Development and Implementation of an Algorithm for the Automatic Computation of Layouts for UML2.0 Class Diagrams

Matthias Schrepfer
Bachelor Thesis

Development and Implementation of an Algorithm for the Automatic Computation of Layouts for UML2.0 Class Diagrams

Matthias Schrepfer

19th June 2008

Department of Computer Science
Växjö University

Supervisor:
Rüdiger Lincke, Phil. Lic.
Abstract
Reverse Engineering of software is a complex task. It is supported by various analyses and the computation of certain software quality metrics. To get an overview of the structure of the analysed software it is often necessary to visualize the relations between different components of software.

The VizzAnalyzer tool was developed for these purposes and allows visualizing the internal structure of software systems as UML2.0 class diagrams. The existing Eclipse-based visualization plug-in does not allow to automatically lay out the computed graph. Applying the layout manually is not feasible for larger graphs. A proper layout is needed to understand the provided information intuitively and the usefulness of the visualization is, on top of that, strongly connected to its layout.

In this thesis, we first state requirements for the development and implementation process of a suitable layout algorithm. We further describe the development, design, and implementation of the Sugiyama layout algorithm into the existing Eclipse-based visualization plug-in. We add these new functionalities to be able to automatically compute a layout for given UML2.0 class diagrams achieving a proper Sugiyama layout for this type of diagrams.

At the end of this thesis, we compare the results achieved by the implemented algorithm with the results achieved by yEd given the same diagrams as input.

Keywords: VizzAnalyzer, GRAIL, visualization plug-in, UML2.0 class diagrams, layout algorithm, Sugiyama
Table of Contents

1 INTRODUCTION ........................................................................................................................................ 5
  1.1 CONTEXT OF THE THESIS ..................................................................................................................... 5
  1.2 PROBLEM .............................................................................................................................................. 5
  1.3 GOALS AND CRITERIA .......................................................................................................................... 6
  1.4 MOTIVATION ......................................................................................................................................... 7
  1.5 OUTLINE ............................................................................................................................................... 7

2 BACKGROUND .......................................................................................................................................... 8
  2.1 VIZZANALYZER .................................................................................................................................... 8
  2.2 GRAIL ................................................................................................................................................... 8
  2.3 UML CLASS DIAGRAMS ....................................................................................................................... 8
  2.4 YED ..................................................................................................................................................... 9

3 REQUIREMENTS ....................................................................................................................................... 10
  3.1 READABILITY ....................................................................................................................................... 10
  3.2 NO MANIPULATION OF GRAPH INPUT .................................................................................................. 11
  3.3 ABSTRACT UML INFORMATION ........................................................................................................... 11
  3.4 SUGIYAMA ALGORITHM ...................................................................................................................... 12
  3.5 DIFFERENT EDGE TYPES ................................................................................................................... 13
  3.6 LABEL OF NODES .................................................................................................................................. 13

4 ANALYSIS OF THE SUGIYAMA ALGORITHM .......................................................................................... 14
  4.1 READING LITERATURE ......................................................................................................................... 14
  4.2 STEPS OF LAYOUT ALGORITHM ........................................................................................................... 15
    4.2.1 Getting a Graph as Input .................................................................................................................. 15
    4.2.2 Making the Input Acyclic ............................................................................................................... 15
    4.2.3 Longest Path Layering .................................................................................................................... 17
    4.2.4 Creating Layers ................................................................................................................................ 19
    4.2.5 Removing Long-span Edges ............................................................................................................ 20
    4.2.6 Incidence Matrices .......................................................................................................................... 22
    4.2.7 Crossing minimization .................................................................................................................... 24
    4.2.8 Layout positioning ............................................................................................................................ 29
    4.2.9 Computing position information .................................................................................................... 31
    4.2.10 Displaying result of layout computing ........................................................................................ 33
  4.3 ADDITIONAL FEATURES....................................................................................................................... 33

5 DESIGN & IMPLEMENTATION .................................................................................................................. 35
  5.1 DIVIDE AND CONQUER ......................................................................................................................... 35
  5.2 CODE REUSAL ..................................................................................................................................... 35
  5.3 STORING IN MEMORY ............................................................................................................................ 35
  5.4 INSTANTIATING THE SUGIYAMA ALGORITHM ..................................................................................... 35
  5.5 DEBUGGING INFORMATION .................................................................................................................. 35
  5.6 TESTING ................................................................................................................................................. 36
  5.7 PERFORMANCE .................................................................................................................................... 36

6 COMPARISON OF SUGIYAMA RESULTS AND RESULTS ACHIEVED BY YED ........................................... 38

7 CONCLUSION AND FUTURE WORK ...................................................................................................... 42
  7.1 CONCLUSION ....................................................................................................................................... 42
  7.2 FUTURE WORK ..................................................................................................................................... 43

REFERENCES ............................................................................................................................................... 45

APPENDIX A ............................................................................................................................................... 46
**Table of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>A UML diagram</td>
<td>8</td>
</tr>
<tr>
<td>2-2</td>
<td>A UML class diagram drawn and laid out by yEd</td>
<td>9</td>
</tr>
<tr>
<td>3-1</td>
<td>Graph showing readability elements</td>
<td>11</td>
</tr>
<tr>
<td>3-2</td>
<td>UML abstraction</td>
<td>12</td>
</tr>
<tr>
<td>4-1</td>
<td>Pseudo code for advanced Greedy heuristic</td>
<td>16</td>
</tr>
<tr>
<td>4-2</td>
<td>Making a graph acyclic</td>
<td>17</td>
</tr>
<tr>
<td>4-3</td>
<td>Pseudo code for longest path layering</td>
<td>18</td>
</tr>
<tr>
<td>4-4</td>
<td>Longest path layering</td>
<td>18</td>
</tr>
<tr>
<td>4-5</td>
<td>Pseudo code for creating layers</td>
<td>19</td>
</tr>
<tr>
<td>4-6</td>
<td>Creating layers</td>
<td>19</td>
</tr>
<tr>
<td>4-7</td>
<td>A long-span edge</td>
<td>20</td>
</tr>
<tr>
<td>4-8</td>
<td>Pseudo code for removing long-span edges</td>
<td>21</td>
</tr>
<tr>
<td>4-9</td>
<td>Removing of long-span edges</td>
<td>22</td>
</tr>
<tr>
<td>4-10</td>
<td>Establishing incidence matrices</td>
<td>23</td>
</tr>
<tr>
<td>4-11</td>
<td>Pseudo code for incidence matrices</td>
<td>23</td>
</tr>
<tr>
<td>4-12</td>
<td>Crossing minimization 2-layer</td>
<td>25</td>
</tr>
<tr>
<td>4-13</td>
<td>Crossing minimization phase1 down</td>
<td>25</td>
</tr>
<tr>
<td>4-14</td>
<td>Crossing minimization overall</td>
<td>26</td>
</tr>
<tr>
<td>4-15</td>
<td>Pseudo code for crossing minimization</td>
<td>26</td>
</tr>
<tr>
<td>4-16</td>
<td>Pseudo code for phase1</td>
<td>27</td>
</tr>
<tr>
<td>4-17</td>
<td>Incidence matrices general</td>
<td>27</td>
</tr>
<tr>
<td>4-18</td>
<td>Incidence matrices with barycentres</td>
<td>28</td>
</tr>
<tr>
<td>4-19</td>
<td>Pseudo code for phase2</td>
<td>28</td>
</tr>
<tr>
<td>4-20</td>
<td>Incidence matrices phase2 up</td>
<td>29</td>
</tr>
<tr>
<td>4-21</td>
<td>Pseudo code for layout positioning</td>
<td>30</td>
</tr>
<tr>
<td>4-22</td>
<td>Layout positioning</td>
<td>31</td>
</tr>
<tr>
<td>4-23</td>
<td>Pseudo code for layout positioning down</td>
<td>31</td>
</tr>
<tr>
<td>4-24</td>
<td>Pseudo code for label parsing</td>
<td>32</td>
</tr>
<tr>
<td>4-25</td>
<td>Pseudo code for layout positioning</td>
<td>32</td>
</tr>
<tr>
<td>5-1</td>
<td>Performance tests</td>
<td>37</td>
</tr>
<tr>
<td>6-1</td>
<td>Graph laid out by yEd and Sugiyama</td>
<td>39</td>
</tr>
<tr>
<td>6-2</td>
<td>Graph with 15 nodes and 30 edges</td>
<td>40</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Today many software products are developed and published. As soon as the software is getting more complex, not only one single engineer can develop the whole program. Many engineers (software analysts, programmers, testers etc.) must work together in order to ensure that the software does what it was originally planned for. One single engineer cannot analyse, develop and program all parts of the final software product. Instead all engineers of that project must work together to produce the final product. The persons involved in the project must continuously communicate and exchange information with each other, e.g. about the current development or implementation steps. The process of sharing and forwarding information is very important for every single participant. For sharing information normally any kind of diagram is suitable, but especially for computer scientists UML (Unified Modelling Language) is widely used when it comes to software developing.

With the help of UML, information can efficiently be shown to all persons taking part in the process of developing, maintaining, and supervising a software system. In UML a whole set of diagrams is defined. Here, we talk about UML class diagrams which show information about single classes, their attributes, methods as well as their relations to other classes. These diagrams are also most commonly used.

For measuring the quality of software, it must be reverse engineered. Hereby, information of the given software must be retrieved or computed by the tool which analyses the software. The results must also be shown and communicated to the analysts, developers and testers. UML class diagrams are also used in this area.

1.1 CONTEXT OF THE THESIS

VizzAnalyzer is a software tool which does reverse engineering of large Java systems. The tool reads software projects and analyses the source code by performing a number of quality analyses. The tool also visualizes the internal structure of the investigated software. VizzAnalyzer is a product for visualizing software quality (Löwe, 2008). The current implementation of the connected visualization plug-in creates a UML conform visualization but the layout of the computed diagram is not laid automatically. UML is the standard for the representation of software systems under development and maintenance. Information of nearly any kind, not only software systems, can be visualized with UML. For the process of information exchange an effective layout is needed to communicate information quickly and understandable to other software engineers. The current plug-in does not further optimize the readability of the diagram.

The goal addressed by this thesis is to develop and implement a suitable layout algorithm for computing the layout on UML class diagrams automatically. The existing implementation of the visualization plug-in shall be extended for this functionality. The algorithm shall take a given diagram and compute a layout with a high readability. This optimized layout is then stored in the graph information.

1.2 PROBLEM

Producing a layout with high readability is not an easy task and some research work is going on in this area. Before the actual layout computation can take place we have to define parameters with which we can measure the readability of a layout on a given graph. These parameters are mostly dependent on human cognition and aesthetics (Sugiyama, 2002). Measurements are necessary to be able to compare different layouts and their readability. The layout algorithm must cover these aspects.

Computing a good layout by hand is, even for a small graph, quite complex and hard to do. For large graphs it is simply infeasible. The layout algorithm shall compute a
layout on a given graph. This graph shall not be limited in its number of nodes or edges. As the computation complexity increases extremely the more nodes and edges are stored in the graph, the development and implementation must follow specific rules in order not to waste computation time needed by the algorithm. Otherwise, large graphs cannot be computed in a predictable amount of time. The implemented algorithm must use heuristics in order to decrease computation time.

Redrawing a graph once it has been modified is not easily possible even if one has the old layout. The new one may completely be different and not necessarily dependent on the previous computed layout. Especially in software projects where a source code is often modified or a new code added, the graph visualization and the corresponding layout might change quite often which increases the need for an automatic layout computing.

The layout algorithm is to be included in the existing GRAIL (see Chapter 2.2) plug-in. It must especially follow up the problem of readability which occurs when graphs are visualized. If the graph is not shown with a high readability, other software engineers might not recognize important information shown. Therefore, it is crucial to present a graph layout with a good readability to improve aspects of human cognition. This is the most critical problem and it must be handled by the algorithm.

1.3 GOALS AND CRITERIA

This section describes the goals pursued by this thesis in order to solve the problems and the criteria used for evaluating the goals.

- The primary goal of this thesis is the selection and implementation of at least one suitable layout algorithm stated in reference literature. We investigated two different algorithms according to their complexity and the costs to implement them into the existing software. The algorithm must store the computed layout information in the given graph.
- The second goal is that the implemented algorithm computes a “good” layout. Several criteria for such a layout were known before the development process started. The layout computed by the algorithm shall consider all of them as well as further criteria which were recognized during the development and implementation phase. The following goal criteria were stated:
  - Single classes shown in the UML class diagram must be placed without any overlapping between them.
  - Edges showing the type of relationship between two classes must not cross any other classes in between.
  - Crossings between edges of different classes shall be minimized to a huge extent.
- The third goal is that the implementation of the software has to have options to configure the algorithm, e.g. maximum amount of computing time. Further parameters have to be chosen while developing and implementing takes place if needed. The software shall provide an application programming interface (API) with which the algorithm can be called as well as configured with parameters.
- The last goal is to compare the results of the algorithm with results produced by yEd where the same diagram is given as input. The output must be evaluated and advantages as well as drawbacks clearly stated.

The main focus is on points one and two as these goals greatly influence the development and implementation process. The goal of this thesis is to develop a better algorithm that considers all goals and all criteria stated. The produced results shall be equivalent or even better than those of yEd. yEd is a tool with which graphs can be visualized and laid out (see Chapter 2.4). Furthermore, the algorithm must be
programmed in Java as the existing plug-in is programmed in that language. The algorithm must also be able to handle various graphs not dependent on the number of nodes or edges.

1.4 MOTIVATION

The diagrams produced by the existing visualization plug-in are not laid out at all. Whenever software engineers discuss software with the help of these diagrams there will probably be a lack of information exchange. This is due to the fact that especially larger diagrams are quite complex and the cognition of information shown in them is up to the reader. An automatic layout algorithm can produce a layout where the cognition is greatly enhanced and therefore the need for such an algorithm increases the larger the diagrams are.

With the help of the VizzAnalyzer generally large systems are reverse engineered which means that also the output in diagram form is large. Now we want to have software which immediately computes a good layout. Not only the person/team who does the reverse engineering can understand the content of the diagram faster, but also other software engineers involved in the overall process benefit from the better layout.

Finally, we also want to have the layout compared to other tools which are also able to compute layouts for our diagrams. Some tools produce good layouts but some criteria do not fit to our needs. The algorithm shall take care of all our needs stated and consequently produce a layout fitting best to our needs.

1.5 OUTLINE

Chapter 2 describes the background and all other topics necessary to understand the context of this thesis.

Chapter 3 covers requirements we stated before starting to develop and implement our layout algorithm. It explains why we considered all goals stated in Chapter 1.3. Here, we also explain why we chose to implement the Sugiyama layout algorithm.

Chapter 4 shows the basic ideas and concepts behind the Sugiyama layout algorithm. The criteria important for the actual development are explained. All single steps are introduced and shown systematically. This chapter also mentions features we additionally added to our software to further improve special criteria.

Chapter 5 gives an outlook on design decisions and the implementation of the algorithm.

Chapter 6 shows the comparison of results achieved by our own implementation and results yEd produced with some parameters.

Chapter 7 concludes this thesis and shows future work.

Appendix A is a tutorial showing how our layout algorithm can be implemented in software and how to use it.
2 BACKGROUND
This section describes some background knowledge necessary to understand certain parts of this thesis.

2.1 VIZZANALYZER
VizzAnalyzer is a reverse engineering framework developed at Växjö University in Sweden with which software quality can be measured. Quality standards and metrics can be defined before analysing the software system. Source code of large software systems is read and analysed according to defined standards and metrics. The tool can also visualize programs with the help of a visualization plug-in. Information about the investigated system is then passed further on to the plug-in.

2.2 GRAIL
The GRAIL (GRAph ImpLementation) is a plug-in which can be connected to the VizzAnalyzer system. The plug-in is used to represent internal data handled within the VizzAnalyzer. All graph information is handled by the plug-in and the corresponding graph properties, e.g. nodes or edges. In this thesis we use GRAIL in order to manipulate the existing graph information and to access all data needed to compute the layout of that graph. This step is further described in Chapter 4.

2.3 UML CLASS DIAGRAMS
After object-oriented programming was introduced, programming changed completely. Object oriented programming is nowadays widely used. In a program not only simple data types are used, but also so-called objects. An object is an instance of a simple class defined in a program. Even more important is the fact that objects can send and receive information from other objects.

The object itself can contain several data: attributes and methods. An attribute can be e.g. an int (integer) variable and a method can be a function which the object contains. The class, its attributes, and methods can have different characteristics, e.g. a method can be private.

In order to describe these objects in a diagram, we use the Unified Modelling Language and the corresponding class diagrams. Those clearly show how an object looks like, meaning which attributes and methods it contains as well as their characteristics in graphical form. Further more important is the fact that objects can have associations between each other. This means one object can, for instance, instantiate another or one class inherits another class. These associations are also clearly shown in the UML class diagram (see Figure 2-1).

The output of GRAIL is in form of a graph and this graph contains the UML information so the graph can be displayed as UML class diagram. For computing the layout, the output of GRAIL must follow certain requirements (see Chapter 3.5). We
use the UML information stored in the graph to compute the individual node position for the layout (see Chapter 4.2.9).

2.4 YED

The tool yEd is published by yWorks. It is a free Java Graph Editor. This editor was previously used to read the output of GRAIL and visualize the graph information. The program displays all UML information (attributes, methods, characteristics, and relations) stored in the graph. It can automatically compute the height and the width of all graph nodes and adapt the layout accordingly.

Different layouts can be applied to the output automatically. As the results were not meeting different readability criteria, this thesis is addressed to develop and implement one layout algorithm which produces the same or even better results than yEd does.

The tool provides next to other automatic layouts, layouts like orthogonal UML style or hierarchical classic layout. The algorithms the program uses are unknown.

The most significant drawbacks of layouts computed by yEd are:

- Long edges are drawn.
- Some edges are not straight.
- With the UML layout no clear hierarchical structure is shown.

These drawbacks reduce the readability for some layouts dramatically.

Figure 2-2 shows a UML class diagram drawn and automatically laid out by yEd.

![Figure 2-2: A UML class diagram drawn and laid out by yEd](image-url)
3 REQUIREMENTS

This chapter describes requirements the layout algorithm must meet. We further state how a standard graph algorithm can be used although the graph output produced by GRAIL contains UML information. After investigation of two algorithms the decision was to implement the Sugiyama layout algorithm and reasons for that are given. Finally, the requirements, which the input given to the layout algorithm must meet, are stated.

3.1 READABILITY

The readability of diagrams is the bottleneck when it comes to cognition of information shown in them. If the readability is not high, one might tend to not recognize some important facts even if they are shown, a fact that is based on human cognition. This is the most critical step a layout algorithm has to take care of. Sugiyama (2002) states that careful consideration is required when diverse demands related to human cognition are investigated. Before automatic graph drawing methods can be developed, these mentioned layout criteria have to be clearly determined.

Now the criteria which have been of most interest for our implementation are stated:

- **Hierarchical layout**: Vertices are placed on horizontal lines, so called layers. Depending on the node’s relations to others, its layer is determined (see Chapter 4.2.4). In a hierarchical layout edges are drawn from top to bottom, meaning all edges going from bottom to top are feedback edges (see Chapter 4.2.2).

- **Edge span**: The edge span between two nodes shall be minimized. The edge shall only be as long as necessarily needed. In other words, the layers between two nodes shall be minimized. This enhances parts of the layout algorithm and for the user these edges are easier to follow. Ideally, the edge span shall always be one.

- **Straightness of edges**: Once we cannot achieve an edge span of one, we try to have these long-span edges as straight as possible. This further enhances the overall readability as straight lines can be traced much easier by the human eye.

- **Minimization of edge crossings**: As soon as there are edge crossings, the human eye has problems to focus on edges. Furthermore, it is extremely difficult to decide where the edge continues after the crossings. As the number of crossings increases, the problem gets even worse. Therefore, the minimization of edge crossings is a central and critical point of a layout algorithm (see Chapter 4.2.7).

- **Close layout**: Vertices placed on adjacent layers shall be placed as close as possible to their neighbours. This also reduces the length of the edges as long edges are harder to follow.

- **Balanced layout**: Vertices shall be placed at the barycentre of neighbouring vertices. This strategy is the best for achieving balance. If the nodes cannot be placed at the barycentre, they shall at least be as close as possible (see Chapter 4.2.8).

Some elements of readability are extremely hard to measure as they are dependent on human cognition. So everybody may recognize the same layout differently. It is first extremely hard to meet all the readability requirements by the layout algorithm. Secondly, everybody has different priorities on different readability elements (Sugiyama, Shojiro, & Mitsuhiko, 1981). The algorithm, therefore, takes the most common criteria and uses constraints to achieve a high readability.

The above described elements are naturally associated with optimization problems (Battista, 1999). However, Battista further claims that these problems are
computationally hard which means that efficient heuristics or approximations must be used in layout algorithms.

Figure 3-1 presents a graph which fulfils the above mentioned readability criteria. First we can see that the nodes are placed in a hierarchical order. Second, the layout is balanced. The nodes four and five are placed beneath node two on both sides. Furthermore, the layout fulfils the criteria for a close layout as the nodes are placed as close as possible to their neighbours. At the right side of the figure a dummy node was inserted to remove a long-span edge, an edge with a span greater than one, going from node one to node six. In the current figure there are no edge crossings, so this readability element is also met.

3.2 NO MANIPULATION OF GRAPH INPUT

The algorithm is called with a “DirectedGraphInterface” as parameter like it is implemented in GRAIL. The graph is fully stored in memory and all related information can be accessed. The input is only read in the beginning. No information is written back to the graph nor updated at that time. Information needed for further computations is retrieved from the graph stored in memory. If data must be stored, a new memory is allocated for storing. The original graph shall not be modified in any way while the computations take place. Finally, when the computations are done and the layout is fully computed, this layout information is stored in the graph.

The initial idea was not to waste too much computing time to manipulate the original graph, for instance for creating dummy nodes (see Chapter 4.2.5). Especially for inserting dummy nodes, manipulating the graph would have cost time as the node and two edges would have had to be inserted. Otherwise the last step would have been to remove these dummy nodes and edges again as they are not relevant for the final layout.

3.3 ABSTRACT UML INFORMATION

Another important step is to abstract the UML information from the graph. This was crucial as most layout algorithms only work for “normal” graphs with nodes and edges. UML information is normally not handled by them. The implemented algorithm must handle UML data as this is a requirement stated.

An interesting observation was that GRAIL produces a normal graph as output and adds all the UML information into one variable. These data is stored for every node itself. When the graph was stored for displaying it in yEd, especially this information (beside other layout information) is stored in a file. yEd displays the normal graph and extracts this “label” tag for displaying the UML information.

We implemented a solution that takes advantage of this fact. For computing the layout we did not care about any UML information even if it is stored in the graph. As already described in Chapter 3.2 we do not manipulate the original graph, except in the last step of the algorithm, so the UML information will not be changed in any way. We can easily abstract this information and treat it as not being there. Later on it is used for
computing the height and width of each node (see Chapter 4.2.9) which is needed for positioning the nodes.

On the left side of Figure 3-2 a UML class diagram is presented where all UML information is displayed. To the right we see the abstraction of the given diagram. Only the nodes, their self-associations and the associations between them are shown.

3.4 SUGIYAMA ALGORITHM

Since the development of our own layout algorithm would have been too difficult we had to find an existing algorithm which is capable of computing a layout for our UML class diagram. This algorithm can then be adapted to our needs. We found two approaches in literature:

- Orthogonal layout using hierarchical and non-hierarchical elements (Gutwenger et. al, 2003).

After having investigated these two papers, we decided to focus on the first one. The reasons for that were the following:

- The layouts described follow a different approach. Sugiyama (2002) and Sugiyama, Shojiro, and Mitsuhiko (1981) always follow a hierarchical approach whereas the method described in Gutwenger et. al (2003) combines this approach with an orthogonal layout. By having compared both final layouts by examples stated in the papers we thought that a hierarchical layout is more likely to fit best to our needs.
- The Sugiyama layout algorithm is most widely used (Kaufmann & Wagner, 2001). Furthermore, this algorithm is described in detail in literature and well known.
- It is also one of the best algorithms to compute a hierarchical layout for graphs.
- The complexity of the Sugiyama method is known and can be handled quite well.
- The basic concepts are fully described and formulas for computing measures for controlling the algorithm are stated in literature.
- The algorithm uses advanced heuristics to perform crossing minimization as well as for layout positioning which is needed especially for larger graphs.
• Eichelberger (2000) introduced a framework for automatically drawing UML class diagrams based on the Sugiyama layout algorithm. This framework also uses other approaches to extend this algorithm according to different needs stated.

We used the paper of Sugiyama (2002) and Sugiyama, Shojiro, and Mitsuhiko (1981) as a guideline for developing and implementing the layout algorithm into the existing visualization plug-in.

3.5 **DIFFERENT EDGE TYPES**

In UML class diagrams the relations between objects (classes in software projects) are shown. There are different relations depending on how the objects are used within the software. Typical relations are inheritance, implementation or instantiation. This results in different edge types in the diagram as the type of relation needs to be clearly shown.

For our software development and implementation we abstracted all edge types to only one type and treated all edge types similarly. Contrary to the hierarchical layout, the orthogonal layout depends on different edge types and treats them differently.

As a consequence of this edge handling, the user must ensure that only edges which shall be shown in the layout are passed to the algorithm. If needed, he must filter out unwanted edges before passing the graph on. Our algorithm does not check whether edge types are correct.

3.6 **LABEL OF NODES**

Within the visualization plug-in each node has specific properties, e.g. incoming edges. Among others a label is set for each node. The label is of type string and it contains all UML information for this single node. The label must follow a specific structure which is needed for the layout positioning (see Chapter 4.2.9).

One requirement for the label is that all information must be stored in one line. Line breaks are not allowed. This is due to restrictions in the existing GRAIL implementation.

The second requirement is regarding the structure of each label. All information has to be inserted in one html-tag as label parsing relies on a special structure. Details on that structure are also explained in Chapter 4.2.9.
4 ANALYSIS OF THE SUGIYAMA ALGORITHM

In this chapter we describe how we used the layout algorithm. We first analysed the Sugiyama algorithm published in literature ((Sugiyama, 2002); (Sugiyama, Shojiro, & Mitsuhiko, 1981)). We studied the mode of operation and understood how these algorithms work. Interesting findings learnt while studying the literature are described in Chapter 4.1. Afterwards we introduce all steps needed to perform an automatic layout computing for a given graph. All single steps are described in detail in Chapter 4.2. As we developed and implemented the algorithm we noticed some steps which were not handled in literature. These points will be presented in Chapter 4.3. Decisions about the real implementation are defined in Chapter 5.

4.1 READING LITERATURE

The first and most important part of the analysis was to read reference literature about the Sugiyama algorithm and the algorithm for orthogonal layouts. While studying the literature, we soon recognized that both algorithms were quite complex and not easy to understand. After we had chosen to develop and implement the Sugiyama layout algorithm (see Chapter 3.4), we did not investigate the orthogonal layout further. Most information we found about Sugiyama layouts was published in Sugiyama (2002) and Sugiyama, Shojiro, and Mitsuhiko (1981). For special parts of the algorithms, e.g. making a graph acyclic or readability criteria, we also studied other books ((Battista, 1999);(Kaufmann & Wagner, 2001)).

An important characteristic of all algorithms studied is that they only work on standard graphs (see Chapter 3.3). Those graphs do not contain any UML information. For the implementation this information must be hidden from the algorithm. We developed our own functionalities to be able to handle this after we have computed the layout on abstracted graphs.

As the main focus is on the Sugiyama layout, we especially investigated this layout and the algorithm for it.

In the given reference literature the basic steps for the development and implementation were stated. In Sugiyama, Shojiro, and Mitsuhiko (1981) four basic steps are needed to achieve a good layout which are the following:

1. “Proper hierarchy”: a proper hierarchy is formed from a given set of directed pairwise relations among elements of the initially given graph. If the given graph does not fulfill the requirement for the layout algorithm, we have to pre-process it. Pre-processing will take care of cycles if a cyclic graph is given as input as well as inserting “dummy nodes” and edges if edges are long-span edges.

2. “Number of crossings”: permuting the orders of vertices contained in different layers shall reduce the number of crossings between different edges in the graph.

3. “Horizontal positions of nodes”: the horizontal position of nodes is computed by taking care of the following readability criteria: straightness of long-span edges, close layout of adjacent nodes and balanced layout of lines coming into or going out of a vertex.

4. A two-dimensional picture of the graph with its layout is finally drawn. The dummy nodes and edges are not shown in this drawing. Contained long-span edges are regenerated by removing all dummy nodes and edges, and displayed correctly.

By comparing our goals with the steps above, we noticed that step four could partly be skipped. The algorithm does not need to draw the graph. It shall only compute a good layout on the given graph and store its layout information. The individual steps are described in detail in Chapter 4.2.
We already knew that the necessary steps for the algorithm are quite complicated. Thus, we assumed that the computation time needed for these steps will be increased. This phenomenon is also explained in literature. Particularly crossing minimization and layout positioning is computationally expensive. In theory there will be optimum solutions for these problems but these are hard to compute. However, Sugiyama, Shojiro, and Mitsuhiko (1981) invented efficient heuristics for these problems. In the mentioned paper the authors present a performance test where they compare theoretical approaches with the heuristic methods. The invented methods are shown to be significantly effective.

After studying the complete layout algorithm we started the development process and first investigated how we can use the described functionalities within the existing visualization plug-in. We analysed how the input (= the graph information) is handed over by the current plug-in as well as the structure of it. The composition of the graph is further described in Chapter 4.2.1. Having developed certain steps we recognized that some descriptions stated in literature allow different interpretations. Knowing that this fact can lead to major problems, we wrote down our own interpretations for the future development. This greatly removed room for interpretations. However, we had to keep in mind that these decisions might influence other future decisions. There were several circumstances which we had to discuss but which were not covered by literature. This was a result of the fact that the Sugiyama algorithm works on and only with standard graphs. In our case we had to consider facts which were simply not essential for standard graphs, e.g. different edge types.

4.2 STEPS OF LAYOUT ALGORITHM
This chapter describes all steps developed for performing the Sugiyama algorithm. Every single phase is explained and important decisions are highlighted. The algorithm performs all methods in the order presented. Problems that occurred during the development process are presented together with their corresponding solutions.

Pseudo code samples and figures shall exemplify each individual step and give a better understanding of the overall algorithm.

4.2.1 Getting a Graph as Input
The input that is passed to the layout algorithm is an object of type “DirectedGraphInterface”. This interface class contains pre-defined methods for accessing and manipulating all information the graph object includes. Per definition we state that the layout algorithm can only be called by passing over this interface. It contains e.g. a method for accessing all nodes, which the graph contains, or the functionality to remove one edge.

Concluding from the interface name, one can see that we only handle directed graphs. In directed graphs every edge has one direction. It is a relation from a source to a target node. In this thesis we do not handle general undirected graphs where one edge can have no or two directions. For the given graphs only one direction per edge is allowed.

For our implementation of the layout algorithm we do not manipulate the original graph as mentioned already before. Only at the final step of the algorithm we update the “GraphProperties” of the given “DirectedGraphInterface” with position information (see 4.2.9).

4.2.2 Making the Input Acyclic
Another characteristic of a graph is that it is either cyclic or acyclic. This criterion is of importance for our development as the Sugiyama algorithm cannot handle directed
cyclic graphs. The method described in this chapter will show how we generate a directed acyclic graph from the given directed cyclic one. This check is always performed in order to be sure to have a correct graph input for the steps described in the following chapters.

The basic idea of making general directed graphs acyclic is to reverse the direction of edges which cause cycles in the given graph. The edges that are reversed build a so-called feedback edge set (FES). As it might be obvious, one might think by just taking out all edges that lead to cycles, we achieve a “good” acyclic graph. This is of course true but it has side effects which affects later steps of the algorithm to a huge extent. Therefore, we must constrain our actions not to increase the difficulty of later steps if possible.

By choosing as less feedback edges as possible we avoid the described effect. The problem of finding a set of the smallest number of feedback edges is a NP-complete problem called the minimum feedback arc set problem (Garey & Johnson, 1983). This emphasizes the necessity of using heuristics to compute a small feedback edge set.

We saw the following methods published in literature: depth first search (Tarjan, 1982), divide and conquer approach (Eades & Sugiyama, 1990), and Greedy algorithm (Kaufmann & Wagner, 2001). As the Greedy algorithm was the one of the fastest and best we chose this one. Kaufmann & Wagner (2001) further state that an Enhanced Greedy heuristic can be used to compute a special order of nodes given in the cyclic graph. The performance of the Greedy heuristic is investigated in Battista (1999) and it is proven to have a better performance and output than the two other methods stated. The feedback edge set is computed only once in the current step and stored in memory but the individual edges contained will be more often accessed in later steps.

We further analysed the latter one and recognized that we can effectively use this heuristic for our purposes. The pseudo code of this heuristic is shown in Figure 4-1.

while G is not empty do
    while G contains a sink v do
        store v in right list
        delete v in G
    while G contains a source v do
        store v in left list
        delete v in G
    while G is not empty do
        choose vertex v with max value
        store feedback edges
        store v in left list
        delete v in G

Figure 4-1: Pseudo code for advanced Greedy heuristic

The algorithm described in pseudo code works as follows. All vertices in the given graph are only visited once. For this purpose we first clone the graph on which we perform this action before we continue to compute the node order and the feedback edge set. This is necessary as we have to delete nodes and edges in the cloned graph. The node order is not needed for computing the feedback edge set but used for the method described in Chapter 4.2.3 and can easily be computed with this algorithm. As long as we have unvisited nodes in graph G, we continue to search for sinks, sources or nodes which do not belong to the first ones. A source is a node which has only outgoing edges, whereas a sink has only incoming edges. If a node has neither of them, we took one where its out degree minus in degree was highest. The out degree is the number of outgoing edges and the in degree the number of incoming edges, respectively. If this node has incoming edges, all of them are feedback edges. Incoming edges are feedback edges due to the fact that the path on which they are positioned would have been
removed otherwise by previous actions. If the path contains sinks, the nodes would have to be removed before this node is chosen. After processing one of the nodes, which is neither a sink nor a source, we continue with a new run of the outer loop if there are still unvisited nodes in G. Once a node that is neither a sink nor a source is removed, it might happen that we produce new sinks and/or sources by deleting this node and its corresponding edges in the graph. After processing any kind of node, we delete it from the graph as we visit it. Consequently, we need to start looking for new sinks and sources. If there are isolated nodes in the graph, we remove them as we treat them as sinks and add them to the right list.

The nodes contained in graph G are stored in two lists for processing the node order. One list is marked as left list and all sources are stored here. The other list is marked as right one and all sinks are stored here, inclusively isolated nodes. The aim of these two lists is to build the order of nodes where the number of edges directing from the right to the left is lowest according to the minimum feedback arc set problem. By simply concatenating the two lists after all nodes have been visited, we get the final node order.

The computed node order is not unique. For the same graph there may be a number of possible node orders. This is due to the fact that nodes may be determined with out degree minus in degree and there might be more nodes with the same value but only one node can be handled at the same time.

Now we computed all feedback edges and we got a node order with minimum feedback edges. All feedback edges are stored in memory in their original direction. This is very important for later processes. The edges are still stored in the original graph in their original order.

The Figure 4-2 shows a graph which is cyclic. On the right side the matching acyclic graph is shown where the feedback edges are dashed. For the given graph the feedback edges are (4 $\rightarrow$ 2) and (3 $\rightarrow$ 1). At the bottom the computed node order is shown.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure42.png}
\caption{Making a graph acyclic}
\end{figure}

\section*{4.2.3 Longest Path Layering}

Once we know which edges are in the directed acyclic graph, we can begin to form a proper hierarchy. This means that we establish different layers and place in the nodes. The layer in which each node fits is depending on its relations to other nodes. For doing this we have to take care of the following readability criteria: uniform distribution and smallest edge length. First, we try to distribute all nodes uniformly to all layers but this greatly relies on the graph itself and its relations. For most graphs this criterion cannot be fulfilled with the algorithm we chose but this is not a mandatory goal addressed by this thesis. The criteria for smallest edge length can always be met. From the start on,
we place all nodes in the top layer. We move those nodes, which have incoming edges from other nodes in the same layer, one layer down. Successively this is done for the next layer until we cannot move any nodes to layers further down. All nodes are layered according to their longest path.

while unvisited node in list do
  access node
  while node has unvisited outgoing edges do
    get edge
    get target of current edge
    if $\lambda$ of target needs to be updated do
      update target node
  for all feedback edges
    if node is target node of feedback edge do
      if $\lambda$ of target needs to be updated do
        update target node

Figure 4-3: Pseudo code for longest path layering

The algorithm we implemented for this step is the Longest Path Layering Algorithm stated in Sugiyama (2002). The approach is the same but we had to modify the algorithm in a way so it can handle feedback edges. First, we need the node order computed in Chapter 4.2.2 as input for this algorithm. This node order also follows the criteria for a topological sort. For every node in the given order a value $\lambda$ is set. $\lambda$ indicates the layer to which the node shall be assigned to. Initially it is set to one for all nodes as every node is in the top layer. Now we access all nodes in the order given by the set. We take a node and follow all its outgoing edges. For every single edge we check whether the target of this edge has a value greater than the one of the current node. If this is not true, we update $\lambda$ of the target node by the value of the current node plus one. If the value is greater, we do not update the target node as it is already on a layer further down than we wanted to. In this case, another path is already assigned to the node in this layer. We repeat this procedure until we handled all nodes. Figure 4-3 shows the pseudo code of the developed algorithm. The number of layers is also determined in this algorithm.

When we trace the outgoing edges of nodes, we cannot simply take those and follow them. As shown in the chapter before we computed a feedback edge set. An outgoing edge can be in the feedback edge set. If we got that edge, we have to stop processing this edge as in the new directed acyclic graph the feedback edge is reversed. This means that it is no outgoing edge anymore. Furthermore, for every node we handle, we not only have to test its outgoing edges as already described. We also have to test the edges in the feedback edge set in reversed order and update the value $\lambda$ for layering the current node.

Figure 4-4 outlines the result of longest path layering carried out on the graph.
presented in Chapter 4.2.2. At the top of the figure we can see the order of nodes (row I) together with normal edges and feedback edges, which are dashed. In row II, we first set the value indicating the layer of each node to one for the top layer. Then we compute the value for each node by performing the presented algorithm. The result is shown in row III.

The current development and implementation also checks all feedback edges and updates lambda if necessary. We developed and implemented this step for safety reasons in order not to compute the layers wrongly. This part can further be investigated to find out if it can possibly be skipped to reduce computation time (see Chapter 7.2).

4.2.4 Creating Layers

The hierarchy we need to set up must be a proper one. The word proper indicates that the hierarchy must meet certain requirements. First, layers are established. This was already done in the previous Chapter. Secondly, the span of edges shall always be one. We explain this further in the next Chapter.

Here, we show how to initially set up the hierarchy and then further process it until we meet the criteria for establishing layers in the hierarchy. The individual layers were already computed in the last chapter. Now we can store each node in its layer. We simply access the node and read its value $\lambda$. Afterwards we use an object of type “ArrayList” for each layer to store nodes belonging to it. All layers are stored in an array of “ArrayList” in the order from top to bottom. Figure 4-5 shows the pseudo code for “Creating Layers”.

```
set up data structure for hierarchy
while unvisited node in list do
    access node
    get lambda of node
    put node in layer
```

Figure 4-5: Pseudo code for creating layers

By taking the value $\lambda$, the integer value shows the layer where the node is placed. The overall height of all layers, the highest number of $\lambda$, is as long as the longest path. As mentioned already in Chapter 4.2.3 the width of hierarchy is not set to a fixed number (uniform distribution). We allow any number of nodes in all layers.

In Figure 4-6 we show how the graph presented in Chapter 4.2.2 looks like after layering. On the left side the numbers of the layers are shown and to the right individual nodes belonging to single layers are printed.

Figure 4-6: Creating layers
The data structure in which single layers with their belonging nodes are stored represents the initially given graph. This structure always contains this information as long as we compute the final layout. Therefore, it is necessary to access and update this data structure several times while other methods of the layout algorithm are performed, e.g. crossing minimization. At the end of the overall algorithm this structure holds the final layout representation of the graph.

4.2.5 Removing Long-span Edges

In the hierarchy shown in Figure 4-6 we can see that not all edges have an edge span of one. The edge span is defined as number of layers between the source node and target node of an edge. As the figure shows, all edges have an edge span of one except the edge from node a to node f. This edge has a span of three. Long-span edges must be removed for achieving a proper hierarchy as explained in the beginning of Chapter 4.2.4. The proper hierarchy is necessary for crossing minimization. This process can only be run when the edge span is always one.

Long-span edges are removed by taking them out of the graph. They are replaced with several dummy nodes and edges depending on their span. In theory the process is as follows: Assuming that there is an edge (a → f) with a span of four, thus it must be replaced. First, we remove the edge (a → f) and replace it with a dummy edge (a → d1). d1 is a dummy node. As the span is three we have to insert another dummy edge as well as another dummy node. We built the edge (d1 → d2) where the source and the target are dummy nodes. Finally, we can connect the dummy node d2 to f and insert the following edge (d2 → f). For a span of n we have to insert n new dummy edges and 1 new dummy nodes. Figure 4-7 shows the original edge with a span of three and its replacement. Each new dummy edge has a span of one.

Replacing the long-span edges is a straightforward process but it also has its limitations. We shall insert as less dummy vertices as possible due to the following statements (Sugiyama, 2002):

- The computation time for crossing minimization is strongly dependent on the number of dummy vertices inserted. For every dummy vertex inserted the minimization process must handle this vertex several times in the worst case. The higher the number of dummy vertices the higher the computation time needed for the process.
- In the final drawings bends will only occur at dummy vertices. This is due to the fact that all edges shall be straight edges if possible but this cannot always be ensured.

![Figure 4-7: A long-span edge](image)
• For the human eyes it is much easier to follow short edges rather than long edges. This statement is contrary to the first one. No inserting of dummy vertices brings higher computation time but crossing minimization cannot be done. Inserting dummy nodes brings higher recognition rates for human viewers and this is exactly what shall be achieved by this thesis.

The best practice of taking care of all three mentioned criteria is to insert only a certain number of dummy vertices and edges. Precisely, the number of dummy vertices for a long-span edge is the span of this edge minus one.

In our program we detect long-span edges by processing all outgoing edges of a node. All layers are taken from top to bottom and all containing nodes are accessed. We take the outgoing edges and test each edge. Is the current edge in the feedback edge set, we do not process this edge any further since this edge is reversed in the acyclic graph. The test must be performed as the original graph is not modified, so all feedback edges are only stored in the feedback edge set in memory. If the edge must be handled, we check if the edge span is greater than one. Only in this case the current edge must be processed. We create a new dummy node and label the node with the “dummyX” where X stands for an integer which is the current number of dummy nodes plus one. The node is of the same object as normal nodes but labelled especially as described. The node is inserted into the layer one further down than the source node of the original edge. Afterwards we check if we need to insert more dummy nodes and repeat the same procedure until we can reach the target node with an edge span of one.

Since we only process outgoing edges in the above described algorithm, we do not handle all cases where long-span edges can occur. Those can appear also for feedback edges which we did not handle so far. We have to check all feedback edges for their span and remove them if needed. Before we can actually process the edge we have to reverse the edge. Feedback edges are stored in their original direction. The same procedure as for normal edges will be used. As there might be a possible improvement for handling feedback edges we decided to handle feedback edges separately to improve future work. Due to the complexity and limitations of this thesis we did not further investigate this (see Chapter 7.2). Figure 4-8 shows the pseudo code for the algorithm explained.

```
for all layers do
  for all nodes in layer do
    get node
    while node has unvisited outgoing edge do
      if edge is not feedback edge
        if span greater one do
          while span greater one do
            if span == 2 do
              insert one dummy node
            else do
              insert as many dummy nodes as needed
          end
        end
      end
    end
  end
for all feedback edges do
  reverse edge
  if span greater one do
    while span greater one do
      if span equal two do
        insert one dummy node
      else do
        insert as many dummy nodes as needed
    end
  end
```

Figure 4-8: Pseudo code for removing long-span edges
Original edges, which are replaced by a path built of dummy edges and nodes, are stored together with their “new” closest dummy node in a new data structure. This allows us to easily test whether this edge was originally a long-span edge. We can also determine which node is the next one of the long-span edge and later trace the path further down. The number of entries in the data structure is equal to the number of long-span edges. The created dummy nodes will be placed in their correct layer and stored in the same data structure as the normal nodes. For the created dummy edges, which connect a dummy node to another dummy node or to the target of the original long-span edge, we create another data structure. Its size is equal to the number of dummy nodes inserted in the whole graph (see Figure 4-9). With the help of the different data structures we can clearly distinguish between feedback edges and dummy edges. Dummy edges have the following characteristics: They always have only one incoming and only one outgoing edge. Furthermore, their edge span is always one (see Figure 4-9). Dummy nodes can never be at the end of one path, meaning at the end position of a long-span edge.

Figure 4-9 shows two long-span edges and their replacements by edges with edge spans of one. The first long-span edge has a span of two and the other one of three. We also present the two data structures and what stored elements look like.

\[ (((1,3),d1), (1,4),d2) \]

\[ (((1,3),d1), (1,4),d2) \]

\[ ((d1,3), (d2,4)) \]

\[ ((d1,3), (d2,4)) \]

**Figure 4-9: Removing of long-span edges**

### 4.2.6 Incidence Matrices

Incidence matrices are representing interconnections between nodes of two layers. They must be established for crossing minimization. For getting the correct relations all edges of all nodes must be considered. This means that we have to check normal edges, feedback edges, and dummy edges as well as to ensure correct matrices for two layers at a time. Having all interconnections stored in the matrices allows crossing minimization between two layers.

For setting up one matrix we need to get information of two adjacent layers. The upper layer is represented by rows in the matrix and the lower layer by columns, respectively. Every node builds one row or a column dependent on its layer. The nodes are stored in the order which is given by the layer data structure (see Chapter 4.2.4). This is very important as we always need exact information about all layers. The matrix itself is set up by integer values. Each integer value can only be zero or one dependent on the fact whether there is an edge from the node in the corresponding row to the node placed in the column. The integer value will be one if there is an edge as described and
zero otherwise. For the mathematical expressions we refer to Sugiyama (2002). In Figure 4-10 three layers and the corresponding matrix realizations are shown. The first matrix shows relations between layer one and two, the second matrix between layer two and layer three.

```
for layers – 1 do
    create and initialize matrix
    for all nodes in upper layer do
        if node is dummy node do
            get new target from memory
            update matrix
        else do
            while node has outgoing edges do
                if feedback
                    stop;
                if long-span && not stop
                    get target from memory
                    update matrix
                if not stop
                    get target from edge
                    update matrix
            for all feedback edges do
                if feedback edge was fragmented
                    reverse edge
                determine source and target
                if source in current layer
                    update matrix
                store matrix for this layer
```

Figure 4-10: Establishing incidence matrices

Figure 4-11: Pseudo code for incidence matrices
first new target of long-span edges. In the case that all these criteria do not meet, the
current edge must be a normal edge and we can easily process it and write the value into
the incidence matrix. After having processed all normal outgoing edges, we have to test
all feedback edges since they can also be long-span edges. We check if the feedback
edge is a long-span edge. If the test is negative, we do not need to handle this edge as it
must have been handled already by the corresponding normal edge. If the test is
positive, we get the new target node from the data structure in memory and update the
matrix accordingly. Figure 4-11 shows the pseudo code of this algorithm and shall help
to better understand the procedure.

4.2.7 Crossing minimization

In this chapter we handle the process of minimizing crossings between edges of
different nodes between two layers. As this procedure was one of the most complex
ones to handle we will explain it in detail and also introduce other procedures, which are
necessary to perform this task. In the beginning of this chapter we describe the
algorithm from a more abstract point of view to give an overall impression. Later on we
focus on each single function and provide detailed descriptions.

The intention of crossing minimization is to achieve an order of nodes in each layer
where the edge crossings are reduced as much as possible. By simply reordering the
nodes in each layer we compulsory find an order with the criteria stated. The number of
crossings is not dependent on the exact position of vertices but instead on the order of
the nodes in each layer. Quite obviously we simply have to test all possible
permutations of a layer and we find one with the least crossings. The problem then is a
combinatorial one and greatly simplified. However, this problem is not necessarily easy
and in reality a quite difficult one. It is known to be NP-complete (Garey & Johnson,
1983). This shows that we cannot easily compute the order of nodes even if there are
only two layers. Thus, we must implement an efficient heuristic instead. The one we
will use and implement is the barycentre method described in Sugiyama, Shojiro, and
Mitsuhiko (1981). This method relies on the fact that each vertex is best placed near its
neighbouring vertices. In Eades and Kelly (1986) it has been shown that this heuristic is
effective.

Sugiyama, Shojiro, and Mitsuhiko (1981) introduced a method to reduce the number
of crossings in a 2-layer graph using the barycentre method. We will shortly focus on
this method and then expand this approach to a $n$-layer graph. In Chapter 4.2.6 we have
seen that we can establish a matrix realization for two layers showing the relations
between these two layers. This information is also used for crossing minimization. First,
we use the matrix realization to compute the number of crossing between these two
layers. Secondly, the barycentres of rows/columns must be computed. Thirdly, we
define two phases for crossing minimization, namely **phase1** and **phase2**. All these
methods are further explained later. The main idea of the algorithm is to arrange all
barycentres to a monotonically increasing order not dependent of rows or columns. For
both phases we introduce two directions as well: up and down. When going down we
fix the upper layer (rows in matrix are fixed) and re-arrange the lower one, respectively
we fix the lower one (columns in matrix are fixed) and reverse the upper one when
going up. This is true for both phases. In every step we perform actions to achieve the
barycentre order already explained. **phase1** performs barycentre ordering in the down
direction (for columns) as well as in the up direction (for rows). **phase2** can only run if
the barycentres are sorted properly. It checks if there are units of equal barycentres and
reverses nodes of equal barycentres with each other. When the algorithm starts, **phase1**
down is performed first. Immediately afterwards **phase2** down is called and reverses a
pair of nodes if possible. Before doing so the barycentre order is checked. Here, **phase2**
calls phase1 with the same direction which rearranges the barycentres accordingly. After running of phase1, phase2 runs again. phase2 alternates between down and up until we reach a stop criterion. This can be zero edge crossings in the graph or a number of overall iterations. A reduction of edge crossings between the given two layers can be achieved by repeatedly alternating row and column barycentre ordering as described. This is the main idea behind this algorithm. A short example of pseudo code for the 2-layer graph is shown in Figure 4-12.

```
phase1 down
while not stop do
    if possible
        phase2 down
    if needed
        phase1 down
    if possible
        phase2 up
    if needed
        phase1 up
```

Figure 4-12: Crossing minimization 2-layer

As a 2-layer graph is not always given, we need to extend this scheme for the functionality to handle any $n$-layer graphs as well. In Sugiyama, Shojiro, & Mitsuhiko (1981) they state that reducing crossings in $n$-layer graphs can be computed and minimized separately. This means that we can reduce crossings between two layers not dependent on other layers if we use a special scheme. If we reverse the node order in one layer, always a second layer is changed (if adjacent to this layer) and obviously dependent on the action taking place. To explain the dependency we have to look at the crossings of this layer. As explained before, changing the node order of a layer may change the number of crossings to the adjacent layer. This exactly occurs between the second and third layer. While we developed the algorithm we found out that the number of crossings (second and third layer) can also increase if the node order is changed. Right now we introduce a scheme with which we perform crossing minimization for multi-layered graphs. Having seen that different layers cannot be handled separately, we handle them sequentially. We already introduced directions for the two phases and explained its application. For all layers one direction (down or up) is executed before the direction is changed. This avoids interferences we see in Figure 4-13. The figure shows that the crossings between layer two and three increases while we get rid of the crossing between layer one and two. Working on the layers from top to bottom (down-procedure) the crossings in these layers are reduced. This is also true for the up-procedure.

![Figure 4-13: Crossing minimization phase1 down](image)

The pseudo-code for handling $n$-layer graphs is now introduced and further explained. We start with one run of phase1 at the top layer in the down direction till we
reach the bottom layer. The overall procedure for crossing minimization performs consequently always phase2 and changes the direction. Phase1 is a sub-algorithm of phase2 and called if barycentre reordering needs to take place for a specific layer and direction. Both phases are performed in the direction till they reach the end layer. If phase2 calls phase1 for barycentre reordering, phase1 does so and consequently phase2 is called again for this layer. If there are nodes with equal barycentres, we reverse them otherwise we move to the next layer in the specified direction. Figure 4-15 describes how the two phases interact with each other. Phase1 down is called once at the start of the overall procedure (column 1). Afterwards phase2 runs continuously and calls phase1 if needed. Within one execution of phase2 for one direction we can have several calls for phase1, which is shown in columns five to nine. Phase1 is called twice and phase2 runs immediately for the same layer if phase1 ended. Figure 4-14 presents the different calls for the two phases graphically. The arrows indicate the direction of the phases.

For performing the two phases we need to introduce some other functions first. For computing the nodes that shall be reversed we have to compute their row/column barycentre. Consequently we decide which nodes are exchanged with each other for achieving a barycentre ordering or reverse nodes with equal barycentres, respectively. The barycentres are computed as stated in Sugiyama, Shojiro, & Mitsuhiko (1981). The computation needs to be done more often and therefore we introduced two methods to do this: one for row barycentres and one for columns. In order to see if our algorithm reduced the number of crossings between two layers it must use the corresponding method. Once the node order in one layer changes the layer information (node order, incidence matrix as well as number of crossings) must be stored in memory. Furthermore, we have to update this information for a second layer if needed. In Figure 4-17 the incidence matrices are shown for each step. In the first step matrix one changes but also the second matrix is influenced by the performed step and must be changed.

```
compute crossings of all layers
phase1(down, top)
while not stop do
  while not stop2 do
    phase2(direction, layer)
    check if phase1 is needed
    if phase1 not needed
      set stop2
    else
      phase1(direction, layer)
    while not stop do
      change direction
      if needed
        restore configuration
```

Figure 4-15: Pseudo code for crossing minimization
Now phase1 is presented in detail. As already described it takes the computed barycentres and reorders the node order of one layer. This layer is dependent on the direction chosen and the layer from which we start executing phase1. In order to understand the process even better the pseudo code for this is shown in Figure 4-16. First of all we check if we reached our stop criteria. These are explained and defined later on. If we have reached them, we stop all further processing and return back to the overall algorithm. The normal step is to check the direction which shall be used. This is necessary due to the fact that the barycentres are computed differently depending on the given direction. The figure to the right shows the algorithm for direction \textit{down}. For direction \textit{up} we do not show a pseudo code as the main computations are the same. Only the barycentre computing, stop criteria for ending phase1, and the layer information are computed differently. The underlying structure remains the same. Phase1 reorders all layers being on the given direction. It ends when the bottom layer is reached. The layer information is fetched out of memory and stored locally. The barycentres are computed and reordering is done until we achieve a barycentre ordering. Under some special circumstances we do not reorder nodes even if a barycentre order is not achieved. If it cannot be guaranteed that the number of crossings is reduced by this step, we do not reorder (see Chapter 4.3). Once reordering is started, the positions to change are determined depending on the barycentres. We implemented two different algorithms for position determination. One is initially taken and two nodes determined for reordering. In each step we exchange two nodes with each other. The incident matrix of the current layers is changed in memory and the new number of crossings computed. For this step a number less than originally stored is expected with a high probability. However, for few situations this number is higher than before. Here, a further

\begin{figure}[h]
\centering
\begin{tabular}{ccc}
3 & 4 & 4 & 3 & 4 & 3 \\
1 & 1 & 0 & 1 & 1 & 0 \\
2 & 0 & 1 & 2 & 0 & 1 \\
\hline
5 & 6 & 5 & 6 & 6 & 5 \\
3 & 1 & 0 & 4 & 0 & 1 \\
4 & 0 & 1 & 3 & 1 & 0 \\
\end{tabular}
\caption{Incidence matrices general}
\end{figure}

\begin{figure}[h]
\centering
\texttt{check stop criteria}
\texttt{if direction==down}
\hspace{1em}
\texttt{while not at bottom layer do}
\hspace{2em}
\texttt{get current layer information}
\hspace{3em}
\texttt{barycentres computation}
\hspace{4em}
\texttt{while not ordered}
\hspace{5em}
\texttt{position determination}
\hspace{6em}
\texttt{change positions}
\hspace{7em}
\texttt{compute new crossings}
\hspace{8em}
\texttt{if new crossings < crossings}
\hspace{9em}
\texttt{update layer information}
\hspace{10em}
\texttt{if second matrix exists}
\hspace{11em}
\texttt{update layer information}
\hspace{12em}
\texttt{barycentres computation}
\hspace{13em}
\texttt{check stop criteria}
\hspace{14em}
\texttt{layer++}
\hspace{15em}
\texttt{else}
\hspace{16em}
\texttt{“do the same for direction UP”}
\end{figure}
minimization of edge crossings is tried by choosing the second algorithm for position
determination. This algorithm may choose two different nodes. The node order is only
rearranged for less crossings and the corresponding layer information updated. Now we
check if another matrix realization must also be changed as a neighbouring one. If so,
we update it accordingly. Now the barycentre order is computed again and if needed,
the actions are repeated. After one layer is sorted monotonically increasing, the
algorithm switches to the next layer but before the stop criteria are checked again. This
is necessary as further processing shall be skipped if we achieved zero crossings in the
graph. In Figure 4-18 the matrix realizations of Figure 4-13 are taken and the same steps
are performed as described earlier. The column barycentres are shown and the nodes are
reversed accordingly. In the second column both matrices changed as result of the first
reordering. In column three only the second matrix is changed.

```
3 4 4 3
1 1 0 1 1 0
2 0 1 2 0 1
BC 2 1 1 2
```

```
5 6 5 6 6 5
3 1 0 4 0 1 4 1 0
4 0 1 3 1 0 3 0 1
BC 1 2 2 1 1 2
```

Figure 4-18: Incidence matrices with barycentres

For reversing nodes with equal barycentres phase2 is introduced. Those nodes can be
exchanged since the monotonically increasing order is not violated by this. While the
algorithm was developed, we defined that only one exchange can be done for one layer.
Phase1 is called if needed or phase2 continues with the next layer. In Figure 4-19 a
sample of pseudo code is shown. Once two positions are determined for exchanging the
number of crossings is computed. Also positions are only reversed if a reduction of
crossings is achieved. Otherwise it is searched for another pair which can be exchanged.
If this is not possible, this layer is skipped and continued with the next one. In the else-

```python
check stop criteria
if direction==down
    while not at bottom layer do
        get current layer information
        barycentres computation
        if not ordered
            call phase1
        else
            while not stop
                position determination
                change positions
                compute new crossings
                if new crossings < crossings
                    update layer information
                    if second matrix exists
                        update layer information
                    if not ordered
                        call phase1
                check stop criteria
            layer++
        else
            "do the same for direction UP"
```

Figure 4-19: Pseudo code for phase2
branch the same steps are performed but for direction up. Congruent to the else-branch in phase1 also phase2 must use different layer information when direction up is performed. The overall usage of exchanging nodes with equal barycentres is to further minimize the number of crossings. Here, we can get a benefit of reversing these nodes. Also reference literature states that this step is shown to be effective. Figure 4-20 presents matrices of a graph where phase2 is performed upwards. The nodes in the lower matrix are changed which influences also the upper matrix. The figure only shows two executions of phase2. The upper matrix is not changed at all as there are no equal barycentres.

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& 3 & 4 & \text{BC} & & 4 & 3 & \text{BC} \\
1 & 1 & 1 & 1.5 & 1 & 1 & 1.5 \\
2 & 0 & 1 & 2 & 2 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
& 5 & 6 & & & 5 & 6 \\
3 & 1 & 1 & 1.5 & 4 & 1 & 1 & 1.5 \\
4 & 1 & 1 & 1.5 & 3 & 1 & 1 & 1.5 \\
\end{array}
\]

Figure 4-20: Incidence matrices phase2 up

The overall algorithm described runs without any interruption if we do not state any criteria for stopping the algorithm. Those are necessary to be able to interrupt the process if certain requirements are already met. While developing the algorithm we defined the following stop criteria:
- Number of crossings equals zero
- Maximum number of iterations reached
- Maximum computation time reached

When computing the number of crossings the goal is to reach total crossings of zero. Once we reached them, we cannot further optimize crossings because there are none. Thus, we immediately skip further processing for crossing minimization and continue with the next steps. Another criterion is the maximum number of iterations. For larger graphs it is unlikely that we achieve zero crossings. Continuously phase1 and phase2 are performed although we never reach the first goal. As this would cause an infinite loop we interrupt the algorithm when we reach the criterion maximum number of iterations. The number can be set statically when an instance of the Sugiyama algorithm is built (see Chapter 7.2). For larger graphs the computation time needed can be large so we decided to skip processing after a certain amount of time. Then the algorithm runs to a position where it is optimal to stop even if the number of crossings can probably be further optimized. The time needed for computing graphs with about 60-100 nodes can be estimated to take one to three hours (or even more) for achieving the minimum of crossings. If the algorithm is stopped earlier, we probably do not get the best output but we may save a lot of computation time (see Chapter 7.2). In the end computing less crossings gets more and more expensive with respect to computing time. This feature was not further investigated in this thesis.

4.2.8 Layout positioning

After crossing minimization is done we can lay out the computed graph with regard to the readability criteria stated in Chapter 3.1. This method computes the placement of nodes within the corresponding layers. All nodes are still layered without using their UML information. Now, not only normal nodes must be layered but also dummy nodes which are necessary to divide long-span edges. The following readability criteria are used: least separation, closeness, balanced layout, and straightness of edges.
The computation of such a layout is not easy and computationally expensive. Sugiyama, Shojiro, and Mitsuhiko (1981) introduced an efficient heuristic to solve this problem. They use priorities which are assigned to nodes before computing the layout. Therefore, their method is called priority layout method. With the help of this method the computing cost can be further reduced. The basic idea is similar to the barycentre method used for crossing minimization. The nodes are reordered according to their barycentre while the nodes are placed according their priority from highest to lowest priority.

Before we can actually place the nodes, new incidence matrices are established as also dummy vertices must be placed. All layers are extended to the amount of nodes the largest layer has. This is necessary to be able to place nodes closest to their neighbours. Empty places in each layer, meaning places where no normal or dummy node is located, are filled with empty nodes so places where nodes can be placed are clearly marked. All incident matrices must be expanded and the old matrices mapped into the new ones so we do not have to compute all edges again. Mapping the old matrices is quite easy. We simply copy all values to the new matrix starting from the top left corner. For placing the nodes a similar scheme as for crossing minimization is introduced. The algorithm also uses two directions: down and up. The problem of placing nodes is separated to smaller ones for each layer. The approach is to go from top to bottom and place all nodes in lower layers iteratively. The upper layers are replaced when the direction is upwards. The method performs two times down and once up. Reference literature states that this ensures a good layout. Figure 4-21 shows the pseudo code for the main algorithm for layout positioning.

We place the nodes according to their upper (downwards) or lower (upwards) barycentres. The method to compute the barycentres is similar to the one for crossing minimization. The exact formulas are described in Sugiyama, Shojiro, and Mitsuhiko (1981). This information is computed for one layer and further used to compute the node priorities. The priorities are sorted from highest barycentres to lowest one. However, dummy nodes must have even higher priorities than the node with the highest barycentres. This is due to the fact that we want to achieve straight lines especially for long edges. Dummy nodes were created to replace long-span edges. The highest barycentre of all normal nodes is computed and added as offset to the barycentre of each dummy node. This ensures that each dummy node has a higher priority than normal nodes and sorted according to their own barycentres, too. Afterwards the nodes are placed as close as possible to their neighbours by always taking the one with the highest priority. Once this node is set, its position is fixed and it cannot be moved by nodes with lower priorities. When the nodes are put to their positions, the node order computed in crossing minimization must remain the same. No overtaking is allowed. In order to exemplify the mode of operation the pseudo code for the algorithm is shown in Figure 4-23. Operating on one layer is necessary as the barycentres for the next layer may change if the positions of nodes change. If there are nodes with equal priorities, the first one in the data structure is chosen and set. Figure 4-22 presents a graph to the left side where the nodes are not positioned. On the right side for the same graph layout
while not at bottom layer do
    compute barycentres and connectivities
    compute offset
    compute priorities
for all nodes in layer do
    compute best position
    compute direction
    if left
        check next empty position
        try to move node leftwards as close as possible to best position
    else
        check next empty position
        try to move node rightwards as close as possible to best position
    layer++

Figure 4-23: Pseudo code for layout positioning down

positioning was done. The top node is placed at its barycentre and the nodes in the bottom layer as well.

The algorithm for layout positioning upwards is the same with one exception. If the method starts from the top, nodes in the next layer must always have at least one edge showing back to the upper layer. This must be true for all nodes otherwise the criteria for a proper hierarchy are not met or the placement was computed wrongly. Once the algorithm starts from the bottom layer, nodes in the upper layer must not have connections to nodes in the lower layer. This results in the fact that their barycentre would be null if no special handling would take place. For those nodes the barycentre is set to the place it had before. They had to be handled in one run downwards and put as close as possible to their neighbours. This is the only difference between phases up and down.

4.2.9 Computing position information
All nodes are now in a canvas with x and y information but so far no UML information was taken into account. The vertices could easily be placed in a layout. This would not ensure that overlapping of classes does not occur. Therefore, the UML information is used to compute the height and the width of all nodes. Afterwards we expand the canvas and place all nodes with correct layout information.

For each node its label is parsed for getting all UML information. The node name, its included attributes, and methods are stored in the html-tag and parsed in order to retrieve them. While doing so, both the number of lines (y-information) and the longest string value (x-information) are stored. In x-direction any name, attribute or method can be the longest, so we must parse all information to get the correct values. We investigated the html-tag and its structure before we developed a method to parse the
tag. The method must handle all situations of different attributes which might occur, e.g. no attributes at all or no methods. In the requirements we stated that the label tag must follow certain specifications. If this is not fulfilled, we cannot parse the label correctly and throw an exception within the program. Then we know for sure that the given input was not correct and we cannot compute the layout correctly. The pseudo code sample is shown in Figure 4-24.

Label parsing is called for all nodes while layout computing takes place. The

```
extract label out of html-tag
compute width
compute height
for all attributes do
  extract string for attribute
  compute height
  compute width
for all methods do
  compute height
  compute width
```

Figure 4-24: Pseudo code for label parsing

“GraphProperties” of each single node are updated with the correct height and width. For each row and each column the maximum height and width is also stored as needed later for computing the gap between columns and rows. Now the exact layout information is fully known and all nodes can be placed in the canvas so no overlapping is possible. All graph information is stored then. To ensure space between classes in both directions two offsets were introduced and set as global variables. Figure 4-25 shows the algorithm for putting the nodes into the canvas. A problem for label parsing

```
initializing variables for boundaries
for all layers do
  for all places in layer do
    if normal node
      parseLabel
      set height
      set width
    for max nodes in layers do
      for all layers do
        if normal node
          set X
          set Y
```

Figure 4-25: Pseudo code for layout positioning

is that we can only compute the length of the string we parse. This is equal to the number of characters in the string. A visualization tool must render the tag with a specific font and display it. This length must not necessarily coincide with the length we computed. The same phenomenon occurs for the height of single classes. For small classes, meaning small height and/or width, the boundaries of nodes can be computed fairly well. The higher or wider the object is the greater the problem of computing their real boundaries. This situation cannot be handled well within the layout algorithm as long as it has to compute the height and the width. A solution to this problem will be stated as future work in Chapter 7.2. At the moment the layout algorithm approximates the height and width of all nodes fairly well but for some graphs this has to be adjusted. In order to adjust this, yEd can be used as this tool can fit the label to nodes. This functionality will further be described in Appendix A.

While we developed the algorithm for layout positioning, a bug in the existing GRAIL plug-in was found. Whenever a graph is stored to file all labels are shortened to
a maximum of 30 characters if needed. Following, the graph information is computed within GRAIL based on the shortened label. Consequently, the position information is wrongly stored in the output file. In order to avoid this fact, the algorithm was investigated to find out how the information can be passed on so the output will be correct. The source code of GRAIL was modified so the full label is output for each node. The node information is still computed wrongly but correcting this code was not within the scope of this thesis. In the current algorithm the width of each node is specially computed so GRAIL stores this information correctly later on. Once the bug in GRAIL is fixed, the algorithm for layout positioning must be modified as well.

4.2.10 Displaying result of layout computing
The result of the previous steps is written back to graph which was initially given. In order to test the computed layout a test class took the graph and stored it in a gml-file. This file can be opened with yEd and the graph with its layout information is shown. With the help of this tool we check the computed layout.

4.3 ADDITIONAL FEATURES
In this chapter additional features to the layout algorithm are introduced which were not described in reference literature. Some features were developed after parts of the algorithm were already implemented. Also important modifications with respect to the algorithm stated in Sugiyama (2002) and Sugiyama, Shojiro, and Mitsuhiko (1981) are presented.

In reference literature nodes are exchanged even so the number of crossings does not decrease in phase1 or phase2 in crossing minimization. After investigating we concluded that nodes are only exchanged with each other if the number of crossings decreases. Only then is it useful to perform actions. Otherwise we skip exchanging and continue. This is contrary to literature. The algorithm stated there exchanges nodes and the number of crossings are not taken into account at all.

For phase1 and phase2 it is not stated how we determine nodes which must be reordered. The position for these nodes can be determined in many different ways. Neither is it not stated whether only two nodes have to be reversed each time or if the complete matrix has to be re-arranged. While developing we focused on exchanging only two nodes at a time as we had to define this. Furthermore, we implemented different algorithms for position determination to perform crossing minimization.

In phase2 nodes with equal barycentres are exchanged with each other. In literature it is not stated how often we have to exchange nodes in one layer if there are several nodes with equal barycentres. The algorithm developed only exchanges one pair of nodes and continues with other actions, either calling phase1 or moving to another layer.

Sugiyama (2002) presents a maximum number of iterations as stop criterion. After a specified number of iterations is reached further processing is skipped. An extended scheme was developed and implemented to the algorithm. The iterations of phase1 and phase2 are stored separately and the sum is compared to the maximum number of iterations. If that number is reached, we have to skip processing at specific points of the algorithm. As testing continued the time needed for computing larger graphs increased while more and more complex graphs were tested. For a graph with 100 nodes the computing time is estimated to one hour up to three hours. Another stop criterion is now given by an amount of time for which the algorithm may reduce the number of crossings. If this criterion is reached, the process is skipped at certain points and ended.

Another interesting finding was that the total number of crossings was sometimes dependent on the maximum number of iterations. By testing a graph with 37 nodes and
50 iterations, the final number of crossings was larger than compared to 30 iterations. This relies on the fact that exchanging equal barycentres or achieving a barycentre ordering in one layer can disrupt another layer and increase the number of crossings accordingly. A scheme for handling this fact was needed and developed. At the beginning of crossing minimization the current configuration (layer information and total number of crossings) is stored. Whenever the number of crossings falls below that threshold, which is always updated to the least crossings, the current configuration is stored again and kept in memory. In a situation where the crossings increase we do not update the configuration. This procedure achieves to always have the configuration with the least crossings in memory and that it will always be accessible. At the end of crossing minimization we checked whether the current situation or the stored configuration has the least crossings. If needed, the configuration can be retrieved for further layout processing.
5 DESIGN & IMPLEMENTATION

In the previous chapter the whole development for the layout algorithm was presented and all steps introduced. This chapter will show important decisions which influence the actual implementation by using the results achieved in Chapter 4.

5.1 DIVIDE AND CONQUER

After having studied reference literature we got to know that the overall algorithm is quite complex. The basic approach was to have a divide and conquer strategy. The algorithm was divided into smaller parts whenever possible which can be handled much more efficient and easier than a large algorithms. Methods were already developed and split up into several ones in Chapter 4 and prepared for implementing. The huge problem of computing the layout is divided into sub-problems which enables an easier solution.

Divide and conquer was already used for the Sugiyama algorithm in the method crossing minimization. The problem to compute a multi-layered graph is divided into problems where only two layers are handled. This makes computation easier. However, it still requires efficient heuristics to be used.

5.2 CODE REUSAL

While the developing took place code reviewing of the existing GRAIL implementation was also done. Thereby, we recognized that some parts of the source code can be reused. This was possible since copying of code and modifying was no problem as the code was within the same project. The implemented algorithm for making cyclic graphs acyclic, Greedy heuristic, was reused. Other parts were also reused, for instance for creating dummy nodes.

5.3 STORING IN MEMORY

In Chapter 3.2 it is stated that the original graph is not modified till the layout is fully computed. Therefore, it was necessary to store information in other data structures. This information was needed to perform several actions within the layout algorithm. Some data structures were read and modified more often, e.g. layer information. For this reason already provided data structures by Java were used for storing information. Most information was stored in objects of “ArrayList” or “HashMap”. By using these types we can be sure that reading a value is always done within a defined time frame.

5.4 INSTANTIATING THE SUGIYAMA ALGORITHM

The whole program was developed in one class at the moment. Due to the complexity of the algorithm it is easier to separate some methods into another class later on. The Sugiyama class has only two public constructors and one public method. An object must be instantiated and parameters given to one constructor or the standard constructor must be used. If the standard constructor is used, internal variables, e.g. maximum number of iterations, are set to default values which cannot be modified. The public method is for passing over an object of type “DirectedGraphInterface” and to start the layout algorithm. All other functions needed to perform the actual computing of the layout are hidden from the user.

5.5 DEBUGGING INFORMATION

For doing analysis of the current program as well as testing certain functionalities debugging information was extremely useful. The last methods of the algorithm are always dependent on the previous one. They are using the results for further computing.
In order to test some functionality many “print-outs” for the console were programmed. These were used first to be able to do a quick check of the results and if needed, having a closer look to single computations which were sometimes also written to the console. Especially for testing methods, investigating known bugs and looking for their root causes, debugging information were useful.

This information is not determined for the user. For developing, testing and maintaining the program a switch was created. This switch is a simple global variable within the Sugiyama algorithm and set to false by standard so no debugging information is shown at all. This enables the user to activate two options for getting debugging information. First, all information can be written to the console, e.g. every single computation of important steps. Second, only relevant information is shown to the engineer, e.g. number of minimum crossings or number of iterations. By standard both debugging modes are switched off.

5.6 TESTING
The complexity of the developed algorithm is quite high. Therefore, we tested the algorithm extensively by computing graphs manually and comparing the results. This was done for each single method. It was too risky not to test this as the next method fully relies on the correctness of its input. A test scenario was programmed which reads a graph from a file, passes it over to the layout algorithm, and stores the output in another file for visualization tests. These tests were only possible after the algorithm was fully implemented. In between we made tests for each method on paper. For instance, on a graph with initially 37 nodes and 68 edges all edges were checked to see whether they were correctly stored in the corresponding data structure. After the single tests were carried out successfully other methods were implemented and iteratively tested. With this scheme we ensured that each single step was valid and the output could be used for other methods.

In certain tests the maximum number of iterations to achieve a minimum number of crossings was evaluated and found to be relatively small but dependent on the graph input. Currently, the maximum number is set statically. Other criteria, e.g. number of long-span edges were tested as well. The most important test was to check the output after the whole algorithm was performed. Here, the graph written to the output file was drawn manually and double-checked to make sure that the number of crossings was equal to that of computed for all layers. The output was also visualized by yEd to see how it was displayed on the computer.

5.7 PERFORMANCE
For investigating the performance of the implemented algorithm several analyses were made. Within these tests the focus was on crossing minimization as this is the criteria with highest importance. Several criteria were analysed, e.g. initial number of crossings. These tests clearly showed that the implemented algorithm works well and can minimize edge crossings to a huge extent. Figure 5-1 presents the results achieved by computing a layout of four different graphs. All information about the initial situation and the final layout is presented.
<table>
<thead>
<tr>
<th>graph</th>
<th>nodes</th>
<th>edges</th>
<th>feedback edges</th>
<th>dummy nodes</th>
<th>longspan edges</th>
<th>initial crossings</th>
<th>minimum crossings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37</td>
<td>68</td>
<td>19</td>
<td>45</td>
<td>30</td>
<td>430</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>51</td>
<td>4</td>
<td>34</td>
<td>18</td>
<td>215</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>46</td>
<td>13</td>
<td>4</td>
<td>4</td>
<td>154</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>26</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5-1: Performance tests

In line one of Figure 5-1 we can see that the graph initially had 430 edge crossings which are reduced to 22 by the implemented algorithm. This table above also shows the inserted feedback edges, dummy nodes and long-span edges while the layout was computed.
6 COMPARISON OF SUGIYAMA RESULTS AND RESULTS ACHIEVED BY YED

The layout algorithm produces a graph output with all layout information. As already described it does not store the graph. With a small test program we take the updated graph and store it in a file. Afterwards this is opened by yEd and the computed result is graphically shown. The goal criteria state important points the layout algorithm must handle. In this chapter the result of the Sugiyama layout is compared with the output of yEd given the same graph as input. Advantages as well as drawbacks between the two layouts are presented and figures show the actual graphical output. The focus is on comparing the goal criteria stated as well as criteria for the Sugiyama layout. The first graph compared contained 7 nodes and 10 edges.

First, the output of yEd is discussed. It is shown in Figure 6-1 on the left side. The layout style used in yEd is *Orthogonal - UML Style*. The labels are fit to nodes with the method available in yEd. By first looking at the design it really follows an orthogonal approach which can be seen by the four classes which are laid on top of each other. The edges connecting these four nodes are quite small but some of them are not straight and have several corners. Two edges are extremely long although they connect nodes located next to each other. However, these edges could be much shorter and connect the next node directly which increases the readability to a huge extent. All goal criteria stated are met in this layout.

Secondly, the Sugiyama layout produced by the implemented algorithm is presented on the right side of Figure 6-1. The labels were adjusted by the function available in yEd. The layout positions were completely computed by the Sugiyama algorithm. The layout clearly follows a hierarchical layout where all edges start at the top layer directed to lower layers except feedback edges. There is always a gap between classes and no overlapping occurs. All UML information is shown as well. Edges are shown as straight lines except one for a self-association which is shown as circle. The individual layers for the hierarchical layout can be recognized. At the top layer there is one node, in the second layer five nodes, and in the lowest layer only one is shown. Totally, the edges are shorter compared to the layout created by yEd. The layout also follows the criteria for a balanced and close layout.

In both layouts there are no edge crossings. It seems that the layout achieved by Sugiyama has a slightly higher readability due to the clear hierarchy and the shorter edges. But this layout visualized with yEd has two significant shortcomings:

- One can recognize that only seven out of ten edges are shown. Obviously three edges were cut out of the layout. This is a result of yEd that is used for displaying the computed output. In three cases two edges are shown at exactly the same place so they overlap. They are not separated in yEd. Again, this is not a result of the layout algorithm neither an error of the implementation. All edges are output correctly. This was tested as explained in Chapter 5.6.

- Edges are crossing classes. This occurs as nodes are set according to their gravity point. For instance in layer two the upper edge of nodes shown is not equal for all nodes. The most right vertex is lower at the top than the node to its left. Therefore, the edge is drawn through this node and overlaps it. This problem occurs more severely for larger graphs and this is explained later on.

With the help of an edge router that is available in yEd, the “hidden” edges can be routed differently. By applying the orthogonal edge routing all edges are shown individually which further emphasizes that an edge router could improve the current layout produced by the implemented algorithm extensively (see Chapter 7.2).

Overlapping of classes can first be handled by adjusting the existing layout algorithm so that the top edges of all nodes are on the same height. Then the upper edge is shown
correctly but it might happen that edges leaving nodes at the bottom are overlapping other nodes. This can be further investigated.

Figure 6-1: Graph laid out by yEd and Sugiyama
Now a graph with 15 nodes 30 edges is laid out by yEd and the Sugiyama layout algorithm. In Figure 6-2 the results of this graph are presented.

Figure 6-2: Graph with 15 nodes and 30 edges
On the left side the layout computed by yEd is shown and discussed first. The layout neither follows a hierarchical nor an orthogonal approach. The nodes are placed with a mix of both approaches. There are no edge crossings. Some edges are quite long although they could be much shorter. Next to the main node in the middle of the layout two bends of edges occur. To the top of this node the edges can be traced by the eyes but the readability is reduced due to this phenomenon. To the right of this node the second bench is shown. Here the edges can definitely not be followed anymore as they are too close to each other. Single edges cannot be distinguished at all. Concerning the readability this is the worst drawback of this layout.

The Sugiyama layout follows, as expected, the hierarchical approach. This is clearly shown. All nodes are put into four layers where the top layer is the largest one. All edges can be traced by human eyes and distinguished clearly although some of them are longer compared to the yEd layout. Edge crossings are minimized as much as possible. In the Sugiyama layout only one edge crossing should be shown. In the picture three crossings are shown but this is due to the fact that yEd only shows edges from the source to the target node. The layout algorithm inserted several dummy nodes before and performed crossing minimization on this graph but in the final layout no dummy nodes are stored. In order to only show crossings computed by the Sugiyama algorithm, edges must be routed via dummy nodes. As this is not fulfilled it could be that many more crossings will be shown. Again, this is a problem of the visualization tool and has nothing to do with the implemented algorithm.

By checking the requirements stated for the Sugiyama layout with the actual output it can be seen that they are fully met. The most obvious requirement is the hierarchy and it can be checked immediately within the layout. Regarding the edge span it is not easily possible to check this in the final drawing but several tests were performed to check if this is handled properly. The tests were successful. The criterion for straightness of edges is also met. Even there are crossings with other classes but this is not up to our implementation as already described. A close layout is achieved but it is hard to recognize as dummy vertices, which had to be inserted, are not shown. The criterion for a balanced layout is fulfilled as nodes are placed as close as possible to their neighbours.

To sum up, the layouts produced by the Sugiyama algorithm follow a clear hierarchical structure. In most graphs we tested it showed that the edges were also shorter. Class overlapping did only occur under special circumstances (see Chapter 4.2.9). Edges were crossing other classes. First, this was due to the fact that nodes are positioned according to their gravity centre. Secondly, edges which were long-span ones were routed through dummy nodes but in the visualization this was not done. Crossing minimization was done successfully. Thus, this criterion was met for the output but the visualization tool did unfortunately not visualize dummy nodes. Consequently, there is an unpredictable number of edge crossings as the edges are simply routed from the source to target node. All readability criteria stated for the Sugiyama algorithm were successful.
CONCLUSION AND FUTURE WORK

This chapter reflects the achieved results within this thesis. It compares the goal criteria with the actual work and shows whether the problem has successfully been solved. Improvements of the current work will shortly be presented as well as ideas to solve them.

7.1 CONCLUSION

The goal criteria shall shortly be summarized and evaluated if they are met.

1. The first criterion stated was to implement at least one suitable layout algorithm into the existing visualization plug-in. All layout information has to be stored in the given graph.

2. Several criteria for a “good” layout were stated: no overlapping between classes, edges are not allowed to cross other classes, and edge crossings shall be minimized.

3. The algorithm must provide an API so it can be instantiated and configured with parameters. Also a graph must be passed to the instance.

4. The results of the layout algorithm are to be compared with those yEd produces. Important advantages as well as shortcomings must be described.

According to the first criterion stated for the development and implementation of a layout algorithm the criterion is fully met. In Chapter 3 we decided to use on the Sugiyama approach. The thesis focused on this layout. In Chapter 4 the algorithm was developed so it could be implemented into the existing visualization plug-in later on. All presented steps shown in this chapter were fully implemented and tested. All presented results were achieved by the developed algorithm.

The second goal criterion was much harder to achieve. Therefore, reference literature was studied to find criteria for a “good” layout. Several facts rely heavily on human cognition and had to be investigated. The requirements for such a layout are stated in Chapter 3. Within the goal criteria we fully met the requirement for minimizing edge crossings. The algorithm was especially developed to minimize them as much as possible. No overlapping between classes was also achieved with some exceptions. Those occurred when objects were extremely high or wide, so boundaries for the objects computed by the layout algorithm differed to a huge extent from boundaries computed by yEd. yEd has to render the objects and does not take the real length in characters or height the layout algorithm computed itself. In order to be able to handle this situation two offsets are introduced. These ensure that there is a gap between all classes. If there is still an overlapping between classes, this offset can be adjusted. Therefore, this criterion is also met. The last goal criterion for a good layout is that edges do not cross other classes is fully met as well. All edges are set from node to target node. If there are long-span edges, dummy nodes are inserted to avoid those edges. If the output is visualized in yEd, one can see there are edges that cross classes. This is only a problem of the visualization tool with regard to the requirements stated for the development of the algorithm. All dummy nodes and edges are removed before the final layout is stored. Only normal nodes are stored and shown in yEd. To overcome this problem we state one point in future work but, however, the criterion is met as this is only a visualization problem. To sum up, all criteria for a “good” layout are met.

In order to be able to configure the implemented algorithm two constructors were implemented within the Sugiyama class. With the help of these the user can set different values and configure the algorithm. The overall algorithm was implemented in one class and this class must be instantiated with a constructor before the algorithm can be started. All important parameters are set and the user can use one method to run the
algorithm by simply calling this method. A graph input must be passed as parameter. Thus, this criterion is met, too.

For the last goal criteria Chapter 6 was reserved. Within this chapter the output of the implemented algorithm is compared to the output yEd produces. Differences and other relevant findings were already discussed in this chapter. It is shown that the outputs are definitely equivalent or better than those produced by yEd.

The conclusion of this thesis highlights that all goal criteria were successfully met. For some criteria future work can be done to further optimize the output.

7.2 FUTURE WORK

This paragraph gives a brief outlook on future work which can be done to further improve the overall performance of the implemented layout algorithm. Possible improvements are not stated in priority order.

As explained in Chapter 4.2.7 crossing minimization always exchanges two nodes in each matrix and checks if the edge crossings are reduced. For larger matrices this process takes quite a long time as the number of crossings must be computed after each step. An improvement can be to compute a fully ordered matrix according to barycentre ordering and only then testing the number of crossings. This saves several runs of edge crossing computing.

Crossing minimization could be skipped for matrices where the number of crossings is zero. Right now the algorithm starts achieving a barycentre ordering even if there are no crossings between the current layers. Processing can be skipped for phase1 and phase2.

For exchanging nodes with equal barycentres in phase2 we can set up a scheme to not change the same nodes all the time. In some cases the same nodes shall always be reversed with each other. It can be investigated if the crossings can be further reduced by not changing the same nodes back.

For all long-span edges dummy vertices are inserted. Also for long-span feedback edges dummy nodes are inserted. While doing so the path on which they are inserted can be checked if it already exists. Then no dummy vertices must be inserted which reduces the overall computing time for crossing minimization.

After having done a maximum number of iterations, the algorithm stops crossing minimization. The current results are not taken into account. For larger graphs this is not a suitable as the number of iterations is set statically. This threshold is dependent on the given input and therefore setting it dynamically, e.g. dependent on the number of nodes and/or edges, will be an advantage.

Another threshold can be used for saving the current configuration when the least number of crossings for that moment is achieved. A scheme can be developed with which a threshold is dynamically set and signals when the best configuration, e.g. dependent on overall iterations in crossing minimization, shall be stored.

For some graphs the same matrices occur over and over again while the reduction of edge crossings is done. If this happens, the process can be interrupted as no further reduction is foreseen.

In Chapter 4.2.9 we stated that the layout algorithm cannot avoid that edges are crossing other classes. This is a requirement stated and although set as goal criteria. This is not a drawback of the implemented algorithm. It is a problem of the visualization tool which only handles normal nodes and edges. To avoid these crossings an edge router can be implemented which defines the way of edges along their path from the source to the target node. Another method is to add dummy nodes and edges to the graph and route all long-span edges through dummy nodes.
With respect to crossing minimization the performance of the existing algorithm is quite good. A performance benchmark shall be executed to monitor criteria like computing time or memory consumption. Especially for larger graphs the current algorithm has performance problems. Based on the gained information the algorithm can further be optimized.

For layout positioning the reference literature states that the optimum output can be produced within three runs: down, up, and down again. It can be investigated if the layout will be better if more runs are performed. The existing algorithm can also be improved to achieve a better balanced layout. The node with the highest priority becomes closest to its neighbours although there are several other nodes which are neighbours as well.

Currently layout positioning cannot compute the height and width of rendered nodes. Therefore, class overlapping might occur. If the algorithm gets the correct information as input and does not compute it, class overlapping can no longer occur.

For further optimizing the output layout positioning can be adjusted to avoid edge crossings with classes. This can be achieved either by placing the nodes differently or using an edge router.

The implemented layout algorithm can be called as already described. Once it completed layout computing it simply stores the layout information in the given graph. No information is returned to the caller. A class containing statistic information about the layout computed (e.g. number of crossings) can be returned to inform the caller about the achieved results.
REFERENCES
APPENDIX A

In order to use the implemented layout algorithm an instance of the algorithm must be created first. While doing so, several variables necessary for internal computations need to be set. These can be set to default values or to specific values given by the user dependent on which constructor the user calls. Currently, there are two constructors implemented as follows:

1) Sugiyama sugi = new Sugiyama();
2) Sugiyama sugi = new Sugiyama(maxIt,maxT);

The first constructor reflects the standard constructor and no parameters are passed over. The second one takes two parameters: the maximum number of iterations and the maximum amount of time. Using the standard constructor the two variables are set to standard values:

- maximum number of iterations = 50
- maximum amount of time = 3600000s = 3600 s = 1 hour

If the maximum number of iterations of phase 1 and 2 is reached, the algorithm stops at a specific position. Thus, this may not be immediately. The final number of iterations might be higher, at maximum +2. This behaviour was already explained earlier.

These values are working well for most of given graph inputs but especially the amount of time is strongly dependent on the speed of the computer used. Using the second constructor these values can be set user-specific allowing limiting one or both stated criteria. A flexible configuration can be achieved which is needed for larger graphs. For the maximum amount of time this is also true. If it is reached, crossing minimization is not stopped immediately but at a specific best position. Therefore, it may happen that algorithm runs "slightly" longer than maxTime. Especially for larger graphs the difference between setting the stop-flag and the end of crossing minimization may significantly be higher, phase1 may run to the end. This is done on purpose and no drawback of the introduced maxTime value. The time is passed over in milliseconds.

Further down a short sequence of code is shown representing how an instance of the layout class can be created. The last line presents a method implemented in this class. Once this method is called the actual computation is started and the layout for the graph passed over is computed.

Sugiyama sugi = null;
//run with default values
//sugi = new Sugiyama();

//run with individual parameters
//Sugiyama(maxIterations,maxTime)
sugi = new Sugiyama(18, 180);
sugi.run(g);

While computing the layout there might be several critical regions where information cannot be accessed or a computation cannot be done. Therefore, the most critical parts of the algorithm are covered by exception handling. Once an exception is thrown, it is passed to the user with a specific message. With the help of it, the user can conclude which part of the algorithm could not be finished successfully. Three exception classes were introduced:

- **SugiyamaException**: This exception is thrown whenever incorrect values are passed over to the constructors or the run-method is not called with a correct graph interface.
- **ParseLabelException**: For computing the height and width of single nodes the label of the nodes must be parsed. If the label is not formatted correctly, the height and width cannot be computed as the label cannot be parsed.
- **CrossingMinimizationException**: Crossing minimization is the most critical part of the whole algorithm. Therefore, both phases are covered by this exception.
Consequently, the user must call the layout algorithm as shown below. A try and catch block is strictly necessary to catch thrown exceptions. The user can also see which problem occurred while layout computing. If needed, the user can catch exceptions separately. In order to assist the user further, a test class was written in which the algorithm is called.

```java
Sugiyama sugi = null;
try {
    //run with default values
    //sugi = new Sugiyama();

    //run with individual parameters
    sugi = new Sugiyama(18,180);

    sugi.run(g);
} catch (Exception e1) {
    e1.printStackTrace();
    System.out.print(e1.getMessage());
}
```