqLen(e), len(e)\}$  and  $\max(e) := \max\{reqLen(e), len(e)\}$ .  $\frac{\min(e)}{\max(e)} = 1$  will occur only, if the length of an edge conforms to the required (minimum) length. If an edge is shorter than the minimum length,  $reqLen(e) < len(e)$  and  $\frac{\min(e)}{\max(e)} < 1$  holds. Similarly, if an edge is longer than the minimum length,  $len(e) < reqLen(e)$  and  $\frac{\min(e)}{\max(e)} < 1$  is true.

Furthermore, let  $filter(E) := filter(E, \{COMMENT, ASSOCCLASS\})$  because comment and association class connecting edges will be considered by individual metrics. Then

$$edgeLengths(E) := \begin{cases} \frac{\sum_{e \in filter(E)} \frac{\min\{reqLen(e), len(e)\}}{\max\{reqLen(e), len(e)\}}}{|filter(E)|} & : \text{ if } |filter(E)| > 0 \\ 1 & : \text{ otherwise} \end{cases}$$

measures the conformance of the edges in  $E$  to their individual required size. Finally, with  $\alpha, \beta \geq 0$  and  $\alpha + \beta \neq 0$  as given scaling values, e.g., to emphasize non-conformance at hierarchical edges,

$$m_{edgeLengths}(G^R) := \frac{\alpha \cdot edgeLengths(E_H) + \beta \cdot edgeLengths(E_N)}{\alpha + \beta}$$

is a metric for GDR\_MIN\_EDGES.

### Positions of Parents and Children

According to UML\_MEDIAN, this metric takes the individual positions of hierarchical edges into account and calculates the sum of the differences to a median balanced layout. For vertical ports let  $E_x(v) := topPorts(v) \cup bottomPorts(v)$  be the set of edges to be considered on node  $v$ . Let furthermore  $x(e, v)$  be the horizontal position where  $e$  is attached to  $v$  given by either  $startX(e)$  or  $endX(e)$ . Then

$$m_x(v) := \begin{cases} \frac{\sum_{e=\{v,w\} \in E_x(v)} x(e,v) - x(e,w)}{\sum_{e=\{v,w\} \in E_x(v)} |x(e,v) - x(e,w)|} & : \text{ if } |E_x(v)| > 0 \\ 0 & : \text{ otherwise} \end{cases}$$

calculates the differences of the edges in  $E_x(v)$  compared to a (theoretical) straightline layout. Let  $m_x(G^R) := \sum_{v \in V} m_x(v)$  be the aggregated distances for all nodes in  $G^R$ . Furthermore, let  $m_y(G^R)$

be similarly defined on the vertical ports of all nodes.

To compose a metric of these two measurements, we have to consider that  $m_x(G^R), m_y(G^R) \in [-1, 1]$ . Therefore, with  $\alpha, \beta \geq 0$  and  $\alpha + \beta \neq 0$  as scaling values, e.g., to disable influences of vertical ports,

$$m_{median}(G^R) := \frac{\alpha \cdot (1 + m_x(G^R)) + \beta \cdot (1 + m_y(G^R))}{2 \cdot (\alpha + \beta)}$$

calculates a metric, which expresses the conformance to UML\_MEDIAN.

### Association Classes

For UML\_ASSOCIATIONCLASSES we simply consider the distance of association classes, because most illegal situations like overlappings or edge crossings are described by other aesthetic criteria and are therefore handled by other metrics. Similar to the positioning of association classes described in Section 4.9.1, the minimum distance of individual association classes can be calculated. We omit a detailed discussion on that here.

Let  $\delta_{min}(v)$  be the minimum distance between an association class  $v$  and the dummy node simulating the hyperedge connection point. Furthermore, let  $\delta(v)$  be the euclidian length of the (dashed) edge connected to  $v$  and let

$$AC(G^R) := \{w : w \in V, type(w) = ASSOCCLASS\}$$

be the set of association classes in  $G^R$ . Similar to the description of  $edgeLengths(E)$ , a part of the edge lengths metric,

$$m_{assocClass}(G^R) := \begin{cases} \frac{1}{|AC(G^R)|} \cdot \sum_{v \in AC(G^R)} \frac{\min\{\delta_{min}(v), \delta(v)\}}{\max\{\delta_{min}(v), \delta(v)\}} & : \text{if } |AC(G^R)| > 0 \\ 1 & : \text{otherwise} \end{cases}$$

defines a metric for association class distances. If  $m_{assocClass}(G^R) = 1$ , no association classes are present or all association classes are positioned in a proper distance. The less  $m_{assocClass}(G^R)$  is, the larger the differences between the minimum distance and the visible distance of association classes appear.

### Comments

As defined for UML\_COMMENTS, connected comments can be treated akin to association classes. Let

$$CS(G^R) := \{w : w \in V, type(w) = COMMENT \wedge d(w) > 0\}$$

be the set of connected comments. As for association classes, let  $\delta_{min}(v, w)$  be the minimum distance between a comment  $v$  and the connected node  $w$ , which also may be the dummy node simulating a hyperedge connection point at an edge. Furthermore, let  $\delta(v, w)$  be the euclidian

length of the (dashed) edge, which connects the comment  $v$  to  $w$ . Then, connected comments can be handled by

$$m_{cc}(G^R) := \begin{cases} \frac{1}{|CS(G^R)|} \cdot \sum_{v \in CS(G^R)} \frac{1}{|V^-(v) \cup V^+(v)|} \cdot \sum_{w \in V^-(v) \cup V^+(v)} \frac{\min\{\delta_{min}(v,w), \delta(v,w)\}}{\max\{\delta_{min}(v,w), \delta(v,w)\}} & : \text{ if } |CS(G^R)| > 0 \\ 1 & : \text{ otherwise} \end{cases}$$

$m_{cs}(G^R) = 1$ , if all connected comments are close to their connected elements. Thereby,  $\delta_{min}(v, w)$  considers cases like all connected elements are placed in a line aside from the comment or all connected elements surround the comment.

For disconnected comments, the closeness to the border of the drawing can be taken into account. Let

$$CD(G^R) := \{w : w \in V, type(w) = COMMENT \wedge d(w) = 0\}$$

be the set of disconnected comments and

$$CB(G^R) := \{w : w \in V, type(w) = COMMENT \wedge \neg \exists_{v \in V} type(v) \neq COMMENT \wedge (left(v) < right(w) \vee right(v) > left(w))\}$$

be the set of comments which are not placed at the border of the graph. Then

$$m_{cd}(G^R) := \begin{cases} \frac{|CB(G^R)|}{|CD(G^R)|} & : \text{ if } |CD(G^R)| > 0 \\ 1 & : \text{ otherwise} \end{cases}$$

considers disconnected comments.  $m_{cd}(G^R) = 1$ , if no other nodes are located between a disconnected comment and the border and  $m_{cd}(G^R) = 0$  if all disconnected comments are surrounded by other model elements.

Finally,

$$m_{comment}(G^R) := \frac{m_{cc}(G^R) + m_{cd}(G^R)}{2}$$

is the combined metric for measuring the positions of comments.

### Conformance of Hierarchical Edges

Hierarchical edges should be compliant to the default UML layout style and should emphasize the (pseudo) hierarchy according to UML\_HIERARCHY. We will assume that the start node of a hierarchical edge is always assigned to a lower rank number than the end node and the (virtual) root of the hierarchy is assigned to rank 0. This induces a standard flow, which, due to the default layout style, might not always be appropriate, i.e., when inheritance relations and dependencies are considered as hierarchy, on dependencies it may be sufficient, that the participants are located below each other in different ranks but the direction of the edge may not be relevant.

Therefore, let  $dir(e) \rightarrow [-1, 1]$  be a function, which returns the valid direction for the hierarchical edge  $e$ , i.e., in certain cases it might also allow reversed hierarchical edges. Hence, if  $dir(e) \cdot (r(w) - r(v)) > 0$  the edge connecting  $w$  and  $v$  is compliant to UML\_HIERARCHY.

Because non-hierarchical edges may be flat or multi-level edges as introduced by definition 17, we do not take them into account here. Therefore, the straightness of hierarchical edges as an aspect of UML\_HIERARCHY can be measured by

$$m_{rankAssign}(G^R) := \begin{cases} \frac{1}{2} + \frac{\sum_{\vec{e}=(v,w) \in E_H(G^R)} dir(\vec{e}) \cdot (r(w) - r(v))}{2 \cdot |E_H(G^R)|} & : \text{ if } |E_H(G^R)| > 0 \\ 1 & : \text{ otherwise} \end{cases}$$

The closer  $m_{rankAssign}(G^R)$  is to 1, the more the hierarchical edges emphasize the (pseudo-) hierarchy due to different ranks and an overall flow influenced by  $dir(e)$ .

### Common Layout for all Hierarchy Levels

Nodes in a rank should be laid out according to a common principle as an aspect of UML\_HIERARCHY and UML\_SPATIAL. The “ranks” required by the aesthetic principle should have a similar shape and should be oriented to the same geometrical direction. Applying this to arbitrary tools, the metric might be confronted with various shapes like concentric ring layouts, hyperbolic trees, stars, horizontal or vertical lines. We will avoid a preselection to a certain shape, because detailed layout issues and, therefore, the spacial distribution of nodes is not specified by UML. Due to our experience, the default UML layout style of horizontal lines is preferred.

Figure 5.4 shows some of the shapes to be considered by our metric. Figure 5.4 (a) depicts ranks according to the default UML layout style. Vertical levels may occur when the default direction of the hierarchy is changed. A combination of horizontal and vertical shapes, as shown in Figure 5.4 (c), may occur when the tip-over convention [Sugiyama 2002; OMG 2003d] is applied to UML class diagrams. Figure 5.4 (e) is dedicated to concentric ring layout or hyperbolic layout. In [Gröhling 2003] the development of this metric and some disappointing trials have been described.

The metric consists of two parts. The first considers the direction of the ranks, the second the overlapping areas of the ranks.

According to experience, the direction  $\varphi(\sigma_r)$  of  $\sigma_r$  in degrees was defined as the angle of the line which maximizes the distance of the central coordinates of all nodes in  $\sigma_r$ . Then

$$\varphi(G^R) := \max_{0 \leq r_1 \neq r_2 < n} |\varphi(r_1) - \varphi(r_2)|$$

is the maximum angle of all ranks in  $G^R$ .

Taking only the direction of ranks into account, circular ranks might receive an appropriate value but this depends on the concrete positions of the nodes admitting the maximum distance. This is illustrated in Figure 5.4 (e). Furthermore more exotic shapes (like a “V” or similar more theoretic ideas) might be considered by average directions only. To provide an appropriate value in these cases, also the shapes of the ranks based on the individual convex hulls are considered. Because even similar rank shapes may have various sizes as depicted in Figure 5.4 (a), comparing the areas of the convex hulls would be misleading.

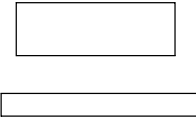

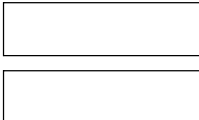
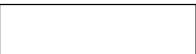
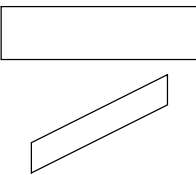



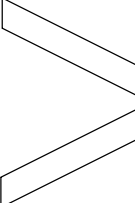
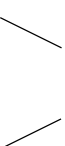
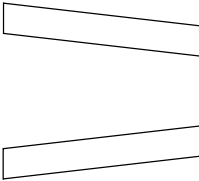

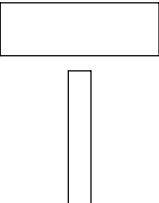

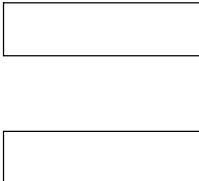

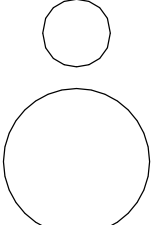
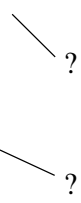
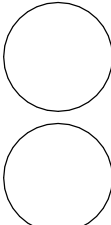
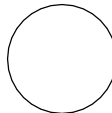
	rank shapes	angle	scaled	common area	expected value
(a)					$1.0 \cdot 1.0 = 1.0$
(b)					$0.7 \cdot 0.8 = 0.56$
(c)					$0.4 \cdot 0.4 = 0.16$
(d)					$0.0 \cdot 1.0 = 0.0$
(e)					$0.75 \cdot 1.0 = 0.75$

Figure 5.4: Various rank shapes to be considered by  $m_{rankShape}$ . The largest rank in the first row was assumed as the size of the entire graph.

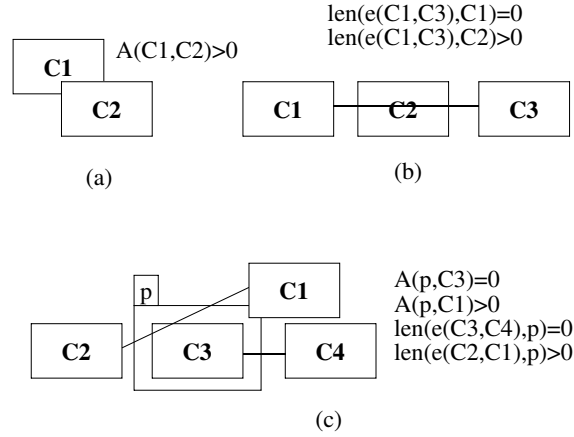


Figure 5.5: Overlapping situations, which are implicitly handled by the overlapping area  $A(v, w)$  and the overlapping length function  $len(e, v)$ .

Therefore, the hulls are smoothed, to avoid variances because of different node sizes, and are scaled with respect to the maximum area induced by the entire graph. Then, the common area of all scaled hulls of all ranks is compared with the area of the graph. This improves the measurement in Figure 5.4 (e) and decreases the values in the case of various rank shapes in the same graph as desired.

Let  $CH(\sigma_r) \subset \mathbb{N}_0^2$  be the convex hull of rank  $r$  considering visible nodes only, let  $A : \mathbb{N}_0^2 \rightarrow \mathbb{R}$  be the area occupied by a specified set of points and let  $A(G^R)$  be the area of the bounding box of  $G^R$ . Furthermore, let  $s : G^R \times \mathbb{N}_0^2 \rightarrow \mathbb{N}^2$  be a scaling which resizes the bounding box of the input area to the bounding box of  $G^R$  to normalize it.

$$m_{rankShape}(G^R) := \begin{cases} \left(1 - \frac{\varphi}{90^\sigma}\right) \cdot \frac{A(\bigcap_{r=0}^{n-1} s(G^R, CH(\sigma_r)))}{A(G^R)} & : \text{ if } |V| > 0 \\ 1 & : \text{ otherwise} \end{cases}$$

Even if the absolute values in some cases like circular layout might be less than expected,  $m_{rankShape}$  admits correct tendencies among similar ranks.

### Overlapping Elements

Due to UML\_NODES and UML\_EDGES, neither nodes nor edges should overlap except for edges, which are allowed to connect inside a node. Even if node overlapping can be avoided using appropriate basic functions for node movement and placement, overlappings between nodes and edges may occur due to layout situations which are not (completely) handled by heuristics.

Let  $len(e, v) \leq len(e)$  be the length of the overlapping part of edge  $e$  with node  $v$ ,  $len(e, f) \leq \min\{len(e), len(f)\}$  be the length of the overlapping part of the edges  $e$  and  $f$ , and  $A(v, w)$  the overlapping area of the nodes  $v$  and  $w$ . As depicted in Figure 5.5, it is assumed that  $len(e, v)$  as well as  $A(v, w)$  handle non-overlapping situations at nested elements correctly.

Consequently,

$$m_{overlap}(G^R) := \begin{cases} 0 & : \exists_{e \in E_H \cup E_N} len(e, v) > 0 \vee \exists_{e, f \in E_H \cup E_N, e \neq f} len(e, f) > 0 \vee \\ & \exists_{v, w \in V, v \neq w} A(v, w) > 0 \\ 1 & : \text{otherwise} \end{cases}$$

denotes the binary overlapping elements decision metric.  $m_{overlap}(G^R) = 1$  if no overlappings occur in  $G^R$  and  $m_{overlap}(G^R) = 0$  in the case of erroneous overlappings.

### Cluster Containment

A node which is contained in another node by nesting relations must neither overlap the boundary of its cluster parent nor be placed completely outside the area of its cluster parent (UML\_SEMANTIC\_CLUSTERS). For this metric we simply count the overlapping nodes. Alternatively, the erroneous overlapping area might be calculated instead but overlappings at clusters are erroneous situations regardless the overlapping area.

$$overlap(v) := \{w : w \in compoundParents(v), left(w) + \delta_l(w) > left(v) \vee \\ right(v) > right(w) - \delta_r(w) \vee top(w) + \delta_t(w) > top(v) \vee \\ bottom(v) > bottom(w) - \delta_b(w)\}$$

is the set of cluster parents which are overlapped by  $v$ . Cluster specific distances to inner nodes are denoted here by  $\delta_x(w), x \in \{t, l, r, b\}$ . Then

$$m_{cluster}(G^R) := \begin{cases} 0 & : \exists_{v \in V} overlap(v) > 0 \\ 1 & : \text{otherwise} \end{cases}$$

denotes the binary cluster containment decision metric.  $m_{cluster}(G^R) = 1$  if all nested nodes are properly contained in all their cluster parent nodes and  $m_{cluster}(G^R) = 0$  in the case of erroneous overlappings.

### Combining Metrics

The metrics defined in this section appear as individual measurements. A single value, which characterizes the layout quality of an UML class diagram would also be appreciated. Let  $M_d$  be the set of decision metrics and  $M_q$  be the set of layout quality metrics so that  $M = M_d \cup M_q$ . To gain a single value reflecting the layout quality of a diagram, we can simply multiply the decision metrics and combine the quality metrics according to the arithmetic mean over the individual metric values:

$$m_{aggregated}(G^R) := \prod_{i \in M_d} m_i(G^R) \cdot \frac{\sum_{i \in M_q} m_i(G^R)}{|M_q|}$$

$m_{aggregated}(G^R)$  treats all quality metric values the same and, except of the decision metrics, does not consider the priorities of our aesthetic principles. Therefore, with respect to priorities, we can



combine the individual metrics as follows:

$$m_{weighted}(G^R) := \prod_{i \in M_d} m_i(G^R) \cdot \frac{\sum_{i \in M_q} \alpha_i \cdot m_i(G^R)}{\sum_{i \in M_q} \alpha_i}$$

where  $\forall_{i \in M} \alpha_i \geq 0$  and  $\sum_{i \in M} \alpha_i \neq 0$  reflect the priorities, e.g., as displayed in Table 3.3. More emphasize on the priorities can be achieved by

$$m_{sqWeighted}(G^R) := \prod_{i \in M_d} m_i(G^R) \cdot \frac{\sum_{i \in M_q} \gamma_i \cdot m_i(G^R)}{\sum_{i \in M_q} \gamma_i} \text{ with } \gamma_i := \left( \frac{\alpha_i}{\max_{j \in M_q} \alpha_j} \right)^2$$

When presenting metric values below, we will mention concrete values for these three combination formulae, even if we prefer  $m_{sqWeighted}$ .

More issues on combining metrics and software measurements can be found in [Zuse 1998].

### Example

To demonstrate the application of our basic metrics set, we consider the following two diagrams. Figure 5.6 shows an UML class diagram laid out by an arbitrary mechanism. Figure 5.7 depicts a readable version of Figure 5.6.

For each individual metric, Table 5.2 shows the parameter values left open above and the individual metric weights  $\alpha_i$  to reflect the priorities according to Table 3.3. Furthermore, the individual metric values as calculated from both diagrams and the aggregated or combined metric value are displayed. In Figure 5.6, an edge crossing (association class connecting line with generalization between C1 and C3), different overlays of two nodes (C4 with I1) or edges with edges (C5 to C2) and a problem at the reflective edge at C5 are shown. It is obvious, that some measurements will not receive the maximum value 1 and the entire judgment of the diagram will be 0, because overlappings are considered as a binary decision metric. Figure 5.7 looks much better, in particular, our aesthetic principles were kept in mind while drawing the diagram.  $m_{median}$  cannot be 1, because C5 is not at median position below C2 and some edges could be drawn shorter.

The concrete values in Table 5.2 emphasize the basic validity of our formulae. More examples will be given in terms of a comparison of three layout algorithms for UML class diagrams in the next section.

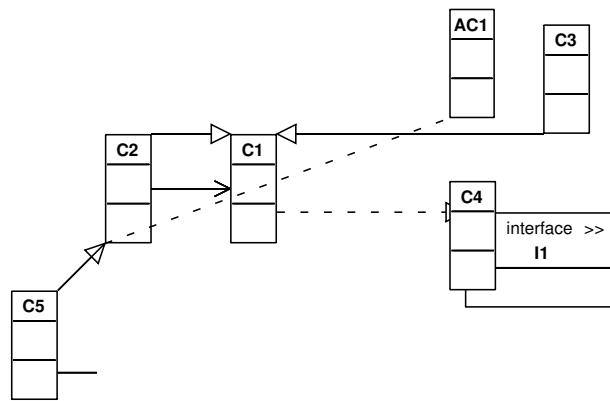


Figure 5.6: An arbitrary UML class diagram for the demonstration of the layout metrics.

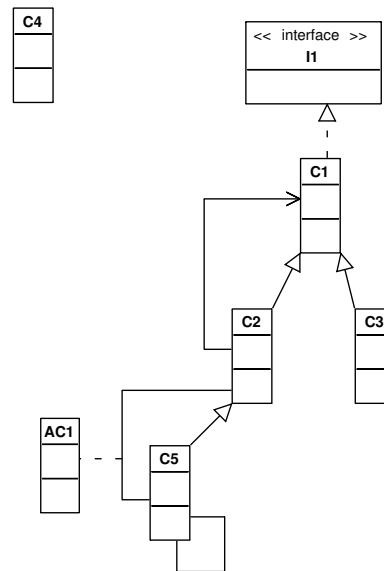


Figure 5.7: More appropriate drawing of Figure 5.6.

metric	parameter	$\alpha_i$	Figure 5.6	Figure 5.7
$m_{cross}$	-	1	0.98	1.0
$m_{rectilin}$	$\alpha = 0, \beta = 1$	1	0.88	1.0
$m_{bends}$	$\alpha = 1, \beta = 1$	1	0.5	1.0
$m_{edgeLengths}$	$\alpha = 0, \beta = 1$	1	1.0	0.71
$m_{median}$	-	6.1	0.91	0.95
$m_{assocClass}$	-	4.5	0.8	1.0
$m_{comment}$	-	4.1	1.0	1.0
$m_{rankAssign}$	$\alpha = 1, \beta = 0$	6.4	1.0	1.0
$m_{rankShape}$	-	6.3	0.67	1.0
$m_{overlap}$	-	5.4	0.0	1.0
$m_{cluster}$	-	6.1	1.0	1.0
$m_{aggregated}$			0.0	0.96
$m_{weighted}$			0.0	0.98
$m_{sqWeighted}$			0.0	0.99

Table 5.2: Metric values of the diagrams shown in this section.

## Conclusions

In this section, we have discussed the definitions of selected layout metrics intended to measure the validity as well as the quality of the layout of UML class diagrams according to our set of aesthetic principles introduced in Section 3.3.6.

Implementing the definitions of the layout metrics, our dream of automatically and objectively comparing the layout results of existing UML tools briefly mentioned in Section 3.3.7 can become reality. Therefore, we need a tool which is able to process the layout results of all UML tools to be analyzed. To get the data, the analyzing tool may

- be directly integrated into the tool to be measured. Most tools provide an Application Programming Interface (API) for plug-ins, e.g. layout engines or code generators. Because no common API for UML tools was specified so far, most tool vendors provide an own, proprietary interface. Hence, it would take an enormous effort to make a small measurement tool compatible to more than 40 different UML tools.
- access the data storage system of the UML tool instead of interacting with the program. Most of the UML tools rely on their own implementation of a repository mechanism and because most tools, as pointed out in Section 2.2.3, are not fully compliant to current versions of the UML, we can expect that these repository mechanisms will also differ in various features. Access interfaces like JMI [SUN JCP 2003] may provide a common API to the storage mechanism of UML tools somewhen in future. Even if JMI seems to be a general mechanism, because it generically relies on XMI, it is closely related to the Java programming language. Furthermore, other metadata interfaces like EMF/UML2<sup>1</sup>

<sup>1</sup><http://www.eclipse.org/uml2/>

might be taken into account by individual tool vendors. As long as no industrial standard is released (e.g. by OMG), there will be no guarantee of convergence of the tools considering that aspect.

- rely on XMI[DI] [OMG 2003b] which was accepted as a final adopted specification of OMG in September 2003. According to the experience on the compliance of UML tools discussed in Section 2.1.3, it will take further years to align the individual metamodel interpretations to XMI and the individual graphical implementations to XMI[DI].

Diagrams provided from outside, in particular if they are given in XMI[DI], carry UML related structural and semantical information and layout specific information, like positions or sizes, to draw the diagram. Some metrics, like  $m_{rankAssign}$  or  $m_{rankShape}$ , require additional information on the edges partitions  $E_H$  and  $E_N$  as well as on the rank assignment  $\sigma$ . This can be achieved by reusing some steps of the preprocessing and rank assignment macro phases of the *SugiBib* algorithm as follows:

1. Create a UML specific input graph  $G_I$  from the external information. As mentioned in Section 4.2, layout information cannot directly be assigned to node and edge instances. Therefore, assign appropriate incremental layout information objects to nodes and edges. Determine the extents of adornments and other textual information and fix the data according to the information object life cycle shown in Figure 4.10.
2. Transform  $G_I$  directly to a result graph  $G_R$ . Thereby, the positions of the graph elements is determined from the information in the incremental layout objects.
3. Store the given identifiers of nodes and attach unique identifiers to all nodes of  $G_R$ .
4. Detect  $E_H$  by considering the type of edges and their positions only.
5. Let  $V' = \{v : v, w \in V, v \neq w, e = \{v, w\} \in E_H\}$  be the set of nodes connected by edges in  $E_H$ . Create the hierarchical subgraph  $G' = (V', E_H)$  of  $G_R$ .
6. Break cycles of  $G'$  by reversing appropriate edges (Section 4.4.11) and perform a basic rank assignment, e.g., by executing the network simplex algorithm as described in Section 4.5.4.
7. Transfer the rank information back into  $G_R$  by considering the unique identifiers on nodes.
8. Assign the remaining nodes of  $V \setminus V'$  to ranks by selecting the most appropriate rank considering positional information only. Additional ranks may be introduced in this step.
9. Restore the initial identifiers on nodes.

Via the metrics implementation, JMI and XMI[DI], *SugiBib* itself becomes its own measuring tool and represents the advent of the realization of our measurement dreams depending on the adaption of the standards and proposals by industry. Furthermore, the layout results of *SugiBib* can be analyzed by the metrics tool and recorded for a metrics based testing approach, which

will be described in Section 6.4.

Compared with more theoretic work from HCI, our approach might appear trivial: Different layouts influence, how many artifacts users have to compare and calculate. Combined with further measurements arising from reading and writing artifacts, the measurement of the visual complexity (ViCo) of diagrams was attacked in [Gärtner et al. 2002]. Even if it would be interesting to validate, if our measurements emphasize similar tendencies, the measurements of the visual complexity requires real user groups or at least a user simulation.

## 5.2 Layout Comparison

The wise man does not permit himself to set up even in his own mind any comparisons of his friends. His friendship is capable of going to extremes with many people, evoked as it is by many qualities.

Charles Dudley Warner (1829 – 1900)

In this section, we will compare implementations of other layout algorithms dedicated to UML class diagrams to the features of *SugiBib*. We decided to focus on GoVisual and yFiles (jarInspector/yWorksUML), which were briefly introduced in Section 2.2.3. Due to the disappointing results in [Eichelberger and von Gudenberg 2003b], we will not take layout mechanisms of CASE tools into account.

A comparison between *SugiBib* and jarInspector/yFiles was presented in [Eiglsperger 2003]. Both algorithms seemed to support the typical graph drawing properties discussed in multiple publications, e.g. [Battista et al. 1999; Waddle 2001]. The topology-shape-metrics approach supports a balanced aspect-ratio but increases the number of edge crossings, the hierarchical approach produces a lower height but a larger width of the drawing and usually admits less edge crossings. Aspects specific to UML class diagrams were not discussed in [Eiglsperger 2003].

For our evaluation, we tried to get implementations of these two layout tools, but both did neither provide an editor for UML class diagrams nor an appropriate input mechanism, e.g., for XMI. Therefore, we decided to use a CASE tool, for which layout plug-ins realized by GoVisual and yFiles are available. We considered Genteware Poseidon as CASE tool, because it also provides input and output of the UML model in XMI[DI] as a common format, which is also supported by *SugiBib*<sup>2</sup>.

To do a fair comparison between the three layout algorithms, we first have to identify common features. This will be done by analyzing the compliance to those of our aesthetic principles from Section 3.3.6, which are closely related to the UML specification, i.e., the mandatory criteria and two optional criteria. Therefore, several diagrams, each reflecting the specific aspects of an individual aesthetic criterion, were specified in Poseidon and laid out by both plug-ins, always taking the manual drawing as input for a layout plug-in. These diagrams can also be laid out by

---

<sup>2</sup>Many thanks to Per Pascal Grube and Dian Dochev for the tricky XMI[DI] implementation, which was realized even if specification and documentation of the used components admitted reverse engineering tasks.

*SugiBib*, due to the support of XMI[DI]. Then, the layout results of diagrams, which share these common features, can be compared.

We used Poseidon 2.5 Professional Edition with GoVisual 1.3 and yWorksUML 1.1 as layout plug-ins<sup>3</sup>. In contradiction to the online information on the plug-ins, GoVisual 1.4 and yWorksUML 1.3 did not run on Poseidon 2.5. Furthermore, we tried Poseidon 3.0 Professional Edition with an evaluation license to also consider the newer versions GoVisual 1.5 and yWorksUML 1.4. Unfortunately, we were not able to run GoVisual 1.4 on Poseidon 3.0 due to a Java exception. yWorksUML 1.4 produced the same results as yWorksUML 1.1 on Poseidon 2.5.

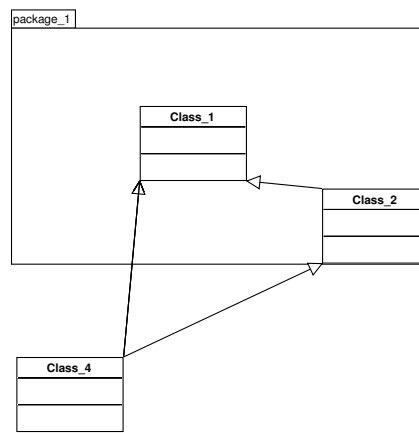


Figure 5.8: Layout of contained elements in yWorksUML. Two associations are hidden below the generalization between `Class_1` and `Class_4`. A similar result was produced by GoVisual.

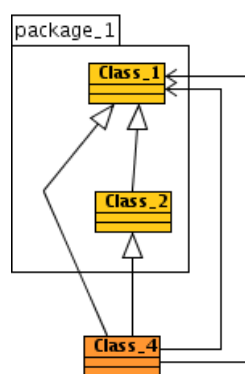


Figure 5.9: Layout of Figure 5.8 by *SugiBib*.

<sup>3</sup>Many thanks to the Genteware Company for providing the appropriate licenses for the test.

	GoVisual 1.3	yWorksUML 1.1/1.4	SugBib
multiple edges (REQ_GRAPH_TYPE)	✓	✓	✓
UML_HIERARCHY	inheritance	inheritance, realization	multiple categories prepared
UML_SPATIAL	global distances	no options	✓
UML_SEMANTIC_CLUSTERS	top-level nodes only	top-level nodes only	✓
UML_MEDIAN	–	–	✓
UML_JOIN	shared (inheritance)	shared/separate	separate (hierarchical)
UML_NODES	size and interior by CASE tool	size and interior by CASE tool	✓
UML_CONTAINER	–	–	✓
UML_CLASS	✓	✓	✓ (scaling)
UML_CENTER	✓ (lower priority)	✓	✓
UML_EDGES	✓	✓	some overlays occur
UML_ASSOCIATIONCLASSES	∇	as usual nodes, overlays	✓
UML_HYPEREDGES	○	○	✓
UML_REFLECTIVE	–	overlays at vertical	✓
UML_ADORNMENTS	✓	as usual nodes	✓
UML_COMMENTS	as usual nodes, ok at edges	somewhere on the grid	✓
UML_DISCONNECTED	somewhere on the grid	○	borders of the drawing optional
UML_EDGECROSSING_SYMBOL	○	○	

Table 5.3: Feature matrix of current UML layout tools. Most optional aesthetic criteria were not considered in this test. ○ signals that the CASE tool does not support the UML feature, ∇ denotes that the plug-in accidentally shuts down the CASE tool.

The GoVisual plug-in provides several layout methods like an UML specific one, orthogonal layout, incremental layout, hierarchical layout, symmetric layout and tree layout. Due to the structure of a UML class diagram, oftentimes only the UML specific one and the incremental layout can be applied. Furthermore, the plug-in provides various user options like component distance, class coloring, preferred orientations or several importance preferences. yWorksUML provides a method for entirely or incrementally laying out a UML class diagram.

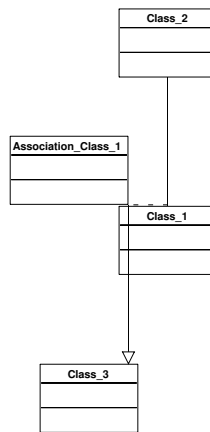


Figure 5.10: Layout of an association class in further relations. GoVisual produced an accidental shutdown of Poseidon.

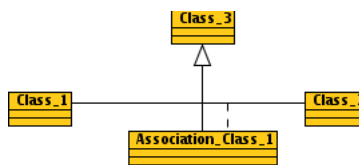


Figure 5.11: Layout of Figure 5.10 by *SugiBib*. A perfect drawing according to the default UML layout style.

Furthermore, options like overlap-free label placement, preferred directions and shared or individual target style on some kinds of edges can be specified by the user.

Table 5.3 shows the summary of the results of the compliance analysis for the layout algorithms. In general, the UML features of GoVisual and yWorksUML are directly aligned to the capabilities of the CASE tool. Unfortunately, Poseidon does not provide a complete UML implementation, e.g., n-ary associations, anchors, hyperedges or the edge crossing symbol are missing in both versions. Poseidon 2.5, which is dedicated to UML 1.x, solely provides packages but not models or subsystems as model management elements. Contained elements can visually be specified in Poseidon.

In particular, the differences shown in Table 5.3 arise from realizing different philosophies: Both approaches from the graph drawing community lay strong emphasize on a more graph



layout specific viewpoint, e.g., by introducing software related hierarchies as constraint into a non-hierarchical layout algorithm. As a different viewpoint, our approach is based on the default layout style implicitly suggested by UML and derived aesthetic criteria discussed in Section 3.3.6.

Both layout plug-ins strongly emphasize a hierarchical orientation for certain edges. In GoVisual, only inheritance edges are considered for a hierarchical layout, the other edges are drawn in orthogonal, non-hierarchical fashion. In yWorksUML, inheritance and realization relations can be laid out in hierarchical style, the other edges are then drawn in rectilinear grid fashion with vertical direction.

In GoVisual as well as in yWorksUML, packages are treated as top-level nodes, e.g., like usual classes. Contained elements are not considered as individuals by the layout algorithms and in the layout result, nodes appear to be unbalanced and edges overlay each other as depicted in Figure 5.8.

We have also tested association classes. It seems that yWorksUML treats an association class like an usual class and produces overlappings as shown in Figure 5.10. GoVisual accidentally produced a shutdown of Poseidon so that the user was not able to store the current work. Surprisingly, GoVisual was able to handle a comment at an association. In other situations, GoVisual as well as yWorksUML treated comments as usual nodes.

On GoVisual as well as on yWorksUML we found that the size of nodes are kept as specified. Instead of scaling individual elements, bends and additional layers are introduced. In particular, yWorksUML partly produces larger diagrams than necessary. In fact, yWorksUML provides more options on the treatment of the semantics of edges than GoVisual, e.g., which kinds of edges to be considered for hierarchical aspects, while GoVisual considers inheritance but no realization edges for hierarchy only. Both programs provide an incremental layout method, which oftentimes produces unpleasing artifacts like overlays or non-orthogonal edges on prerouted edges, e.g., after importing a XMI[DI] file written by *SugiBib*. In yWorksUML, oftentimes also the non-incremental layout mechanism produced unpleasing drawings due to prerouted edges, e.g., to obtain Figure 5.13 all bends were removed from the diagram before executing the layout algorithm.

Hence, the set of common features with respect to Poseidon as underlying CASE tool are simple class diagrams containing classes, various types of edges but no packages, n-ary associations, hyperedges, etc.

As a summary, Table 5.4 shows the judgment of the layout results according to the layout metrics defined in Section 5.1. The values show that the metric formulae are able capture the layout situation of the shown diagrams. For GoVisual and yWorksUML,  $m_{overlap}$  forced 0 as result on the package diagram, because the associations were overlaid by an inheritance edge. The same is true for the association class test on yWorksUML. On that diagram, the hierarchy detection of the metrics calculation considers association edges as hierarchy. Therefore,  $m_{rankAssign}$  also returns a perfect measurement.

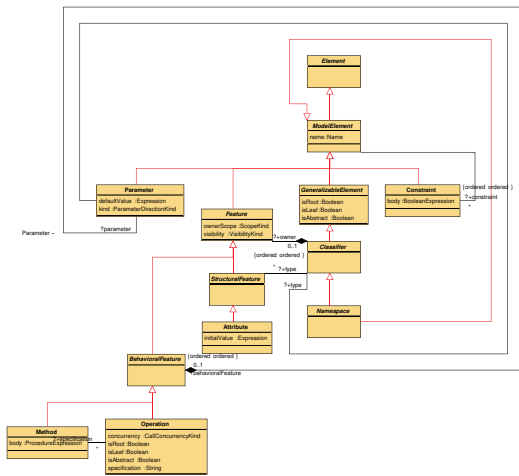


Figure 5.12: Layout of a simple class diagram by GoVisual.

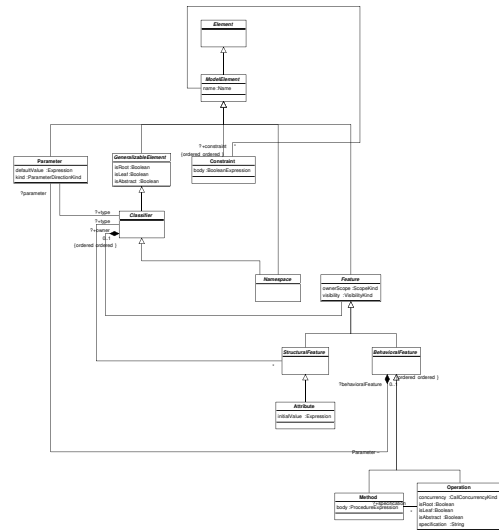


Figure 5.13: Layout of Figure 5.12 diagram by yWorksUML.

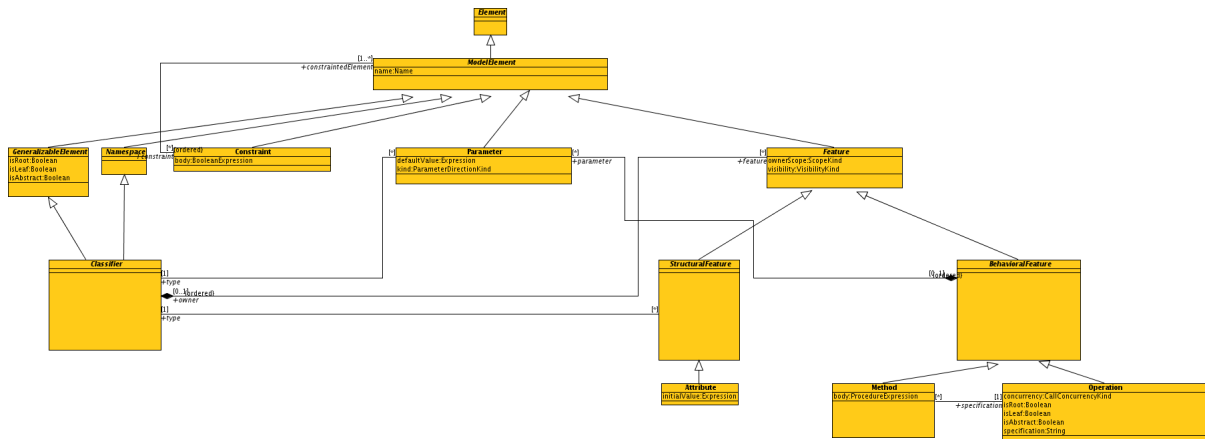


Figure 5.14: Layout of Figure 5.12 or Figure 5.13 by SugiBib.

metric	package Figure 5.9		association class Figure 5.11		simple diagram Figure 5.14		
	GoVisual	yWorksUML	SugiBib	yWorksUML	SugiBib	yWorksUML	SugiBib
$m_{cross}$	1.0	1.0	1.0	1.0	1.0	0.86	0.99
$m_{rectilin}$	0.0	0.0	1.0	1.0	1.0	1.0	1.0
$m_{bends}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$m_{edgeLengths}$	1.0	1.0	0.82	1.0	0.67	0.93	0.87
$m_{median}$	0.81	0.81	0.98	0.94	0.98	0.96	0.98
$m_{assocClass}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$m_{comment}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$m_{rankAssign}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$m_{rankShape}$	1.0	1.0	1.0	1.0	0.92	1.0	0.96
$m_{overlap}$	0.0	0.0	1.0	0.0	0.98	1.0	1.0
$m_{cluster}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$m_{aggregated}$	0.0	0.0	0.98	0.0	0.95	0.87	0.97
$m_{weighted}$	0.0	0.0	0.99	0.0	0.97	0.81	0.98
$m_{sqWeighted}$	0.0	0.0	1.00	0.0	0.97	0.77	0.99

Table 5.4: Metric values of the diagrams shown in this section. Parameters and priority weights were chosen to be the same as in Table 5.2. No values for the GoVisual version of Figure 5.11, because the layout plug-in accidentally caused Poseidon to terminate.

For the simple class diagram, the values of GoVisual are low due to the long association routes and the inheritance edge from Namespace to ModelElement, which was laid out against the general bottom-up flow for hierarchical edges. The values of *SugiBib* are slightly lower than the values of yWorksUML, because of the node scaling feature:  $m_{edgeLengths}$  and  $m_{rankShape}$  return better values for smaller, equal sized nodes.

Figure 5.12 up to 5.14 show the drawing of a simple class diagram by GoVisual, yWorksUML or *SugiBib*, respectively. The drawing by yWorksUML and *SugiBib* reveal a clear hierarchical structure of inheritance edges, while the drawing by GoVisual produces a rectangular drawing with without consequently considering the semantics of the edges.

One-layer layouts, which were discussed as design quality indicators in Section 3.3.7, were not considered in this comparison.

In conjunction with Section 2.2.3, we can conclude that GoVisual and yWorksUML pay more attention to graph drawing aspects of UML class diagrams than to specific structural and semantical issues of UML class diagrams. Even if our aesthetic criteria in Section 3.3.6 might appear as artificial at a first glance, some main aspects are also considered by graph drawing tools like yWorksUML. Hence, we can conclude that our layout algorithm realized by *SugiBib* provides a unique set of aesthetic principles deduced from UML, graph drawing, HCI, software engineering and software visualization.

In comparisons of layout algorithms, e.g., in [Eiglsperger 2003], often the runtime allocated by the individual implementation is measured. In the next section, various results for *SugiBib* will be given. To also provide an impression of the runtime allocation of the three implementations considered in this section, we have laid out a diagram with initial 67 classes and 105 relations: *SugiBib* (Java) took 2080 ms, yFiles (Java) 19887 ms and GoVisual (native) 71196 ms, whereby the general features of the drawings reveal similar properties as in Figure 5.12 up to 5.14.

## 5.3 Runtime Measurements

Effort moves toward whatever is measured.

DeMarco's Principle

In Chapter 4, we have intensively discussed practical and theoretical aspects of our layout approach for UML class diagrams. In this section, runtime values for each macro phase of the algorithm will be presented. From Section 4.6.5, the question, which concrete crossing reduction algorithm is appropriate for which kind of graph, was left open. Therefore, in this section, we will make a decision on finding the most appropriate edge crossing reduction strategy for UML class diagrams depending on the runtime of the implementation and the number of edge crossings produced by individual strategies.

While the other measurements in this chapter were application domain specific metrics, in this section we will focus on usual measurements from algorithm theory and graph drawing, namely the runtime of parts of the implementation as well as the number of edge crossings both related to the number of nodes or the density (the number of edges related to the number of nodes).

On finding the most appropriate edge crossing reduction strategy, we might rely on general statements in literature like:

On graphs that are more random and/or have several nodes of high degree, barycenter does better than median (high-degree nodes need to be centrally located with respect to their neighbors). Jünger and Mutzel [Jünger and Mutzel 1997] observed that the barycenter heuristic does better than the median on random graphs of various sizes and densities.

[Stallmann et al. 2001]

Such statements may apply to general graphs, but on application domain specific graphs like UML class diagrams, other results might be obtained. Therefore, on a set of test class diagrams, most of the edge crossing reduction algorithms realized in *SugiBib* will be taken into account for a runtime and crossing number evaluation on UML class diagrams.

Even if *SugiBib* still provides the implementation of the barycenter and the median method from the first version, both cannot be taken into account in general, because both do neither ensure cluster-validity nor consider the node naming function. Therefore, we can select from the extended barycenter approach (with intertwined or postprocessing enforcement of the cluster validity), the extended intertwined or postprocessing median method with transpose heuristics and our hierarchical approach.

The measurements have been collected running *SugiBib* on the standard JVM included in the SUN JDK 1.5.0 and a Pentium 4, 3.0 GHz with 2 GByte main memory on SuSE Linux 9.1. 175 diagrams have been considered for runtime measurement. One of the diagrams is displayed in Figure 5.15, further drawings will be presented in the appendix.

Figure 5.16-5.19 show the number of visible edge crossings in the layout results. We have also registered the results of the backtracking approaches, which were briefly mentioned in Section 4.6.5. The  $n$ -level backtracking algorithm [Eichelberger and von Gudenberg 2003a] visits all nodes of a graph in a depth-first traversal on the hierarchical edges and permutes only nodes in the current subtree. Unfortunately, this strategy is not able to change the initial sequence of clusters due to issues of consistency. The deep  $n$ -level backtracking algorithm exchanges subtrees instead of individual nodes and is able to revisit all hierarchical children after a positional change of a node.

From Figure 5.16-5.19 we can conclude that on most graphs in the test set the hierarchical method calculates the lowest number of crossings, but also the extended median method (with postprocessing cluster enforcement) and the deep  $n$ -level backtracking method deliver noteworthy results.

In Figure 5.20-5.23, the runtime allocated by the edge crossing algorithms is depicted. On small graphs the  $n$ -level backtracking is faster than the other strategies, but on larger graphs it takes an enormous time to compute a result. As predicted in Section 4.6.5, the median method outperforms the other methods in runtime.

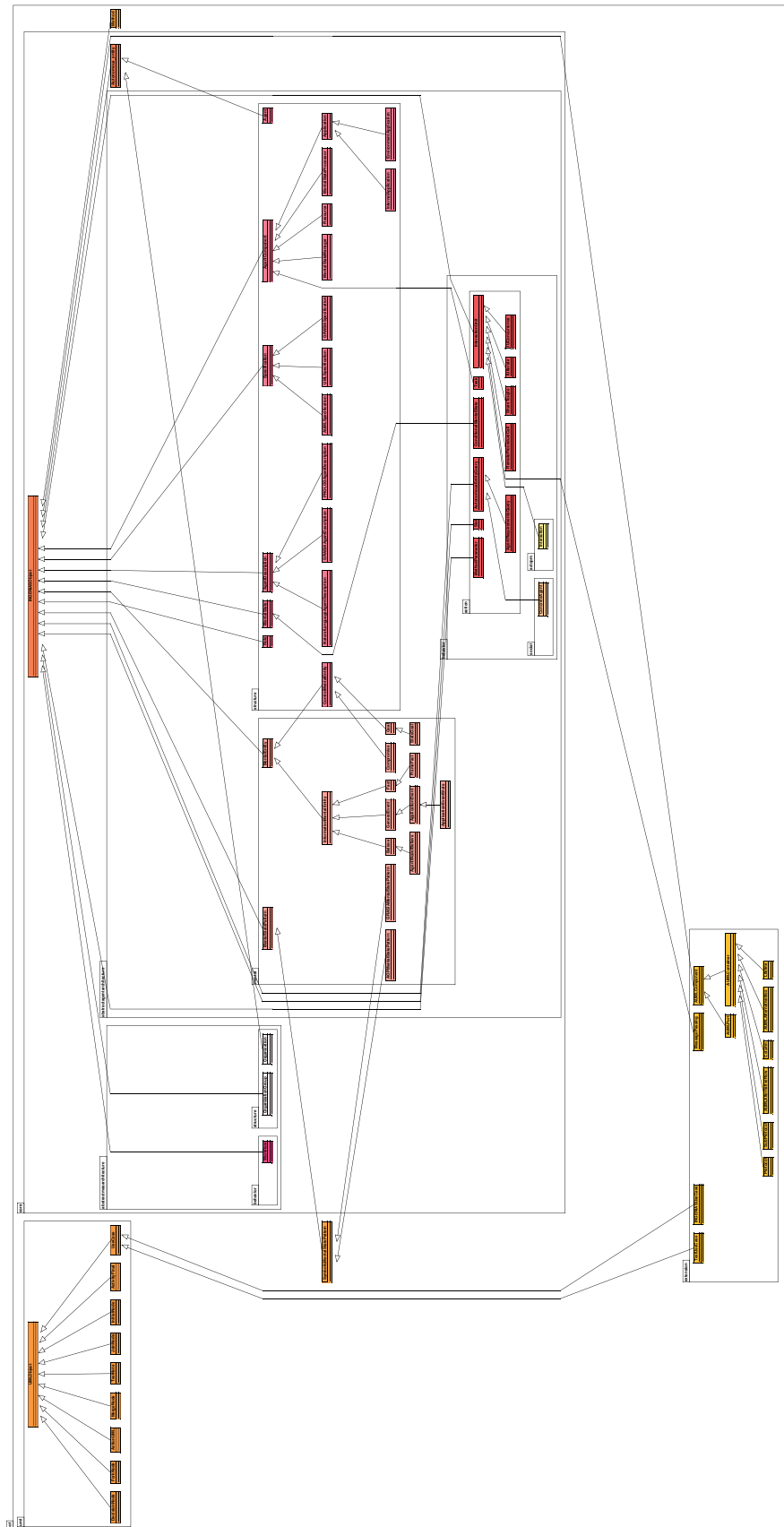


Figure 5.15: *SugiBib* drawing of a compound diagram with 90 visible nodes used for runtime measurements. Edge crossing reduction was done by the hierarchical method, hopping of hidden and visible nodes was activated.

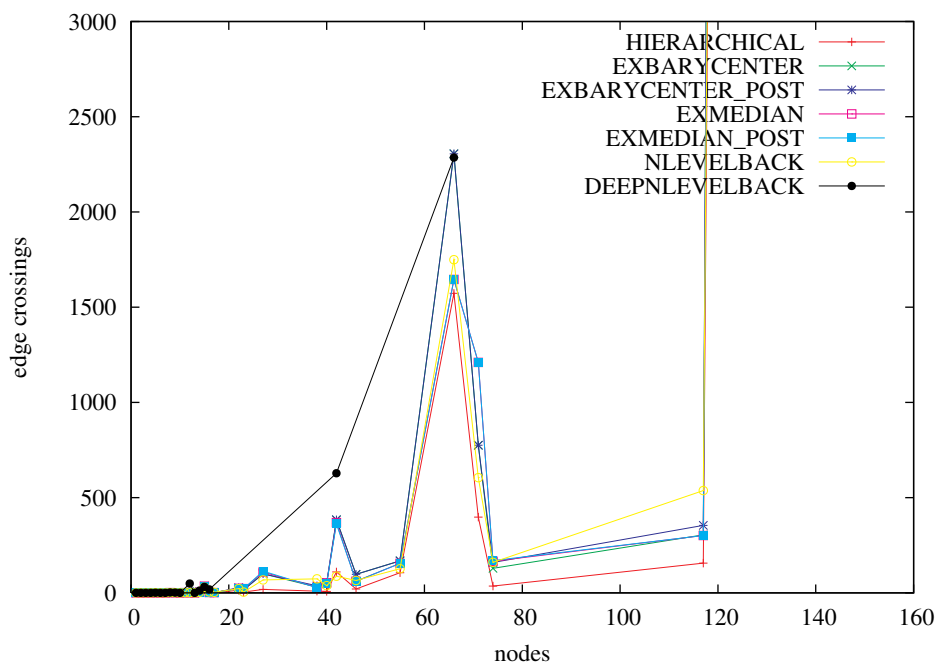


Figure 5.16: Various edge crossing reduction algorithms on non-compound graphs by nodes.

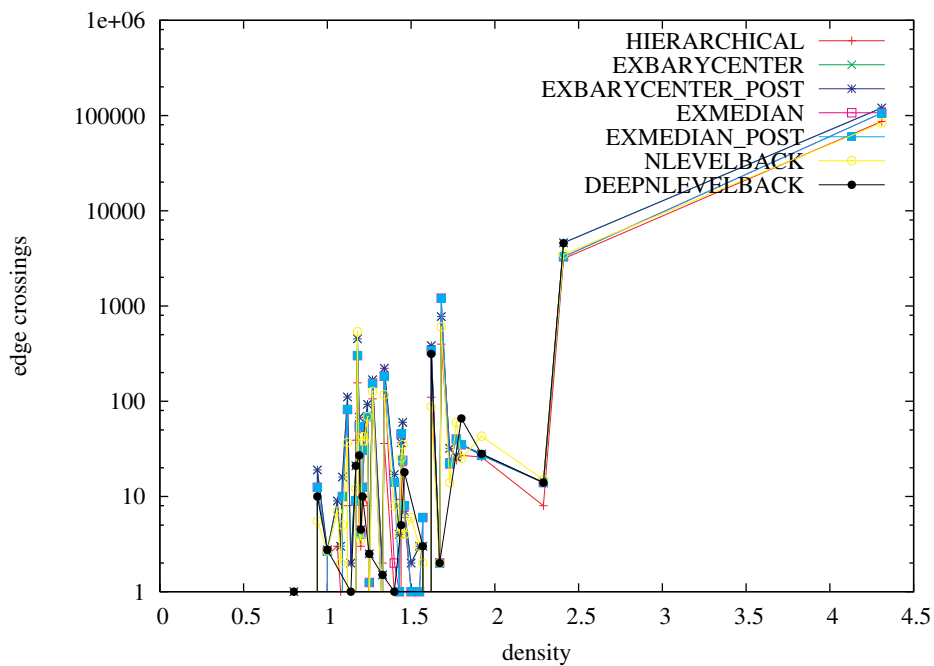


Figure 5.17: Various edge crossing reduction algorithms on non-compound graphs by density.

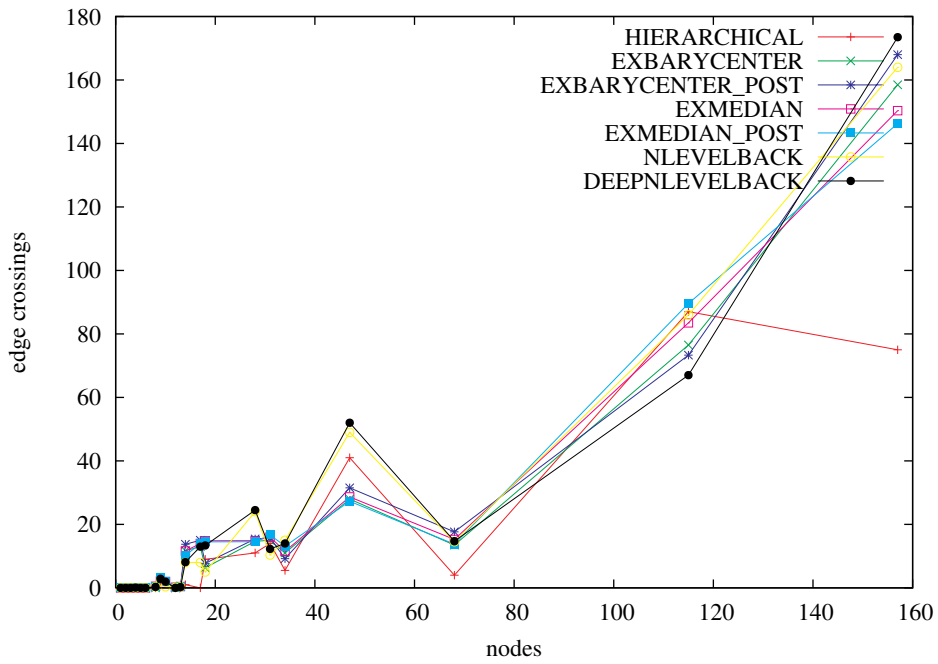


Figure 5.18: Various edge crossing reduction algorithms on compound graphs by nodes.

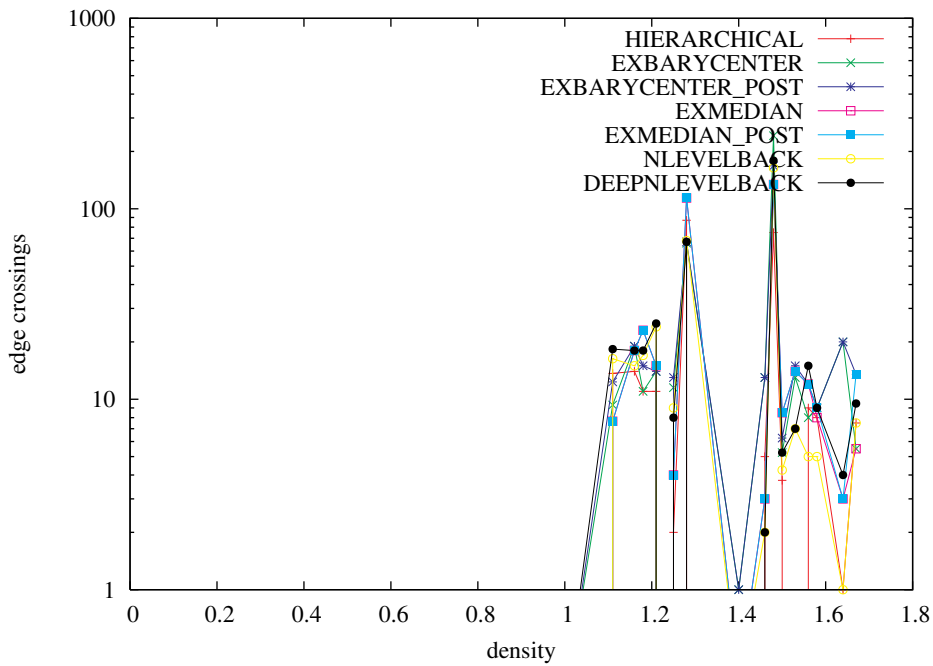


Figure 5.19: Various edge crossing reduction algorithms on compound graphs by density.



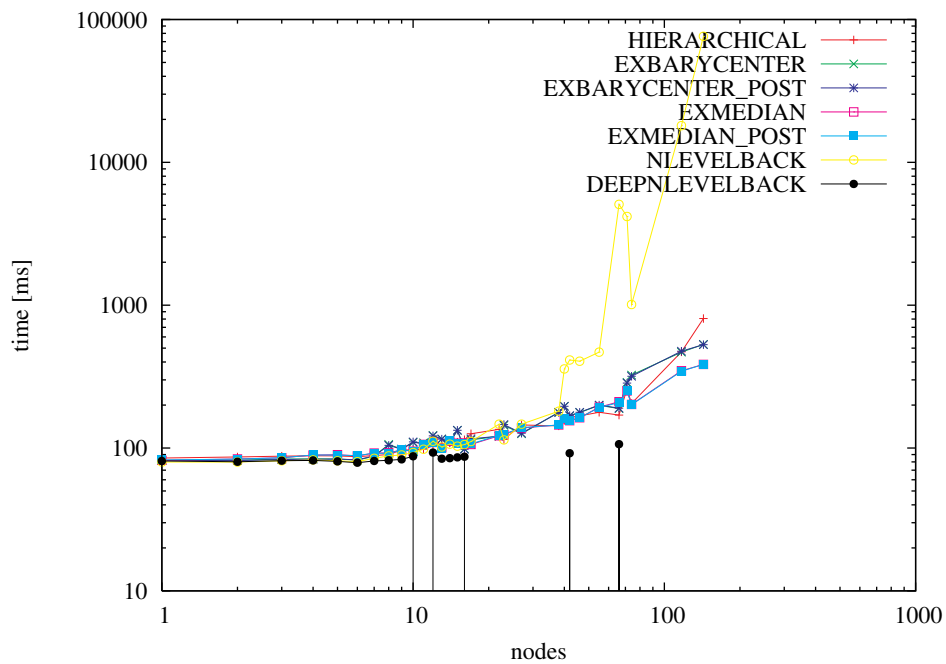


Figure 5.20: Runtime of various edge crossing reduction algorithms on non-compound graphs by nodes.

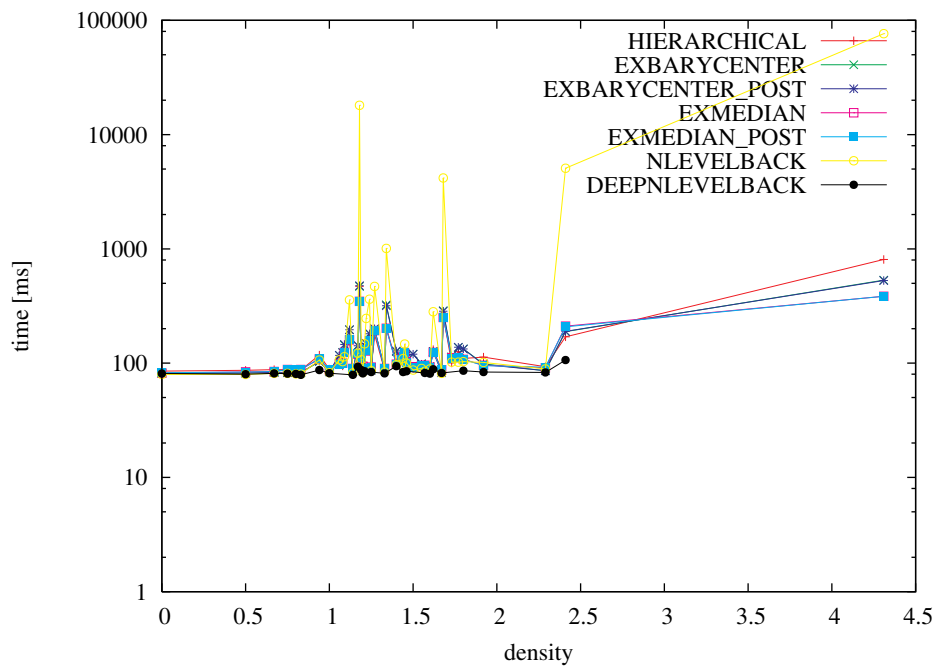


Figure 5.21: Runtime of various edge crossing reduction algorithms on non-compound graphs by density.

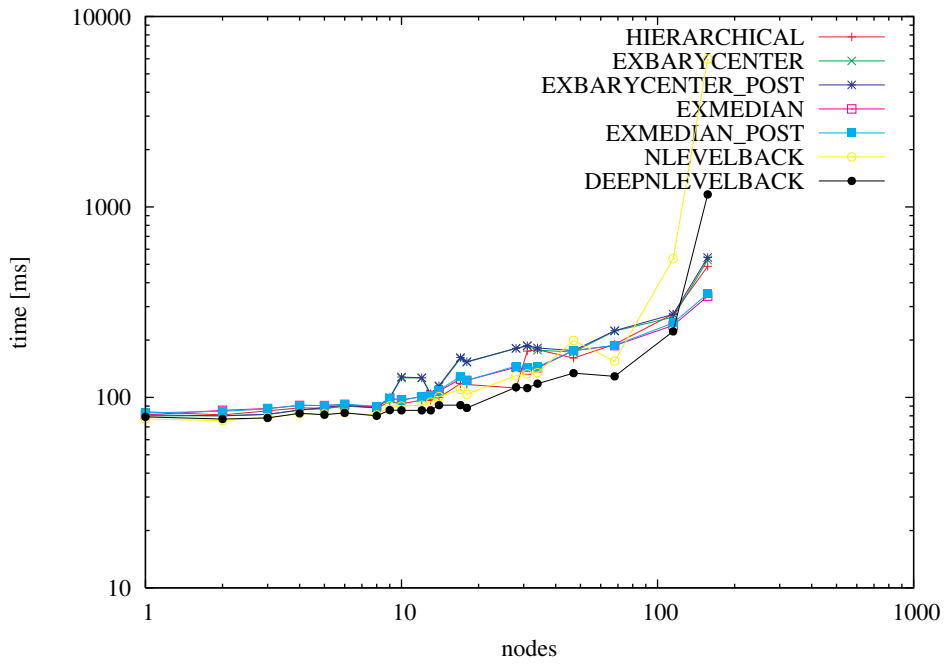


Figure 5.22: Runtime of various edge crossing reduction algorithms on compound graphs by nodes.

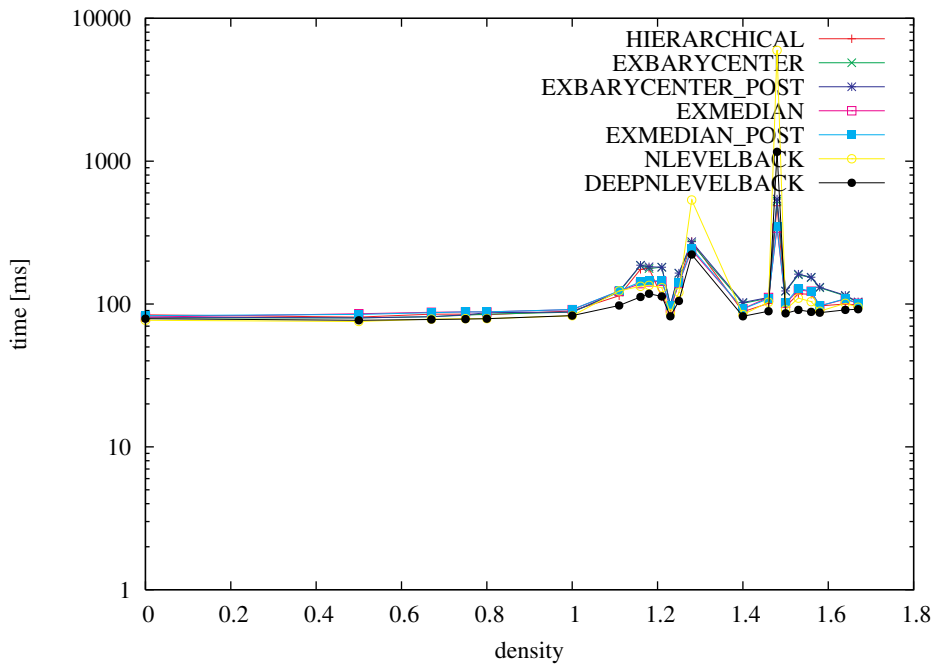


Figure 5.23: Runtime of various edge crossing reduction algorithms on compound graphs by density.

Considering runtime and crossing results from the test diagrams, we recommend to suggest two strategies to the user: A fast crossing reduction with a probably higher edge crossing number by the extended median method with postprocessed cluster alignment and the hierarchical method for higher quality and longer runtime.

The theoretical runtime complexities have been summarized in Table 4.8. As mentioned in Section 4.10, the theoretical results may differ from the effective runtime, which can be retrieved from a concrete implementation. Therefore, in Figure 5.24-5.37, the results of the runtime measurements for each individual macro phase related to the number of nodes or to the density, respectively, are depicted.

The predicted runtime complexities fit to the measurements, even if the values for S11 and S15 seem to have an exponential component. In particular, Figure 5.28 and Figure 5.32 appear to be critical. Currently, not the entire source code of *SugiBib* has been revised according to optimization issues. Only parts have been tuned, and we believe, that inefficient code from ancient versions is responsible for these effects. In the initial measurements, all macro steps required a certain setup time, which was consumed on each individual processing step by the graph copy mechanism and by at least one main loop searching for the nodes or edges to be processed.

In the current version of *SugiBib*, the graph copy mechanism as well as certain processing steps are executed only, if they are required to process the concrete input graph. For example, S7, S12 and S16 are not executed, if no association classes are given in the input graph. Therefore, the preprocessing macro phase (Figure 5.24-5.25), the intermediary macro phase (Figure 5.30-5.31) and the postprocessing macro phase (Figure 5.34-5.35) show a high variation in the runtime measurements compared with the other macro phases. Furthermore, the entire runtime of the algorithm was reduced. Further source code and runtime optimizations will be discussed in Section 6.5.

A significant runtime improvement without modifications to source code or the algorithms of the framework can be obtained, by processing the Java byte code with a native compiler. A native compiler produces an executable specific to a certain platform. Thereby, the platform independence of the Java byte code is stripped off, various additional optimizations for native code can be applied and security issues like different class loaders or runtime checks on the integrity of the byte code can be removed or tuned. The program itself remains platform independent, because the byte code can still be executed on a JVM, but on certain platforms, a significant speedup can be obtained easily by running the native code. In [Eichelberger and von Gudenberg 2004], we have shown that the runtime of a native compiled version of *JTransform*, our object oriented Java source code processing framework, can be improved by more than 50%. We conducted the native measurements on a Pentium 4, 3 GHz, 2 GByte main memory under Microsoft Windows XP Professional, because the current version of the native compiler JET<sup>4</sup> is unfortunately not able to generate native code for *SugiBib* under Linux. JET 3.60 professional was prepared for a SUN JDK 1.4.2, because JET is currently not able to process byte code for Java 5.

---

<sup>4</sup><http://www.excelsior-usa.com>

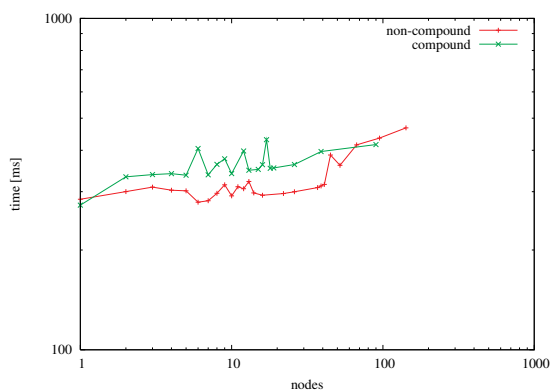


Figure 5.24: Preprocessing by nodes.

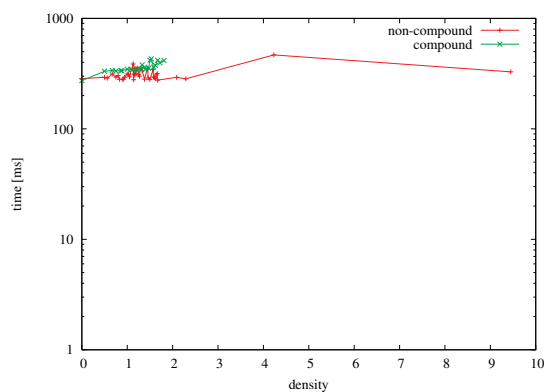


Figure 5.25: Preprocessing by density.

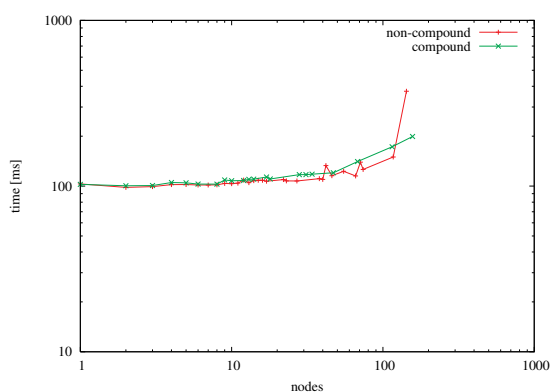


Figure 5.26: Rank assignment by nodes.

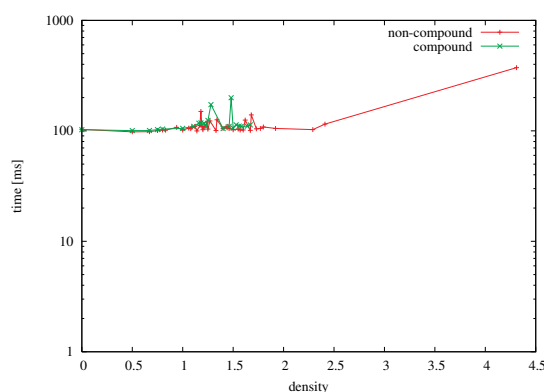


Figure 5.27: Rank assignment by density.

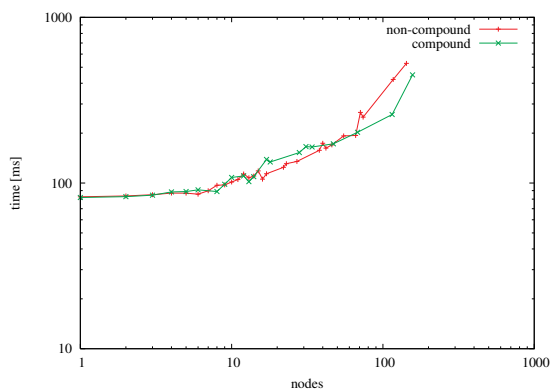


Figure 5.28: Edge crossing reduction by nodes (mean values).

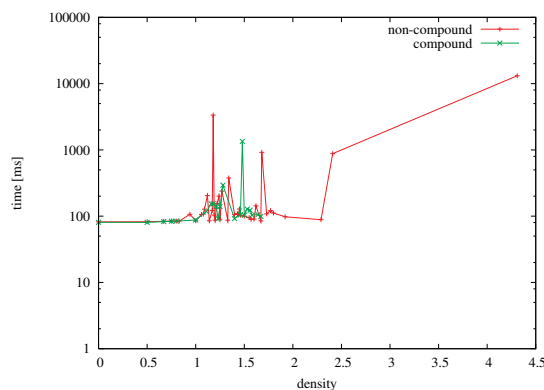


Figure 5.29: Edge crossing reduction by density (mean values).

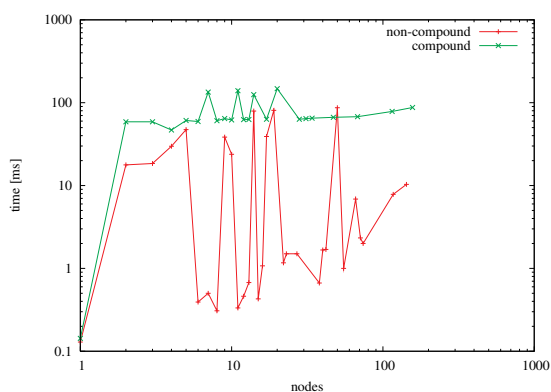


Figure 5.30: Intermediary processing by nodes.

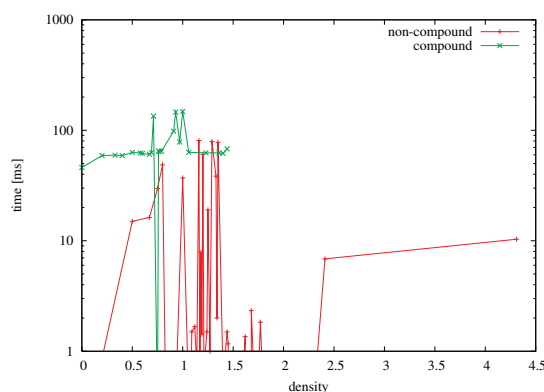


Figure 5.31: Intermediary processing by density.

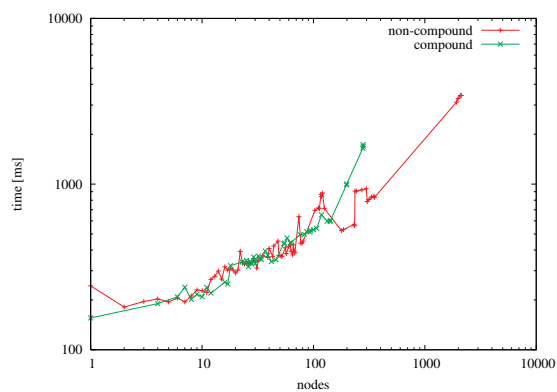


Figure 5.32: Coordinates assignment by nodes.

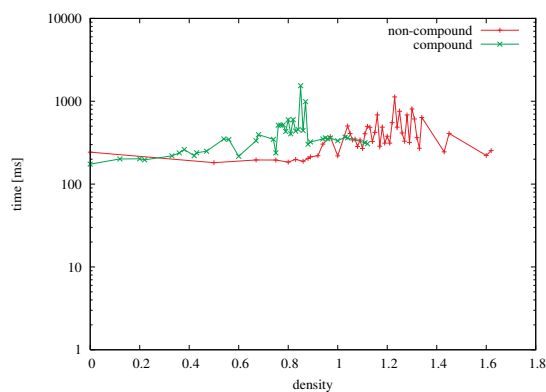


Figure 5.33: Coordinates assignment by density.

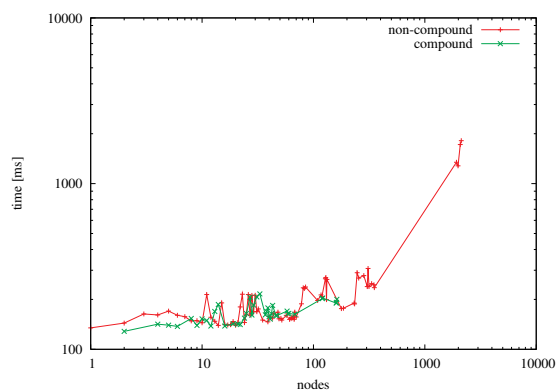


Figure 5.34: Postprocessing by nodes.

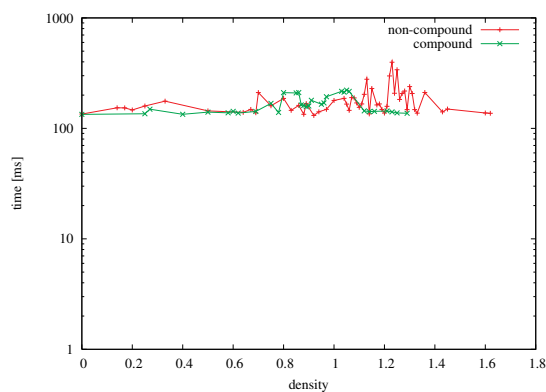


Figure 5.35: Postprocessing by density.

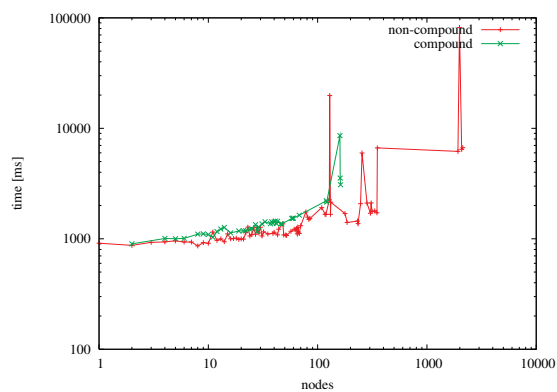


Figure 5.36: Entire algorithm by nodes.

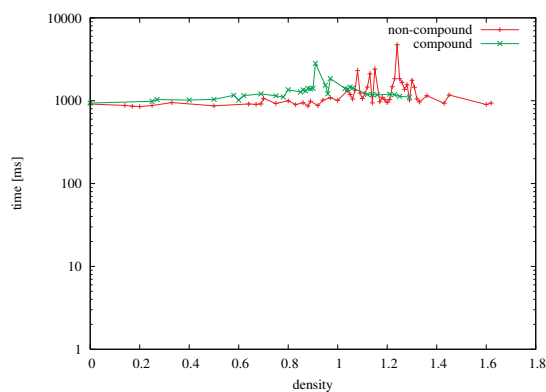


Figure 5.37: Entire algorithm by density.

To run the same measurement scripts as on Linux, cygwin<sup>5</sup>, which provides a Linux bash environment for Windows, was also installed. Figure 5.38 and

<sup>5</sup><http://www.cygwin.com>

5.39 show similar results as in Figure 5.36 and 5.37, but the concrete runtime is approximately 10% of running *SugiBib* as an interpreted Java program.

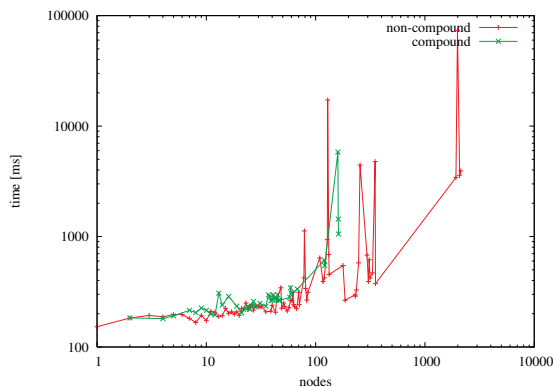


Figure 5.38: Entire algorithm by nodes (native).

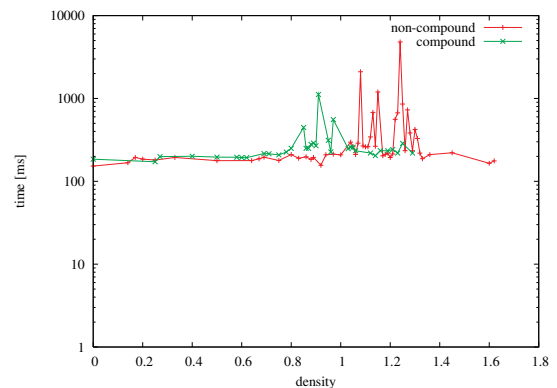


Figure 5.39: Entire algorithm by density (native).

So far, only measurement results on graphs with maximum 1923 nodes and 2396 edges (initially 142 nodes and 601 edges) were presented. To also show scaling effects for large graphs, we will now discuss the measurement results on a class diagram of *SugiBib* itself. The input file in UMLscript format was produced by the reverse engineering application of *JTransform* [Eichelberger and von Gudenberg 2004].

Table 5.5 shows the results of the measurement. Unfortunately, we were not able to collect runtime results of the native executable due to errors in the memory management mechanism of JET.

After running the layout algorithm, the number of nodes in the non-compound graph grows by factor 4.2 and in the compound graph by factor 11. Comparing the numbers of edge crossings and edge-compound crossings in the layout result, on that large non-compound graph, the hierarchical crossing reduction performs better than the barycentric and the median method (in the given sequence). The runtime of the algorithm increases by factor 13.6 using the hierarchical method instead of the median method or by factor 3.3 using the barycentric method instead of the median method. On the compound graph, also the hierarchical method performs better than the postprocessed barycentric, intertwined barycentric and both median variants (in that sequence). The runtime of the algorithm increases by factor 5.1 using the hierarchical method instead of the median method or by factor 6.6 using the barycentric method instead of the median method.

We can conclude that the choice of the appropriate edge crossing reduction mechanism heavily

<b>macro phase</b>	<i>SugiBib</i> version 1.33 863 nodes, 1318 edges	<i>SugiBib</i> version 1.33 (cluster) 1063 nodes, 2380 edges
preprocessing	1227	1643
rank assignment	675	10590
hierarchical method	2775806	5180007
intertwined barycenter	657445	7105939
postprocessed barycenter	660683	7061713
intertwined median	182739	3911368
postprocessed median	183357	3927547
intermediary processing	1170	7022
coordinates assignment	17653	1057978
postprocessing	2296	15211
complete: hierarchical method	2799741	6322691
complete: intermediary barycenter	680561	8193137
complete: postprocessed barycenter	683503	8119320
complete: intermediary median	205764	1294759
complete: postprocessed median	205589	1246766
crossings: hierarchical method	19669	59987 (4849)
crossings: intertwined barycenter	90524	180853 (20074)
crossings: postprocessed barycenter	90524	155542 (13729)
crossings: intertwined median	117368	152417 (20433)
crossings: postprocessed median	117368	152417 (20433)

Table 5.5: Runtime values of UML class diagrams visualizing *SugiBib* itself with and without packages. The crossing numbers are collected immediately after executing the individual crossing reduction algorithms. Compound crossing numbers are denoted in braces.

depends on the structure and the properties of the input graph. With a test set, which also contains several large graphs like the one used to obtain the results in Table 5.5, a break-even point between the individual crossing methods might become obvious. Also the runtime of the hierarchical method might be improved by further incremental algorithms, in particular those for the cluster consistency.

# 6 Implementation

---

In this chapter, we will discuss several aspects of the realization of our layout algorithm by the framework *SugiBib*. Beside knowledge on the layout algorithm, details on the concrete implementation are required to maintain, modify and extend the framework.

First, we will introduce the basic architecture of the framework. A general core framework will realize more common, reusable parts of the layout algorithm. A specialized extension will configure the core framework for UML class diagrams. Details on the general and the specialized parts of the 7 macro steps of the layout algorithm, which were introduced in Table 4.1, will be outlined in Section 6.2. In the next section, further details on concrete applications, which can be realized by configuring a basic application library, as well as the implementation of the layout metrics introduced in Section 5.1 will be given. Finally, issues on testing, runtime optimization and anticipated extensions to the framework and the layout algorithm will be described.

The layout results of some of the diagrams in this chapter automatically drawn by *SugiBib* will be presented in the appendix.

## 6.1 Architecture of *SugiBib*

Good architecture is where the whole is greater than the parts.

Mies van der Rohe (1886 – 1969)

So far we have discussed diagrams, disciplines and algorithmic principles for drawing UML class diagrams automatically, but, except for the basic introduction into our algorithm and its graph model, we did not describe main design aspects of an implementation. In this section, will introduce basically the main design aspects of the implementing framework, because the next chapter will discuss intensively the individual algorithmic steps and, as an overview, issues of the implementation will be given.

Fortunately, the basic design aspects of *SugiBib* [Eichelberger 1999] meet most of our basic requirements listed in Section 3.1. To easily fulfill REQ\_PLATFORM, we decided in 1999 to implement the framework in Java, an object-oriented, platform independent programming language. Even if in the late 1990th the advent of Java as a popular programming language was at



the beginning, we strongly believed that additional libraries for generic printing, image formats, database or repository access, which were not available in these days, will appear soon.

name	language	KLOC <sup>1</sup>	references <sup>2</sup>
AGD	C++		[Jünger et al. 2003]
Boost Graph Library 1.31.0	C++	570	[Siek et al. 2002]
GGCL	C++, STL		[Lee et al. 1999]
GTL	C++		[Forster et al. n. d.]
HGV 1.1.0	C++	5	[Raitner 2002]
LEDA 4.5	C++	230	[Mehlhorn and Näher n. d.]
Netlib			<a href="http://www.netlib.org">http://www.netlib.org</a>
Stanford GraphBase	C, T <sub>E</sub> X	37	[Knuth 1993]
yFiles	Java		[Wiese et al. 2003]

Table 6.1: Graph drawing libraries.

In other programming languages, like C++, some of the required libraries existed, but the usage on different platforms requires a strong discipline of the programmer. In Java many features for platform independent programming are basically provided by the language itself, unless certain rules have to be respected to meet REQ\_PLATFORM. Similar languages like .NET did not exist at that time.

Before designing a completely new graph drawing library or framework from scratch, existing work should be taken into account. In Table 6.1 and 6.2 some of the known graph drawing libraries and systems are listed. A more detailed overview was given in [Willhalm 2001]. Even if nowadays some systems are implemented in Java, in 1999 only few of them existed and some of those systems are not maintained anymore today. Furthermore, some of these systems are (now) commercial and at least academic licenses are required. Despite of the fact that we could have based our work on one of these systems, we decided to design a new one. Considering some of the non-graph-drawing features, which will be mentioned below and will also be discussed in Section 6.1, we are convinced that it was an adequate decision.

In Figure 6.1 the most relevant modules (packages) of *SugiBib* are depicted. Some classes are given as examples to show important relations. Parts of the graph model described in Section 4.2 can be found in the upper left corner, in the package `sugi::admin`<sup>3</sup>. Graph as a representant of the basic graph, its nodes and edges, the storage mechanisms for nodes and all the basic interfaces implemented by these classes are located in `sugi::admin`. The interfaces of the information to be attached to realize different kinds of nodes, edges and the external options on a graph are member of `sugi::admin::elementInformation`, the information due to incremental layout can be found in `sugi::admin::incrementalLayout`.

<sup>1</sup>Kilo Lines Of Code; taken from literature/WWW or calculated from source if available

<sup>2</sup>The most recent references are shown only.

<sup>3</sup>We use the fully qualified UML notation for design issues instead of the dot notation known from Java.

The package also provides interfaces introducing the classes of the basic algorithmic steps according to the Sugiyama algorithm. Furthermore, the implementation of the algorithms responsible for cluster validity, common constants, basic matrices for the realization of some edge crossing reduction algorithms, resource handling according to I18N<sup>4</sup>, a common plug-in mechanism and utilities like object pools and formatting helpers are part of `sugi::admin`.

name	language	KLOC <sup>5</sup>	references <sup>6</sup>
aiSee			<a href="http://www.aisee.com">http://www.aisee.com</a>
CABRI			[Carbonneaux et al. 1996]
DAG, DynaDag			[Gansner et al. 1988]
da Vinci	A <sup>7</sup> SpecT → <sup>7</sup> C		[Fröhlich and Werner 1995]
DBE	C		[Nummenmaa and Tuomi 1990]
EDGE	C, C++, Tcl/Tk	30	[Paulisch 1993]
G-ABDUCTOR			[Sugiyama and Misue 1996]
GDS	Java, plug-ins		[Bridgeman and Tamassia 2003]
GDToolkit	C++		[Willhalm 2001]
GD-Workbench			[Buti et al. 1996]
GEM	Borland C		[Frick et al. 1995]
GIOTTO	Pascal		[Tamassia et al. 1988]
GMB	C, C++		[Jablonowski and Guarna, Jr. 1989]
GoVisual	Java, C++, .NET		[Gutwenger et al. 2003a]
GRAB			[Rowe et al. 1987]
Graph <sup>Ed</sup>	C, C++	240	[Himsolt 1995]
Graph Editor Toolkit			[Madden et al. 1996]
Graph Layout Toolkit	Java, MFC, .NET		[Doğrusöz et al. 1998]
Graphlet	Tcl/Tk, C++		[Himsolt 1997]
GraphViz, DOT	C, C++, Java	76	[Ellson et al. 2003]
Henry			[Henry and Hudson 1991]
JViews			[Diguglielmo et al. 2002]
STATEMATE			[Harel 1988]
VCG	C++	73	[Sander 1995]
VGJ <sup>8</sup>	Java	17	[McCreary and Barowski 1998]
WilmaScope	Java		[Dwyer and Eckersley 2003]

Table 6.2: Some graph drawing systems.

<sup>4</sup>Internationalization denotes human language independent programming and the opportunity to switch the display/output language externally without the necessity of recompiling the program.

<sup>5</sup>Kilo Lines Of Code; taken from literature/WWW or calculated from source if available

<sup>6</sup>The most recent references are shown only.

<sup>7</sup>converted by an intermediary compiler

<sup>8</sup>The current version is 1.03 released on 4/20/98.

Obviously, these packages require different graph types to be implemented. `sugi::admin::input` as well as `sugi::admin::work` provide own graph realizations to be used as base classes for more specific packages. To prevent defining data specific to individual algorithmic aspects or application domain specific graph transformations in basic graph, node or edge instances, we decided that using the graph copy mechanism each algorithmic step creates its own graph, which is allowed to subsequently create its own node and edge instances. Therefore, each graph implements its own factory methods to be called while executing the graph copy mechanism. While copying the local input graph and creating appropriate instances, data from the preceding steps is taken over by the concrete step as far as required. On the one side, this reduces the speed of the implementation (REQ\_SPEED), on the other side, common data or dynamic data structures are not required and responsibilities can be spread over the concrete graph, its nodes and edges instead of collecting all that in an external algorithm class (REQ\_ARCHITECTURE).

Furthermore, we decided to bundle the term “graph algorithm” with the term “graph”, so that usually no external algorithm transforms a graph but the graph does that itself. Hence, a specialized graph represents itself as a graph, a specific algorithm and, after calling its constructor, it represents the result after executing the algorithm realized by the graph.

As described in Section 4.2, different access paths through a graph are provided. A graph has access to its nodes and edges, a node to its attached edges and an edge to its start and end node. This allows easy navigation in a concrete implementation but defining and traversing a subgraph is not possible without further mechanisms. A subgraph can be created by copying the appropriate instances, or a graph mapping proxy/decorator can be used instead. A graph mapper, which also may represent augmented graphs, retrieves its data (e.g. the elements of the subgraph) from a graph elements filter. The subgraph mechanism is generically implemented in `sugi::admin::work`.

Because Graph itself, its node and edge class serve for multiple purposes, e.g. as foundation for manipulating and transforming an graph in algorithms, to specify an input graph as well as for the result of the layout process, a lot of signatures are present and implemented because of consistency reasons, but should not be usable everywhere. For example, changing the nesting relations might be appropriate to a general input mechanism or the implementation of the layout algorithm, but the usage on result instances is discouraged. Therefore, the classes in `sugi::admin::input` restrict certain signatures like specifying internal basic node and edge types from being applied to input instances. Similarly, `MutableGraph`, which can explicitly be locked to prevent (accidental) changes, is used as superclass for the result graph, which then is locked by default. As an option, result instances may be unlocked in certain situations.

Beside `sugi::admin`, three further packages are located on the same level. `sugi::applicationLibrary` implements the basic facilities to define different, consistent applications for *SugiBib*. It provides message handling through the framework, a generic command line interpreter and a Java-independent graphics subframework, which also encapsulates the GUI (Graphical User Interface). Further issues on that package and its concrete instantiation for UML class diagrams will be discussed in Section 6.3. To provide input and output plug-ins to the application library, `sugi::io` defines the basic interfaces for file in- and output as well as general repository access. This package is not located in `sugi::applicationLibrary`,

because these facilities might be used outside the application library. Furthermore, even if not explicitly shown in Figure 6.1, both packages use the interfaces in `sugi::admin` but cannot be seen as part of the core layout package.

On the same level, the implementation of the basic steps of the Sugiyama algorithm `sugi::ranking`, `sugi::ordering` and `sugi::coordinates` are located. Basic pre- and postprocessing classes are defined in `sugi::preprocessing` and `sugi::postprocessing`. All these classes can be tailored to a specific application domain via factory, strategy and configuration methods. These graph, node and edge classes extend the classes in `sugi::admin::work`. Individual general algorithms from graph drawing and computational geometry, which are intended for reuse in the other packages, are members of `sugi::algorithms`. Even if not explicitly shown in Figure 6.1, also this package imports `sugi::admin`.

On top of the graph layout core, the application library and the basic implementation of the individual steps of the Sugiyama algorithm as well as the extension for UML class diagrams is defined. `sugi::uml` basically introduces some general UML classes, e.g., adornments or basic classes holding stereotypes, constraints and tag-value lists each with individual coordinates and fonts due to the information object life cycle. Furthermore, a generic algorithm for enumerating and drawing edge crossings of a laid out graphs according to `UML_EDGE_CROSSING_SYMBOL` is provided.

These basic facilities are then extended for UML class diagrams in `sugi::uml::classDiagrams`. To comfortably specify UML class diagrams as input, the graph, node and edge class from `sugi::admin::input` are extended. Further constants and resources as well as a specialized implementation of the mechanism for `UML_EDGE_CROSSING_SYMBOL` are defined. As described in Section 4.2 and above in this section, nodes and edges are not able to draw themselves. More specialized information instances have to be attached. In `basics` the extended interfaces and basic implementations of these information classes are located. A factory for information instances in `basics` supports that user specific implementations of these interfaces can be simply be introduced. One of these realizations is the set of default implementations in `standard`. The information on incremental layout is handled similarly. The preprocessing package uses a hierarchy detection plug-in to realize `UML_HIERARCHY` and a metrics plug-in to calculate the scaling of nodes (`UML_SIZE_NODES`) and edges (`UML_SIZE_EDGES`) according to issues of object-oriented software engineering.

A more specialized version of the application library defines the common facilities of the different applications of *SugiBib* for UML class diagrams. In principle, an application interprets command line options, loads one or more diagrams from different sources, calculates the layout according to several user specified options and writes the result in a given format. Some applications provide a single or multiple document GUI, others work as command line applications only. Using the basic application library in `sugi::applicationLibrary`, such a common basic application is defined in `sugi::uml::classDiagrams::applicationLibrary`. As discussed in Section 3.2, the storage format specific handling for UMLscript (in `UMLscript`), XUMLscript and XMI via XSLT ([Reiniger 2003], both in `XUMLscript`) and XMI as well as XMI[DI] via JMI (both in `xmi`) are realized as input/output plug-ins.

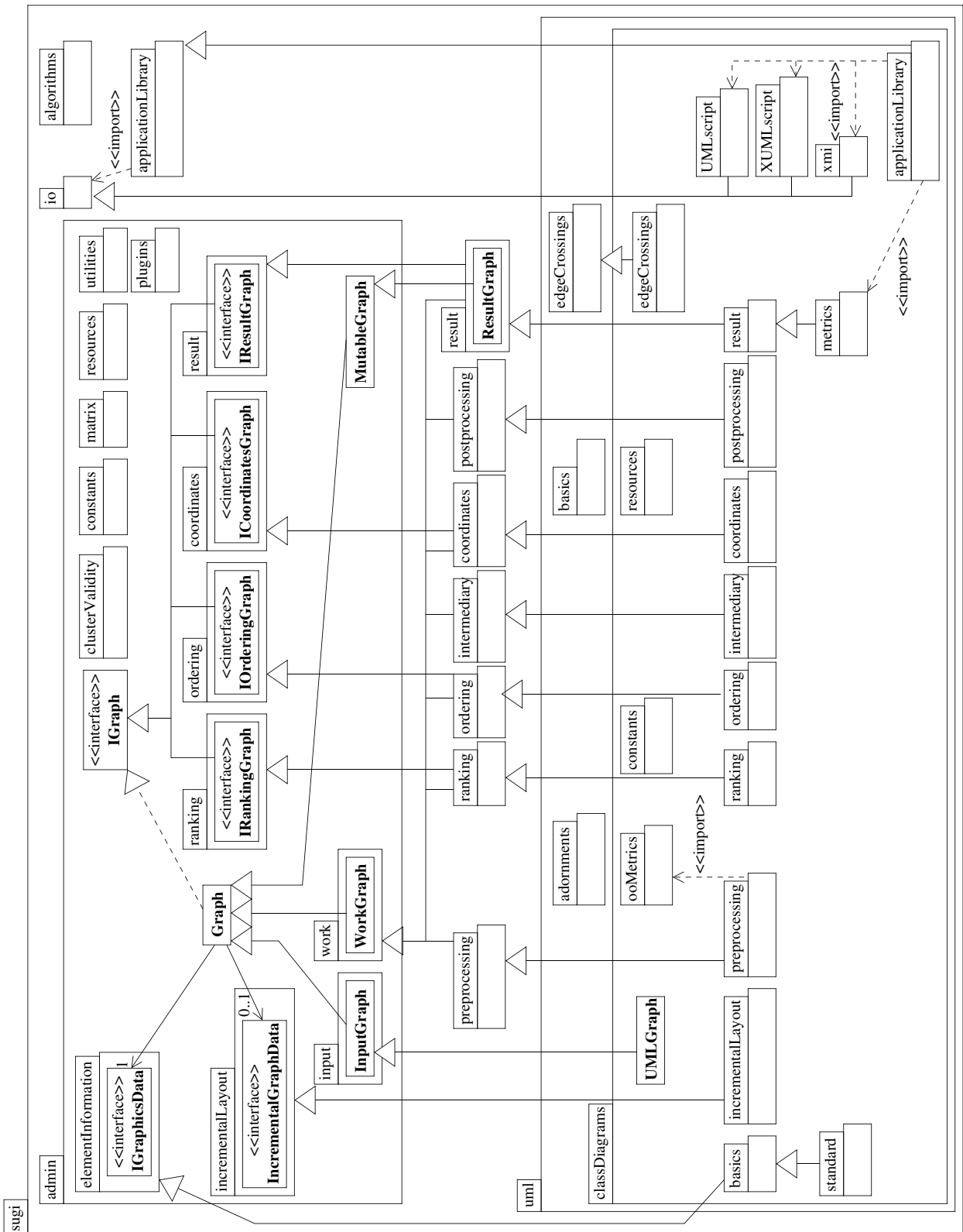


Figure 6.1: Overview on the main structures of SugiBib.

Additionally, the application library references metrics, the implementation of the UML class diagram specific aesthetics as layout metrics. These metrics are used to objectively judge the concrete result of the layout algorithm and can be used for regression testing. Furthermore, these metrics can also be used as a tool for evaluating the results of user studies on aesthetics of UML class diagrams.

Beside compositions, the graph drawing framework *SugiBib* has different factories (e.g. for UML information objects), provides strategy and factory methods (described along with the individual methods in the source code documentation), decorators (e.g. the subgraph handling mechanism), plug-ins (e.g. graph input/output), singletons (e.g. repository access), visitors (e.g. generic graph output), delegates (e.g. information instances for nodes and edges) and strategy classes (e.g. parts of the rank assignment algorithm). Because of efficiency reasons, iterators are used rarely in *SugiBib*. This is discussed in detail in Section 6.5, because it is an issue of implementation and optimization rather than a design topic.

## Class Diagram Aesthetics Revisited

On Figure 6.1, which was drawn manually, some aspects of class diagram aesthetics should be discussed again.

The main purpose of Figure 6.1 is to give an overview on the architecture and the main structures. Beside delegation (configuration by plug-ins, template methods and strategies) and composition of the graph data structures, realization of interfaces, inheritance and nesting of elements are the most relevant criteria for structuring the diagram on the given level of granularity. Therefore, we decided to compose the hierarchy out of nesting, realization and inheritance edges (UML\_HIERARCHY). Obviously, nesting of elements into packages is respected even if most packages occur as individual elements without contained elements (UML\_SEMANTIC\_CLUSTERS). Due to space limitations, we also included dependencies in the hierarchy, applied UML\_JOIN for inheritance edges (horizontal joins) and a vertical join for dependency edges. The sequence of edges at joined elements is respected as mentioned along with Figure 3.8. To emphasize the top-down direction of the hierarchical edges, UML\_MEDIAN is supported. A typical strict layering is not applied for `IGraphicsData`, `IncrementalGraphData` and `Graph` because of space limitations and to prevent edge crossings.

To prevent overlays of package tabs (UML\_CONTAINER) and edges, the inheritance edge between `sugi::admin::elementInformation` and `sugi::uml::classDiagrams::basics` is routed with a low number of bends outside the packages `sugi::admin` and `sugi::uml`. Also due to space limitations, we connected most of the packages at their tabs instead of the package region below and therefore violated UML\_CONTAINER. Implicitly, the packages are ordered (UML\_CONSTRAINT\_SEQUENCE) in sequence of the layout algorithm, in `sugi::admin` according to the basic STT algorithm, in `sugi` as well as in `sugi::uml::classDiagrams` subject to the sequence of the macro phases in Section 4.1. Generally we did not draw edge crossing symbols (UML\_EDGE\_CROSSING\_SYMBOL) in this diagram.

From viewpoint of graph drawing (UML\_GRAPHDRAWING) at least the usage of the drawing area might not be efficient, but it fulfills the criteria with higher priority, espe-

cially regarding UML\_SPATIAL and UML\_CONSTRAINT\_VICINITY and rectangular clusters (UML\_SEMANTIC\_CLUSTERS).

## 6.2 Layout Algorithm

For every thousand hacking at the leaves  
of evil, there is one striking at the root.

Henry David Thoreau (1817 – 1862)

In this section, details on the implementation of the algorithmic steps introduced in Section 4.1 and explained in Chapter 4 will be given. Each of the 7 macro steps of our layout algorithm will be discussed in an own section.

This section should be understood as a brief introduction to basic structural information required as a foundation when maintaining or extending *SugiBib*. It will also become obvious that applying the layout algorithm requires a valid input graph and access to the class `sugi::uml::classDiagrams::InfoIIAlgorithm`, which realizes the main process flow of our layout algorithm and returns a result graph.

### 6.2.1 Preprocessing

Good teaching is one-fourth preparation  
and three-fourths theater.

Gail Godwin

Figure 6.2 depicts the relevant classes realizing the preprocessing macro phase of the layout algorithm. `InfoIIAlgorithm` implements the process flow of our layout algorithm as introduced in Section 4.1. The individual algorithmic steps are executed by instantiating the appropriate graph classes as described in Section 4.2.

In *SugiBib*, step S1, which adjusts basic semantical issues of the input graph, is implemented in the class `AdjustSemanticalGraph` in `sugi::uml::classDiagrams::preprocessing`. The graph copy mechanism, which creates the first internal copy of the input graph, is basically implemented in the top-level graph class `sugi::admin::Graph` and refined in `sugi::admin::work::WorkGraph` to also handle global internal temporary data. The plugins for the calculation of the complexity of model elements, which are considered in S1 as a preparation of the scaling of model elements according to their magnitude, can be found in `sugi::uml::classDiagrams::oopMetrics`. Both, the node complexity and the edge complexity metric are currently implemented as a proof of concept only: The node complexity metric simply calculates a weighted sum of the number of attributes, methods, aggregations, compositions and associations of a class. The edge complexity metric always returns a constant value.

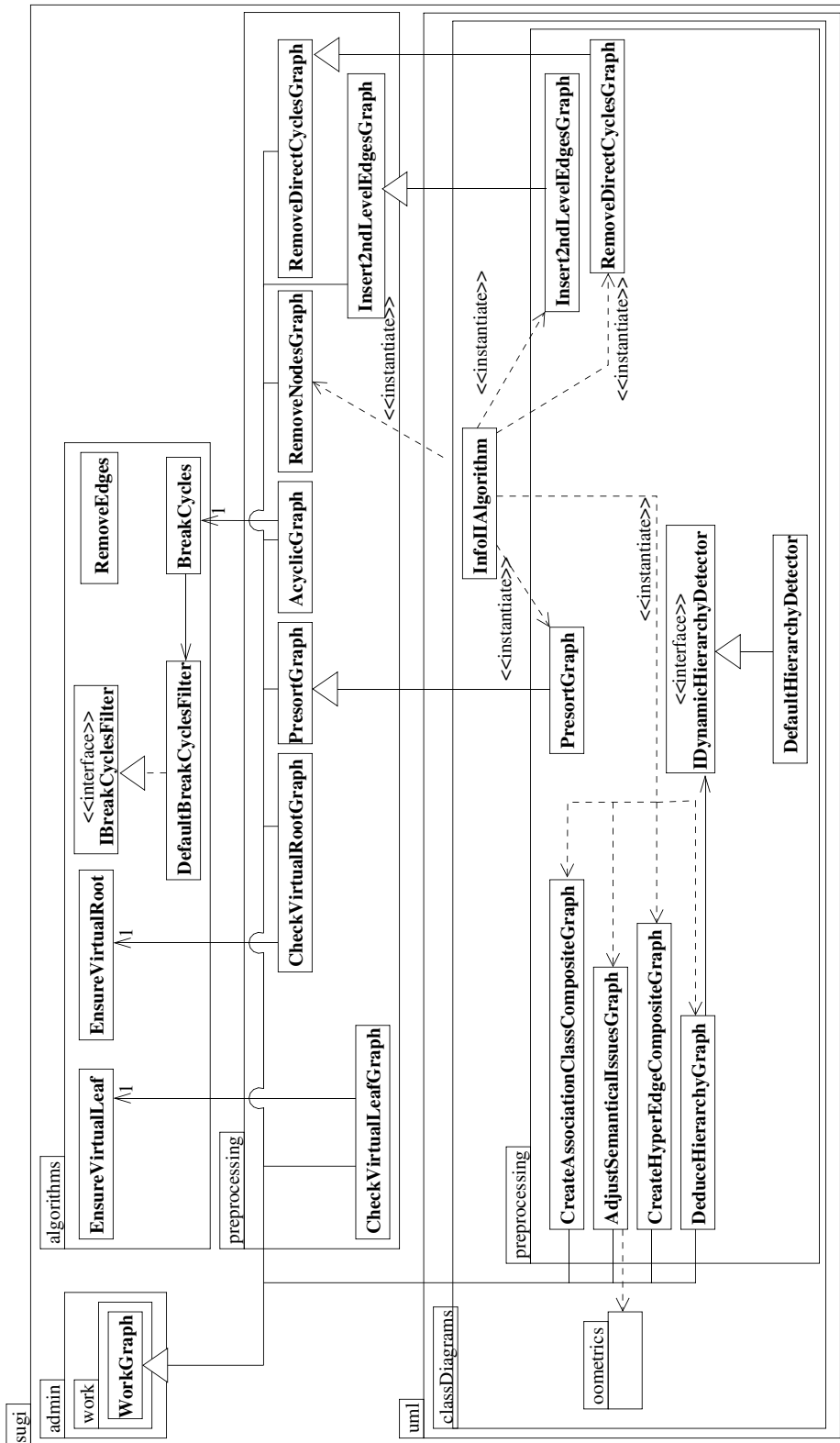


Figure 6.2: Simplified view on the classes and relations of the preprocessing steps in *SugiBib*.



The implementation of the semantic ordering of graph elements (S2) is separated into a common and a UML specific implementation, because other layout algorithms may reuse this step for a similar normalization. As extension of the classes in `sugi::admin::work`, a basic implementation is given in `sugi::preprocessing` and a UML specific one in `sugi::uml::classDiagrams::preprocessing`.

In S3 the pseudo hierarchy, which represents the main skeleton of the result drawing according to `UML_HIERARCHY`, is assembled. This is implemented in the class `DeduceHierarchyGraph` as a subclass of `sugi::admin::work::WorkGraph`. To simplify further development, various hierarchy detectors can be provided, registered and selected. Currently, edges can be considered as hierarchical edges only, if all edges of the same kind, e.g., anchors, aggregations, dependencies or inheritance edges are selected. This basic behaviour is implemented as a default hierarchy detector (not shown in Figure 6.2), which can be refined for own implementations, e.g., if a user study admits certain rules according to which an automatic grouping can be performed.

At a first glance, inserting edges to represent the containment relations in S4 looks quite simple, but the basic implementation in `sugi::preprocessing::Insert2ndLevelEdgesGraph` provides different filtering mechanisms to generically select the nodes and edges to be processed. Therefore, it uses the graph element query mechanism introduced by the basic interfaces. As described in Section 4.2, a node, an edge or an attached information object receives thereby an arbitrary object and returns a positive or negative response. A version specialized to create information instances appropriate for UML class diagrams is provided in `sugi::uml::classDiagrams::preprocessing`.

Some of the following steps, like preparing hyperedges (S5) or association classifiers (S7) are specific to UML class diagrams so that only specific graphs like `CreateHyperEdgeCompositeGraph` or `CreateAssociationClassCompositeGraph` in `sugi::uml::classDiagrams::preprocessing` are provided. Others, like removing reflective edges (S6) or disconnected nodes (S9) can reuse common algorithms, which can be configured via the graph element query mechanism.

Furthermore, the preparation steps, which ensure a root of the hierarchy as well as an acyclic graph, required for implementing the rank assignment, are provided as common static implementations in `sugi::algorithms`. These steps can directly be referenced from the rank assignment without inducing the necessity of probably copying the input graph. Common, self-transforming graphs, as usual in *SugiBib*, are provided in `sugi::preprocessing` and simply delegate their work to the static algorithms. As discussed in Section 4.4.11, some edges may be reversed with a higher priority than others when breaking cycles in a graph. This is realized as an optional filter to, which must implement the interface `IBreakCyclesFilter`.

## 6.2.2 Rank Assignment

The only limit to our realization of tomorrow will be our doubts of today.

Franklin D. Roosevelt (1882 – 1945)

Figure 6.3 depicts the implementation of the rank assignment step in *SugiBib*. Nodes and edges

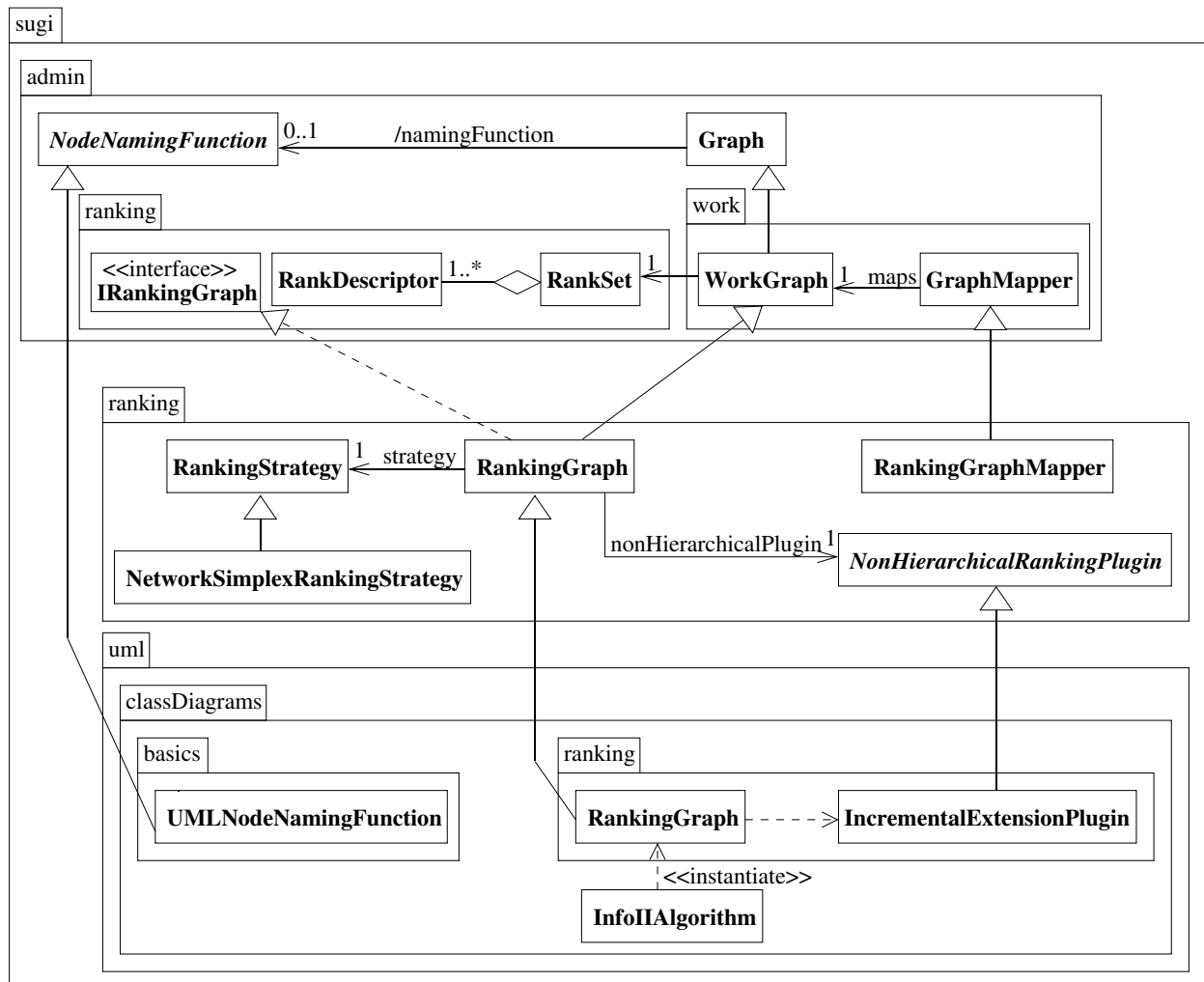


Figure 6.3: Simplified class diagram on the implementation of the rank assignment in *SugiBib*

are omitted, because their inheritance relations are handled similarly to graphs.

The implementation of the node naming function is separated as suggested in Section 4.3.4. The abstract class `sugi::admin::NodeNamingFunction` ensures some basic consistency issues between node based and criterion based methods and specifies the abstract hierarchical node naming function. The concrete UML specific implementation, which

realizes the hierarchical naming system defined on fully qualified names, is given in `sugi::uml::classDiagrams::basics::UMLNodeNamingFunction`.

`sugi::admin::ranking` provides `RankSet`, the data structure which holds the hierarchy levels. It consists of instances of `RankDescriptor` which, like the `RankSet` itself, may be reconfigured by subclassing and overriding factory methods in the graph class. To avoid replicated code, the `WorkGraph` owns the rank set and considers the instances of the graph copying mechanism.

In the package `sugi::ranking` the basic rank assignment algorithm applicable to general graphs is implemented by delegating the hierarchical and the non-hierarchical task to two different plug-ins. Only the default network simplex algorithm as described in Section 4.5.4 is given as default in this package. If another implementation of the rank assignment algorithm, e.g. based on longest path layering or linear programming as mentioned in Section 4.5.1, should be realized, an appropriate strategy class, which implements `sugi::uml::classDiagrams::ranking::RankingStrategy`, has to be provided and has to be passed as a constructor parameter to the `RankingGraph`.

The non-hierarchical rank assignment is then implemented along with UML specific adaptations to the basic rank assignment in `sugi::uml::classDiagrams::ranking`.

### 6.2.3 Edge Crossings

Strategy is a style of thinking, a conscious and deliberate process, an intensive implementation system, the science of insuring future success.

Pete Johnson

The foundation of the implementation of the edge crossing reduction is defined in `sugi::admin::ordering` by specifying some common interfaces. An `IOrderingGraph` is intended to delegate the work specific for a crossing reduction method to a certain `IOrderingStrategy`. Obviously, the `RankSet`, displayed in Figure 6.3, which represents the  $n$ -level hierarchy is important for the implementation of the edge crossing reduction mechanism. Due to space limitations and to avoid lengthy edges we did not depict it in Figure 6.4.

In `sugi::ordering`, the implementation of `IOrderingGraph` and the basic implementations of the ordering strategy are located. `BasicOrderingStrategy` relates the graph with the strategy to define the data to work on. `BasicMatrixOrderingStrategy` uses the matrix implementations in `sugi::admin::matrix` and does not respect cluster-validity. The implementation of that abstract class, `BarycentricOrderingStrategy` and `MedianOrderingStrategy` are implementations from the first version of *SugiBib* and were refactored to fit into this strategy concept. `BasicCrossingOrderingStrategy` attaches a crossing number calculation plug-in to the ordering strategies. It is assumed that strategies, which are derived from that class, directly use the cluster valid position implementation in `sugi::admin::clusterValidity` (the relation is not shown in Figure 6.4).



`PermutationOrderingStrategy` realizes algorithm 4.12 and therefore ensures cluster-  
validity by intertwined or postprocessing execution for non-incremental edge cross-  
ing reduction strategies. As opposite, the hierarchical strategy directly inherits from  
`BasicCrossingOrderingStrategy` and facilitates the incremental validity mechanism.

The implementation of the hierarchical crossing reduction algorithm and the  
`OrderingGraph` have to be configured for the use with UML class diagrams.  
Due to code reuse and consistency issues, the most configuration work is done in  
`sugi::uml::classDiagrams::ordering::OrderingGraph`. Dependent on the prese-  
lected ordering strategy and the concrete type of the input graph, `InfoIIAlgorithm` instantiates  
one of the strategies and passes it to an instance of `OrderingGraph` which then reorders the  
nodes in their individual ranks according to the concrete strategy.

As mentioned in Section 4.6, comparing two orderings should also be delegated to a sep-  
arate strategy, e.g. to experiment with secondary ordering criteria. Such a criterion is also  
capable of preventing from superfluous crossing number calculations or representing different  
orderings: the best one found so far and the working one while executing the edge crossing  
reduction. Due to space limitations, in Figure 6.4 it is not shown that a basic ordering criteria  
is defined in `sugi::admin::ordering`, basically implemented along with the basic strategies  
in `sugi::ordering::basics` and partly specialized for individual edge crossing reduction  
strategies.

## 6.2.4 Intermediary Processing

There's a fine line between genius and in-  
sanity. I have erased this line.

Oscar Levant (1906 – 1972)

The three algorithms from the intermediary macro processing step can be realized by two  
dedicated classes in `sugi::uml::classDiagrams::intermediary`<sup>9</sup>. Expanding association  
classes (S12) as well as expanding hyperedges (S13) are processing steps, which can generi-  
cally be realized in the same class (`ReIntegrateCompositeNodesGraph`). `InfoIIAlgorithm`  
provides both graph instances with the correct parameters for the graph element query mecha-  
nism.

Similar to the 2-level implementation of S4, which provides a common and a specific ver-  
sion for inserting containment relations as edges, the complementary step (S14) is real-  
ized. A basic version is provided in `sugi::intermediary` and a specialized version in  
`sugi::uml::classDiagrams::intermediary`.

---

<sup>9</sup>We do not display the five involved classes in an own diagram.



provided. Both strategies as well as all coordinates methods were located in one graph class with attached node and edge classes. In further versions this implementation was refactored towards the structure displayed in Figure 6.5. `CoordinatesBaseGraph` implements a set of basic methods for graphs without non-hierarchical edges. The attached node and edge classes are not shown in Figure 6.5. The methods which work on non-hierarchical edges were moved into the class `CoordinatesFlatGraph`. The deterministic (`gansner`) and the currently unused `springEmbedder` assignment strategies have been encapsulated in subpackages. Depending on the presence of non-hierarchical edges, further assignment steps have to be executed. Therefore, the two packages contain specialized strategies for both types of graphs. Algorithm 4.19 is directly implemented in `GansnerFlatStrategy`.

The graph distance functions for adjacent nodes or ranks are represented by `GraphDistance`. The abstract class `ClusterSeparator` provides a plug-in into the basic coordinates graph to insert cluster separator nodes dependent on the application domain.

To maintain rank specific information, like the number of flat edges or the maximum height of a node, the basic `RankDescriptor` has been subclassed to more specific versions required by the coordinates implementation.

The concrete specialization, which adds UML specific conditions and rules to the implementation, is located in `uml::classDiagrams::coordinates`. A concrete mechanism specialized for packages and subsystems realizes the cluster separator plug-in mentioned above. Furthermore, a `CoordinatesGraph` internally maintains a set of `InitXYConstraints`. These instances represent the data structure used to initialize the coordinates of the nodes to obtain a cluster-valid graph<sup>10</sup>.

Finally `InfoIIAlgorithm` creates the UML specific `CoordinatesGraph` and passes an instance of `GansnerFlatStrategy` as parameter. Subsequent instances like the cluster separator plug-in are automatically created by specialized factory methods in `CoordinatesGraph`.

## 6.2.6 Postprocessing

[The superior man] acts before he speaks,  
and afterwards speaks according to his actions.

Confucius (551 BC – 479 BC)

Figure 6.6 depicts the individual classes involved in the postprocessing macro step. Similar to some of the other algorithmic steps described in this section so far, the implementation of the postprocessing is distinguished in a common and a UML specific implementation.

Because laying out association classes (S16), hyper edges (S17), annotations (S18) and disconnected elements (S19) share a common basic behaviour, namely the reintegration and layout of individual graph elements, several basic classes are successively refined and specialized in `sugi::uml::classDiagrams::postprocessing`.

<sup>10</sup>Because of historical reasons this function is still located in the UML specific part. A future implementation may be refactored towards a more general version in `sugi::coordinates` and a specialized version for UML class diagrams

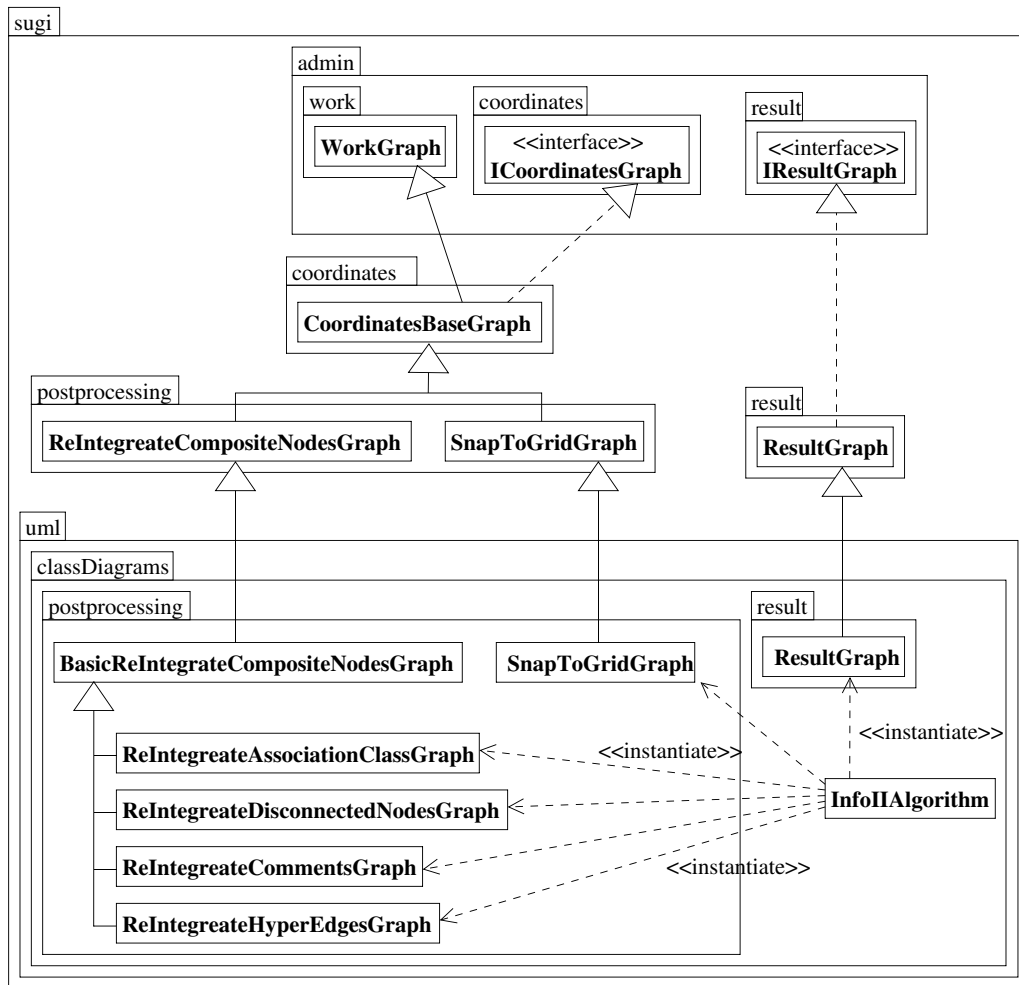


Figure 6.6: Simplified view on the classes and relations of the postprocessing steps in *SugiBib*.

Also the implementation of the optional snap-to-grid feature (S20), which relies on the basic node positioning rules, e.g., for nested nodes, too, provides a general and a more specialized version.

As a final processing step, the result graph is generated (S21). The general `ResultGraph` is subclassed, because various implicit UML related information, like the positions of reflective edges, has to be made accessible to arbitrary output formats. Unlike the graphs realizing the other processing steps of our layout algorithm, a result graph is not subclass of `WorkGraph`, because internal, temporary information should be disposed and not be available to an application presenting or processing a result graph.



## 6.3 Applications for UML class diagrams

A manager is responsible for the application and performance of knowledge.

Peter F. Drucker

Designing and implementing a layout algorithm is one task, providing applications for different contexts and testing the algorithm another task. In this section we will discuss some applications delivered with *SugiBib*. Concrete applications integrate the basic facilities provided by the *SugiBib* framework with the UML specific extensions. Thereby, information classes specific to UML class diagrams as well as input and output mechanisms appropriate to the application play an important role. Beside these reusable functionality, many concrete applications basically realize the same flow of information:

- External options, e.g. obtained from the command line, are interpreted.
- One or more class diagrams in probably different formats are read in and transformed to *SugiBib* compliant input graphs according to Section 4.2.
- The diagrams are processed by the layout algorithm respecting external, global and diagram specific options.
- Finally, the result is, however, presented, e.g. on a screen, as a file, on a printer, etc.

In *SugiBib* this basic flow was implemented as an application library which, furthermore, is independent from the underlying GUI. In this section, architectural features going more into detail than the general overview given in Section 6.1 will be discussed.

### 6.3.1 Information Classes for UML Class Diagrams

I find that a great part of the information I have was acquired by looking up something and finding something else on the way.

Franklin P. Adams (1881 – 1960)

As described in Section 4.2, in *SugiBib* application domain specific information classes can be attached to nodes and edges. In Figure 6.7 basic relations between the core framework and the UML extension in `sugi : :uml` are depicted. Due to space restrictions and issues of understandability, only few classes and relations are shown.

The basic graph classes `Node`, `Edge` and `Graph` are extended to provide UML class diagram specific signatures to simplify creating the input graph. As described in the preceding sections, the instances of `UMLGraph` are transformed by the graph copy mechanism and therefore only the structural relations and the information objects are accessible to the layout algorithm.

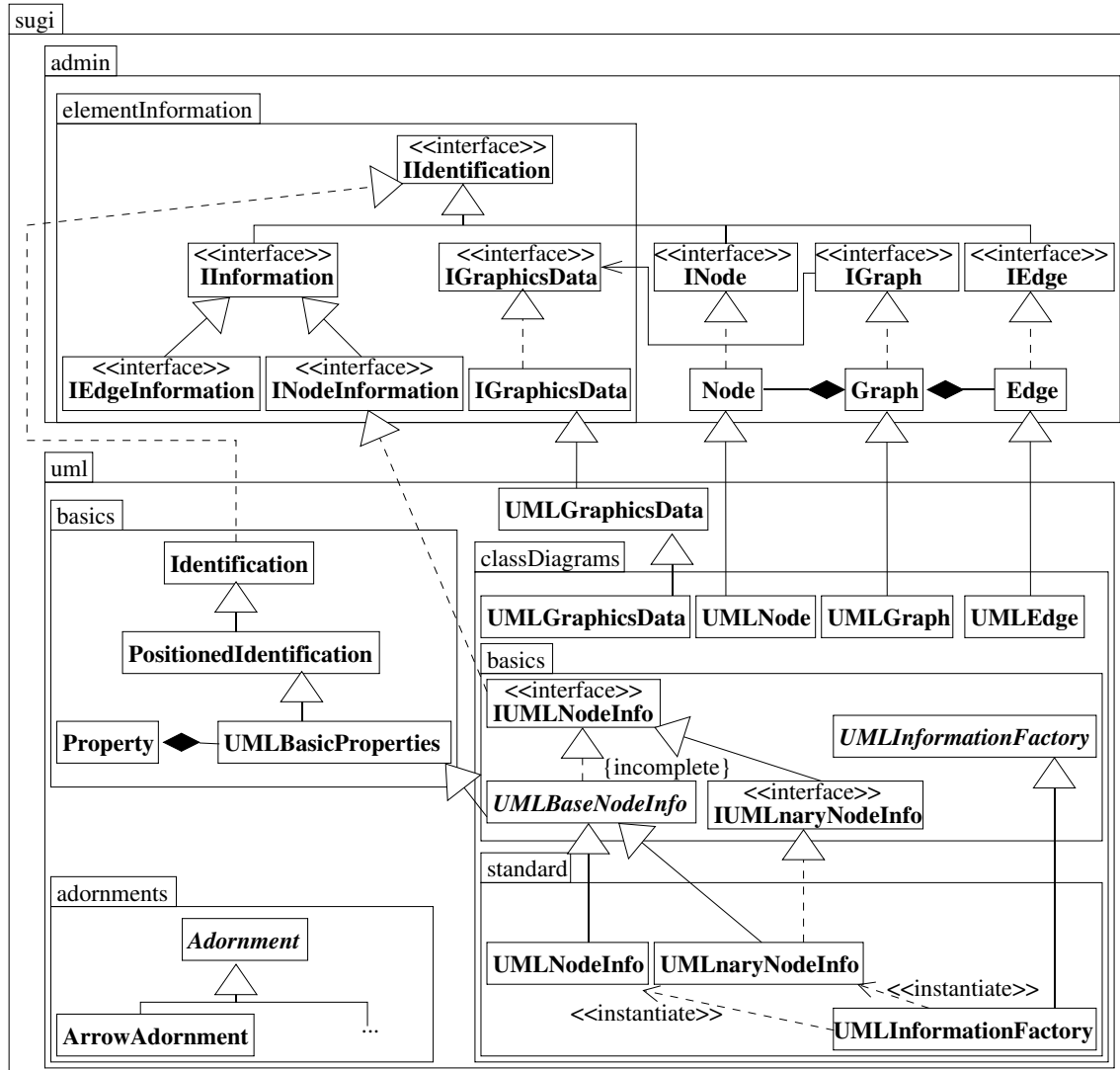


Figure 6.7: Simplified view on the classes and relations of the UML information infrastructure.

Furthermore, the graph information class was extended over two layers. In `sugi::uml::UMLGraphicsData`, basic support to draw UML-like lines was added, in `sugi::uml::classDiagrams::UMLGraphicsData` further UML class diagram specific configuration options like fonts were integrated.

The implementation of the information classes was split into several packages on different layers. This information must not be stored as implicit data in the information classes, because an output filter should be able to store detailed information of the layout result, like coordinates, directions or sizes of adornments. Furthermore, to support `REQ_INCREMENTAL_ALGORITHM` in future releases, detailed positional information, which can probably be obtained from the input format has also to be stored as partly immutable information. Therefore, in `sugi::uml::basics`, common data storage classes were defined, for example various adornments or basic classes which hold stereotypes, constraints and tag-value lists (each with individual positions and fonts). In `sugi::uml::classDiagrams` these classes were aggregated to information classes specific to UML class diagrams. The package `basics` contains more specific interfaces and basic implementations. In `standard`, concrete information classes to be used primarily with the standard applications of *SugiBib* were defined.

When *SugiBib* is used as a plug-in by another application, shapes for connected elements instead of detailed structures might be sufficient. Then, other implementations than those in `standard` may be provided to simplify the exchange of modeling and layout data. Information classes for edges follow the same structural principles.

To support different concrete implementations of information classes, factories have been defined. The factory and the underlying constant classes in `sugi::uml::classDiagrams::constants` were split physically into an internal and an external part, because *SugiBib* also creates instances of information classes which should be used internally only. The UML class diagram specific implementation is encouraged to create information objects via these factories only. This was furthermore supported by specifying appropriate visibility modifiers, e.g. package local visibility, for the concrete information classes.

Because of historical reasons, the structure of the information classes for nodes is more fine-grained than that part for edges. Each node type, e.g. classes, n-ary associations, comments, etc. is implemented by an own class. For edges only two different classes were provided. The first version of *SugiBib* supported inheritance edges, realizations, associations, compositions and aggregations only. Even if all novices are told to avoid subclassing for code reuse, this mistake was made in the past and we disliked a refactoring so far. Therefore, one class realizes inheritance-like edges and another the remaining (association-like) edges.

Usually an input graph is not created manually. This task is done by input graph filters, so called *contents interpreters*. The layout result may be transformed into a (standardized) persistent format by various *contents exporters*. Such a data filter can easily be selected from the set of available filters, because each filter is able to return, if it is applicable to an input graph, a result graph or a certain format.

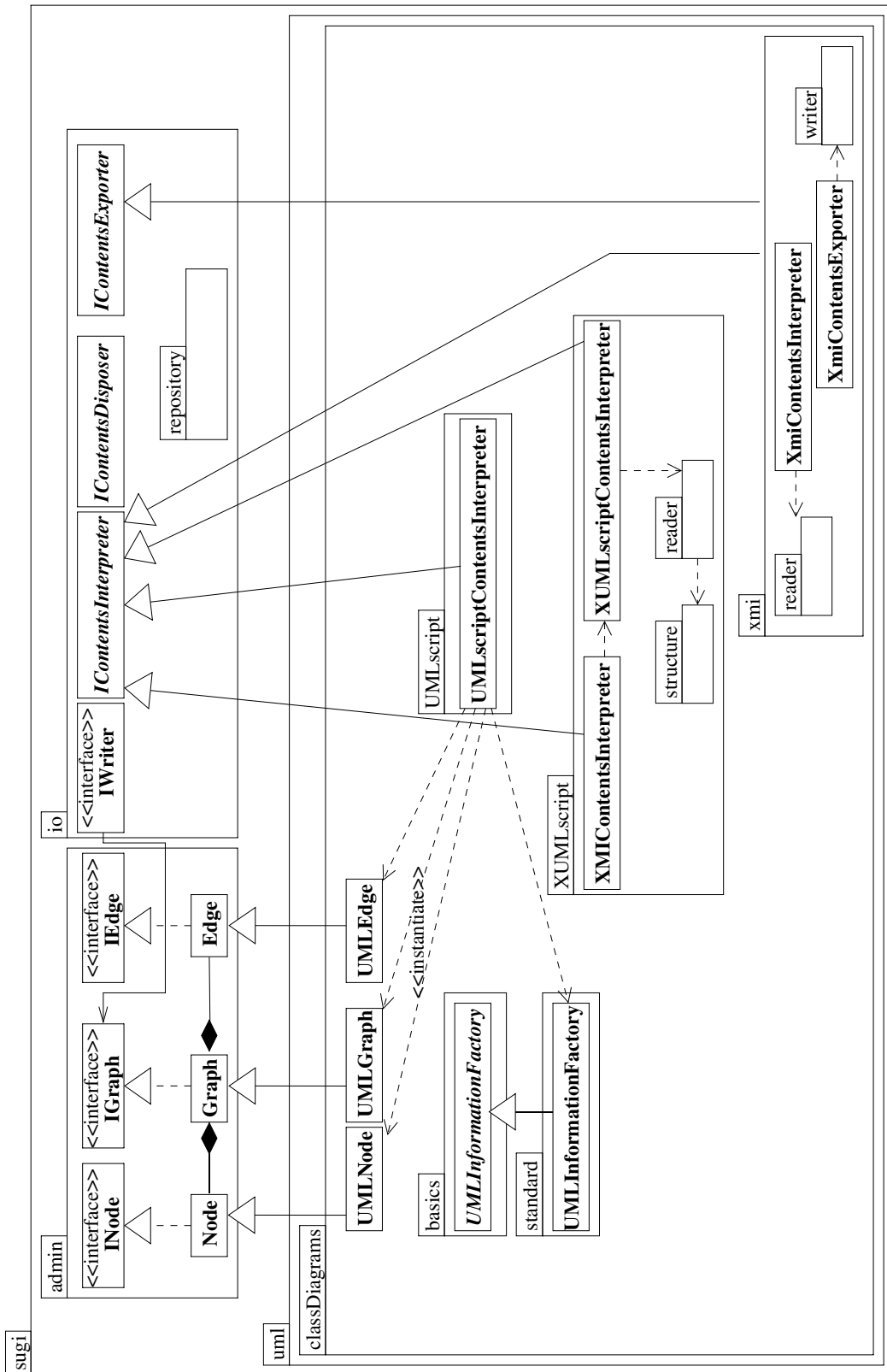


Figure 6.8: Simplified view on the Input/Output subframework.

Dependent on the underlying mechanism, temporary data might be present even after finishing input or output operations. For example, an input graph could be loaded using a repository mechanism and it might be appropriate to export the result by accessing the original information used for creating the input graph. Therefore, at least when terminating the application, such mechanisms have to be informed that temporary data should be disposed. To notify a filter mechanism, an application calls registered *contents disposer* instances in certain situations, e.g., when the application is shutting down. The classes in `sugi::io` discussed so far are not given as interfaces, because each of these classes define static signatures to realize type specific parts of the dynamic registration mechanism. Interpreters, exporters as well as disposers are automatically registered via a plug-in mechanism. In Figure 6.8 the basic classes provided by `sugi::io` are depicted.

A concrete contents exporter needs to traverse the result produced by *SugiBib* and may store information on each element of the graph. An object-oriented approach would be to send a request to each graph element, which then generates the appropriate output of itself. This can be realized by deriving the information classes and defining a new information factory. As described above, further implementations of the information classes might be provided and even these classes would have to be subclassed to be applicable to the export mechanism. To circumvent this structural problem, the classes in `basic` could be modified directly. In this case, the code size would grow with the number of export mechanisms and basic object-oriented principles would be violated. Hence, traversing the graph and accessing the information via public signatures is the remaining approach. This is supported by *SugiBib* via the *IWriter* interface. In combination with the core graph implementation, a simplified visitor mechanism is provided. An instance of that interface is passed to the `write` method of a graph, which then traverses itself according to a sequence specified by the concrete writer. On each graph element, the `write` method of the *IWriter* interface is called with the individual graph element.

Furthermore, to introduce generic readable and writable access to repositories, `sugi::io` realizes a general repository browsing mechanism.

According to the different formats discussed in Section 3.2, several packages have been implemented to realize

- an UMLscript importer based on a parser generated by the  $LL(k)$  parser generator *Coco/R*<sup>11</sup>.
- a XUMLscript importer [Reiniger 2003]. It creates a secondary data structure which is traversed twice to generate a *SugiBib* compliant input graph. Furthermore, by preprocessing XMI data via XSLT, the implementation can be used to directly read XMI.
- an importer and exporter based on an underlying XMI[DI] repository<sup>12</sup>. It relies on general JMI mechanisms, the Netbeans metadata repository<sup>13</sup> and an XMI[DI] enhanced meta-model.

---

<sup>11</sup><http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/>

<sup>12</sup>Unfortunately that part of the implementation is currently not complete.

<sup>13</sup><http://www.netbeans.org>

## 6.3.2 Application Library

When I step into this library, I cannot understand why I ever step out of it.

Marie de Seigne

As mentioned in the introduction of this section, the basic information flow, which is similar for a lot of applications using the *SugiBib* layout algorithm, can generally be implemented and reused. In the first version of *SugiBib*, we started with a simple browsing application for the Abstract Window Toolkit (AWT), which is part of the core Java library. From the beginning, this browser provided various features to be manipulated by a user. The user options of this application (UMLAWT) do not collide with REQ\_USER\_OPTIONS, because it is intended for testing and exploring the layout features of *SugiBib*.

With the advent of Swing, the lightweight platform independent GUI implementation in the extended Java library, a similar application (UMLSwing) was realized. Furthermore, two command line applications were provided: UMLBatch processes multiple diagrams in batch mode, UMLNet supports online rendering by appending the result as an attachment of an email.

When realizing new global options of the layout algorithm, all dependent applications had to be adjusted to keep them in a consistent state. Because this was a tedious, time-consuming job, we started to implement all applications on top of one library and to perform changes only once. Therefore, an intermediary layer providing an additional abstraction between the representation of the options and their concrete GUI presentation was introduced. By applying the bridge design pattern, for each (required) element of the GUI, a delegate interface and appropriate implementations for AWT, Swing and commandline applications were realized. Then the common parts of the implementations were compiled into one library.

Furthermore, a generally configurable command line parser was implemented, so that the application library can provide common command line arguments and the individual applications may redefine that set. A general purpose command line interpreter was not used, because we wanted to implicitly distinguish between different groups of arguments. For example, static arguments have to be processed before the application object is created, while other arguments are allowed to influence the state of the application object.

Due to the discussion between SUN and IBM on the Standard Widget Toolkit (SWT), the proprietary GUI library of the Eclipse Integrated Development Environment (IDE)<sup>14</sup>, we also considered the mapping of the graphical elements in Java. Beside a native access to the GTK libraries, SWT provides an own mechanism for drawing graphics with Java. Unfortunately, no classes or interfaces common to the AWT are supported. As an experiment, we also included delegates for (required) features like the graphical context, colors, fonts and dialogs. Thereby, another browser which natively relies on SWT (UMLSWT) was realized. This graphical independence can help integrating *SugiBib* into tools which rely on non-Java mechanisms.

---

<sup>14</sup><http://www.eclipse.org>



Figure 6.9 illustrates the architecture of the application library discussed above. All GUI delegates are created through a factory. A concrete application creates the appropriate factory instance, registers it in the application library, and all subsequent GUI operations are automatically delegated to the corresponding GUI implementation. Hence, all classes outside `sugi::applicationLibrary` realize their tasks independent from the underlying GUI.

A concrete application library realizing the basic flow for UML class diagrams was provided in `sugi::uml::classDiagrams::applicationLibrary`. Here concrete command line options, menu options (in the case of a GUI application) and input/output facilities are compiled into one implementation. Furthermore, plug-ins for automatic features, e.g. coloring classes according to certain criteria, and interactive features, activated by mouse actions, were defined. Dialogs, specific to the application, e.g. for browsing a given repository, were implemented on top of the application library using GUI delegates only. The basic initial options, e.g. what crossing reduction algorithm should be applied, are injected into the library by an initializer plug-in. Concrete plug-ins may return static defaults, read basic options from a configuration file or initialize the application library for certain purposes like testing or debugging.

In all diagrams so far we have omitted direct relations to the resource management basically implemented in `sugi::resource`. These classes realize a hierarchical resource message bundle facility according to localization and I18N, i.e. if appropriate message bundles exist, the application can be run in any language without recompiling the source code. The application library relies on one of the most specific message bundles so that the language of each dialog, each menu entry and most of the internal error messages for exceptions can be determined from outside the application.

Further subclasses of the basic application library for UML class diagrams provide facilities like multiple document interfaces or default code for command line applications. Finally, the five concrete implementations briefly described above, were implemented as subclasses of the same application library.

### 6.3.3 Layout Metrics

You have all the reason in the world to  
achieve your grandest dreams. Imagina-  
tion plus innovation equals realization.

Denis Waitley

The result of the layout process was copied into a usually immutable graph in S21. This information has to be reconstructed from coordinates, because additional information like individual layers have been removed by creating the result graph. In fact, `MetricsGraph`, `MetricsNode` and `MetricsEdge` reimplement parts of the classes related with `WorkGraph` to simplify navigation and hierarchical access.



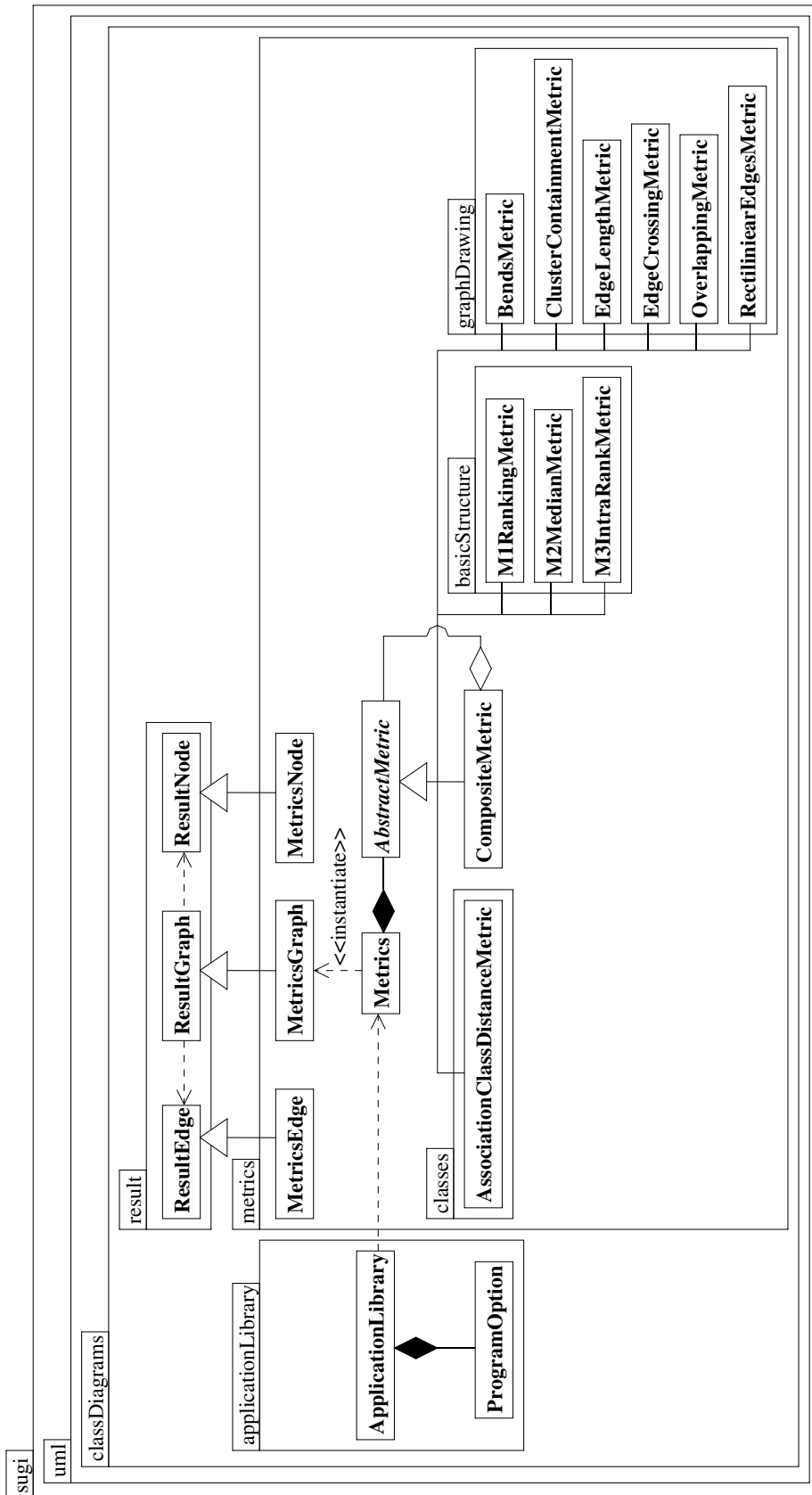


Figure 6.10: Implementation of the layout metrics in *SugiBib*.

At a first glance, this seems to be a superfluous effort, but because this subframework may be reused to also analyze graphs created by external tools without executing the *SugiBib* algorithm, a unified approach was realized even for results delivered by *SugiBib*.

The individual metrics were implemented as subclasses of `AbstractMetric` and are organized in several packages. Individual metrics are registered in `Metric`, the class which implements the main flow of control for metrics calculation. `Metric` is the only class accessible from outside.

This information has to be passed by the `ApplicationLibrary` to the `Metric` class, because program options like `UML_COUPLING` do not only influence the layout result but may be considered in some metrics to calculate a objective judgment.

## 6.4 Testing & Debugging *SugiBib*

Never stop testing, and your advertising  
will never stop improving.

David Ogilvy (1911 – 1999)

Over the years of development, *SugiBib* has grown to a size of 278 KLOC containing approximately 40% comments (about 900 classes in 110 packages). At a first glance, this might appear as a large, probably oversized project for only drawing UML class diagrams. Even if it is well known that code sizes do not automatically lead to an objective judgment of a program, we have displayed the sizes of other, graph drawing related software in Table 6.1 and 6.2. According to that information, *SugiBib* seems to be comparable with the sizes of LEDA or Graph<sup>Ed</sup>. In Figure 6.11 the individual sizes of the logical parts of *SugiBib* are displayed. Excluding parts of the

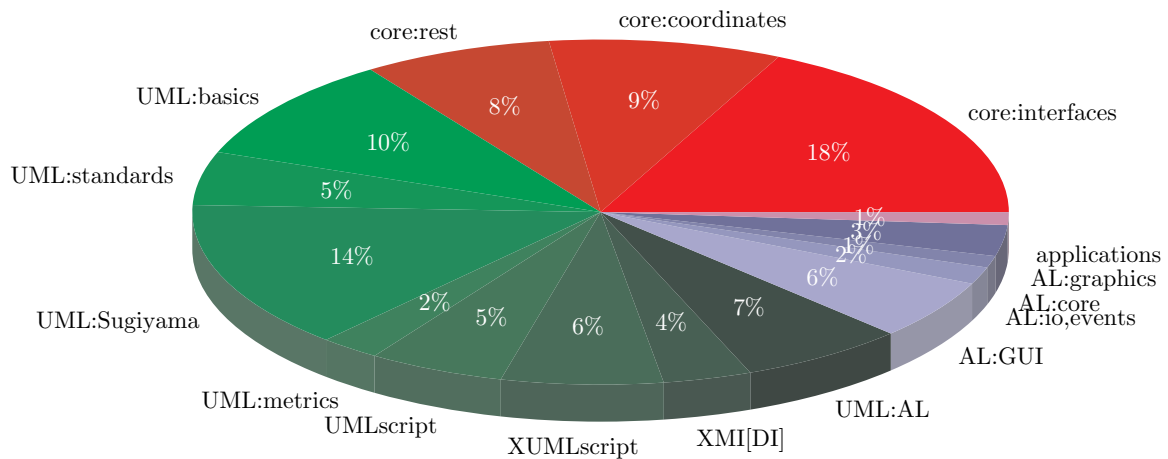


Figure 6.11: Distribution of sizes of the main parts in *SugiBib*. “core” as prefix denotes the core layout implementation excluding applications, UML (prefix “UML”) and application library (“AL”).

application library and the input mechanisms, we can conclude that the general graph drawing

part takes 35% (97.3 KLOC) and the code for drawing UML class diagrams takes 71%-88% (197.4-244.6 KLOC) of the entire framework. Hence, the general framework compares in size to VCG or GraphViz and size of the part specific to UML class diagrams appears to be reasonable.

When modifying the source code of a program, usually various kinds of tests are required to ensure consistency across more or less invasive changes. Typical test methodologies known in literature are blackbox and whitebox tests. A *blackbox test* considers a module to be tested as a closed part, which is accessible through defined signatures only. Such tests pass different types of input to the module and analyze the output by comparing it against expected data. A *whitebox test* is allowed to make certain assumptions on the implementation and even to test individual parts of the unit, which are not required to be visible or accessible from outside. Furthermore, different kinds of stress tests should be applied to analyze the behavior in unexpected or exceptional situations. Hence, beside valid input, invalid data, borderline situations as well as manipulations of temporary data, memory or certain (online) attacks to a program might be considered. Even if the latter testing issues are appropriate to our online rendering application UMLNet, we will discuss tests for local applications here only.

For blackbox testing we need a set of representative class diagram input files in different formats. These diagrams should capture a variety of situations, which may occur in UML class diagrams. A lot of diagrams arise from programming when thereby certain situations are (re)produced, e.g., to validate new parts of the implementation. Unfortunately, only few users, who obtained a version of *SugiBib* from our homepage, responded in the case of successful evaluation or problems. For example Yann-Gaël Guéhéneuc extended his reverse engineering tool Ptidej<sup>15</sup> by an output implementation for UMLscript and contributed some files which currently produce the largest diagrams in test. Jorge Gomez Sanz described the diagrams of the INGENIAS development process metamodel<sup>16</sup> in UMLscript and used *SugiBib* to automatically generate diagrams<sup>17</sup>. Florian Grupp created a set of input files in UMLscript and XUMLscript, which describe class diagram examples in the UML specification.

The more tests are provided, the easier negative impacts, which may arise from modified code, can be detected. Therefore, the example files have been compiled to a black box test suite which currently consists of 175 files. Unfortunately, looking at some diagrams after changing code does not really help ensuring consistency. Hence, an automated regression test mechanism would be appreciated. In contradiction to usual black box tests, comparing the output of a layout mechanism with expected data appears to be more complex than implementing a layout algorithm. For that reason we decided to use the implementation for measuring compliance to aesthetic criteria by metrics for automated testing.

A general test recorder was implemented in the package `regressionTesting` within the UML class diagram specific application library. Various concrete implementations, e.g. storing the test results in a database, can be realized. Currently a XML based recording is provided. A shell script iterates over all available input files, provides UMLBatch with appropriate options for activating the test recorder and ensures that the program is killed in the case of an endless loop.

<sup>15</sup><http://www.yann-gael.gueheneuc.net/Work/Research/Ptidej/Download/>

<sup>16</sup><http://ingenias.sourceforge.net/>

<sup>17</sup><http://grasia.fdi.ucm.es/ingenias/metamodel/>

	association class distance				bends (1,1)				cluster containment				edge crossings				edge lengths			
	2004-07-08	2004-07-10	2004-07-12	2004-07-12	2004-07-08	2004-07-10	2004-07-12	2004-07-12	2004-07-08	2004-07-10	2004-07-12	2004-07-12	2004-07-08	2004-07-10	2004-07-12	2004-07-12	2004-07-08	2004-07-10	2004-07-12	2004-07-12
one.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	0.98	0.98	0.98	0.65	0.65	0.65	0.65
st.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
st1.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
st2.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
st3.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
st4.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
st5.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
st6.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
p12.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
p29.uml	1.0	1.0	1.0	0.82	0.82	0.82	0.82	0.82	1.0	1.0	1.0	1.0	0.91	0.97(+)	0.97	0.97	0.65	0.65	0.65	0.65
p3.uml	1.0	1.0	1.0	0.9	0.9	0.9	0.9	0.9	1.0	1.0	1.0	1.0	0.98	0.98	0.98	0.98	0.74	0.70(-)	0.70	0.70
p36.uml	1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	1.0	1.0	1.0	1.0	0.96	0.98(+)	0.98	0.98	0.19	0.19	0.19	0.19
p36_Short.uml	1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	1.0	1.0	1.0	1.0	0.95	0.98(+)	0.98	0.98	0.17	0.17	0.17	0.17
p36_Simple.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.17	0.17	0.17	0.17
p37.uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
p4.uml	1.0	1.0	1.0	0.88	0.88	0.96(+)	0.96	0.96	1.0	1.0	1.0	1.0	0.93	0.95(+)	0.95	0.95	0.64	0.71(+)	0.71	0.71
p4_Short.uml	1.0	1.0	1.0	0.9	0.9	0.96(+)	0.96	0.96	1.0	1.0	1.0	1.0	0.90	0.92(+)	0.92	0.92	0.60	0.62(+)	0.62	0.62
l	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.74	0.74	0.74	0.74
m	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.74	0.74	0.74	0.74
uml	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.16	0.16	0.16	0.16
.uml	0.61	0.61	0.61	0.61	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.61	0.61	0.61	0.61

Figure 6.12: Screenshot of the HTML output created from the regression test results.

To automatically test various options, e.g. considering all implemented edge crossing reduction algorithms, on each input file, another script is provided with *SugiBib*. Combined with the runtime measurement facilities, the diagrams in Section 5.3 have been produced as a side effect on a complete regression test run. To visualize the test results, the XML data is postprocessed by an XSLT sheet to generate HTML. In Figure 6.12 an excerpt of the test protocol is shown. Metric values are encoded by colors and change tendencies within one metric for an individual file can be highlighted by symbols. Furthermore, changes in runtime behavior are collected and displayed.

As an experience, regression tests on graphs are a helpful tool to ensure stability and consistency, but they may also lead to disappointing results. As quoted in Section 4.8.1, due to interferences, some values may improve, others may show a tendency in the other direction. Thereby, it is not always easy to make the correct decision, if the last changes should be committed or rejected. Unfortunately, regression tests do not help when searching for concrete errors in the source code. When a failure, e.g. an exception, is spotted by the automated test, even when the stack trace is available, it might be difficult to find the source of the error, because often a preceding piece of code in another class causes the error.

Hence, whitebox tests are required to provide a mechanism to help the programmer. Even if this thesis provides several definitions, which can be implemented as local checks for certain units, we did not apply a unit test framework like JUnit<sup>18</sup> to realize this task so far. Similar to the hierarchy of graph classes, a set of attached consistency test classes have been realized. A global flag in `sugi:GlobalFlags` statically activates certain calls to these test classes. For example, consistency checks are then activated directly after edge crossing minimization or after moving an individual node in the coordinates assignment. Thereby the graph is analyzed for conformance to definition 13, 23 or 25, respectively. If appropriate checks have been implemented and the consistency mechanism is called frequently while debugging, the location where the error is produced can easily be spotted. Unfortunately, activating these consistency checks, slows down the runtime by at least factor 10.

On simple or small graphs, the Java debugger can be used to step through certain parts of code when searching an error. The larger the graph, the more complicated it is to find out the correct conditions for a breakpoint next to the erroneous code. Therefore, *SugiBib* also provides a simple logging mechanism to write data onto the standard streams. When appropriate messages are emitted, it is most times much simpler to trace an error due to information of the context. Simply placing print statements in the code increases the time to get rid of these statements after the error is found. Therefore, the logging mechanism also emits information on the source of the logging call. Thereby, not all direct sources should be mentioned, e.g. if the logging is encapsulated in a convenience method like `printRanks`. Then the source of the call to `printRanks` is more appropriate. Therefore, the logging mechanism provides a general feature to register certain methods/classes to be excluded as source.

Even if a specialized runtime consistency test for the input graph is provided, it is a complicated task to find certain structural errors in a new input format reader. Especially if structurally equal files in different formats are present, debugging such errors is simplified, when the internal data structure of an existing implementation can be compared to that of the new one. For this reason, *SugiBib* provides a graph output filter, which produces a normalized, proprietary format that can

---

<sup>18</sup><http://junit.sourceforge.net>

be compared by tools like `diff`<sup>19</sup>. Because memory addresses or Java hashcodes for objects will probably differ between various input format filters or implementations, a contents dependent mechanism is more appropriate. By considering UML specific information, to each graph element a unique identifier can be assigned: For each class UML requires a unique name and edges as well as unnamed nodes like n-ary nodes can be named according to the connected nodes. By sorting the output according to these unique identifiers the output is normalized. Furthermore, for each graph element and information object, the contents of the attributes can generically be retrieved via Java reflection. For normalization, a type specific handling of object references was implemented. Unfortunately, the generic dynamic data access in Java induces additional maintenance effort: Generic dynamic data access can only be realized via the reflection API and thereby class members can only be referenced as strings, which then are not always correctly handled by automatic refactoring tools.

A fragment of an output produced by that filter is shown below:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<sugiBibGraph>
  <object description="graph" type="sugi.uml.classDiagrams.UMLGraph"
    id="graph:10918860222">
    <field name="activeCluster" value="0"/>
    <field name="autoInsert" value="false"/>
    <field name="clusters" value=" [[]]"/>
    <list name="edges">
      <object description="edges entry" type="sugi.uml.classDiagrams.UMLEdge"
        id="edge:node:BehavioralFeature|node:Method|UMLInheritanceInfo[|null|null]">
        <field name="minLength" value="1"/>
        <field name="type" value="USUALEDGE"/>
        <field name="weight" value="1"/>
        <object description="edgeinfo"
          type="sugi.uml.classDiagrams.standard.UMLInheritanceInfo"
          id="edgeInfo:edge:node:BehavioralFeature|node:Method|
            UMLInheritanceInfo[|null|null]">
          <field name="hierarchyEdge" value="false"/>
          <field name="visibility" value="UNDEFINED"/>
          <field name="color" value="BLACK"/>
          <list name="constraints">
          </list>
          <field name="inheritanceKind" value="INHERITANCEEDGE"/>
          <list name="stereotypes">
          </list>
        </object>
        <objectRef name="from" reference="node:BehavioralFeature"/>
        <objectRef name="to" reference="node:Method"/>
      </object>
    </list>
  </object>
  ...
```

<sup>19</sup><http://www.gnu.org/software/diffutils/diffutils.html>

## 6.5 Runtime Optimizations

To improve is to change; to be perfect is to change often.

Winston Churchill (1874 – 1965)

Even if we decided by `REQ_SPEED` that runtime speed is not the main aspect when implementing our layout algorithm, generally, various influences to the code can be identified which then may lead to speed improvements. On the one side, improved and more sophisticated algorithms obviously can influence the time required for calculating the layout of a given graph. For example, in Section 4.6.3 we discussed different aspects of realizing the crossing calculation. On the other side, depending on the underlying programming language, structural changes and opportunities to refactor the code for speed improvements without modifying individual algorithms can be taken into account.

In [Shirazi 2000] useful hints on performance tuning for the Java programming language were described. In the last years the implementation of the Java compilers and the Java Virtual Machine (JVM) have been improved a lot. But even if experts believe that these days changes to the source code do not significantly influence the compiled program, the basic rules in [Shirazi 2000] are still valid. Furthermore, were presented in [Eichelberger and von Gudenberg 2004] an extended set of rules, which has been applied successfully to *JTransform*, an object-oriented source code transformation framework for Java.

In this section we will first discuss structural changes based on the results in [Shirazi 2000; Eichelberger and von Gudenberg 2004] which can be applied to the source code of *SugiBib*. Then we will list the effects of applying some of the rules and deeper algorithmic changes to *SugiBib*. In this section we will exclusively refer to the current Sun HotSpot JVM 1.4.2.

- O1: Avoid excessive instance creation [Shirazi 2000]:** Every instance creation requires memory allocation for the object itself and its internal structures. It takes time to initialize an object, especially if it points to deeper data structures or it is member of a larger hierarchy, because the constructors of all superclasses have to be called. That is a problem, in particular if constructors frequently delegate the implementation by `this(...)` calls. In special cases of flat initialization a call of the `clone` method might be appropriate and faster than the usual initialization calling a constructor. Furthermore, the fewer (temporary) instances are created, the less time the automatic garbage collector requires.
  
- O2: Avoid iterators [Shirazi 2000]:** The iterator design pattern [Gamma et al. 2000] is extremely useful when generically traversing data structures in a defined sequence without internal knowledge of a data structure. Unfortunately, the current compiler optimizations do not respect iterators to be compiled without instance creation and method invocations. In [Trapp 2001] techniques and results for optimizing compilers by implementing such techniques were described. On own, container-like data structures, methods, which return the number of elements and the *i*-th element, usually provide a better performance. Beside

the effect of tuning a program for performance, typed signatures help avoiding errors at compile time and may lead to an increased understandability of code and design.

- O3: Use pooling [Shirazi 2000]:** Sometimes it is beneficial to "recycle", or pool, certain, otherwise short-living, objects while executing a program. Preserving and reusing a single object is often better than repeated creation and garbage collection of multiple objects if the cost of repeating the above operations outweighs the inconvenience of introducing additional object pool management code. This might be the case for collections, maps and frequently used own data structures. Pooling implies collections which store the temporary objects while they are not in use. These temporary instances are then received ("created") from and released by special methods. Hence, the scope of objects has to be respected and code looks more like C code with explicit memory management. As a drawback, as in programming languages without automatic memory management, accidentally releasing the same object multiple times may cause unpredictable behavior of the program. Not releasing a poolable object does not result in memory leaks because of the automatic garbage collecting mechanism. Unfortunately, Java and the JVM itself do not provide appropriate standard mechanisms for pooling.
- O4: Use caching [Shirazi 2000]:** Results, which are requested frequently may be cached and the time for recalculation can be saved. Note, however, naive caching may lead to inefficient memory usage. Caching is usually implemented using hash tables from the Java collection framework, which are based on an array of internally created map entries. These entries should (optionally) be treated according to O1 and O3. Unfortunately, changes to these internal structures by reusing the original implementation is not permitted due to visibility modifiers and missing signatures. Hence, an overhead in memory usage arises, because for each stored object a new corresponding map entry has to be created. Furthermore, usually own implementations are executed with a lower priority by the JVM, because they are not loaded via the bootstrap class path and not treated as trustable libraries. Usually, tricks like replacing library classes as advocated in [Shirazi 2000] are inappropriate techniques due to incompatibilities.
- O5: Avoid excessive exception throwing [Shirazi 2000]:** Exceptions are the default technique to signal problems or errors in Java programs, especially in public interfaces. While runtime, an exception itself is an object, which has to be created (see O1). The catch-throw mechanism is time consuming, e.g., if exceptions are thrown repeatedly from inside loop bodies. Until version 1.4 of the JDK, the stack trace stored in an exception was not accessible and therefore an instance of an exception usually was not reusable. In newer versions of the JDK, access to the stack trace of an exception is provided and can be manipulated to realize pooling of exception instances [Zukowski 2003a].  
In *SugiBib* a more defensive programming style was applied. Errors, if possible, are caught and exceptions are thrown only for methods to be called from outside the framework or when the execution should be terminated immediately.



- O6: Avoid excessive local variable declaration [Shirazi 2000]:** Local variable declarations should be avoided in loops, because there are JVMs that repeatedly execute these declarations and create the appropriate stack structures. Optimizing Java compilers automatically move the variable declaration outside the most appropriate inner loop, but this is not true for the default compiler by SUN<sup>20</sup>.
- O7: Avoid long parameter lists [Shirazi 2000]:** The JVM is optimized for a maximum method parameter number of 4. If this number is exceeded, the remaining parameters have to be handled explicitly on the call stack and these method calls become time consuming. This can be circumvented by (temporary) objects containing the method parameters (respect O3).
- O8: Avoid irrelevant or convenient method calls:** Sometimes overloaded methods with default parameter settings ultimately result in a call to the same base method. Especially if these methods are called frequently, irrelevant method calls are executed. Due to the single source principle some method implementations are shared, but if these methods are called repeatedly, a refactoring of the interface and a merge towards one method should be performed. Even if the Java compiler by SUN now supports "aggressive method inlining", calls over chains of (static) methods are much slower than a direct call. Simple code might be replicated to gain performance, in particular, if methods which originally contained the code will never be overridden. Especially classes providing a basic implementation like adaptors for convenience introduce another drawback because of additional effort due to dynamic linking.
- In particular, this applies to template methods in *SugiBib*. Because of the framework architecture often template methods are used to make certain algorithms configurable. These methods are called frequently and often they also refer to the basic implementation to ensure consistency. Hence, a certain trade-off is induced by REQ\_ARCHITECTURE, because not every superfluous or convenient method may be removed due to the extendibility aspect of the framework.
- Modern JVM implementations such as SUN's HotSpot JVM, contain a just-in-time or dynamic compiler. Even if the bytecode produced by the compiler does not show inlining, the adaptive compiler technology by SUN does this at runtime and eliminates guesswork in determining which optimizations will yield the largest performance benefit in the Java compiler. Therefore, introducing additional abstraction layers heavily relying on the delegates design pattern usually does not significantly slow down the execution speed. But even when such transparent techniques were applied, we gained improvements in certain situations by eliminating convenient method calls.
- O9: Use object comparisons carefully:** Comparing objects in Java is usually done by calling the equals method. As defined in Section 4.2, the elements of a graph are required to represent unique objects and hence the much faster reference equality can be used instead

---

<sup>20</sup>Newer versions of the JDK seem to implement internal optimizations and in extreme runtime tests, variables outside a loop body cause performance penalties.

of calling the `equals` method. Especially when loops are combined with methods, which cannot be inlined, this significantly reduces the speed of execution.

Without further consideration of internal mechanisms of Java, that optimization cannot be applied to strings, because most string operations return unique objects, which may hold the same contents than other string objects but are accessed by different object references. This can be circumvented by the `intern` method, which maps equal string contents to unique object references. A discussion on comparisons for strings was given in [Eichelberger and von Gudenberg 2004]. We do not go into details here, but an equality comparison of strings, which starts at the end instead of the beginning, implemented in `java.lang.String` may also speed up the implementation of *SugiBib*.

**O10: Avoid type casts [Shirazi 2000]:** For security reasons the JVM has to check each type cast and throws an exception in the case of erroneous usage [Gosling et al. 2000]. Unfortunately, the layered architecture of *SugiBib* requires an enormous effort in type casts. More specific implementations which use the immediate type of the nodes or edges in a graph implementation instead of the top level interfaces would be desirable. As a side effect, reading the source code could be simplified and hidden errors due to runtime type conflicts could be avoided.

On the one side, this implies that standard collections, which are implemented for the top-level class `Object`, have to be avoided and, on the other side, code for more specialized classes is replicated. By applying class templates as introduced with Java generics [Bracha 2004], plenty type casts could be eliminated.

**O11: Avoid unnecessary use of standard collection classes:** Wherever the number of elements is fixed or a maximum number is known, arrays should be used. The classes of the Java collections framework using `Object` as entity type should be avoided or replaced by specialized versions (see O10). If dynamically resizing collections are required, the new collection classes should be used instead of the legacy collections [McCluskey 1999; Shirazi 2000; Zukowski 2003b; Zukowski 2003c] such as `java.lang.Vector` because they are not synchronized by default. Executing each synchronized code section when synchronization is not really needed carries a performance penalty.

**O12: Estimate the initial size of dynamic data structures [Shirazi 2000]:** The initial size of most dynamic data structures can be estimated via the maximum number of elements to be stored. By presizing e.g. collections, time consuming resize operations (O4) can be avoided.

**O13: Consider short-circuit operators [Shirazi 2000]:** Put costly operations at the end in a short-circuit (short cut) logic expressions to avoid the evaluation of these operations in the case that the evaluation of the expression terminates before considering the costly operations.

**O14: Avoid `instanceof` for equality checks:** `instanceof T` returns, if a given object is instance of the type `T` or of one of `T`'s superclasses or superinterfaces. If `instanceof` is

frequently called, it might be replaced by equality checking of dedicated constants or the object query mechanism.

- O15: Avoid recursive implementations:** As far as possible, recursive implementations should be replaced by iterative implementations (see O8).
- O16: Check the sequence in switch-statements and if-chains [Shirazi 2000]:** According to the expected number of executions, the sequence of individual cases in switch-statements or chained if-statements should be modified to avoid unnecessary evaluation of conditions.
- O17: Use Java-style listeners:** If user defined code should be called in certain situations, hooks or listeners can be used. In the case of a hook, a method in the same class is defined empty and called when the event occurs. To implement the hook, the class has to be subclassed. The hook method is called every time the event occurs. This is a disadvantage (see O8), especially if a specific event is not of interest. If a listener interface and a corresponding registration method can be provided, the default case, when no listeners are registered, can be considered by a simple if-statement. Then the listener method is executed only if at least one listener was registered.  
The same applies to plug-ins: In the case of optional plug-ins the technique described above should be considered. If the presence of a plug-in is required and a default one is provided, checks for the presence can be avoided.
- O18: Avoid reflection for class information retrieval:** On the one side, for factory methods it would be extremely comfortable to retrieve the instances to be created automatically. This would unburden the programmer, e.g., in the case of the graph copy mechanism in *SugiBib*, from deep internal mechanisms. On the other side, except for the standard constructor, dynamically compiling the parameters of a method or constructor call is time consuming due to O1 especially for array and wrapper instances. Furthermore, dynamic method calls are much slower than usual method calls and exception handling (O5) is required when using the Java reflection mechanism. Obviously the overhead grows when such code fragments are called repeatedly in loops.
- O19: Restrict the reuse of class signatures:** In a framework, the programmer is tempted to make a large number of signatures visible, accessible or overridable. Therefore, at least time critical sections should be analyzed, if the corresponding methods could be defined static, private or at least final to avoid dynamic linking.

The following concrete changes have been carried out during the development of the current version of *SugiBib*. Most of the speed improvements can be validated by the runtime data collected while running automatic tests which will be described in Section 6.4. Some changes have been done before the automatic testing mechanisms were introduced. The speedup in these cases has been estimated according to layout samples of some individual diagrams. The percentage data should be interpreted as successive improvements relative to the speed of the implementation before an individual optimization was carried out.

- Global object pools (O3) in `sugi::admin::utilities::ObjectPools` have been provided for most of the temporary data structures like collections, maps, some arrays, points, minimizing and maximizing helpers, object pairs etc. Also standard Integer wrapper instances have been considered for pooling (11% speedup).
- When introducing rank descriptors as a dedicated data structure to represent a rank. First this was done by a simple wrapper of a standard collection. Then, to each rank a map, which relates the nodes to their index positions, was associated (O4), because position based access is used frequently in *SugiBib*. Thereby, we accept a possible slowdown in the case that extensive modifications like moving nodes induce costly changes to the position map, because usually the rank structures are not changed frequently (12% speedup). Further 5% speed improvement was gained after replacing the map, an instance of the standard collections framework, by a specialized map similar to that in [Laux 2004], which relates objects and the basic type `int`.
- Iterators (O2), except for those to be used with maps and in code pieces implemented by students, have been substituted by equivalent code for position based access (8% speedup).
- After replacing the simple calculation of flat edge crossings by the matrix method described in Section 4.6.3, 31% speedup was gained. Thereby, a frequently called method with 6 parameters was replaced by successive calls to two methods, each with not more than 4 parameters (O7) and an appropriate initial size for container data structures was chosen (O12). Furthermore, the calls to the layout plug-in for subsystems have been refactored (O17).
- In ancient versions, the graph copy mechanism was executed at the beginning of each algorithmic step. By copying a graph only if new elements are not assignable from the existing ones (O1), the runtime was decreased by 8%. Testing for reference equality instead of calling `equals` (O9) in the implementation of the copy mechanism, improved the runtime by further 4% – 16.5% depending on the input graph.
- The initial implementation of the coordinates assignment for compound graphs did not augment the graph by inserting cluster border nodes. It considered the contained nodes as node or rectangular coordinates closures and all node movement operations had to explicitly take these closures into account. Cluster parent nodes were allowed to adjust their position automatically when contained nodes were moved, similar to the rules described in Section 4.8.5. Even if the containment data was excessively cached, update operations were (probably due to some bugs) time consuming. After replacing that realization, which contained different inefficient algorithms, by the one based on cluster border nodes, the runtime decreased by 45%.

Even if different runtime bottlenecks have been eliminated so far, plenty more can still be found in the current version. Most of the `equals` calls (O9) could be substituted, even in specialized versions of the standard hash maps (O9,O11), various methods especially in the coordinates assignment implementation receive more than 4 parameters (O7), using relative positions

for edges instead of absolute would speedup moving nodes a lot, retrieving information on information classes via the internal query mechanism is much faster then checking the attached information class by instanceof (O14), etc.

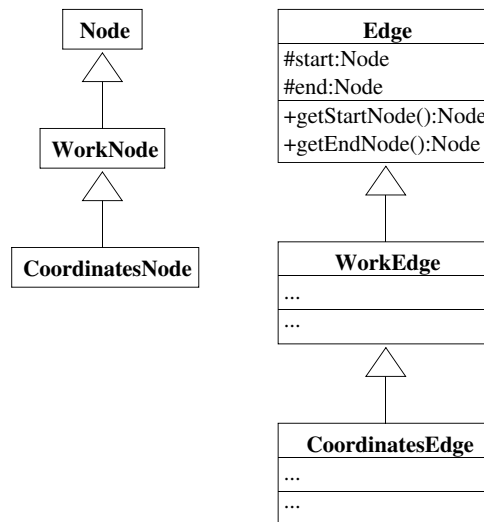


Figure 6.13: The current architecture, which leads to a large number of type casts.

A lot of superfluous type casts in *SugiBib* could be eliminated to meet O10. An appropriate programming tool, which automatically detects superfluous casts, would be appreciated. But getting rid of one of the main bottlenecks, which arise from ubiquitous type casts due to the architecture, would require new mechanisms like templates in the programming language itself or a change of the architecture.

Figure 6.13 depicts a part of the class hierarchy of *SugiBib*. As usual in object oriented programming, attributes are defined in the most common base class Edge as types of other base classes, e.g., Node. When an instance of a more specific class, e.g. WorkNode, is stored in an attribute of Edge, no type casts are required. The same is true, when methods defined in Node are accessed. By defining abstract, empty or default methods as callbacks in the base class, which are then implemented or overridden in subclasses, the template method pattern is often applied in *SugiBib*. Negative impacts occur, when methods, which are introduced in subclasses, are called from outside, because then time consuming type casts are required.

One alternative to get rid of type casts is to rethink the location of the definitions of attributes in larger class hierarchies. If reading access outweighs writing access to object references, it may be an option to declare the attributes using the most specific type in the leaf classes and to define appropriate typed accessors. Figure 6.14 (a) illustrates this idea, which will be called *lazy attributes*. The basic idea is to provide accessors, which return the most specific instance as appropriate types on each level of the class hierarchy and to structurally avoid type casts. Thereby, the declaration of attributes is deferred into the most specific (leaf) class.

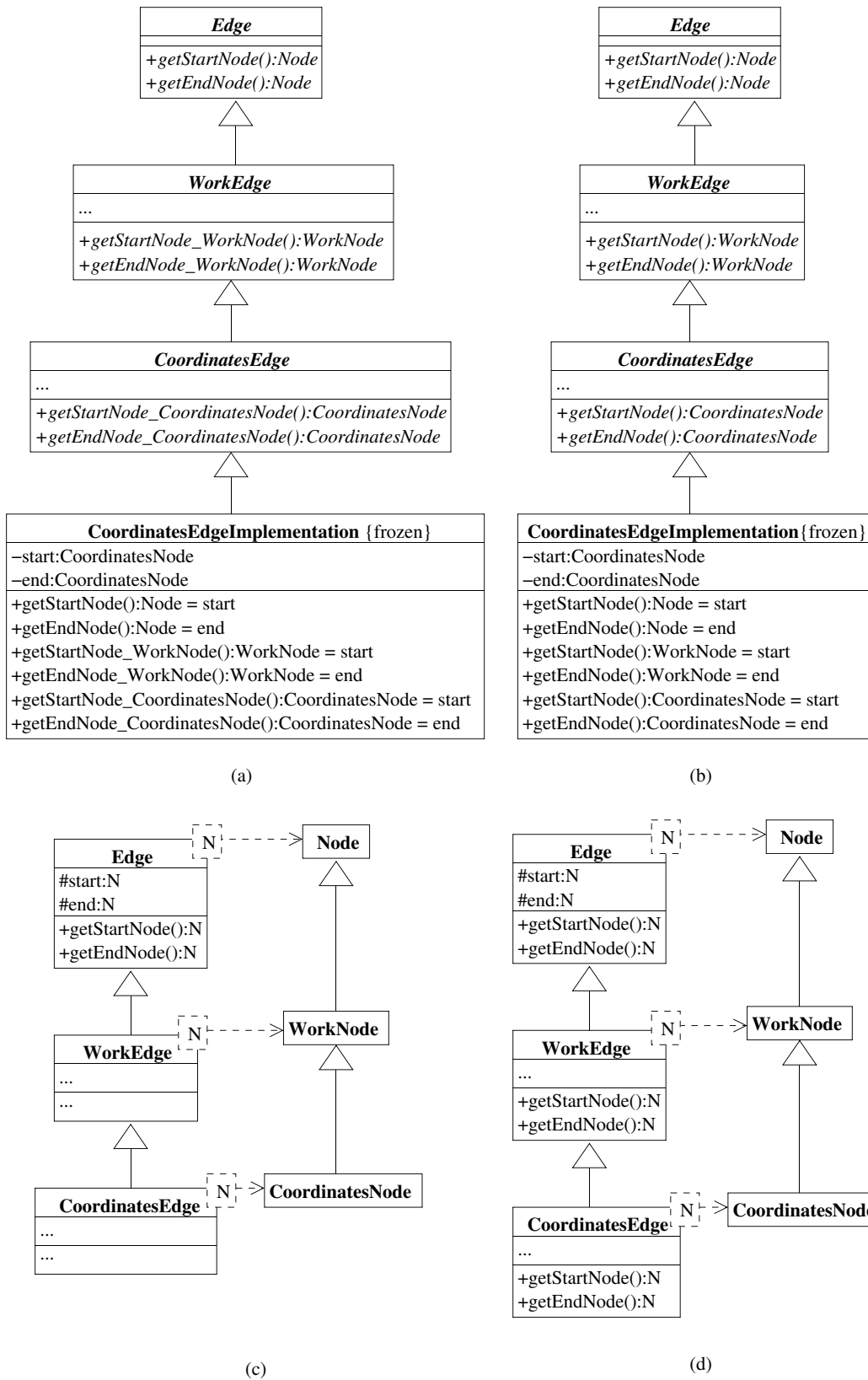


Figure 6.14: Four methods to eliminate superfluous casts: (a) depicts the *lazy attributes* pattern, (b) a combination of lazy attributes and covariant return types, (c) the application of templates (Java generics) and (d) the combination of templates and covariant return types.

alternative	figure	reading only [ms]	reading & writing [ms]
current architecture	6.13	9632	16502
lazy attributes	6.14 (a)	9276	15462
covariant lazy attributes	6.14 (b)	9444	15237
Java generics	6.14 (c)	5662	11936
covariant Java generics	6.14 (d)	1235	3637

Table 6.3: Runtime measurements of alternatives to eliminate type casts. 10000000 node and edge instances were created and 100000000 times the start node was accessed by two different reading methods.

In Figure 6.14 (a), the method names include the name of the return type as suffix, because in the type system of Java 2, an overridden method must specify the same return type as the method in the superclass.

A *covariant* type system allows that an overridden method in a subclass may return an object, whose type is a subtype of that returned by the method it overrides [McCluskey 2004; McCluskey 2005]. This feature was introduced in Java 5 along with Java generics [Bracha 2004], the template mechanism of Java. Figure 6.14 (b) depicts lazy attributes realized by covariant reading accessors.

Figure 6.14 (c) shows the modifications of the architecture, which arise from defining template parameters for the type of nodes in edges. Furthermore, as provided by Java generics, the type of the individual template parameters is restricted by appropriate node types. Similarly, the nodes can be parametrized by the types of edges. In fact, this alternative can be obtained from Figure 6.13 by inserting template parameters and replacing the node types. By applying templates, most type casts can be removed from the source code. In the generic byte code, the concrete type in the base class is represented by the minimum restriction of the template parameters, i.e., in our case by `Node`. The type conformance is kept by the compiler by automatically inserting type casts.

In Figure 6.14 (d), Java generics as described for Figure 6.14 (c) are combined with covariant return types. Thereby, the accessors are redefined in each class and via the type restrictions of the template parameter, covariant return types are introduced.

To optimize *SugiBib* for speed by removing type casts, the minimum invasive but maximum effective method should be applied. Therefore, we simulated the five alternatives depicted in Figure 6.13 and 6.14 as follows: For each alternative, 10000000 times an edge and a node object was created and the node object was set as start node of the edge. Then, the node object was accessed 10 times by the appropriate getter for `Node` and `WorkNode`. The runtime results are summarized in Table 6.3.

On the one side, lazy attributes require minor changes to the existing classes by introducing additional abstract typed or covariant reading accessors but also a large number of new leaf classes. On the other side, we can assume that the runtime will not significantly change, because the real number of accessors called on usual graphs in *SugiBib* is lower than the number of simulated test runs. In fact, the same is true for simply applying Java generics, but refactoring *SugiBib* towards

covariant Java generics will help getting rid of the type cast bottleneck. This suggestion also depends on the choice of the version of the underlying programming language. If Java 2 must be kept because of compatibility issues, the lazy attributes pattern helps simplifying the source code by eliminating type casts. But if the source code is not restricted to Java 2 style or the restrictions will be released in the next future, the covariant Java generics architecture is suggested as target for refactoring.

When comparing a pure Java implementation with the runtime of natively compiled programs written in other programming languages, it should always be considered that the JVM interprets the resulting bytecode which, beside various security issues, is one of the reasons of negative runtime performance. Hence, a comparison should also always include timing results gained from a native compiled version. The results of applying a native compiler to *SugiBib* were shown in Section 5.3.

Performance tuning may improve the runtime but also may cause runtime errors due to hidden dependencies, which have not been considered accidentally. Therefore, without excessive testing, as discussed in Section 6.4, no performance tuning should be done.

## 6.6 Extensions & Modifications to *SugiBib*

Every extension of knowledge arises from making the conscious the unconscious.

Friedrich Nietzsche (1844 – 1900)

So far in this thesis, we have discussed various aspects of aesthetics and automatic layout for UML class diagrams. Furthermore, we introduced a prototypical implementation. For practical use, several questions on how to use the implementation were left open. In this section a brief guide on using or extending the implementation will be given.

A native compiled executable of the latest stable version can be obtained from <http://www.sugibib.de>. Currently only native versions for windows, which include an installer and some examples, are available. Also the related documentation generated by JavaDoc is available online for browsing.

The source code of *SugiBib* can be obtained via CVS access only<sup>21</sup>. The source code is prepared for development with the eclipse<sup>22</sup> IDE. Scripts for generating the hypertext documentation, generating the JAR archive or running *SugiBib* under Linux and Windows are included. For example the script to run the AWT browser application for Linux is located in `bin/linux/startAWT.sh`. Without parameters, the GUI is started and an input file can be selected. When calling the script with the parameter `--help` the command line argument help is displayed.

Before considering changes to the framework, always the hypertext documentation should be consulted. Beside a description of all accessible methods and signatures, it provides additional important details related to the release and the implementation.

<sup>21</sup>The information required for the access is available from the author on justified request.

<sup>22</sup><http://www.eclipse.org>



- **How to add a certain plug-in, e.g. an input format reader?** Beside choosing the appropriate interface, abstract class or basic implementation, a new plug-in needs to register itself, usually by calling a static method in one of the parent classes. But how can a plug-in register itself, if no class outside executed in the framework holds a reference to the plug-in?

This is done by the plug-in mechanism, which searches for plug-ins while starting an application. The basic classes involved in that mechanism are depicted in Figure 6.15. A `PluginManager` uses specialized resource locating implementations by considering the current classpath and the bootstrap classpath of the JVM. To avoid searching the entire framework for plug-ins, certain packages can be specified to be considered. Optionally,

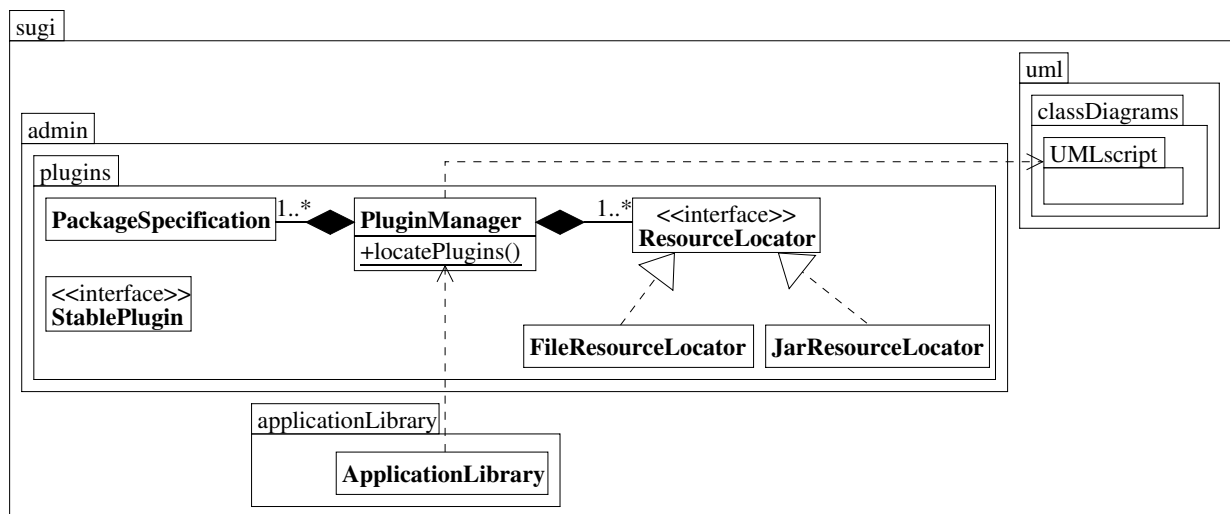


Figure 6.15: The basic relations of the plug-in mechanism in *SugiBib*.

these packages can be traversed recursively for finding plug-ins. Some standard packages like the one containing the parser for UMLscript are referenced by default. To realize a quick mechanism not implying slow type checking or reflection mechanisms, the `PluginManager` simply loads and initializes each class in the specified package. Hence, a plug-in simply has to provide a static initializer which registers itself as mentioned above. As long as own plug-ins represent non-standard extensions, certain runtime switches at JVM level can be used to add packages to be searched. For native or obfuscated versions ASCII tables which contain the classes to be loaded by the plug-in mechanism have to be provided. These tables are located in the resources directory, which can transparently be accessed by `sugi::admin::resources::FileResources` even if *SugiBib* is loaded as a library into the JVM or provided via network mechanisms.

*SugiBib* distinguishes between a development and a snapshot mode, which is activated for a public release. For example, the snapshot mode disables runtime consistency checks as well as plug-ins currently being developed. Therefore, stable plug-ins have to be marked by the interface `StablePlugin` to be visible in a snapshot version.

- **How can the basic applications be modified, e.g., to provide more specialized menu items or mouse operations?** Mouse operations and general postprocessing actions, like coloring the entire graph according to certain rules, can simply be added as interactive or automatic plug-ins, respectively. The menus in the browsing applications and partly the command line options will automatically be adjusted to handle the new features. Deeper modifications in the application library should only be considered, if changes to all applications should be performed, which cannot be handled by plug-ins. In the case of GUI and platform specific changes, the applications itself can be modified (or, less preferable, subclassed). For example, the Swing browser application extends the standard menus by adding a menu item, which allows switching the look & feel of the GUI.
- **How to reuse the *SugiBib* algorithm in another project?** First, the application library can be reused by writing a new frontend class and passing the appropriate data through that class into the library. It might not be appropriate, to reuse the application library in a completely different application, because it handles lots of internal details for providing the user with various options. Hence, parts of the basic mechanisms of that library may have to be rebuild. Basic examples on directly calling the algorithm can be found in the Examples package.

As shown in most of the illustrating class diagrams in this thesis, the main flow of control for the layout algorithm is implemented in `sugi::uml::classDiagrams::InfoIIAlgorithm`. The method `execute` of that class receives two parameters, an `InputGraph` and a `GraphicsDelegate`.

The graphics delegate can simply be obtained by instantiating the appropriate graphics delegate factory from one of the subpackages in `sugi::applicationLibrary` and calling `createGraphicsDelegate` on that. A valid input graph can simply be obtained by facilitating one of the input format readers for UMLscript, XUMLscript XMI or XMI[DI]. Usually, the readers are automatically registered in `sugi::io::ContentsImporter` and the most appropriate one is selected by the application library. Therefore, similar to the application library, the loading of plug-ins has to be initiated by calling `PluginManager.loadPlugins()`. By iterating over all registered contents importers, the correct one can easily be retrieved. When calling such an input reader, the source and an application instance implementing `sugi::applicationLibrary::IApplication` has to be specified. From the viewpoint of an algorithm, an application is an object, which which may receive messages from the input and the rendering process, provides information on the state of processing and returns the current graphics delegate instance. Hence, this interface provides signatures to directly cooperate with the calling application and can easily be implemented by an outstanding class.

- **How can the sequence in the layout algorithm be changed?** As discussed in this thesis, the sequence of the steps in the main layout algorithm was chosen carefully to meet the priorities of the aesthetic criteria. Therefore, usually changes to the sequence of steps are discouraged. The processing sequence of certain steps within a macro phase like pre-processing (S1-S9), rank assignment (S10), edge crossing reduction (S11), intermediary

processing (S12-S14), coordinates assignment (S15) and postprocessing (S16-S21) might be changed. Steps like creating the result graph (S21) or determining the pseudo-hierarchy (S3) should be kept in their absolute or relative position, respectively.

- **How can individual facilities of the display of elements in UML class diagrams be changed or extended?** The individual elements of UML class diagrams are realized via information classes attached to nodes and edges. The basic interfaces can be found in `sugi::uml::classDiagrams::basics` and the concrete implementations in `sugi::uml::classDiagrams::standard`. By implementing the basic interfaces, new classes can be provided. By subclassing, existing ones can be modified according to the principles of object-oriented programming. Finally the new implementation has to be made accessible through an appropriate modification of the information object factory. A discussion on using inheritance for reconfiguration was given in Section 6.3.1.
- **How can a new layout step, like one of those in the main algorithm, be integrated? Can parts of the implementation be replaced by other, alternative or faster realizations?** As long as new steps fulfill certain consistency conditions, new or alternative steps can be integrated into the layout process. As defined in the preceding chapters, individual steps require various definitions to be respected. Furthermore, information, required by following steps through the graph copy mechanism has to be provided by the new or alternative implementation. Therefore, some of the basic interfaces have to be implemented depending on the macro phase the new step should be used in.  
As a first task it should always be investigated, if existing classes can be used to realize the desired modification, because these classes meet most of the basic mentioned described above. Furthermore, modifications to the source code of the class `InfoIIAlgorithm` are required to respect the new implementation, because the current layout algorithm does not allow dynamic changes of the sequence of algorithmic steps.  
Each graph class provides various factory methods required to realize the graph copy mechanism. If new node or edge classes, extended rank descriptors or another rank set implementation should be used by the new graph, appropriate instances have to be created by the factory methods. Even the graph itself has to follow this principle. An important part of the copy mechanism is implemented in the constructors of the basic classes. Therefore, own constructors should be declared similar to the existing ones of the superclasses and the new one must call the appropriate constructor of the superclass. The rules, which have to be respected for the graph copy mechanism, might also have been realized by dynamic mechanisms like runtime reflection. We still rely on the static handling which was introduced in the first version of *SugiBib*, because a reflection based implementation is time consuming (O18), in particular, when methods are called dynamically or instances are created frequently. Introducing new node and edge classes should be avoided, because copying the graph slows down the execution. In future releases, inner classes might be used to make factory methods and template methods more obvious.  
Some tips: If unexpected class cast errors or null pointer exceptions occur after inserting a new step in code, which belongs to the framework itself, it should be ensured that the new

instances meet the requirements of the following algorithmic steps. Furthermore, the new factory methods should be investigated, if the appropriate instances are created. Similarly, the copy constructors or clone-like methods should be reviewed for the appropriate super class constructor invocations and if all (relevant) attributes have been considered. To improve the speed of the implementation, *SugiBib* heavily relies on pooling instances (O3). If totally unexpected results or runtime errors occur, some of the explicit memory management calls defined in `sugi::admin::utilities::ObjectPools` might have been mixed up. In each of these mysterious situations, we recommend to activate the runtime consistency checks while can be enabled by the appropriate flag in `sugi::GlobalFlags`. This will also enable runtime checks which terminate the execution in the case that instances are released twice to the global pools.

- **Can *SugiBib* also be applied to other types of diagrams?** Due to the alignment to the Sugiyama algorithm, *SugiBib* primarily is intended to apply a hierarchical layout algorithm. But this is no restriction, because other layout algorithms (in individual classes) or new layout steps can be introduced as described above. Similarly, by combining parts of the existing algorithms, a general hierarchical layout algorithm can easily be obtained. According to the desired application domain, the information classes for UML class diagrams might not be appropriate. Therefore, if non-UML diagrams should be drawn, the packages located in `sugi::uml` can be ignored completely. An example for drawing a general graph by the Sugiyama algorithm implemented in *SugiBib* is given in the package `Examples`. Thereby, nodes are displayed as circles and edges are shown as simple straight lines. When considering an implementation of other diagrams relevant for software engineering, the following ideas can be taken into account:
  - UML statecharts can be laid out by ViSta [Davidson and Harel 1996; Castelló et al. 2001; Castelló et al. 2002; Castelló et al. 2003], another, specialized hierarchical layout algorithm. Also Statemate or higraphs [Harel 1988] seem to be an appropriate approach.
  - Aesthetic principles and a layout algorithm for UML sequence diagrams were discussed in [Rhode 2003].
  - In [Six and Tollis 2002] a linear time algorithm for processing flowcharts with top-left to bottom-right direction was discussed. A flowchart algorithm, which also respects swimlines and object flows, might be appropriate, because in the current version of UML, activity diagrams are closely related to flowcharts. A grid based algorithm for data flow diagrams as discussed in [Protsko et al. 1991] might also be a starting point.
  - Outside the UML, non-standard approaches like [Davis et al. 2003] for layout of UML diagrams in three dimensions or Crococosmos [Lewerentz et al. 1995; Lewerentz and Noack 2003] might be interesting.

As a final hint to own modifications of *SugiBib*, we repeat the programming rule discussed in Section 4.7:

*Solve a layout problem at the point of time when it occurs and try to avoid postprocessing.*



# 7 Drawing Class Diagrams – an Ongoing History

---

## 7.1 Future Work

Work is a necessary evil to be avoided.

Mark Twain (1835 – 1910)

A lot of research has been invested to explore algorithms for drawing UML class diagrams automatically. We have designed an algorithm, which is capable of realizing the complex relationships between diagram elements in UML class diagrams and the aesthetic criteria defined in Section 3.3.6. As shown in Section 6.4, plenty of code has been produced for realizing the algorithm by a concrete program. Does future work simply consist of maintenance, optimization and improvement?

Even if all elements of UML class diagrams have been considered in our approach, not all alternatives have been realized. In the first version of *SugiBib*, a naive postprocessing in the result graph supported UML\_JOIN. This implementation was disabled, because it was able to consider generalizations only, did not work in all cases and was not compliant to the requirements of XMI[DI]. Joining dependencies, aggregations, compositions or generalizations while coordinates assignment would be a desirable feature.

In Section 2.1, when UML and more specifically class diagrams have been described, it was mentioned that the future issues of the next version of UML [OMG 2003d] have not been considered in this work. Fortunately, for class diagrams only few changes are required to meet [OMG 2003d]. Subsystems have been removed and therefore the complexity of the layout algorithm considering illegal edge crossings will be reduced. To support the specification of component architectures, so called compound classifiers as well as a refined “lolly” notation have been introduced. Furthermore, wiring of components including explicit ports at the border of the components and directed connectors in the sense of a secondary flow from the ports to the contained components can be specified. As described, the layout algorithm of *SugiBib* is able to handle compounds and the secondary flow can be considered by adjusting the various edge crossing reduction plug-ins. Stacked layout of classes, i.e. like the tip-over convention known

in graph drawing, was also introduced [OMG 2003d]. In the first version of *SugiBib*, a simple postprocessing feature was already present to realize a similar feature, but a more specific rank assignment for tip-over layout would be a more appropriate realization.

So far, in most algorithms only the elements, which directly participate in a class diagram are considered. Further information may be used as hints for the layout algorithm to place even disconnected or loosely connected elements in a more pleasing and semantical sufficient way. We mentioned invisible relations to accomplish this task. The data can be deduced from input information, which explicitly has been marked as invisible, or which can be retrieved from a repository by taking relations of the underlying model outside the diagram into account. Also constraints from graph drawing could be interesting for a realization. In [Waddle 2001], dynamic constraints were realized by proxy nodes (similar to our composite nodes) and a postprocessing step of the rank assignment. Further work on constraints, like [Böhringer and Paulish 1990; Protsko et al. 1991; Kamps et al. 1996; Tamassia 1998; Brockenauer and Cornelsen 2001; Brandenburg et al. 2003] can be considered, but complications on introducing conflicting constraints are also known: It is easy to accidentally define conflicting constraints [Hansen et al. 2002].

Currently, the set of metrics discussed in Section 5.1 used to measure the aesthetic criteria introduced in Section 3.3.6 is not complete. Further metrics have to be formalized and implemented. The more metrics are available, the more the results of other UML specific layout algorithms and UML tools will become comparable. Using a UML tool as input mechanism, a user study on validating the aesthetic criteria as proposed in Section 3.3.7 can be conducted. In such a study, our focus would be on UML experts and system analysts similar to the study mentioned in [Dwyer 2001]. The data collected by conducting the study can be stored in a standard exchange format like XMI[DI] and analyzed by a metrics implementation, e.g. , the one provided with *SugiBib*. As a side effect, hypotheses on finding rules for automatic deduction of hierarchies might thereby be examined. Furthermore, having a validated set of aesthetics and metrics at hands, non-deterministic layout algorithms can be driven and controlled by them.

Various improvements on the drawing mechanisms can be considered for integration with *SugiBib*. So far, we did not consider sophisticated mechanisms for edge labels. For text adornments, which are not closely related to nodes, mechanisms like those mentioned in [Kakoulis and Tollis 1997a; Kakoulis and Tollis 1997b; Kakoulis and Tollis 1998; Binucci et al. 2002] might improve the results. Furthermore, fast crossing calculation [Chazelle 1986; Chazelle and Edelsbrunner 1992; Barth et al. 2002] as well as layer-independent crossing number calculation or floorplanning techniques as in VLSI or [Messinger et al. 1991; Hershberger and Snoeyink 1994; Sander 1996a; Dobkin et al. 1997] can be taken into account. Decomposition of a graph into trivial or disconnected subgraphs and replacement into multiple macro-layers [Messinger et al. 1991] as well as local layering [Schreiber 2002; Brandenburg et al. 2003] may also improve the results but might also collide with UML\_HIERARCHY.

Beside the theoretical and algorithmic improvements, which may arise from realizing the features mentioned above, various runtime optimizations as discussed in Section 6.5 can help tuning the execution speed. But also structural changes to the source code may improve the readability and understandability. For example, the method and class names can be checked for conformance to an internal naming convention and configuration as well as template methods might be made more obvious by refactoring them into dedicated inner classes.

As discussed in Section 2.1 and 3.1, navigation techniques, in particular for CASE tools, are currently out of scope for this work. Basic semantical zooming, like class or package folding, as well as edge type changes as discussed in [Köth 2001] can easily be realized. Thereby, the mental map would basically be kept by *SugiBib* due to the normalization in S2. A technique, which is often used for navigation, can be realized by coordinates transformations, like fisheye-views [Formella and Keller 1996; Storey and Mueller 1996; Köth 2001]. Unfortunately, these techniques introduce hard mental operations due to implicitly comparing the resulting diagrams with the default UML layout. Automated abstractions as described in [Egyed 2002], which support physical, logical and goal-driven refinement, seem to be more appropriate from the viewpoint of software engineering. Also a semantic and context sensitive navigation technique, based on dynamically generated diagrams and information paths retrieved from the information stored in the repository of a CASE tool, might be interesting for future work.

One major future research interest will be the realization of an appropriate mechanism for incremental layout (REQ\_INCREMENTAL\_ALGORITHM). Due to our application domain, structural and semantical information on changes between two diagrams can be collected and supplied in a (currently not) standardized format. Similar to taxonomies for program changes as discussed in [Gustavsson and Assmann 2002; Gustavsson 2003], changes for UML diagrams can be categorized and collected [Briand et al. 2003]. The following types of changes may be considered:

- Modifications of elements in a node (change of a name, adding, deleting or modifying attributes or operation signatures) or at an edge (adornments, name, stereotype, qualifiers, etc.)
- Adding or removing nodes or edges.
- Changing the containment of classes in packages, models or subsystems.
- Collapsing/expanding subgraphs or compound nodes.

Due to the normalization in the preprocessing of the *SugiBib* algorithm, simple changes are handled implicitly. To minimize changes induced by the other categories, the algorithm has to be extended at various points: Detection of the pseudo-hierarchy and the rank assignment on the input, mechanisms for keeping unchanged nodes in predefined vicinity, etc. The difference between two UML diagrams in sequence can be retrieved as follows: On the one side, the UML tool, in particular, the editor of the tool and the history of the repository, can directly provide the change information. On the other side, the information can be deduced from structured formats like XMI [Ohst et al. 2003]. In graph drawing, usually constraints are used to implement incremental layout features as described in [Böhringer and Paulish 1990; Misue et al. 1995; North 1996; Storey and Mueller 1996; Köth 2001; Waddle 2001]. But it seems to be a hard task to integrate constraints into the Sugiyama and therefore also into our approach as mentioned in [Waddle 2001]. Hence, we believe that smaller changes can be handled by encapsulating unchanged structures in composite nodes and running the algorithm as usual. When the differences between two inputs get too large, iteratively smaller parts of the changes might be processed instead of handling all in one step. Additional metrics for incremental features could be based on



[Diehl and Görg 2002]. Also the cluster model and the measures of distribution and differences on proximity graphs described in [Lyons et al. 1998] can thereby be taken into account.

## 7.2 Conclusions

None of the drawing methods so far presented in the literature produces pleasing drawings for all applications and for all input graphs. It is presumable that such an algorithms will never be presented.

[Mäkinen 1990]

In this thesis, we have given a description of a complete layout algorithm for UML class diagrams according to the UML 1.x specifications. That layout algorithm was developed with respect to a set of requirements (Chapter 3), which take the viewpoints of graph drawing, HCI, software engineering and software visualization into account. To enumerate these requirements, we have presented an intensive discussion on aesthetic principles for diagrams in general and UML class diagrams as our target application domain. Our unique set of aesthetic principles was derived from the disciplines mentioned above and the UML specification as the only source, which describes the validity of UML class diagrams.

The general process flow of the layout algorithm for (mixed) compound graph was given in 7 macro steps and 21 individual processing steps as an extensive modification of the well known Sugiyama algorithm. In Chapter 4, we have also described extended algorithms from graph drawing for rank assignment, edge crossing reduction and coordinates assignment. Furthermore, we have introduced a penalty mechanism for invalid edge crossings, techniques for producing cluster valid graphs, the hierarchical edge crossing reduction technique and several detail algorithms in coordinates assignment. To realize the priorities of our aesthetic principles, further processing steps dedicated to elements specific to UML class diagrams were introduced: A grouping mechanism for nodes, namely composite nodes, which is responsible for the implementation of association classes, comments and hyper edges. Furthermore, scaling techniques and spatial distribution of nested elements to emphasize important issues of software engineering like coupling were invented.

To objectively compare layout results of layout algorithms, a basic set of application domain specific metrics for the aesthetic principles was introduced. Furthermore, the algorithmic framework *SugiBib*, which realizes the layout algorithm, was extended by mechanisms to communicate with other UML tools, e.g., by direct JMI-repository access or the standardized data format XMI[DI]. An example of applying these metrics was given in Section 5.2, where two graph drawing approaches for UML class diagrams, which both do not fully implement the UML specification, were compared with our approach. Having the implementation of layout metrics at hands, a new testing mechanism, the metrics based regression testing for (UML class diagram) layout algorithms was invented and applied to *SugiBib*. That mechanism produced detailed information, which was visualized in Section 5.3, to describe runtime and scaling issues of the layout frame-

work. As a conclusion of the runtime measurements, we have presented a list of performance tuning opportunities for Java programs while describing the architecture of GUI-independent framework *SugiBib* in Chapter 6.

In fact, drawing UML class diagrams automatically appears to be not only a world of its own as a kind of overlapping between different disciplines, but more a universe of various worlds with complex interactions. In the introduction of this thesis, a software engineering fairy tale was given, in which software engineers had built themselves a perfect world to work in. Obviously, this world is not reality now, because neither the (visual) languages for specifying software nor most of the tools and their concepts nor the disciplines reasoning on perception and layout of diagrams are currently able to realize the entire dream world.

One of the reasons why this software engineering fairy tale is currently not realized, is the problem of interpreting and realizing standards. Instead of simply realizing the UML specifications and the related standards, most of the tools introduce their own habitat in the world of UML. As shown in Section 2.2.3, most of the CASE tools have been identified to be not fully compliant even to ancient versions of UML, rely on own interpretations of the standards, hence do not properly support the migration between tools and usually provide a low quality layout mechanism. Most of the tool vendors prefer picking popular parts from UML and thereby realizing own sublanguages of UML in contradiction to our approach (REQ\_COMPLETE\_UML, REQ\_COMPLETE\_DIAGRAM). As also described in Section 2.2.3 and Section 5.2, vendors of layout algorithms and plug-ins for UML class diagrams apply the same simplifications to fit into the world of UML tools.

Is the *SugiBib* algorithm (one of) the final solutions for drawing UML class diagrams?

Different approaches each with its own advantages and disadvantages may exist. In fact, tradeoffs occur between the diagrammatic, the computational and the heuristical complexity. Reducing the diagrammatic complexity to gain a more handy and maintainable algorithm or implementation is generally not acceptable due to the nature of UML as an international standard. Hence, computational and heuristical complexity increase.

It is a natural approach to select a layout algorithm, which supports by its construction important structural and semantic aspects of the diagrams to be drawn. Hence, the selection of these aspects, which can be seen in our case as a philosophy of drawing UML class diagrams, has major influence on the choice of the layout algorithm. GoVisual and yFiles, the proposals from the graph drawing community, rely on a modified topology-shape-metrics approach. Additional constraints were used to produce drawings, in which inheritance (and realization) edges form a hierarchy. Both algorithms support aesthetic criteria common to general graphs and well-known to the graph drawing community. Taking into account the default layout rules implicitly introduced by UML, a set of application domain specific layout rules can be deduced. From our point of view, an approach, which is based on a hierarchical algorithm and, therefore, naturally supports hierarchy and nesting as the most important aesthetic principles for UML class diagrams, is more appropriate. Finally, the decision on the best layout algorithm for UML class diagrams will be deferred into future, depending on the acceptance of UML, CASE tools and automatic layout mechanisms by the users.

The application domain specific aesthetic principles, the layout algorithm, which is capable of realizing these principles, the basic set of metrics for measuring the aesthetic principles on a concrete diagram and the prototypical implementation of the layout algorithm discussed in this thesis represent a contribution to various disciplines of computer science.

The set of aesthetic principles represent the foundation for work on a standardization of the layout of UML diagrams to simplify the communication between human beings involved in the software development process (software engineers, stakeholders, etc.) This might introduce a technique like "quality engineering by layout", an improvement of the quality of the development process and the produced software by the layout of software engineering diagrams.

For software engineers as well as software visualizers, this work shows that also the complex and more exotic elements of class diagrams, which are responsible for large degree of freedom as well as their own requirements can be considered by an automatic layout algorithm.

This also may influence the graph drawing community. We have contributed some ideas and algorithms in rank assignment, edge crossing reduction and coordinates assignment for drawing application domain specific hierarchical (mixed) compound digraphs. Furthermore, we have shown that, by combining the knowledge of four computer science discipline, the basic aesthetic principles from graph drawing can be extended to characterize the requirements and aesthetics of UML class diagrams. A step to also consider more application domain specific rules in drawing diagrams, analyzing them by metrics and automatically testing an implementation by regression tests.

# A1 Example Drawings by *SugiBib*

---

So far in this thesis, plenty details on algorithms for drawing UML class diagrams automatically have been discussed, but most class diagrams were produced manually using xfig<sup>1</sup>. To emphasize certain aspects in the drawings, we did not use one of the (probably non-compliant) CASE tools for this task. In this section, some diagrams drawn by *SugiBib*, the prototypical implementation of our layout algorithm for UML class diagrams, will be presented. The nodes in the diagrams are scaled to minimum area occupied by their contents and comments are currently disabled. Color or shading is assigned to individual nodes due to package containment as a non-standard UML feature. Visual artifacts in some drawings arise from bugs in the implementation, which have not been fixed so far.

---

<sup>1</sup><http://www.xfig.org>

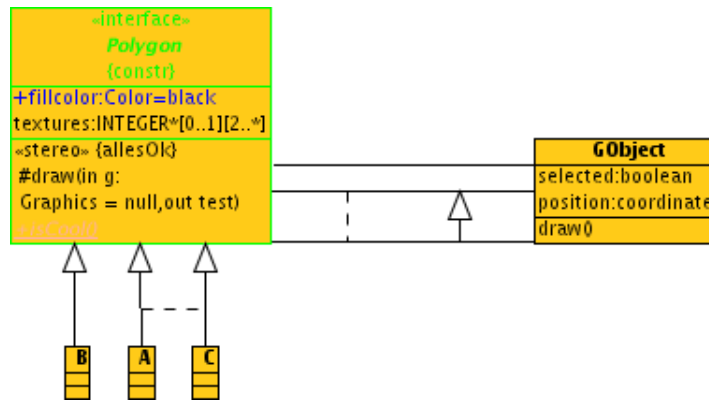


Figure A1.1: *SugiBib* drawing of some hyperedges (hierarchical edge crossing reduction).

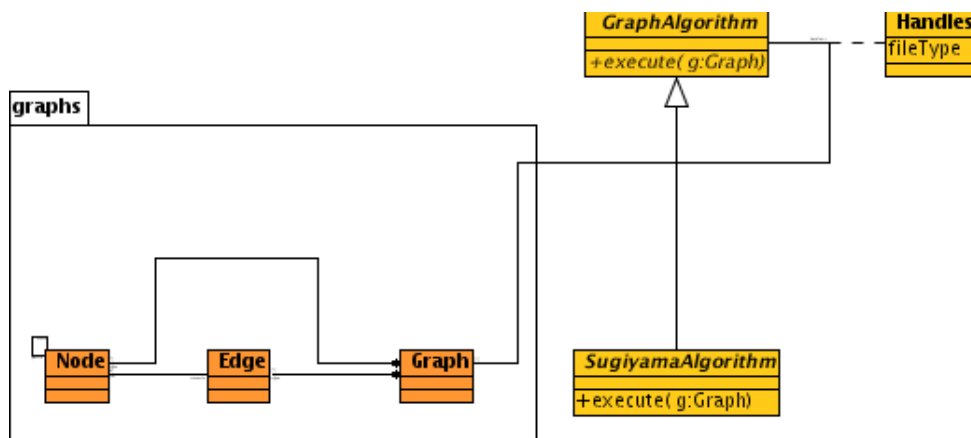


Figure A1.2: *SugiBib* drawing of Figure 2.9 using the hierarchical edge crossing reduction without optimization for association classes. Some additional rank assignment heuristics considering hierarchical subtrees of an initial forest may help to produce a drawing as shown in Figure 2.9.

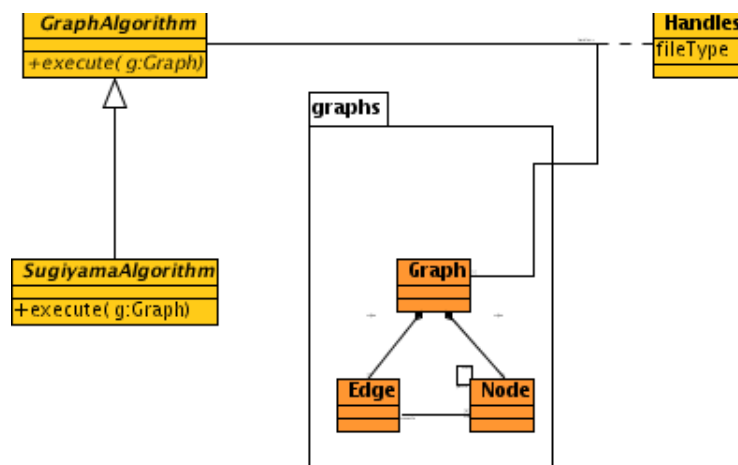


Figure A1.3: *SugiBib* drawing of Figure 2.9 using the hierarchical edge crossing reduction without optimization for association classes. Aggregations are included in the pseudo-hierarchy by user selection.

UMLscript source code of Figure A1.2 and A1.3:

```

UMLscript VERSION 1 MINOR 14 { *ApplyPackages, *reflectiveAssociations }
DIAGRAM
  CLASS {abstract} GraphAlgorithm
    OPERATIONS
      {abstract}+execute(g:Graph)
    RELATIONS
      GENERALIZE 'SugiyamaAlgorithm'
      ASSOC handles >
        ASSOC-CLASS
        CLASS Handles
          ATTRIBUTES
            fileType
          TO [1] 'graphs::Graph'
      END
  CLASS {abstract} SugiyamaAlgorithm
    OPERATIONS
      +execute(g:Graph)
  PACKAGE
  // COMMENTS 'encapsulates graph-related structures' END COMMENTS
  graphs
  SUB DIAGRAM
    CLASS Graph
      RELATIONS
        ASSOC composite TO [*] ROLE edges 'graphs::Edge'
        ASSOC composite TO [*] ROLE edges 'graphs::Node'
      END
    CLASS Node
      RELATIONS
        ASSOC ROLE parent TO ROLE child 'graphs::Node'
        ASSOC [2] ROLE end TO ROLE connects 'graphs::Edge'
      END
    CLASS Edge
  END DIAGRAM
END DIAGRAM

```



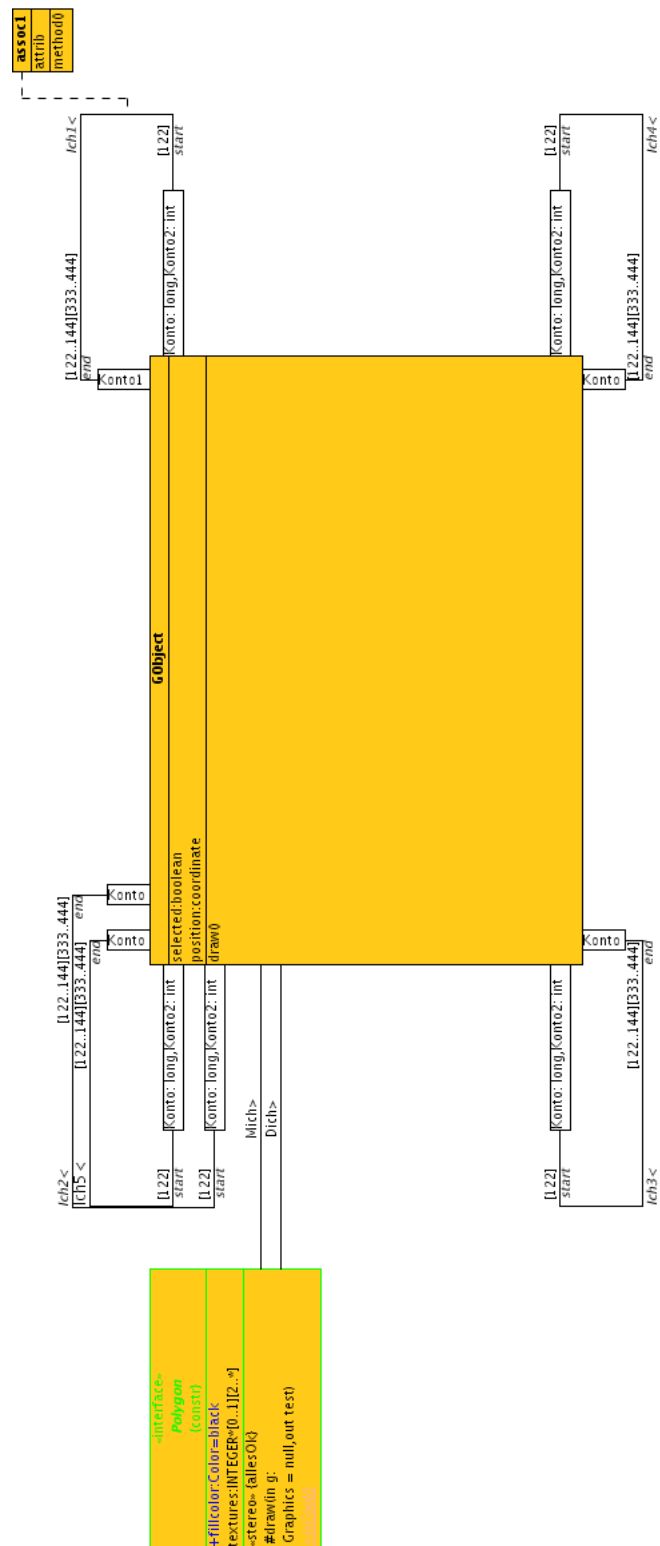


Figure A1.5: *SugiBib* drawing of some reflective edges and an association class at a reflective edge. Scaling of nodes and overlays of reflective association should be improved.







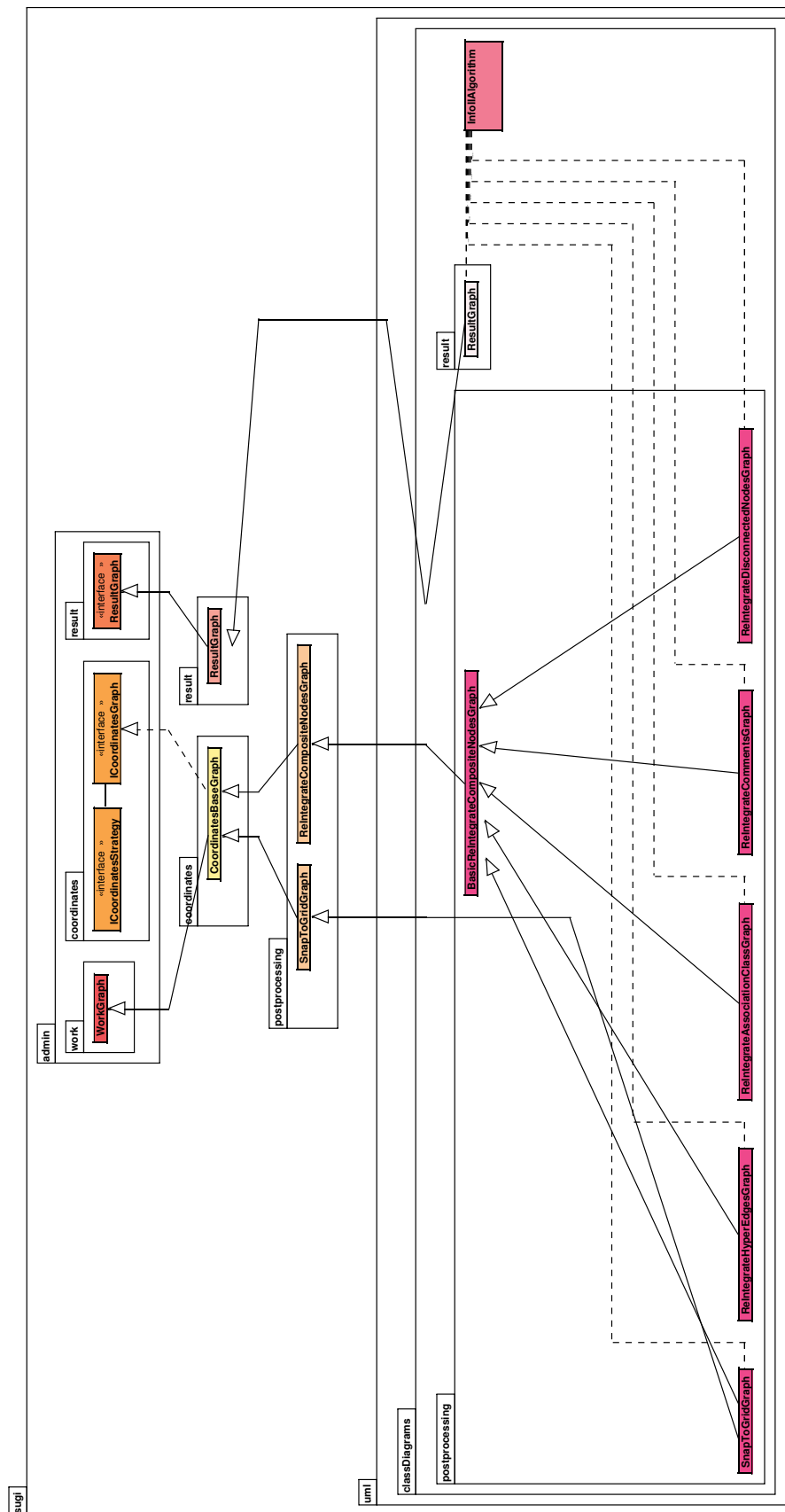


Figure A1.8: *SugiBib* drawing of Figure 6.6: hierarchical edge crossing reduction and nodes hopping are activated.

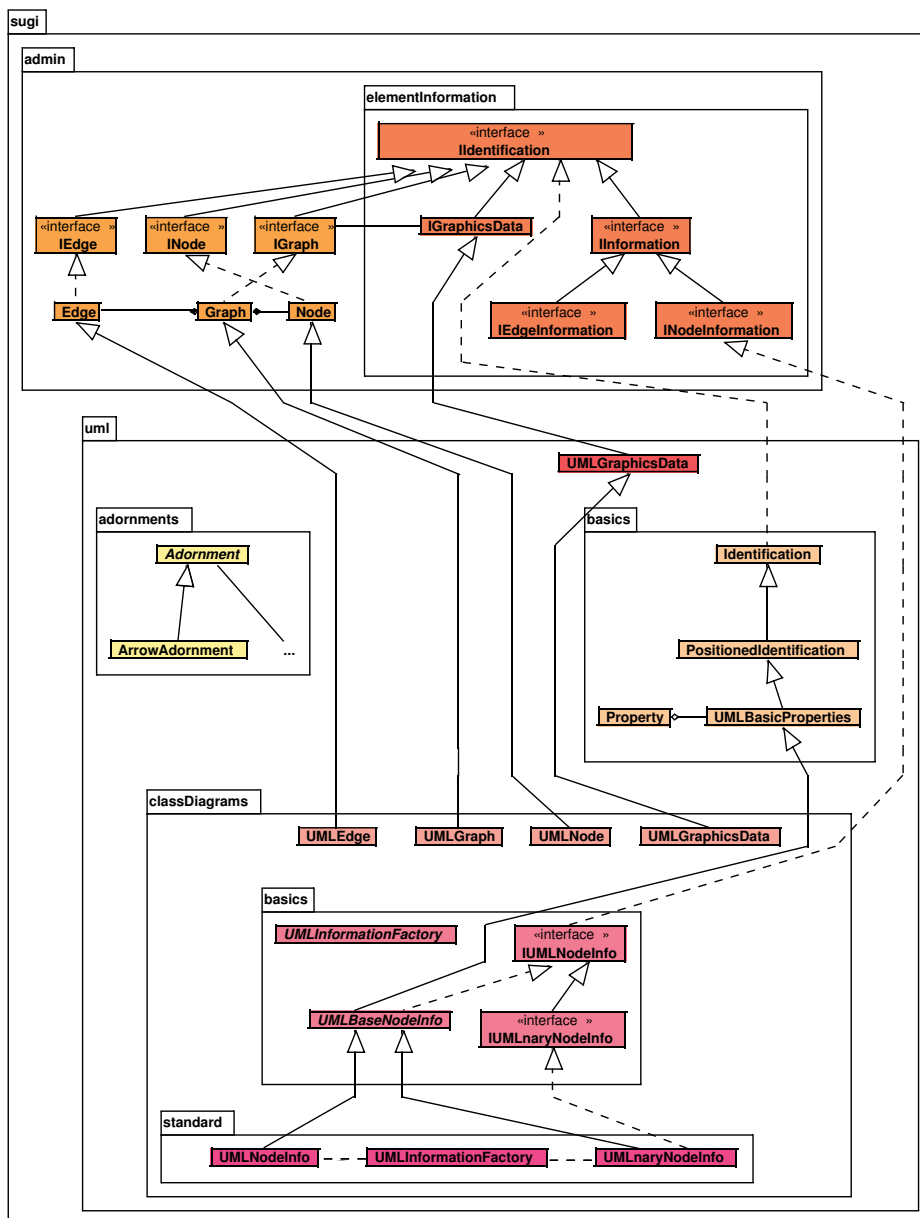


Figure A1.9: *SugiBib* drawing of Figure 6.7: hierarchical edge crossing reduction and nodes hopping are activated.





## A2 Lists and Hashtables

---

In this section, the formal definition for lists and hashtables, which are used in this thesis are given. At a first glance, for these data structures, a hint that the operations will behave as known from Java, would be sufficient. But then implicit behavior like returning  $\perp$  (null in Java) in certain situations would not become obvious. Therefore, directly implementing the pseudocode algorithms in another programming language or using other collection implementations would lead to several errors and tedious debugging.

### Definition 27 (lists)

A list  $L$  is defined as

$$L : \{0 \dots (n-1)\} \rightarrow O, n \in \mathbb{N}_0$$

where  $o \in O$  denotes an arbitrary object. We assume an empty list for  $n = 0$ . Furthermore, we will internally denote in this and subsequent definitions  $L_l$  as the  $l$ -th element in the sequence and  $L_l \rightarrow o$  for element assignments.

- a new list is created by  $L := \{\}$
- $|L| := listSize(L) := n$
- Let  $M : \{0 \dots (n_M - 1)\} \rightarrow O, n_M \in \mathbb{N}$  be a list. Then

$$listAddAll(L, M) : \{0 \dots (n + n_M - 1)\} \rightarrow O$$

$$listAddAll(L, M)_l \rightarrow \begin{cases} L_l & : \text{if } l < n \\ M_{l-n} & : \text{if } n \leq l < n + n_M \end{cases}$$

- $listPos(L, o) := \begin{cases} i & : \text{if } \{i \rightarrow o\} \in L \\ -1 & : \text{otherwise} \end{cases}$
- $o \in L \Leftrightarrow listPos(L, o) \geq 0$

- $listAdd(L, i, o) : \{0 \dots n\} \rightarrow O, listAdd(L, i, o)_l \rightarrow \begin{cases} L_l & : \text{if } i < 0 \vee i > n \\ L_l & : \text{if } l < i \leq n \\ o & : \text{if } l = i \\ L_{l-1} & : \text{if } i < l \leq n \end{cases}$
- $listAdd(L, o) := listAdd(L, n, o), listAddFront(L, o) := listAdd(L, 0, o)$
- $listGet(L, i) := \begin{cases} o & : \text{if } \{i \rightarrow o\} \in L \\ \perp & : \text{otherwise} \end{cases}$
- $\{o : o \in L, property(v)\}$  constructs a new list in which the element sequence from the source  $L$  is (partly) present in the result set. *property* denotes an arbitrary selector function on  $o$ . The element before the colon in specifies the elements to be part of the target list.
- $\{o, \dots\}$  is a list constructed according to the given sequence of elements.
- let  $M$  be a list, then  $L \cup M := listAddAll(L, M)$  is defined as a shortcut.
- $L \setminus M := \{o : o \in L, o \notin M\}$
- $L \cap M := \{o : o \in L, o \in M\}$
- $L = M \Leftrightarrow \forall_{o \in L} o \in M \wedge \forall_{o \in M} o \in L$ .  
 $L =^* M \Leftrightarrow L = M \wedge \forall_{o \in L} listPos(L, o) = listPos(M, o)$  also requires equal sequences.
- Similar to sets,  $\wp L$  denotes the powerset of lists of  $L$ .

### Definition 28 (hashtables)

Let  $K^* := K_0 \setminus \{\perp\}$  be the set of keys to be used with a hashtable where  $K_0$  denotes an arbitrary set of keys.

$$H : K \subset K^* \rightarrow O$$

is a hashtable  $H$  probably mapping from a certain key set  $keys(H) := K$  to an element  $o$ , which is member of an arbitrary set  $O$ . Let  $k \in K^*$ .

- a new hashtable is constructed by  $H := \{\}$
- $hashPut(H, k, v) : K \cup \{k\} \rightarrow O$  via

$$hashPut(H, k, v)_l \rightarrow \begin{cases} w & : \text{if } l \neq k \wedge \{l \rightarrow w\} \in H \\ v & : \text{if } l = k \end{cases}$$

- $hashGet(H, k) = \begin{cases} v & : \text{if } \{k \rightarrow v\} \in H \\ \perp & : \text{otherwise} \end{cases}$



- $hashRemove(H, k) : K \setminus \{k\} \rightarrow O$

$$hashRemove(H, k)_l \rightarrow w \text{ if } l \neq k \wedge \{l \rightarrow w\} \in H$$

- $hashContainsKey(H, k) = \begin{cases} true & : \text{ if } \exists_v \{k \rightarrow v\} \in H \\ false & : \text{ otherwise} \end{cases}$

# A3 Improved Algorithms

---

While revising the realization of the framework, shortly before publishing this thesis, we were able to improve some of the algorithms discussed in this work. These improvements are described in this section.

## A3.1 Cluster Validity

Similar to algorithm 4.11, in which the basic positions for determining the intra-rank validity were precalculated, the basic data required to make the decision on inter-rank validity can be handled. By calculating the rank-local inter-rank valid region for the node to be inserted, the validity test in algorithm 4.11, which takes  $O(|V|)$  in the naive version, can be realized in constant time. Algorithm A3.1 depicts the call of the preprocessing algorithm and the simplified test.

`prepareInterRankTest` determines the subrank of  $\sigma_r$ , which contains inter-rank valid positions for the insertion of  $v$ . This can be done by the following steps:

1. If no alignment rank  $\sigma_{r+d}$  is given, e.g. for the first rank in the hierarchy, or if  $global(v)$  holds, set  $irValid_l = 0$  and  $irValid_r = |\sigma_r|$ .
2. Retrieve the most specific cluster criterion  $c$ , which is member of  $\sigma_{r+d}$  or  $\sigma_r$ .
3. If  $c$  is not member of  $\sigma_{r+d}$ ,  $irValid_l = 0$  and  $irValid_r = |\sigma_r|$  is the result. If  $global(c)$  holds or  $c$  is member of either  $\sigma_{r+d}$  or  $\sigma_r$ , i.e.,  $c$  was not aligned to  $\sigma_{r+d}$  before, positions of  $\sigma_{r+d}$  are projected to  $\sigma_r$ . The projection is visualized in Figure A3.1. Therefore, the area of  $c$  in  $\sigma_{r+d}$  is searched, clusters not aligned so far are ignored and within the neighbored clusters the most specific aligned ones are determined. Then the corresponding positions of these clusters are retrieved in  $\sigma_r$ . The valid area in  $\sigma_r$  is returned dependent on the existence of the projected clusters.
4. Otherwise, let  $C(r, c) := \{i : 0 \leq i < |\sigma_r|, \sigma_r[i] \preceq c\}$  be the index positions of the nodes of  $\sigma_r$  contained in  $c$ . The valid area is determined by  $irValid_l = \min C(r, c)$  and  $irValid_r = \max C(r, c) + 1$

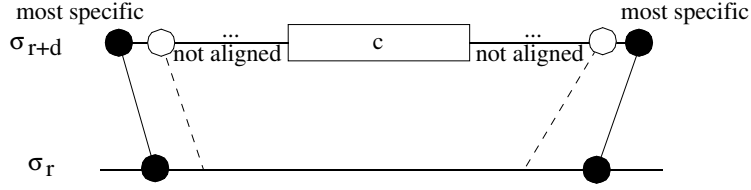


Figure A3.1: Projecting the positions from the align rank  $\sigma_{r+d}$  to the target rank  $\sigma_r$ .

---

**Algorithm A3.1** *ip\_handleNonGlobal\_improved*


---

**input:**  $\bar{G} = (V, E_H, E_N, n, \sigma), r, v, d \in \{-1, 1\}$

**output:** cluster-valid positions if  $\neg \text{global}(v)$   $\{0 \dots |\sigma_r|\} \rightarrow \text{insertPosResult}$

$sAlign := \text{augmentClusters}(\sigma_{r+d})$

$sRank := \text{augmentClusters}(\sigma_r)$

$(irValid_l, irValid_r) := \text{prepareInterRankTest}(\sigma_{r+d}, \sigma_r)$

$result : \{0 \dots |\sigma_r|\} \rightarrow \text{insertPosResult}$

$S := \{i : 0 \leq i < |\sigma_r|, v \preceq \sigma_r(i) \wedge \exists_{w \in \sigma_r} v \prec w \preceq \sigma_r[i]\}$

$minSpec := \min_{-1} S$

$maxSpec := \max_{-1} S$

**for**  $i := 0$  **to**  $|\sigma_r|$  **do**

**if**  $\text{insideClusterBorderNodes}(\sigma_r, v, i)$  **then**

$listPos(result, i) := ip\_intraRankPosition(\sigma_r, i, v, minSpec, maxSpec)$

**else**

$listPos(result, i) := NO$

**end if**

**if**  $listPos(result, i) \neq NO \wedge listPos(result, i) \neq EQUAL \wedge listPos(result, i) \neq GLOBAL$  **then**

**if**  $irValid_l \leq i \leq irValid_r$  **then**

$listPos(result, i) := NO$

**end if**

**end if**

**end for**

**return**  $result$

---

The precalculation can obviously be done in  $O(|V|)$ . Hence, algorithm A3.1 runs in  $O(|V|)$ .

The algorithm described above can also be transformed into an incremental version, which keeps some data on the clusters of all ranks. If at least the left and the right position of a cluster, the next position of a direct cluster member to the right of the node to be inserted and the most specific left and right clusters, which have been aligned so far, are stored in that data structure, `prepareInterRankTest` can be implemented in constant time with linear time complexity for the incremental updates when inserting (or deleting) a node.

## A3.2 Edge Crossing Reduction

With the improvements on cluster validity, corollary 15 can be rewritten so that Algorithm 4.12 runs now in  $O(|V|^2)$  due to the improved complexity of the inter-rank validity test. Consequently, the extended median and barycentric edge crossing reduction algorithms run now in  $O(|V|^2)$  or  $O(|E_H| + |V|^2)$ , respectively.

Basically, the runtime complexity of the hierarchical edge crossing reduction algorithm is not tainted by the results described above. As described in Section 4.6.5, the complexity heavily depends on the operations carried out in the order loop, which determines the position of the nodes to be inserted successively. As a variant of the basic algorithm, which determines the position of an individual node by calculating the incrementally changing number of edge crossings of all valid positions in the target rank, simply the barycenter or median position, which was forced to the closest cluster-valid position, can be used instead. Now, the order loop takes  $O(|V|)$  to calculate the cluster-valid positions and  $O(|V|)$  to force the single node to a cluster-valid position by applying the methods described in Section 4.6.4. Hence, the median and barycenter variants of the hierarchical crossing reduction run both in  $O(|V|^2)$ .

Measurements have shown that on compound graphs, the improved hierarchical method oftentimes outperforms the other crossing reduction methods considering the number of edge and edge-region crossings. Oftentimes, the variants of the hierarchical method lead to a better number of edge crossings and compound-region crossings than median and barycenter. Considering the runtime, the median method is faster than the variants of the hierarchical edge crossing reduction, the hierarchical edge crossing reduction and the barycentric method. Unfortunately, the median method produces a lot of edge-region crossings. Applying the improved transpose heuristic of the hierarchical crossing reduction, may crossings can be reduced but the runtime increases significantly.

Multiple edge crossing reduction strategies provide the choice of a compromise considering runtime and quality of the layout result as discussed in Section 5.3. Even if the basic hierarchical edge crossing reduction implies a high complexity, it also has advantages like the influence of edge lengths on the result, which usually lead to a better and more intuitive layout in particular on small diagrams (less than approx. 200 nodes). Therefore, we propose to suggest the basic hierarchical algorithm for small diagrams, the heuristic hierarchical algorithms for larger diagrams and the median edge crossing reduction if speed is more important than quality.

## A4 Bibliography

---

- ANDERSSON, E. 1998. *Automatic Layout of Diagrams in Rational Rose*. Master's thesis, Computing Science Department, Uppsala University, Uppsala, Sweden.
- BABURIN, D. E. 2002. Some Modifications of Sugiyama Approach. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodricht and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 366–367.
- BACHL, S., AND BRANDENBURG, F.-J. 2002. Computing and Drawing Isomorphic Subgraphs. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodricht and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 74–85.
- BANSIYA, J., ETZKORN, L., DAVIS, C., AND LI, W. 1999. A Class Cohesion Metric for Object-Oriented Designs. *Journal of Object-Oriented Programming* 11, 8 (Jan.), 47–52.
- BARTH, W., JÜNGER, M., AND MUTZEL, P. 2002. Simple and Efficient Bilayer Cross Counting. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodricht and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 130–141.
- BASIL, V. R., BRIAND, L. C., AND MELO, W. L. 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators. *Software Engineering* 22, 10, 751–761.
- BASSIL, S., AND KELLER, R. K. 2001. Software Visualization Tools: Survey and Analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'2001)*, IEEE, 7–17.
- BASTERT, O., AND MATUSZEWSKI, C. 2001. Layered Drawings of Digraphs. In *Drawing Graphs: Methods and Models* [Kaufmann and Wagner 2001], 87–120.
- BATINI, C., FURLANI, L., AND NARDELLI, E. 1985. What is a Good Diagram? A Pragmatic Approach. In *Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation, Proceedings of the Fourth International Conference on Entity-Relationship Approach*,

---

Chicago, Illinois, USA, 29-30 October 1985, IEEE Computer Society and North-Holland, P. P. Chen, Ed., 312–319.

BATTISTA, G. D., EADES, P., TAMASSIA, R., AND TOLLIS, I. 1999. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

BERNER, S., GLINZ, M., AND JOOS, S. 1999. A Classification of Stereotypes for Object-Oriented Modeling Languages. In *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Springer, R. France and B. Rumpe, Eds., vol. 1723 of *LNCS*, 249–264.

BERNHART, M. 2001. Semantische Optimierung der Anordnung von Diagrammelementen in UML. Dokumentation zum Praktikum, Research Industrial Software Engineering, Institut für Softwaretechnik und Interaktive Systeme, Technische Universität Wien.

BINUCCI, C., DIDIMO, W., LIOTTA, G., AND NONATO, M. 2002. Computing Labeled Orthogonal Drawings. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 66–73.

BÖHRINGER, K.-F., AND PAULISH, F. N. 1990. Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, 43–51.

BRACHA, G., 2004. Generics in the Java Programming Language, July. <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>.

BRANDENBURG, F. J., FORSTER, M., PICK, A., RAITNER, M., AND SCHREIBER, F. 2003. BioPath - Exploration and Visualization of Biochemical Pathways. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 215–235.

BRANDES, U., AND KÖPF, B. 2002. Fast and Simple Horizontal Coordinate Assignment. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 31–44.

BRANDES, U., AND WAGNER, D. 2003. Visone - Analysis and Visualization of Social Networks. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 321–340.

BRANDES, U., EIGLSPERGER, M., HERMAN, I., HIMSOLT, M., AND MARSHALL, M. 2002. GraphML Progress Report: Structural Layer Proposal. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 501–512.

- BRIAND, L. C., DALY, J. W., AND WÜST, J. K. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 25, 1 (January/February), 91–121.
- BRIAND, L. C., LABICHE, Y., AND O’SULLIVAN, L. 2003. Impact Analysis and Change Management of UML Models. In *Proceedings of Software Maintenance 2003, IEEE Conference on Software Maintenance*, IEEE, 256–265.
- BRIDGEMAN, S., AND TAMASSIA, R. 2003. GDS - A Graph Drawing Server on the Internet. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 193–213.
- BRITO E ABREU, F., AND MELO, W. 1996. Evaluating the Impact of Object-Oriented Design on Software Quality. In *Proceedings of the 3rd International Software Metrics Symposium*, 90–99.
- BROCKENAUER, R., AND CORNELSEN, S. 2001. Drawing Clusters and Hierarchies. In *Drawing Graphs: Methods and Models* [Kaufmann and Wagner 2001], 193–227.
- BUCHHEIM, C., JUNGER, M., AND LEIPERT, S. 2001. A Fast Layout Algorithm for k -Level Graphs. In *Graph Drawing: 8th International Symposium GD 2000, Colonial Williamsburg, Va, USA, September 20–23, 2000: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Marks, Ed., vol. 1984 of *Lecture Notes in Computer Science*, 229–240.
- BUTI, L., DI BATTISTA, G., LIOTTA, G., AND TASSINARI, E. 1996. GD-Workbench: A System for Prototyping and Testing Graph Drawing Algorithms. In *Graph Drawing: Symposium on Graph Drawing, GD ’95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 111–122.
- CARBONNEAUX, Y., LABORDE, J.-M., AND MADANI, M. 1996. CABRI-Graph: A Tool for Research and Teaching in Graph Theory. In *Graph Drawing: Symposium on Graph Drawing, GD ’95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 123–126.
- CARPANO, M. 1980. Automatic Display of Hierarchized Graphs for Computer-Aided Decision Analysis. *IEEE Transactions on Software Engineering SE-12*, 4 (Apr.), 538–546.
- CASTELLÓ, R., MILI, R., AND TOLLIS, I. G. 2001. An Algorithmic Framework for Visualizing Statecharts. In *Graph Drawing: 8th International Symposium GD 2000, Colonial Williamsburg, Va, USA, September 20–23, 2000: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Marks, Ed., vol. 1984 of *Lecture Notes in Computer Science*, 139–149.
- CASTELLÓ, R., MILI, R., AND TOLLIS, I. G. 2002. ViSta: A Tool Suite for the Visualization of Behavioral Requirements. *J. Syst. Softw.* 62, 3, 141–159.

CASTELLÓ, R., MILI, R., AND TOLLIS, I. G. 2003. ViSta - Visualizing Statecharts. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 299–320.

CATARCI, C. 1995. The Assignment Heuristic for Crossing Reduction. *IEEE Transactions on Systems, Man and Cybernetics SMC-25*, 3 (Mar.), 515–521.

CHARTERS, S. M., THOMAS, N., AND MUNRO, M. 2003. The end of the line for Software Visualization? In *Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, A. van Deursen, C. Knight, J. I. Maletic, and M.-A. Storey, Eds., 110–112.

CHAZELLE, B., AND EDELSBRUNNER, H. 1992. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM* 39, 1, 1–54.

CHAZELLE, B. 1986. Reporting and counting segment intersections. *Journal of Computer and System Sciences* 32, 156–182.

CHIDAMBER, S., AND KEMERER, C. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* (June), 476–492. Copy.

COFFMAN, E. G., AND GRAHAM, R. L. 1972. Optimal Scheduling for Two-Processor Systems. *Acta Informatica* 1, 3 (Feb.), 200–213.

COLEMAN, M. K., AND PARKER, D. S. 1996. Aesthetics-based Graph Layout for Human Consumption. *Software – Practice and Experience* 26, 12, 1415–1438.

CUNNINGHAM, W. 1976. A Network Simplex Method. *Mathematical Programming* 11, 105–116.

DAVIDSON, R., AND HAREL, D. 1996. Drawing Graphs Nicely Using Simulated Annealing. *ACM Transactions on Graphics* 15, 4 (Oct.), 301–331.

DAVIS, T. A., PESTKA, K., AND KAPLAN, A. 2003. KScope: A Modularized Tool for 3D Visualization of Object-Oriented Programs. In *Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, A. van Deursen, C. Knight, J. I. Maletic, and M.-A. Storey, Eds., 98–103.

DENGLER, E., AND COWAN, W. 1998. Human Perception of Laid-Out Graphs. In *Graph Drawing: 6th International Symposium, GD'98, Montréal, Canada, August 1998. Proceedings*, Springer-Verlag Inc., New York, NY, USA, S. H. Whitesides, Ed., vol. 1547 of *Lecture Notes in Computer Science*, 441–443.

DI BATTISTA, G., AND TAMASSIA, R. 1988. Algorithms for Plane Representations of Acyclic Digraphs. *Theoretical Computer Science* 61, 2-3 (Nov.), 175–198.



- DI BATTISTA, G., EADES, P., TAMASSIA, R., AND TOLLIS, I. G. 1994. Algorithms for Drawing Graphs: an Annotated Bibliography. *Computational Geometry: Theory and Applications* 4, 235–282. quarterly updates to the Geometry Literature Database on <ftp://ftp.cs.usask.ca/pub/geometry/>.
- DI BATTISTA, G., DIDIMO, W., PATRIGNANI, M., AND PIZZONIA, M. 2002. Drawing Database Schemas. *Software – Practice and Experience* 32, 11, 1065–1098.
- DI BATTISTA, G., DIDIMO, W., PATRIGNANI, M., AND PIZZONIA, M. 2003. DBdraw - Automatic Layout of Relational Database Schemas. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 238–256.
- DIEHL, S., AND GÖRG, C. 2002. Graphs, They Are Changing. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 23–30.
- DIGUGLIELMO, G., DUROCHER, E., KAPLAN, P., SANDER, G., AND VASILIU, A. 2002. Graph Layout for Workflow Applications with ILOG JViews. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 362–363.
- DISKIN, Z., KADISH, B., AND PIESSENS, F. 1999. What vs. How of Visual Modeling: The Arrow Logic of Graphic Notations. In *Behavioral Specifications of Business and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Kluwer Academic Publications, 27–44.
- DISKIN, Z., KADISH, B., PIESSENS, F., AND JOHNSON, M. 2000. Universal Arrow Foundations for Visual Modeling. In *Diagrams*, Springer, M. Anderson, P. Cheng, and V. Haarslev, Eds., vol. 1889 of *Lecture Notes in Computer Science*, 345–360.
- DISKIN, Z. 2002. Visualization vs. Spezification in Diagrammatic Notations: A Case Study With UML. In *Diagrams 2002*, Springer-Verlag, M. Hegarty, B. Meyer, and H. Narayanan, Eds., vol. 2317 of *LNAI*, 112–115.
- DISKIN, Z. 2002. Visualization vs. Specification in Diagrammatic Notations: A Case Study with the UML. In *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002, Callaway Gardens, GA, USA, April 18-20, 2002, Proceedings*, Springer-Verlag, M. Hegarty, B. Meyer, and N. H. Narayanan, Eds., vol. 2317 of *Lecture Notes in Computer Science*, 112–115.
- DOBKIN, D. P., GANSNER, E. R., KOUTSOFIOS, E., AND NORTH, S. C. 1997. Implementing a General-Purpose Edge Router. In *Graph Drawing: 5th International Symposium, GD'97, Rome, Italy, September 18–20, 1997: proceedings*, Springer-Verlag Inc., New York, NY, USA, G. Di Battista, Ed., vol. 1353 of *Lecture Notes in Computer Science*, 261–271.

- 
- DOĞRUSÖZ, U., KAKOULIS, K. G., MADDEN, B., AND TOLLIS, I. G. 1998. Edge Labeling in the Graph Layout Toolkit. In *Graph Drawing: 6th International Symposium, GD'98, Montréal, Canada, August 1998. Proceedings*, Springer-Verlag Inc., New York, NY, USA, S. H. Whitesides, Ed., vol. 1547 of *Lecture Notes in Computer Science*, 356–363.
- DWYER, T., AND ECKERSLEY, P. 2003. WilmaScope - A 3D Graph Visualisation System. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 55–75.
- DWYER, T. 2001. Three dimensional UML using force directed layout. In *Australian symposium on Information visualisation*, Australian Computer Society, Inc., 77–85.
- EADES, P., AND FENG, Q.-W. 1997. Multilevel Visualization of Clustered Graphs. In *Graph Drawing: Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18–20, 1996: proceedings*, Springer-Verlag Inc., New York, NY, USA, S. C. North, Ed., vol. 1190 of *Lecture Notes in Computer Science*, 101–112.
- EADES, P., AND KELLY, D. 1986. Heuristics for Drawing 2-Layered Networks. *Ars Combinatorica* 21-A, 89–98.
- EADES, P., AND SUGIYAMA, K. 1990. How to Draw a Directed Graph. *Journal of Information Processing* 13, 4, 424–437.
- EADES, P., AND WHITESIDES, S. 1994. Drawing Graphs in Two Layers. *Theoretical Computer Science* 131, 2 (Sept.), 361–374.
- EADES, P., AND WORMALD, N. C. 1994. Edge Crossings in Drawings of Bipartite Graphs. *Algorithmica* 11, 4 (Apr.), 379–403.
- EADES, P., MCKAY, B., AND WORMALD, N. 1986. On an edge crossing problem. In *Proc. 9th Australian Computer Science Conference*, 327–334.
- EADES, P., FENG, Q. W., AND LIN, X. 1997. Straight-Line Drawing Algorithms for Hierarchical Graphs and Clustered Graphs. In *Graph Drawing: Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18–20, 1996: proceedings*, Springer-Verlag Inc., New York, NY, USA, S. C. North, Ed., vol. 1190 of *Lecture Notes in Computer Science*, 113–127.
- EADES, P. 1984. A Heuristic for Graph Drawing. *Congressus Numerantium* 42, 149–160.
- EGYED, A. 2002. Automated Abstraction of Class Diagrams. *ACM Transactions on Software Engineering and Methodology* 11, 4, 449–491.
- EICHELBERGER, H., AND V. GUDENBERG, J. W. 2000. UML Description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*.
- EICHELBERGER, H., AND VON GUDENBERG, J. W. 2001. UMLscript Sprachspezifikation. Technical Report 272, Institut für Informatik, University of Würzburg, Feb. Institut für Informatik, University of Würzburg.

- EICHELBERGER, H., AND VON GUDENBERG, J. W. 2003. On the Visualization of Java Programs. In *Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, S. Diehl, Ed., vol. 2269 of *Lecture Notes in Computer Science*, 295–306.
- EICHELBERGER, H., AND VON GUDENBERG, J. W. 2003. UML Class Diagrams - State of the Art in Layout Techniques. In *Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, A. van Deursen, C. Knight, J. I. Maletic, and M.-A. Storey, Eds., 30–34.
- EICHELBERGER, H., AND VON GUDENBERG, J. W. 2004. Object-oriented Processing of Java Source Code. *Software – Practice and Experience* 34, 12 (Oct.), 1157–1185.
- EICHELBERGER, H. 1999. *Entwicklung eines Frameworks zum Zeichnen von Softwareentwurfsdiagrammen*. Diploma thesis, University of Würzburg.
- EICHELBERGER, H. 2002. Aesthetics of Class Diagrams. In *Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, 23–31.
- EICHELBERGER, H. 2002. Evaluation-Report on the Layout Facilities of UML Tools. Technical Report 298, Institut für Informatik, University of Würzburg, July. Institut für Informatik, University of Würzburg.
- EICHELBERGER, H. 2003. Nice Class Diagrams Admit Good Design? In *Proceedings of the 2003 ACM Symposium on Software Visualization*, ACM Press, 159–167.
- EIGLSPERGER, M., KAUFMANN, M., AND SIEBENHALLER, M. 2003. A Topology-Shape-Metrics Approach for the Automatic Layout of UML Class Diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, ACM Press, 189–216.
- EIGLSPERGER, M. 2003. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach*. Ph.D. thesis, Fakultät für Informations- und Kognitionswissenschaften (Wilhelm-Schickard Institut für Informatik), Tübingen University, Germany.
- ELLSON, J., GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND WOODHULL, G. 2003. Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 127–148.
- ESCHBACH, T., GÜNTHER, W., DRECHSLER, R., AND BECKER, B. 2002. Crossing Reduction by Windows Optimization. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 285–294.
- FENG, Q. 1997. *Algorithms for Drawing Clustered Graphs*. Ph.D. thesis, The University of Newcastle, Department of Computer Science and Software Engineering, Australia.

FLEISCHER, R., AND HIRSCH, C. 2001. Graph Drawing and Its Applications. In *Drawing Graphs: Methods and Models* [Kaufmann and Wagner 2001], 1–22.

FORMELLA, A., AND KELLER, J. 1996. Generalized Fisheye Views of Graphs. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 242–253.

FORSTER, M., PICK, A., AND RAITNER, M. Graph Template Library. <http://www.fmi.uni-passau.de/Graphlet/GTL>.

FORSTER, M. 2002. Applying Crossing Reduction Strategies to Layered Compound Graphs. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 276–284.

FREIVALDS, K., DOGRUSOZ, U., AND KIKUSTS, P. 2002. Disconnected Graph Layout and the Polyomino Packing Approach. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 378–391.

FRICK, A., LUDWIG, A., AND MEHLDAU, H. 1995. A Fast Adaptive Layout Algorithm for Undirected Graphs (Extended Abstract and System Demonstration). In *Graph Drawing: DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994: proceedings*, Springer-Verlag Inc., New York, NY, USA, R. Tamassia and I. G. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*, 388–403.

FRÖHLICH, M., AND WERNER, M. 1994. The Graph Visualization System daVinci - A User Interface for Applications. Technical Report 5/94, Department of Computer Science, Bremen University.

FRÖHLICH, M., AND WERNER, M. 1995. Demonstration of the Interactive Graph-Visualization System *da Vinci*. In *Graph Drawing: DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994: proceedings*, Springer-Verlag Inc., New York, NY, USA, R. Tamassia and I. G. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*, 266–269.

FRUCHTERMAN, T. M. J., AND REINGOLD, E. M. 1991. Graph Drawing by Force-directed Placement. *Software – Practice and Experience* 21, 11 (Nov.), 1129–1164.

FUGGETTA, A. 1993. A Classification of CASE Technology. *IEEE Software* 26, 12, 25–38.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 2000. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts.

- GANSNER, E. R., AND NORTH, S. C. 1998. Improved Force-Directed Layouts. In *Graph Drawing: 6th International Symposium, GD'98, Montréal, Canada, August 1998. Proceedings*, Springer-Verlag Inc., New York, NY, USA, S. H. Whitesides, Ed., vol. 1547 of *Lecture Notes in Computer Science*, 364–373.
- GANSNER, E. R., NORTH, S. C., AND VO, K. P. 1988. DAG: A Program that Draws Directed Graphs. *Software – Practice and Experience* 18, 11 (Nov.), 1047–1062.
- GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND VO, K.-P. 1993. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering* 19, 3 (Mar.), 214–230.
- GAREY, M. R., AND JOHNSON, D. S. 1983. Crossing Number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods* 4, 3 (Sept.), 312–316.
- GÄRTNER, J., MIKSCH, S., AND CARL-MCGRATH, S. 2002. ViCo: A Metric for the Complexity of Information Visualizations. In *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002, Callaway Gardens, GA, USA, April 18-20, 2002, Proceedings*, Springer-Verlag, M. Hegarty, B. Meyer, and N. H. Narayanan, Eds., vol. 2317 of *Lecture Notes in Computer Science*, 249–263.
- GENERO, M., PIATTINI, M., AND CALERO, C. 2000. Early measures for UML class Diagrams. *L'Objet* 6, 4.
- GIL, J., HOWSE, J., AND KENT, S. 2002. Advanced Visual Modeling: Beyond UML. In *Proceedings of the 24th International Conference on Software Engineering*, ACM Press, 697–698.
- GOLDBERG, A., AND TARJAN, R. 1986. A New Approach to the Maximum Flow Problem. *Proc. of the 8th Annual ACM Symposium on Theory of Computing*, 136–146.
- GOLDBERG, A., AND TARJAN, R. 1990. Finding Minimum-Cost Circulations by Successive Approximation. *Mathematics of Operations Research* 15, 3, 430–466.
- GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc.
- GREEN, T., AND BLACKWELL, A., 1998. Cognitive Dimensions of Information Artefacts: a tutorial. Version 1.2, October 1998, via <http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/>, <http://www.ndirect.co.uk/thomas.green/workStuff/Papers/>.
- GROHE, M. 2001. Computing Crossing Numbers in Quadratic Time. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, ACM Press, 231–236.
- GRÖHLING, B., 2003. Implementation of Selected Layout Metrics. internal paper, available on request.

GUÉHÉNEUC, Y.-G. 2003. *Un cadre pour la traçabilité des motifs de conception*. PhD thesis, École des Mines de Nantes. À paraître.

GÜNTHER, W., SCHÖNFELDER, R., BECKER, B., AND MOLITOR, P. 2001. *k*-Layer Straight-line Crossing Minimization by Speeding Up Sifting. In *Graph Drawing: 8th International Symposium GD 2000, Colonial Williamsburg, Va, USA, September 20–23, 2000: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Marks, Ed., vol. 1984 of *Lecture Notes in Computer Science*, 253–258.

GUSTAVSSON, J., AND ASSMANN, U. 2002. A Classification of Runtime Software Changes. In *Proceedings of the First International Workshop on Unanticipated Software Evolution*.

GUSTAVSSON, J. 2003. A Classification of Unanticipated Runtime Software Changes in Java. In *Proceedings of Software Maintenance 2003, IEEE Conference on Software Maintenance*, IEEE, 4–12.

GUTWENGER, C., JÜNGER, M., KLEIN, K., KUPKE, J., LEIPERT, S., AND MUTZEL, P. 2002. Caesar Automatic Layout of UML Class Diagrams. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 461–462.

GUTWENGER, C., JÜNGER, M., KLEIN, K., KUPKE, J., LEIPERT, S., AND MUTZEL, P. 2003. GoVisual - A Diagramming Software for UML Class Diagrams. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 257–278.

GUTWENGER, C., JÜNGER, M., KLEIN, K., KUPKE, J., LEIPERT, S., AND MUTZEL, P. 2003. A New Approach for Visualizing UML Class Diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, ACM Press, 179–188.

GUTWENGER, C., KUPKE, J., KLEIN, K., AND LEIPERT, S. 2003. GoVisual for CASE Tools Borland Together ControlCenter and Gentleware Poseidon – System Demonstration. In *Graph Drawing: 11th International Symposium, GD 2003, Perugia, Italy, September 2003, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, G. Liotta, Ed., vol. 2912 of *Lecture Notes in Computer Science*, 123–128.

HACKBUSCH, W. 1997. On the Feedback Vertex Set Problem for a Planar Graph. *Computing* 58, 129–155.

HAHN, J., AND KIM, J. 1999. Why Are Some Diagrams Easier to Work With? Effects of Diagrammatic Representation on the Cognitive Intergration Process of Systems Analysis and Design. *ACM Transactions on Computer-Human Interaction* 6, 3, 181–213.

HANSEN, T., MARRIOTT, K., MEYER, B., AND STUCKEY, P. J. 2002. Flexible Graph Layout for the Web. *Journal of Visual Languages and Computing* 13, 35–60.

HAREL, D. 1988. On Visual Formalisms. *Communications of the ACM* 31, 5, 514–530.

HEALY, P., AND KUUSIK, A. 2000. The Vertex-Exchange Graph: A New Concept for Multi-level Crossing Minimisation. In *Graph Drawing: 7th International Symposium, GD'99, Stiriin Castle, Czech Republic, September 15–19, 1999: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Kratochvil, Ed., vol. 1731 of *Lecture Notes in Computer Science*, 205–216.

HEALY, P., AND NIKOLOV, N. S. 2002. A Branch-and-Cut Approach to the Directed Acyclic Graph Layering Problem. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodricht and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 98–109.

HEALY, P., AND NIKOLOV, N. S. 2002. How to Layer a Directed Acyclic Graph. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 16–30.

HEISE, W., AND QUATTROCCHI, P. 1995. *Informations- und Codierungstheorie*. Springer, Berlin.

HENRY, T. R., AND HUDSON, S. E. 1991. Interactive Graph Layout. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 55–64.

HERSHBERGER, J., AND SNOEYINK, J. 1994. Computing minimum length paths of a given homotopy class. *Comput. Geom. Theory Appl.* 4, 2, 63–97.

HIMSOLT, M. 1995. GraphEd: a Graphical Platform for the Implementation of Graph Algorithms. In *Graph Drawing: DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994: proceedings*, Springer-Verlag Inc., New York, NY, USA, R. Tamassia and I. G. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*, 182–193.

HIMSOLT, M. 1997. The Graphlet System (System Demonstration). In *Graph Drawing: Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18–20, 1996: proceedings*, Springer-Verlag Inc., New York, NY, USA, S. C. North, Ed., vol. 1190 of *Lecture Notes in Computer Science*, 233–240.

JABLONOWSKI, D., AND GUARNA, JR., V. A. 1989. GMB: A Tool for Manipulating and Animating Graph Data Structures. *Softw. Pract. Exper.* 19, 3, 283–301.

JACOBSON, I., RUMBAUGH, J., AND BOOCH, G. 1999. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading, MA.

JINGWEI, H., AND ZHUO, K. 1994. A Genetic Algorithm for the Feedback Set Problem. Technical report, Computer School, Wuhan University, China, January.

JOHNSON, D. S. 1982. The NP-Completeness Column: An Ongoing Guide. *Journal of Algorithms* 3, 1 (March), 89–99.

JÜNGER, M., AND MUTZEL, P. 1996. Exact and Heuristic Algorithms for 2-Layer Straightline Crossing Minimization. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 337–348.

JÜNGER, M., AND MUTZEL, P. 1997. 2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms. *Journal of Graph Algorithms and Applications* 1, 1, 1–25.

JÜNGER, M., AND MUTZEL, P. 2003. *Graph Drawing Software*. Springer-Verlag, Berlin.

JÜNGER, M., AND MUTZEL, P. 2003. Technical Foundations. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 9–53.

JÜNGER, M., KLAU, G. W., MUTZEL, P., AND WEISKIRCHER, R. 2003. AGD - A Library of Algorithms for Graph Drawing. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 149–172.

KAKOULIS, K. C., AND TOLLIS, I. G. 1997. An Algorithm for Labeling Edges of Hierarchical Drawings. In *Graph Drawing: 5th International Symposium, GD'97, Rome, Italy, September 18–20, 1997: proceedings*, Springer-Verlag Inc., New York, NY, USA, G. Di Battista, Ed., vol. 1353 of *Lecture Notes in Computer Science*, 169–180.

KAKOULIS, K. G., AND TOLLIS, I. G. 1997. On the Edge Label Placement Problem. In *Graph Drawing: Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18–20, 1996: proceedings*, Springer-Verlag Inc., New York, NY, USA, S. C. North, Ed., vol. 1190 of *Lecture Notes in Computer Science*, 241–256.

KAKOULIS, K. G., AND TOLLIS, I. G. 1998. On the Multiple Label Placement Problem. In *Proc. 10th Canadian Conf. Computational Geometry (CCCG'98)*, 66–67.

KAMADA, T., AND KAWAI, S. 1989. An algorithm for drawing general undirected graphs. *Information Processing Letters* 31, 1 (Apr.), 7–15.

KAMPS, T., KLEINZ, J., AND READ, J. 1996. Constraint-Based Spring-Model Algorithm for Graph Layout. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 349–360.

KAUFMANN, M., AND WAGNER, D., Eds. 2001. *Drawing Graphs: Methods and Models*. No. 2025 in *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA.

KELLER, R. K., SCHAUER, R., ROBITAILLE, S., AND PAGÉ, P. 1999. Pattern-Based Reverse-Engineering of Design Components. In *Proceedings of the 21st International Conference on Software Engineering*, IEEE Computer Society Press, 226–235.



- KNUTH, D. E. 1993. *Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press.
- KOBRYN, C. 1999. UML 2001: A Standardization Odyssey. *Communications of the ACM* 42, 10, 29–37.
- KOBRYN, C. 2000. Modeling Components and Frameworks with UML. *Communications of the ACM* 43, 10, 31–38.
- KOBRYN, C. 2002. Will UML 2.0 Be Agile or Awkward? *Communications of the ACM* 45, 1, 107–110.
- KOSAK, C., MARKS, J., AND SHIEBER, S. 1994. Automating the Layout of Network Diagrams with Specified Visual Organization. *IEEE Trans. Systems, Man and Cybernetics* 24, 3, 440–454.
- KOSCHKE, R. 2003. Software Visualization in Software Maintenance, Reverse Engineering, and Reengineering: A Research Survey. *Journal on Software Maintenance and Evolution* 15, 2 (Mar./Apr.), 87–109.
- KÖTH, O., AND MINAS, M. 2002. Structure, Abstraction, and Direct Manipulation in Diagram Editors. In *Diagrammatic Representation and Inference, Second International Conference, Diagrams 2002, Callaway Gardens, GA, USA, April 18-20, 2002, Proceedings*, Springer-Verlag, M. Hegarty, B. Meyer, and N. H. Narayanan, Eds., vol. 2317 of *Lecture Notes in Artificial Intelligence*, 290–304.
- KÖTH, O., 2001. Semantisches Zoomen in Diagrammeditoren am Beispiel von UML. diploma thesis, Erlangen-Nürnberg University, <http://www2.informatik.uni-erlangen.de/DiaGen/Doc/oliver-thesis2.pdf>.
- LAGUNA, M., AND MARTÍ, R. 1999. GRASP and Path Relinking for 2-Layer Straight Line Crossing Minimization. *INFORMS Journal on Computing* 11, 44–52.
- LAGUNA, M., MARTÍ, R., AND VALLS, V. 1997. Arc Crossing Minimization in Hierarchical Digraphs with Tabu Search. *Computers and Operations Research* 11, 2, 1175–1186.
- LANZA, M. 2003. CodeCrawler - A Lightweight Software Visualization Tool. In *Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, A. van Deursen, C. Knight, J. I. Maletic, and M.-A. Storey, Eds., 54–55.
- LAUX, M. 2004. Eine schnelle Hash Map für ganzzahlige Schlüssel. *Java Magazin* (Jan.), 30–48.
- LEE, L.-Q., SIEK, J. G., AND LUMSDAINE, A. 1999. The Generic Graph Component Library. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 399–414.

- LEWERENTZ, C., AND NOACK, A. 2003. CrocoCosmos - 3D Visualization of Large Object-oriented Programs. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 279–297.
- LEWERENTZ, C., LINDNER, T., RÜPING, A., AND SEKERINSKI, E. 1995. On Object-Oriented Design and Verification. In *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, M. Broy and S. Jähnichen, Eds., vol. 1009. Springer-Verlag, New York, N.Y., 92–111.
- LI, Q., AND MOON, B. 2001. Indexing and Querying XML Data for Regular Path Expressions. In *The VLDB Journal*, 361–370.
- LI, W., HENRY, S., KAFURA, D., AND SCHULMAN, R. 1995. Measuring object-oriented design. *Journal of Object-Oriented Programming* (July-August), 48–55.
- LORENZ, M., AND KIDD, J. 1994. *Object-Oriented Software Metrics*. Prentice Hall.
- LYONS, K. A., MEIJER, H., AND RAPPAPORT, D. 1998. Algorithms for Cluster Busting in Anchored Graph Drawing. *Journal of Graph Algorithms and Applications* 2, 1, 1–24.
- MADDEN, B., MADDEN, P., POWERS, S., AND HIMSOLT, M. 1996. Portable Graph Layout and Editing. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 385–395.
- MAGUIRE, S. 1993. *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs*. Microsoft Press.
- MÄKINEN, E., AND SIERANTA, M. 1994. Genetic Algorithms for Drawing Bipartite Graphs. Technical Report A-1994-1, Department of Computer Science, University of Tampere, January.
- MÄKINEN, E. 1990. Experiments on Drawing 2-Level Hierarchical Graphs. *International Journal of Computer Mathematics* 37, 129–135.
- MARCHESI, M. 1998. OOA Metrics for the Unified Modeling Language. In *Proceedings of the 2<sup>nd</sup> Euromicro Conference on Software Maintenance and Reengineering*, 67–73.
- MARKWITZ, S. 1998. Layoutalgorithmen für die UML. *OBJEKTSpektrum* 98, 6, 56–61.
- MARTÍ, R., AND LAGUNA, M. 2003. Heuristics and Meta-Heuristics for 2-Layer Straight Line Crossing Minimization. *Discrete Applied Mathematics* 127, 3, 665–678.
- MATUSZEWSKI, C., SCHÖNFELD, R., AND MOLITOR, P. 2000. Using Sifting for  $k$ -Layer Straightline Crossing Minimization. In *Graph Drawing: 7th International Symposium, GD'99, Stirin Castle, Czech Republic, September 15–19, 1999: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Kratochvil, Ed., vol. 1731 of *Lecture Notes in Computer Science*, 217–224.
- MCCLUSKEY, G., 1999. Core Java Technologies Technical Tips, August 9, 1999: Using List Collections Efficiently. <http://java.sun.com/developer/TechTips/1999/tt0809.html>.

- MCCLUSKEY, G., 2004. Core Java Technologies Technical Tips, December 1, 2004: Covariant Return Types. <http://java.sun.com/developer/TechTips/2004/tt0112.html>.
- MCCLUSKEY, G., 2005. Core Java Technologies Technical Tips, January 4, 2005: Covariant Parameter Types. <http://java.sun.com/developer/TechTips/2005/tt0401.html>.
- MCCREARY, C., AND BAROWSKI, L. 1998. VGJ: Visualizing Graphs Through Java. In *Graph Drawing: 6th International Symposium, GD'98, Montréal, Canada, August 1998. Proceedings*, Springer-Verlag Inc., New York, NY, USA, S. H. Whitesides, Ed., vol. 1547 of *Lecture Notes in Computer Science*, 454–455.
- MEHLHORN, K., AND NÄHER, S. LEDA. <http://www.mpi-sb.mpg.de/LEDA/leda.html>.
- MELLOR, S. J., WILKIE, I., FONTAINE, J. S., AND STATEZNI, D. E., 1999. An Approach to Adding Precision and Clarity to UML. via <http://www.omg.org/docs/ad/99-12-25.pdf>, <http://www.omg.org/docs/ad/99-12-25.pdf>.
- MESSINGER, E. B., ROWE, L. A., AND HENRY, R. R. 1991. A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs. *IEEE Transactions on Systems, Man and Cybernetics SMC-21*, 1 (Jan./Feb.), 1–11.
- MISUE, K., EADES, P., LAI, W., AND SUGIYAMA, K. 1995. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing* 6, 2, 183–210.
- MUTZEL, P., AND WEISKIRCHER, R. 1998. Two-Layer Planarization in Graph Drawing. In *ISAAC: 9th International Symposium on Algorithms and Computation*.
- MUTZEL, P. 1997. An Alternative Method to Crossing Minimization on Hierarchical Graphs. In *Graph Drawing: Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18–20, 1996: proceedings*, Springer-Verlag Inc., New York, NY, USA, S. C. North, Ed., vol. 1190 of *Lecture Notes in Computer Science*, 318–333.
- NOGUCHI, T., AND TANAKA, J. 1998. New Automatic Layout Method based on Magnetic Spring Model for Object Diagrams of OMT. In *Proceedings of International Symposium on Future Software Technology 1998 (ISFST'98)*, 89–94.
- NOGUCHI, T., AND TANAKA, J. 1999. Interactive Layout Method for Object Diagrams of OMT. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC '99)*, 110–117.
- NOLTEMEIER, H. 1988. *Informatik III - Einführung in Datenstrukturen*. Hanser, München.
- NORTH, S. C. 1996. Incremental Layout in DynaDAG. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 409–418.

- 
- NUMMENMAA, J., AND TUOMI, J. 1990. Constructing Layouts for ER-Diagrams from Visibility-Representations. In *Proceedings of the 9th International Conference on Entity-Relationship Approach (ER'90), 8-10 October, 1990, Lausanne, Switzerland*, ER Institute, H. Kangassalo, Ed., 303–317.
- OHST, D., WELLE, M., AND KELTER, U. 2003. Different Tools for Analysis and Design Documents. In *Proceedings of Software Maintenance 2003, IEEE Conference on Software Maintenance*, IEEE, 13–22.
- OMG, 1997. Unified Modeling Language Specification, Nov. Version 1.1, <http://www.omg.org>.
- OMG, 2002. XML Metadata Interchange (XMI), May. Version 2.0, <http://cgi.omg.org/docs/ad/99-10-02.pdf>.
- OMG, 2003. Model Driven Architecture Specification, June. Version 1.0.1, <http://www.omg.org/cgi-bin/doc?omg/03-06-01>.
- OMG, 2003. UML 2.0 Diagram Interchange Specification, Sept. ptc/03-09-01 OMG Final Adopted Specification, <http://www.omg.org/cgi-bin/doc?ptc/2003-09-01>.
- OMG, 2003. Unified Modeling Language Specification, Mar. Version 1.5, March 2003, <http://www.omg.org/cgi-bin/doc?formal/03-03-01>.
- OMG, 2003. Unified Modeling Language Specification, June. Version 2.0, <http://www.uml.org>.
- PACH, J., AND TÓTH, G. 2000. Which Crossing Number Is It Anyway? *J. Comb. Theory Ser. B* 80, 2, 225–246.
- PACH, J. 1998. Crossing Numbers. In *Discrete and Computational Geometry: Japanese Conference, JCDCG'98 Tokyo, Japan, December 9-12, 1998. Revised Papers*, Springer-Verlag Inc., New York, NY, USA, J. Akiyama, M. Kano, and M. Urabe, Eds., vol. 1763 of *Lecture Notes in Computer Science*, 267–273.
- PANKOWSKI, T. 2004. Processing XPath Expressions in Relational Databases. In *SOFSEM 2004: Theory and Practice of Computer Science*, Springer-Verlag Inc., New York, NY, USA, P. Van Emde Boas, J. Pokorný, M. Bieliková, and J. Stuller, Eds., vol. 2932 of *Lecture Notes in Computer Science*, 265–276.
- PAULISCH, F. 1993. *The Design of an Extendible Graph Editor*. No. 704 in LNCS. Springer-Verlag.
- PETRE, M. 1995. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Commun. ACM* 38, 6, 33–44.
- PROTSKO, L. B., SORENSON, P. G., TREMBLAY, J. P., AND SCHAEFER, D. A. 1991. Towards the Automatic Generation of Software Diagrams. *IEEE Transactions on Software Engineering* 17, 1 (January), 10–21.

- PURCHASE, H. C., COHEN, R. F., AND JAMES, M. 1996. Validating Graph Drawing Aesthetics. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 435–446.
- PURCHASE, H. C., COHEN, R. F., AND JAMES, M. I. 1997. An Experimental Study of the Basis for Graph Drawing Algorithms. *J. Exp. Algorithmics* 2, 4.
- PURCHASE, H. C., ALLDER, J.-A., AND CARRINGTON, D. 2001. User Preference of Graph Layout Aesthetics: A UML Study. In *Graph Drawing: 8th International Symposium GD 2000, Colonial Williamsburg, Va, USA, September 20–23, 2000: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Marks, Ed., vol. 1984 of *Lecture Notes in Computer Science*, 5–18.
- PURCHASE, H. C., MCGILL, M., COLPOYS, L., AND CARRINGTON, D. 2001. Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In *Proceedings of the Australian Symposium on Information Visualisation*, Australian Computer Society, Inc., P. Eades and T. Pattison, Eds., 129–137.
- PURCHASE, H., ALLDER, J.-A., AND CARRINGTON, D. 2002. Graph Layout Aesthetics in UML Diagrams: User Preferences. *Journal of Graph Algorithms and Applications* 6, 3, 255–279.
- PURCHASE, H., COLPOYS, L., CARRINGTON, D., AND MCGILL, M. 2003. UML Class Diagrams: An Empirical Study of Comprehension. In *Software Visualization - From Theory to Practice*, K. Zhang, Ed. Kluwer, 149–178.
- PURCHASE, H. C. 1997. Which Aesthetic has the Greatest Effect on Human Understanding. In *Graph Drawing: 5th International Symposium, GD'97, Rome, Italy, September 18–20, 1997: proceedings*, Springer-Verlag Inc., New York, NY, USA, G. Di Battista, Ed., vol. 1353 of *Lecture Notes in Computer Science*, 248–261.
- PURCHASE, H. C. 1998. Performance of Layout Algorithms: Comprehension, not Computation. *Journal of Visual Languages and Computing* 1998, 9, 647–657.
- PURCHASE, H. 2000. Effective information visualization: A study of graph drawing aesthetics and algorithms. *Interacting with Computers* 13, 147–162.
- PURCHASE, H. C. 2002. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages and Computing* 13, 5, 501–516.
- PURCHASE, H. 2004. Evaluating Graph Drawing Aesthetics: defining and exploring a new empirical research area. In *Computer Graphics and Multimedia: Applications, Problems and Solutions*, J. DiMarco, Ed. Idea Group Publishing, 145–178.
- RAITNER, M. 2002. HGV: A Library for Hierarchies, Graphs, and Views. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*,

---

Springer-Verlag Inc., New York, NY, USA, M. T. Goodricht and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 236–243.

REINGOLD, E. M., AND TILFORD, J. S. 1981. Tidier Drawing of Trees. *IEEE Transactions on Software Engineering SE-7*, 2 (Mar.), 223–228.

REINIGER, C. 2003. *Comparison and Transformation of Languages for UML Diagram Exchange*. Diploma thesis, Lehrstuhl für Informatik, University of Würzburg.

RHODE, S. 2003. *Automatisches Layout von UML-Sequenzdiagrammen*. Diploma thesis, Lehrstuhl für Informatik, University of Würzburg.

ROWE, L. A., DAVIS, M., MESSINGER, E., MAYER, C., SPIRAKIS, C., AND TUAN, A. 1987. A Browser for Directed Graphs. *Software – Practice and Experience 17*, 1 (Jan.), 61–76.

SAAB, Y. 2001. A Fast and Effective Algorithm for the Feedback Arc Set Problem. *Journal of Heuristics 7*, 3, 235–250.

SANDER, G. 1995. Graph Layout through the VCG Tool (Extended Abstract and System Demonstration). In *Graph Drawing: DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994: proceedings*, Springer-Verlag Inc., New York, NY, USA, R. Tamassia and I. G. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*, 194–205.

SANDER, G. 1996. A Fast Heuristic for Hierarchical Manhattan Layout. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 447–548.

SANDER, G. 1996. Layout of Compound Directed Graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken.

SCHAUER, R., AND KELLER, R. K. 1998. Pattern Visualization for Software Comprehension. In *Proceedings of the Sixth International Workshop on Program Comprehension (IWPC'1998)*, IEEE, 4–12.

SCHREIBER, F. 2002. High quality visualization of biochemical pathways in biopath. *Silico Biology 2*, 0006. <http://www.bioinfo.de/isb/2002/02/0006/>.

SEEMANN, J., AND VON GUDENBERG, J. W. 1998. UMLscript: A Programming Language for Object-Oriented Design. In *The Unified Modeling Language – Technical Aspects and Applications*, M. Schader and A. Korthaus, Eds. Physica-Verlag, Heidelberg, 160–169.

SEEMANN, J. 1997. Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams. In *Graph Drawing: 5th International Symposium, GD'97, Rome, Italy, September 18–20, 1997: proceedings*, Springer-Verlag Inc., New York, NY, USA, G. Di Battista, Ed., vol. 1353 of *Lecture Notes in Computer Science*, 415–423.

SHIRAZI, J. 2000. *Java Performance Tuning*. O'Reilly.

SIEK, J. G., LEE, L.-Q., AND LUMSDAINE, A. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Pearson Education Inc.

SIX, J. M., AND TOLLIS, I. G. 2002. Automated Visualization of Process Diagrams. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 45–59.

SPINELLIS, D. 2003. On the Declarative Specification of Models. *IEEE Software* 20, 2, 94–96. March/April.

STALLMANN, M., BRGLEZ, F., AND GHOSH, D. 1999. Heuristics and Experimental Design for Bigraph Crossing Number Minimization. In *Algorithm engineering and experimentation: international workshop ALENEX '99, Baltimore, MD, USA, January 15–16, 1999: selected papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and C. C. McGeoch, Eds., vol. 1619 of *Lecture Notes in Computer Science*, 74–93.

STALLMANN, M., BRGLEZ, F., AND GHOSH, D. 2001. Heuristics, Experimental Subjects, and Treatment Evaluation in Bigraph Crossing Minimization. *J. Exp. Algorithmics* 6, 8.

STOREY, M.-A. D., AND MUELLER, H. A. 1996. Graph Layout Adjustment Strategies. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 487–499.

SUGIYAMA, K., AND MISUE, K. 1991. Visualization of Structural Information: Automatic Drawing of Compound Digraphs. *IEEE Transactions on Systems, Man and Cybernetics SMC-21*, 4 (July/August), 876–891.

SUGIYAMA, K., AND MISUE, K. 1995. A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm. In *Graph Drawing: DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994: proceedings*, Springer-Verlag Inc., New York, NY, USA, R. Tamassia and I. G. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*, 364–375.

SUGIYAMA, K., AND MISUE, K. 1996. A Generic Compound Graph Visualizer/Manipulator: D-ABDUCTOR. In *Graph Drawing: Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995: proceedings*, Springer-Verlag Inc., New York, NY, USA, F. J. Brandenburg, Ed., vol. 1027 of *Lecture Notes in Computer Science*, 500–503.

SUGIYAMA, K., TAGAWA, S., AND TODA, M. 1981. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics SMC-11*, 2 (Feb.), 109–125.

SUGIYAMA, K. 2002. *Graph Drawing and Applications for Software and Knowledge Engineers*. World Scientific, New York.

SUMMERVILLE, I. 1996. *Software Engineering*. Prentice Hall.

SUN JCP, 2003. JMI Specification API 1.0, June. Final Release, June 28, 2002, <http://java.sun.com/products/jmi>.

TAMASSIA, R., DI BATTISTA, G., AND BATINI, C. 1988. Automatic Graph Drawing and Readability of Diagrams. *IEEE Transactions on Systems, Man and Cybernetics* 18, 1 (Jan./Feb.), 61–79.

TAMASSIA, R. 1985. New Layout Techniques for Entity-Relationship Diagrams. In *Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation, Proceedings of the Fourth International Conference on Entity-Relationship Approach, Chicago, Illinois, USA, 29-30 October 1985*, IEEE Computer Society and North-Holland, P. P. Chen, Ed., 304–311.

TAMASSIA, R. 1987. On Embedding a Graph in the Grid with the Minimum Number of Bends. *SIAM Journal on Computing* 16, 3, 421–444.

TAMASSIA, R. 1998. Constraints in Graph Drawing Algorithms. *Constraints* 3, 87–120.

TEOH, S., AND MA, K.-L. 2002. RINGS: A Technique for Visualizing Large Hierarchies. In *Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, M. T. Goodrich and S. G. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 268–275.

TRAPP, M. 2001. *Optimierung objektorientierter Programme (PhD. thesis, in German)*. Springer.

TUTTE, W. T. 1963. How to draw a graph. In *Proceedings London Math. Society*, no. 13 in 3, 743–768. Graph drawing.

UTECH, J., BRANKE, J., SCHMECK, H., AND EADES, P. 1998. An Evolutionary Algorithm for Drawing Directed Graphs. In *Proceedings of the 1998 International Conference on Imaging Science, Systems, and Technology (CISST'98), Las Vegas, USA*, 154 – 160.

VALLS, V., MARTI, R., AND LINO, P. 1996. A Branch and Bound Algorithm for Minimizing the Number of Crossing Arcs in Bipartite Graphs. *Central European Journal of Operational Research* 90, 303–319.

WADDLE, V., AND MALHOTRA, A. 2000. An  $E \log E$  Line Crossing Algorithm for Levelled Graphs. In *Graph Drawing: 7th International Symposium, GD'99, Stirin Castle, Czech Republic, September 15–19, 1999: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Kratochvil, Ed., vol. 1731 of *Lecture Notes in Computer Science*, 59–71.



- WADDLE, V. 2001. Graph Layout for Displaying Data Structures. In *Graph Drawing: 8th International Symposium GD 2000, Colonial Williamsburg, Va, USA, September 20–23, 2000: proceedings*, Springer-Verlag Inc., New York, NY, USA, J. Marks, Ed., vol. 1984 of *Lecture Notes in Computer Science*, 241–252.
- WALKER II, J. Q. 1990. A Node-positioning Algorithm for General Trees. *Software – Practice and Experience* 20, 7 (July), 685–705.
- WARE, C., HUI, D., AND FRANCK, G. 1993. Visualizing Object Oriented Software in Three Dimensions. In *Proc. IBM Centre for Advanced Studies Conf., CASCON*.
- WARE, C., PURCHASE, H., COLPOYS, L., AND MCGILL, M. 2002. Cognitive Measurements of Graph Aesthetics. *Information Visualization* 1, 2, 103–110.
- WARE, C. 2000. *Information Visualization: Perception for Design*. Academic Press, Morgan Kaufmann Publishers.
- WARFIELD, J. 1977. Crossing Theory and Hierarchy Mapping. *IEEE Transactions on Systems, Man and Cybernetics SMC-7*, 7 (July), 505–523.
- WETHERELL, C., AND SHANNON, A. 1979. Tidy drawings of trees. *IEEE Transactions on Software Engineering SE-5*, 5, 514–520.
- WIESE, R., EIGLSPERGER, M., AND KAUFMANN, M. 2002. yFiles: Visualization and Automatic Layout of Graphs. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 453–454.
- WIESE, R., EIGLSPERGER, M., AND KAUFMANN, M. 2003. yFiles - Visualization and Automatic Layout of Graphs. In *Graph Drawing Software* [Jünger and Mutzel 2003a], 173–191.
- WILLHALM, T. 2001. Software Packages. In *Drawing Graphs: Methods and Models* [Kaufmann and Wagner 2001], 274–281.
- WINTER, A. 2002. Exchanging Graphs with GXL. In *Graph Drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, Springer-Verlag Inc., New York, NY, USA, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 479–500.
- ZUKOWSKI, J., 2003. Core Java Technologies Technical Tips, April 22, 2003: Reusing Exceptions. <http://java.sun.com/developer/JDCTechTips/2003/tt0422.html>.
- ZUKOWSKI, J., 2003. Core Java Technologies Technical Tips, February 20, 2003: Choosing A Collections Framework Implementation. <http://java.sun.com/developer/JDCTechTips/2003/tt0220.html>.

ZUKOWSKI, J., 2003. Core Java Technologies Technical Tips, October 21, 2003: Converting Between Old and New Collections. <http://java.sun.com/developer/JDCTechTips/2003/tt1021.html>.

ZUSE, H. 1998. *A framework of Software Measurement*. deGruyter.

## A5 List of Figures

---

2.1	UML: a class . . . . .	11
2.2	UML: is-a relation . . . . .	12
2.3	UML: an association . . . . .	12
2.4	UML: compositions . . . . .	13
2.5	UML: an association class . . . . .	14
2.6	UML: more on associations . . . . .	14
2.7	UML: packages . . . . .	15
2.8	UML: subsystems . . . . .	16
2.9	UML: comments . . . . .	16
2.10	UML: composition and anchor notation . . . . .	18
2.11	UML: shared target style . . . . .	18
2.12	evaluation test class diagram . . . . .	21
2.13	a 3D class diagram . . . . .	25
2.14	diagram languages . . . . .	29
2.15	UML tools: TNI OpenTool . . . . .	37
2.16	UML tools: Popkin SystemArchitect . . . . .	37
2.17	UML tools: IBM/Rational Rose . . . . .	38
2.18	UML tools: NoMagic MagicDrawUML . . . . .	38
3.1	visual grammar of node-link diagrams . . . . .	60
3.2	class diagram example to be represented as geon diagram . . . . .	62
3.3	a geon diagram example . . . . .	63
3.4	class size scaling . . . . .	67
3.5	polymetric view . . . . .	70
3.6	good aesthetics example class diagram . . . . .	72
3.7	spatial distribution in class diagrams . . . . .	81
3.8	edge sequences for joined edges . . . . .	82
3.9	aesthetic class diagram example drawn by <i>SugiBib</i> . . . . .	85
3.10	layout without hierarchies in the main layout approaches . . . . .	90

4.1	<i>SugiBib</i> algorithm: the input graph . . . . .	95
4.2	composite node technique . . . . .	96
4.3	<i>SugiBib</i> algorithm: step S1 to step S6 . . . . .	97
4.4	<i>SugiBib</i> algorithm: step S7 to step S11 . . . . .	99
4.5	<i>SugiBib</i> algorithm: step S12 to step S16 . . . . .	100
4.6	<i>SugiBib</i> algorithm: step S18 to step S21 . . . . .	103
4.7	<i>SugiBib</i> graph model: structural input rules . . . . .	106
4.8	<i>SugiBib</i> graph model: clusters . . . . .	108
4.9	<i>SugiBib</i> graph model: class diagram . . . . .	109
4.10	<i>SugiBib</i> graph model: information object life cycle . . . . .	110
4.11	transforming a hyper edge to composite nodes . . . . .	128
4.12	transforming an association class to a composite node . . . . .	129
4.13	various comment situations . . . . .	130
4.14	<i>n</i> -level hierarchy . . . . .	139
4.15	intra-rank-validity example . . . . .	140
4.16	inter-rank-validity example . . . . .	141
4.17	compound dependency graph . . . . .	146
4.18	hidden nodes for hierarchical cluster relations . . . . .	149
4.19	UML rank assignment: edges connected to the virtual root . . . . .	151
4.20	UML rank assignment: adjust coupled cluster members . . . . .	152
4.21	matrix realization example . . . . .	161
4.22	alternative matrix realizations . . . . .	166
4.23	different types of non-hierarchical edges . . . . .	172
4.24	example for the number of flat crossings . . . . .	174
4.25	subsystems with different compartments . . . . .	186
4.26	polymorphic shapes of subsystems . . . . .	187
4.27	unpleasing edges at a subsystem . . . . .	188
4.28	edge crossing problems . . . . .	203
4.29	area for nodes and edges . . . . .	210
4.30	coordinates-valid graph . . . . .	213
4.31	cluster-valid graph . . . . .	215
4.32	partitioning of vertical ports . . . . .	221
4.33	moving nodes and clusters . . . . .	224
4.34	hidden nodes hopping at an UML package . . . . .	226
4.35	certain shapes considered by minpath heuristic . . . . .	228
4.36	preparing the orthogonalization of non-hierarchical edges . . . . .	232
4.37	edge routing (second step) . . . . .	233
4.38	postprocessing of hyperedges . . . . .	237
5.1	layout metrics: edge crossings . . . . .	248
5.2	layout metrics: avoidable bend situations . . . . .	250
5.3	layout metrics: edge lengths . . . . .	251
5.4	layout metrics: various rank shapes . . . . .	255

5.5	layout metrics: handling of overlapping elements . . . . .	256
5.6	erroneous metrics example diagram . . . . .	259
5.7	aesthetic metrics example diagram . . . . .	259
5.8	layout comparison: packages by yWorksUML . . . . .	263
5.9	layout comparison: packages by <i>SugiBib</i> . . . . .	263
5.10	layout comparison: association class by yWorksUML . . . . .	265
5.11	layout comparison: association class by <i>SugiBib</i> . . . . .	265
5.12	layout comparison: simple class diagram by GoVisual . . . . .	267
5.13	layout comparison: simple class diagram by yWorksUML . . . . .	267
5.14	layout comparison: simple class diagram by <i>SugiBib</i> . . . . .	267
5.15	<i>SugiBib</i> drawing of a large diagram used for runtime measurements . . . . .	271
5.16	crossing numbers on non-compound graphs by nodes . . . . .	272
5.17	crossing numbers on non-compound graphs by density . . . . .	272
5.18	crossing numbers on compound graphs by nodes . . . . .	273
5.19	crossing numbers on compound graphs by density . . . . .	273
5.20	runtime of the edge crossing reduction on non-compound graphs by nodes . . .	274
5.21	runtime of the edge crossing reduction on non-compound graphs by density . .	274
5.22	runtime of the edge crossing reduction on compound graphs by nodes . . . . .	275
5.23	runtime of the edge crossing reduction on compound graphs by density . . . . .	275
5.24	runtime of S1-S9 by nodes . . . . .	277
5.25	runtime of S1-S9 by density . . . . .	277
5.26	runtime of the rank assignment by nodes . . . . .	277
5.27	runtime of the rank assignment by density . . . . .	277
5.28	runtime of the edge crossing reduction by nodes (mean values) . . . . .	277
5.29	runtime of the edge crossing reduction by density (mean values) . . . . .	277
5.30	runtime of S12-S14 by nodes . . . . .	277
5.31	runtime of S12-S14 by density . . . . .	277
5.32	runtime of the coordinates assignment by nodes . . . . .	278
5.33	runtime of the coordinates assignment by density . . . . .	278
5.34	runtime of S16-S21 by nodes . . . . .	278
5.35	runtime of S16-S21 by density . . . . .	278
5.36	runtime of the entire layout algorithm by nodes . . . . .	278
5.37	runtime of the entire layout algorithm by density . . . . .	278
5.38	runtime of the entire layout algorithm by nodes (native) . . . . .	279
5.39	runtime of the entire layout algorithm by density (native) . . . . .	279
6.1	<i>SugiBib</i> architectural overview . . . . .	286
6.2	preprocessing macro phase: class diagram . . . . .	289
6.3	rank assignment: class diagram . . . . .	291
6.4	edge crossing reduction: class diagram . . . . .	293
6.5	coordinates assignment: class diagram . . . . .	295
6.6	postprocessing macro phase: class diagrams . . . . .	297
6.7	UML information classes: class diagram . . . . .	299

---

6.8	I/O subframework: class diagram . . . . .	301
6.9	application subframework: class diagram . . . . .	304
6.10	layout metrics: class diagram . . . . .	306
6.11	relative sizes of the main parts of <i>SugiBib</i> . . . . .	307
6.12	screenshot of the regression test results . . . . .	309
6.13	type cast bottleneck . . . . .	318
6.14	architectural changes to eliminate type casts . . . . .	319
6.15	plug-in mechanism: class diagram . . . . .	322
A1.1	drawing of hyperedges . . . . .	334
A1.2	drawing of an example diagram from the UML introduction . . . . .	334
A1.3	alternative drawing of an example diagram from the UML introduction . . . . .	334
A1.4	drawing of the UML tool evaluation diagram . . . . .	336
A1.5	drawing of some reflective associations and an association classes . . . . .	337
A1.6	<i>SugiBib</i> drawing of the UML overview class diagram . . . . .	338
A1.7	<i>SugiBib</i> drawing of the UML edge crossing reduction class diagram . . . . .	339
A1.8	<i>SugiBib</i> drawing of the UML postprocessing class diagram . . . . .	340
A1.9	<i>SugiBib</i> drawing of the UML infrastructure class diagram . . . . .	341
A1.10	<i>SugiBib</i> drawing of the layout metrics class diagram . . . . .	342
A1.11	drawing of a reversed engineered students' project . . . . .	343
A3.1	cluster projection . . . . .	348

## A6 List of Algorithms

---

4.1	adjustSemanticalIssues . . . . .	123
4.2	rankAssignment . . . . .	144
4.3	hierarchicalRankAssignment . . . . .	145
4.4	preprocess . . . . .	146
4.5	enforceHierarchicalClusterDependencies . . . . .	147
4.6	postprocess . . . . .	151
4.7	calculate flat matrices . . . . .	183
4.8	update flat matrices after adding $v$ . . . . .	185
4.9	calculateInsertPosCluster . . . . .	191
4.10	ip_handleGlobal . . . . .	192
4.11	ip_handleNonGlobal . . . . .	192
4.12	sortAndAlignRank . . . . .	195
4.13	updateSortValues . . . . .	196
4.14	hierarchical . . . . .	200
4.15	h_insertNodeOrCluster . . . . .	200
4.16	h_insertCluster . . . . .	201
4.17	basicInitializeXYCoordinates . . . . .	207
4.18	xcoordinate . . . . .	208
4.19	coordinatesAssignment . . . . .	216
4.20	c_preprocessing . . . . .	217
4.21	initializeClusterX (as a part of initializeXYCoordinates) . . . . .	219
4.22	sortPorts . . . . .	220
4.23	c_iteration . . . . .	223
4.24	packcutAreas . . . . .	229
4.25	c_postprocessing . . . . .	230
4.26	prepareEdgePorts . . . . .	231
4.27	postprocess_hyperedges . . . . .	238
4.28	snapToGrid . . . . .	240
4.29	createResult . . . . .	241
A3.1	ip_handleNonGlobal_improved . . . . .	348

## A7 List of Tables

---

3.1	model levels of UML and XMI . . . . .	48
3.2	mandatory UML criteria . . . . .	80
3.3	UML criteria to be realized . . . . .	84
4.1	macro steps of the <i>SugiBib</i> algorithm . . . . .	104
4.2	runtime complexities of the preprocessing macro phase . . . . .	135
4.3	runtime complexities of the rank assignment . . . . .	155
4.4	runtime complexities of the edge crossing reduction macro phase . . . . .	203
4.5	runtime complexities of the intermediary macro phase . . . . .	206
4.6	runtime complexities of the coordinates assignment macro phase . . . . .	235
4.7	runtime complexities of the postprocessing macro phase . . . . .	242
4.8	complexities of the macro steps . . . . .	243
5.1	realized metrics for aesthetic principles . . . . .	247
5.2	metric values of the example diagrams . . . . .	260
5.3	features of current UML layout tools . . . . .	264
5.4	metric values of the tool comparison . . . . .	268
5.5	runtime values of large graphs . . . . .	280
6.1	graph drawing libraries . . . . .	282
6.2	some graph drawing systems . . . . .	283
6.3	runtime measurements of alternatives to eliminate type casts . . . . .	320



# A8 Index

---

=, 113  
:=, 113  
⊥, 112  
*n*-level hierarchy, 138  
.NET, 282  
3D visualization, 24  
  
abstract node naming function, 121  
abstract window toolkit, *see* AWT  
acyclic graph, 29  
API, 260  
application initializer plug-in, 305  
application library, 284  
application programming interface, *see* API  
association class, 202, 236  
association classifier, 14, 205  
automatic documentation, 27  
AWT, 303  
  
BDP, 156  
bend, 54, 137, 233  
bipartite crossing numbers, 163  
bipartite drawing problem, *see* BDP  
bipartite graph, 156  
blackbox test, 308  
bottom flat area, 233  
bottom flat edge, 172  
browsing, 23  
  
C++, 113, 282  
CASE, 1, 36

CDIF, 47  
circular layout, 24  
cluster  
    augmented graph, 213  
    base node, *see* compound  
    border node, 148, 208, 317  
    clustered graph, 31  
    dependency, 205, 226  
    parent node, *see* compound  
    separator node, 213  
colors in UML, 21, 24  
comment node, 202  
compartment, 10  
complexity, 112  
component coupling, 68  
composite node, 96, 110, 202  
compound  
    augmented graph, *see* cluster  
    base node, 106, 223  
    border node, *see* cluster  
    compound graph, 31, 44, 106  
    dependency, *see* cluster  
    parent node, 106, 223  
    separator node, *see* cluster  
computer aided software engineering, *see*  
    CASE  
concentrical layout, 24  
concurrent versions system, *see* CVS  
contents disposer, 302

- 
- contents exporter, 300
  - contents interpreter, 300
  - coordinates assignment
    - global layout method, 208
    - gravity-driven algorithm, 209
    - pendulum method, 209
    - priority layout method, 207
    - quadratic programming method, 207
  - coordinates- and cluster-valid graph, 214
  - coordinates-valid graph, 212
  - costs of communication, 27
  - coupling, 67, 81, 152
  - covariant type system, 320
  - crossing matrix, 163
  - crossing number calculation plug-in, 292
  - CVS, 23, 321
  - cyclic graph, 29
  
  - declarative modeling approach, 48
  - default UML layout, 19
  - depth-first cycle-breaking, 134
  - design pattern, 19, 287
    - bridge, 303
    - visitor, 302
  - DiaGen, 40
  - diagram interchange, 49
  - diagram language, 30
  - diffable graph output, 311
  - digraph, 29
  - directed graph, 29
  - dummy leaf, 133
  - dummy node, 101, 139, 144, 173, 189, 202, 208, 225, 228, 233, 237
  - dummy nodes, 138
  - dummy root, 132, 151
  - dynamic layout rules, 53
  - dynamic stability, 124
  
  - Eclipse, 303, 321
  - edge complexity metric, 288
  - edge crossing
    - hierarchical crossings, 161
  - edge crossings
    - n*-layer algorithms, 157
    - n*-level backtracking method, 199, 270
    - 2-layer algorithms, 157
    - adaptive insertion, 159
    - adjacent exchange method, 158
    - assignment method, 158
    - averaging method, 158
    - barycenter method, 158, 198, 270
    - branch and bound method, 158
    - context free, 157
    - context sensitive, 157
    - dot heuristic, 158
    - greedy insertion, 158
    - greedy switching, 158
    - guided breadth-first search, 159
    - hierarchical method, 199, 270
    - hybrid approaches, 157
    - intertwined cluster enforcement, 196
    - layer-by-layer sweep, 157
    - median method, 158, 199, 270
    - non-incremental, 157
    - postprocessing cluster enforcement, 197
    - relative degree method, 158
    - semimedian method, 158
    - sifting method, 159
    - splitting method, 158
    - switch method, 158
    - tabu search, 159
    - vertex-exchange graph, 159
  - effective runtime, 112, 242
  - entity-relationship diagram, *see* ER
  - ER, 15, 39
  - exception, 313
  - factory, 284, 300, 305
  - feedback arc set, 133
  - flat edge, 172
  - flat flat edge, 172, 202
  - font, 300
  
  - general graph, 31
  - geon diagram, 61
  - GoVisual, 32, 40, 87, 262
  - graph, 115

- graph copy mechanism, 111, 276, 284, 298, 316, 317, 324
- graph drawing
  - algorithmic approach, 32
  - augmentation approach, 35
  - declarative approach, 35
  - divide and conquer approach, 34
  - evolutionary approach, 35
  - force-directed approach, 34, 40
  - genetic approach, 35
  - hierarchical approach, 33
  - STT method, 33
  - Sugiyama approach, 33
  - topology-shape-metrics approach, 32, 40
  - visibility approach, 34
- graph element factory, 284
- graph mapper, 284
- graphical standards, 30
- graphical user interface, *see* GUI
- GraphML, 48
- GRASP, 159
- gravity algorithms, 35
- grid line layout, 233
- GTXL, 47
- GUI, 284, 298
- GXL, 47
  
- handicapped engineers, 48
- hashtable, 345
- hidden leaf, 133
- hidden node, *see* dummy node
- hidden nodes hopping, 225
- hidden root, 132, 151
- higraph, 31
- HTML, 310
- huffman encoding, 121
- hyperedge, 76
  
- I18N, 283, 305
- IDE, 303, 321
- identification object, 111
- incremental editing, 27
- incremental extension, 94, 149
- incremental layout, 45, 329
- information instance life cycle, 111
- information object, 110
- initial coordinates assignment, 206
- inner node area, 210
- integrated development environment, *see* IDE
- inter-rank validity, 141
- interaction coupling, 68
- interconnect matrix, 160
- internationalization, *see* I18N
- intra-package coupling, 68
- intra-rank validity, 141
- iterator, 287, 312, 317
  
- jarInspector, 32, 40, 87, 262
- Java, 243, 281
- Java compiler, 312
- Java generics, 315
- java metadata interface, *see* JMI
- Java virtual machine, *see* JVM
- JET, 276
- JMI, 50, 260, 302
- JTransform, 49, 276, 279, 312
- JVM, 270, 312, 322
  
- layout stability, 124
- lazy attributes, 318
- LCC, 120
- length of the hierarchy, 138
- line row, 209, 231
- Linux, 270, 276
- list, 344
- local layering, 137
- locked graph, 284
- longest path layering, 137
  
- macro phase, 104
- magnetic fields, 35, 40
- mandatory aesthetic criteria, 72
- matrix realization, 160
- MDA, 23
- MDI, *see* multiple document interface

- mental map, 45
- meta-model levels, 48
- metadata repository, 50
- Microsoft Windows, 276
- middle rank area, 233
- minimum feedback arc set problem, 133
- mixed compound graph, 44, 127, 215, 218
- model driven architecture, *see* MDA
- MOF, 48
- moving nodes, 223
- multi-level non-hierarchical edge, 172
- multiple document interface, 285
- mutable graph, 284
  
- name compartment, 10
- native compiler, 276
- native version, 322
- navigation, 23
- navigation path, 108
- nesting relation, 106
- network simplex method, 143, 145
- node complexity metric, 288
- node hopping, 202
- node naming function, 119, 144
  - abstract, 121
  - huffman encoding, 121
  - preorder-size numbering, 121
- node storage strategy, 110
- node-link diagrams, 61
- not intended hierarchical level, 89
- null, 112
  
- obfuscated version, 322
- Object Modeling Technique, 34
- object pools, 317, *see* pooling
- object-oriented metrics plug-in, 285
- OCL, 13
- Oh-notation, 112
- OMG, 21, 48, 94, 261
- OMT, 34
- one sided crossing minimization problem, 157
- one-layer layout, 89, 269
  
- optional aesthetic criteria, 72
- ordering strategy, 292
- outer node area, 211
  
- path, 29
- planar embedding, 29
- plug-in manager, 322
- polymetric view, 70
- polyomino packing algorithm, 239
- pooling, 313, 325
- port, 188, 205, 212, 231
- port penalty, 188
- postprocessing programming rule, 204, 325
- preorder-size numbering, 121
- private edge, 211
- pseudo-coordinates, 202
- Ptide, 49
- public edge, 211
  
- query mechanism, 290, 316
  
- rank number, 138
- reference equality, 314
- reverse engineering, 24, 27
- reversed edges, 133
- ringed layout, 24
- round-trip engineering, 26
- running *SugiBib*, 321
- runtime consistency checks, 325
- runtime consistency test, 310
  
- SCC, 134
- SDI, *see* single document interface
- Seemann algorithm, 40, 93, 148
- self-loop, 28, 177
- semantic constraints, 53
- simple graph, 28
- simplex method, 137, 143
- simulated annealing, 35
- simulated sintering, 35
- single document interface, 285
- single-layer layout, *see* one-layer layout
- span, 136
- standard widget toolkit, *see* SWT
- standardization, 27

- static layout rules, 52
- stress test, 308
- strongly connected component, 134
- structural stability, 124
- STT algorithm, 33
- SugiBib, 40
- Sugiyama algorithm, 33, 148, 242
- Swing, 303
- SWT, 303
  
- theoretical complexity, 112, 242
- three amigos, 34
- top flat area, 233
- top flat edge, 172
  
- UML
  - abstract, 11
  - active object, 19
  - activity diagram, 8
  - aggregation, 13
  - anchor notation, 18
  - annotation, 16
  - association, 12
  - association class, 14
  - association classifier, 14
  - attributes, 10
  - basic style guide, 9
  - class, 10
  - class diagram, 8, 10
  - class instance, 19
  - collaboration, 19
  - collaboration diagram, 8, 19
  - comment, 16
  - component diagram, 9
  - composition, 13, 17
  - constraint, 10
  - default layout, 19
  - dependency relation, 17
  - deployment diagram, 9
  - discriminator, 11
  - ellipses, 11
  - higher association, 15
  - inheritance relation, 11
  - inner class, 18
  - interface, 11
  - joined arcs, 18
  - lolly, 17
  - model, 15
  - multi-object, 19
  - multiple dependencies, 17
  - multiplicity, 12, 17
  - n-ary association, 15
  - namespace, 15
  - navigational indicator, 12
  - nested class, 18
  - node naming function, 119
  - object, 19
  - operations, 10
  - package, 15
  - package diagram, 8
  - qualifier, 12
  - realize relation, 11
  - role, 12
  - separate target style, 18
  - sequence diagram, 8
  - shared target style, 18
  - statechart diagram, 8
  - stereotype, 10
  - style guide, 10
  - subsystem, 15
  - tag-value list, 10
  - templates, 17
  - ternary association, 15
  - use case, 15
  - use case diagram, 8
  - visibility, 11, 12
  - xor-constraint, 14, 76
- UML 2.0, 9, 327
- UML-XMI, 48
- UMLscript, 48, 279, 302
- universal arrow diagram logic, 24
- upward edge, 134
- user defined layout constraints, 72
  
- Very Large Scale Integration, *see* VLSI
- virtual leaf, 133, 206, 233

virtual node, *see* dummy node

virtual root, 132, 151, 206, 228, 233

VLSI, 39, 233, 328

W3C, 47

whitebox test, 308

XMI, 9, 48, 260, 302

XMI[DI], 49, 261, 302

XML, 47, 310

XML debugging format, 311

XML Metadata Interchange, *see* XMI

XSLT, 302, 310

XUMLscript, 49, 302

yFiles, 40, 87, 262

yWorksUML, 32, 40, 87, 262