Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams

Jochen Seemann

Institut für Informatik, Am Hubland, 97074 Würzburg,
seemann@informatik.uni-wuerzburg.de

Abstract. The automatic layout of software diagrams is a very attractive graph drawing application for use in software tools. Object-oriented software may be modelled using a visual language called the Unified Modeling Language (UML). In this paper we present an algorithm for the automatic layout of UML class diagrams using an extension of the Sugiyama algorithm together with orthogonal drawing. These diagrams visualize the static structure of object-oriented software systems and are characterised by the use of two main types of edges corresponding to different relationships between the classes. The graph drawing algorithm accounts for these concepts by treating the different edge types in different ways.

1 Introduction

Object-oriented modeling techniques such as Booch [1] or OMT (Object Modeling Technique) [10] and their graphical representations of object-oriented software design have become very popular in recent years. The Unified Modeling Language (UML) [14] is an up-coming standard for specifying and visualizing various aspects of object-oriented software systems. UML is a graphical language derived from several existing notations commonly used to specify the design of object-oriented software. Some important diagrams are those representing the architecture of the software. In UML these are known as static structure diagrams. Class diagrams form a subset of these which is used to represent the static structure of classes and the relationships between them.

In this paper we present a technique for the automatic layout of UML class diagrams. Our algorithm is based on a combination of an extension of the well-known Sugiyama algorithm and orthogonal drawing techniques. We have implemented the algorithm as a part of a tool called UML workbench, which is used to demonstrate new techniques for software engineering tools.

The paper is organised as follows. Section 2 gives an overview of the diagrams and the related graphs that should be drawn. Section 3 describes the phases of our layout algorithm. In section 4 we show an example of the output from our drawing algorithm. Finally, we discuss the application of the algorithm and give some ideas for further work in this field.

2 UML Class Diagrams

Figure 1 shows an example of a UML class diagram. The classes are depicted by rectangles containing a list of the attributes and operations (methods) of the class. There are two main categories of relationships between classes:

- inheritance relationships (Fig. 1: Expression / TypeExpression or ModelElement / Instance): these represent generalization-specialization relationships allowing a hierarchical classification of classes. These relationships are known from object-oriented programming languages.
- associations (Fig. 1: Type / TypeExpression): these represent a more general relationship between classes as for example in ER-diagrams. An aggregation (Fig. 1: Instance / Value) is a special association representing a containment of classes and is drawn with a special symbol.

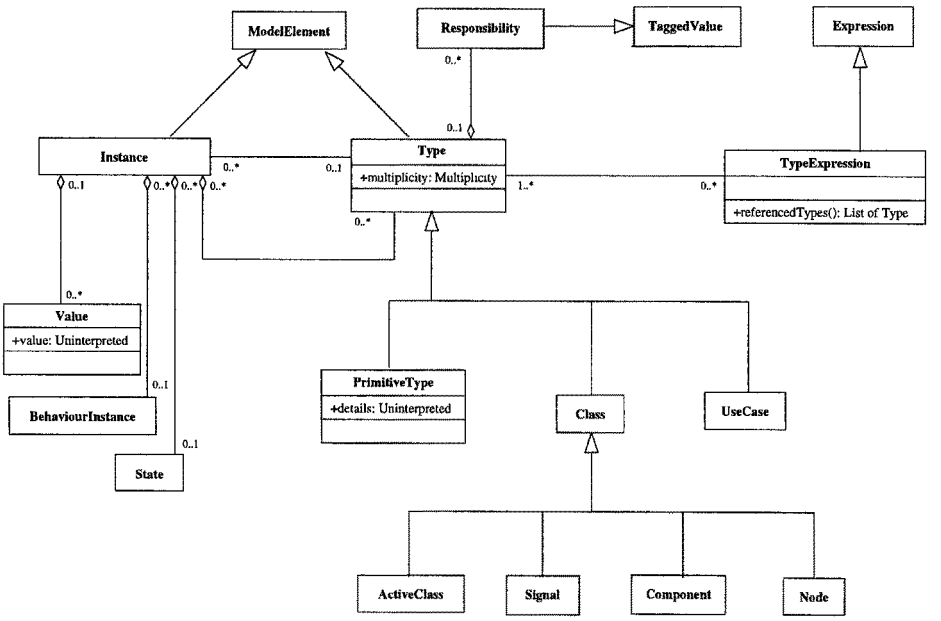

Fig. 1. Example class diagram in UML notation (as printed in [14])

A diagram can be seen as a graph whose nodes represent classes and whose edges represent relationships. Because of the semantic richness of UML both the nodes and the edges may have attributes (Fig. 1). These attributes are represented textually or in some cases graphically.

We have identified some important conditions for the drawing of UML class diagrams that lead to the idea of our algorithm:

- Nodes do not have a fixed size.
- The main difference to usual graph drawing problems is that we have two types of edges between the nodes, that must be treated by the layout algorithm in two different ways.
- The most important condition for the placement of classes is the inheritance structure of the graph. The subclasses (specializations) should always be placed below their superclasses (generalizations). This leads to a hierarchical structure with the classes placed on several levels.
- The subgraph of a diagram, which contains only the classes and the inheritance relationships, is a directed acyclic graph.
- The placement of classes connected to other classes by association edges is free, but should lead to short edges and few crossings.
- The subgraph of a diagram, which contains the classes and the association relationships, is a general graph that may be cyclic.
- The software engineer expects a layout of the diagram that satisfies these constraints even if there are alternative layouts for the graph with, for example, fewer crossings. This means the diagrams drawn by our algorithm may be far from an edge-crossing minimal solution.

We conclude the following drawing strategy from these characteristics:

- First place the classes involved in inheritance relationships in a hierarchical structure.
- Then place the remaining classes preserving the basic structure.

3 Algorithm

The Sugiyama algorithm is a widely used technique for drawing directed graphs [13], which has been well-analyzed [4, 5] and improved in many ways during recent years [6, 9].

Our approach is to use the Sugiyama algorithm with some modifications for the placement of all classes involved in inheritance relations. We then place the remaining classes using our extension to the basic algorithm. Then the other classes are placed using an incremental placement algorithm. This placement may influence the first Sugiyama placement again.

We represent a UML class diagram as a graph G, whose nodes are classes and whose edges are relationships among these classes.

Our algorithm consists of several phases:

Phase 1 - Preparation

1. First we remove direct cycles in G. These edges are removed and stored as attributes of the nodes involved in the cycle.
2. Then we compute the subgraph I of G that contains only the classes related through inheritance together with their inheritance connections. If I does not form a single graph, we add a new (hidden) node as root of all the different partial graphs. I is a directed acyclic graph.

Phase 2 - Sugiyama Layout

1. For the graph I we compute a first layering as described in [13]. The nodes are assigned to different layers according to the structure of the inheritance graph.
2. Reduction of crossings: this phase reorders the nodes in each layer to reduce the number of crossings. As in [13], we use barycentric ordering [15] as a heuristic method for this purpose.
   We also take into account that there may be other edges between nodes of graph I due to association relationships between the classes. If there are such edges, the nodes should be placed next to each other where possible. We compute the set O of these additional edges between nodes of I. From O we compute priorities for the placement of the nodes on each layer. If O contains an edge between two nodes on the same layer, these nodes should be placed as neighbours on the layer.
   In some cases, the drawing of an association edge between nodes placed in different layers can be improved. For example, if a node X placed in the layer L_i has an association edge to a node Y placed in layer L_j $(i < j)$, and there are no inheritance edges from X to nodes in layers between L_i and L_j, this node X can be placed in layer L_j. If X has several such edges, it is placed in the layer with the lowest index.
3. After the previous step is completed, mark all nodes that are neighbours in a layer and an edge of O exists between them. Remove this edge from O.

Phase 3 - Incremental Extension

In this phase, we extend the layout incrementally, until all nodes have been placed in the diagram.

1. Compute the set S, which contains the nodes of Graph I.
2. Select nodes from G that are not in S and which are connected to nodes of graph S due to association relationships. We construct sets S_i for each node N_i of S in two steps as follows:
   - First, for each node X in G, not in S, select X for S_i if it has one or more connections to exactly one node N_i.
   - Next, for each node X in G, not in S and not in any S_i, choose X if it has more than one connection to nodes in S. The node X is added to the set S_i that relates to a node N_i connected to X and that has the minimum number of elements.

   Now we are able to extend the existing layout in the neighbourhood of node N_i with each of these sets S_i. If there are one or two nodes in S_i we simply place those nodes in the layer to the right and to the left of N_i. If there are more than two nodes in S_i we insert another sublayer into the diagram. The nodes of S_i are placed in that layer with directed edges to N_i. If there is already a new sublayer above or below the layer of node N_i, we use this existing sublayer. If N_i is marked in the last step of Phase 2, we have to use

a sublayer even if there are less than three nodes in S_i, because we cannot add a new node to the right or/and to the left of N_i on the layer.

3. Add all nodes of all sets S_i to S.
4. Repeat steps 2. and 3. until all nodes of the original graph G are placed.

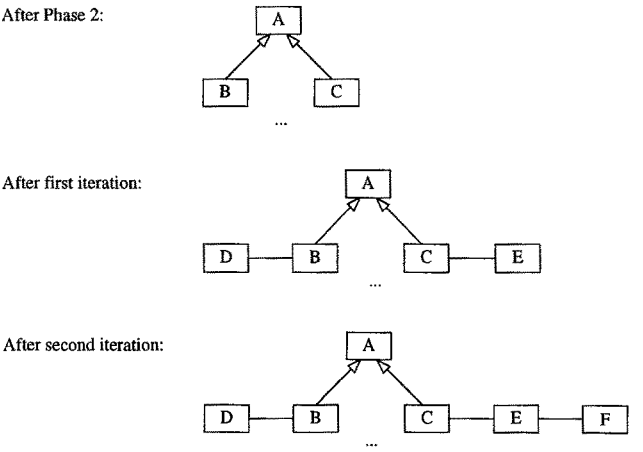

Fig. 2. Incremental extension

5. Optimize the node positions in each layer. In this step we try to reduce the number of crossings and bends of the association edges between classes on a layer. This step also improves the aspect ratio of the drawing. The nodes involved in inheritance relationships are not moved anymore. Fig. 3 shows an example of such a transformation.

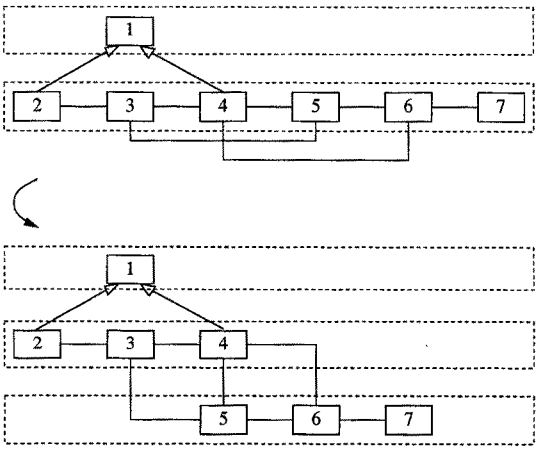

Fig. 3. Optimization using sublayers

Phase 4 - Orthogonal Drawing of Association Connection Edges

In UML diagrams, association relationships are usually drawn as orthogonal line segments and inheritance relationships as straight lines of any angle. As shown in Fig. 1 inheritance relationships may also be drawn as orthogonal segments. Straight line drawing is preferred however, because it is easier to distinguish the different edge types.

1. Compute the node sizes (Fig. 4). We have to take into account that our nodes consist of the class with their attributes and parts of the attributes of the relationships. The node size depends also on the position of other nodes that are connected to the node. For this reason the node size is calculated as in Fig. 4.

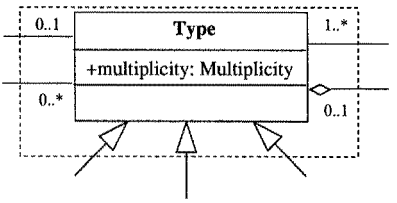

Fig. 4. Calculation of the nodesize

2. Add hidden nodes to construct the drawing of the edges remaining in set O and the edges handled in the second part of Phase 3, Step 2. We add a hidden node next to (right or left) each of the two nodes we have to connect. If the edges cross at least one layer, we add hidden nodes on all layers that are between these two nodes.
3. Now we are able to do the fine tuning of the node positions on the layers as in other Sugiyama implementations [6, 7].
4. Compute line positions of association relationships between adjacent nodes on the same layer. These will be drawn as horizontal lines. The hidden nodes may be expanded to improve the drawing of the connecting lines (For example in Fig. 5. the hidden node on the right side of node 'Shape').
5. Compute line positions of the straight lines representing the inheritance relations.
6. Compute line segment positions of association relationships between classes on the same layer or on adjacent layers. We also connect the hidden nodes added in Step 2 of this phase. We insert horizontal gridlines between the existing layers to construct the drawing of the edges. This technique is similar to the known construction of orthogonal grid drawings as in [3]. The gridlines can be shared between several edges. If it is necessary, additional gridlines are inserted. We use a sweep line algorithm as in [11] for this task.

4 Implementation

We have developed a tool called UML workbench. Using this tool we have investigated the drawing algorithms for UML class diagrams described by an underlying scripting language [12]. The following example (Fig. 5), with a graph taken from [10], shows the layout produced by our algorithm. Further examples are shown in Fig. 6 and Fig. 7.

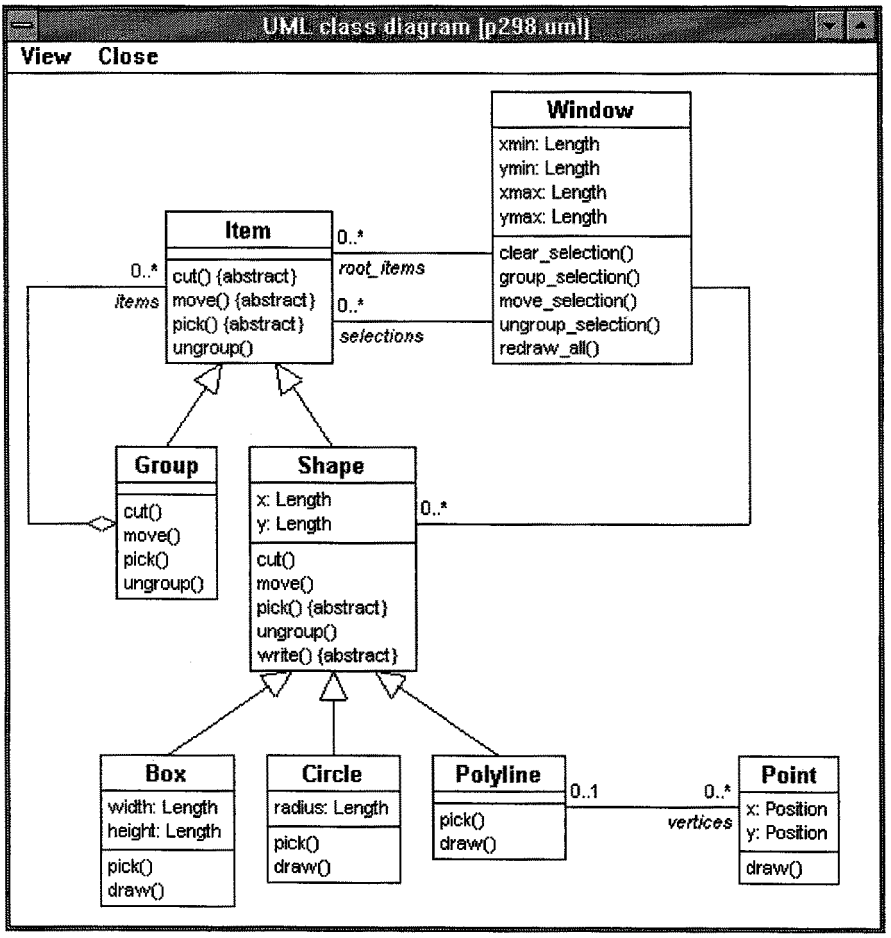

Fig. 5. Example from [10] drawn by the algorithm

5 Conclusion

We have presented a technique for the automatic generation of UML class diagrams. It is fundamentally the adaptation and combination of various algorithms

developed during recent years among the graph drawing community. The growing use of visual languages in the field of object-oriented software engineering will lead to interesting graph drawing applications such as graphical browsers and CASE-Tools.

This work is a first step towards the automatic layout of object-oriented software diagrams. We have implemented the most important subset of the UML static class diagram notation.

We have good results for object-oriented architectures with considerable use of inheritance relationships but poor results where the architecture is based heavily on associations. In order to overcome this problem we plan to investigate an extension to planar hierarchical drawing algorithms [3]. It is expected that the graph characteristics can be used to select the most appropriate layout algorithm for a particular diagram.

For interactive applications such as CASE tools our algorithm has a useful property; when the software developer changes the relationships between the classes, we preserve the fundamental structure of the diagram. In this way we preserve the class hierarchy as a "mental map" for the user.

The algorithm will be used in a reverse-engineering tool for C++ software projects. Further work will focus on clustering and folding techniques in the diagrams, because software diagrams may consist of a few hundred classes. For large diagrams node clustering and diagram folding algorithms are as important as the graph layout algorithm itself.

References

1. G. Booch: *Object-Oriented Design*, Benjamin/Cummings Publishing, 1991
2. F.J. Brandenburg, editor: Proceedings of Graph Drawing '95, Vol. 1027 of *Lecture Notes in Computer Science*, Springer Verlag, 1996
3. G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis: Algorithms for drawing graphs : an Annotated Bibliography, *Comput. Geometry Theory Appl.*, 4:235-282, 1994
4. Peter Eades, Kozo Sugiyama: How to draw a directed graph, *Journal of Information Processing*, 14(4):424-437, 1990
5. A. Frick: Upper bounds on the Number of Hidden Nodes in the Sugiyama Algorithm, in [8], pp. 169–183
6. E.R. Ganser, E. Koutsofios, S. North, K.-P. Vo: A technique for drawing directed graphs, *IEEE Transactions on Software Engineering*, 19(3): 214–230, March 1993
7. F. Newbery-Paulisch, W. F. Tichy: Edge: An extendible graph editor, *Software - Practice and Experience*, 20(1): 63–88, June 1990
8. S. North, editor: Proceedings of Graph Drawing '96, Vol. 1190 of *Lecture Notes in Computer Science*, Springer Verlag, 1997
9. P. Mutzel: An Alternative Method to Crossing Minimization on Hierarchical Graphs, in [8], pp. 318–333
10. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson: *Object-Oriented Modeling and Design*, Prentice-Hall, 1991
11. G. Sander: A Fast Heuristic for Hierarchical Manhattan Layout, in [2], pp. 447–458

12. J. Seemann, J. Wolff von Gudenberg: *OMTscript - eine Programmiersprache für objekt-orientierten Software-Entwurf*, Technical Report, Department of Computer Science, Würzburg University, 1997
13. Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda: Methods for visual understanding of hierarchical system structures, *IEEE Transactions on Systems, Man, and Cybernetics* SMC-11(2): 109–125, February 1981
14. Rational Software Corporation: *The Unified Modeling Language 1.0*, only available via WWW: `http://www.rational.com`, January 1997
15. J. Warfield: Crossing Theory and Hierarchy Mapping, *IEEE Transactions on Systems, Man, and Cybernetics* SMC-7(7): 505–523, July 1977

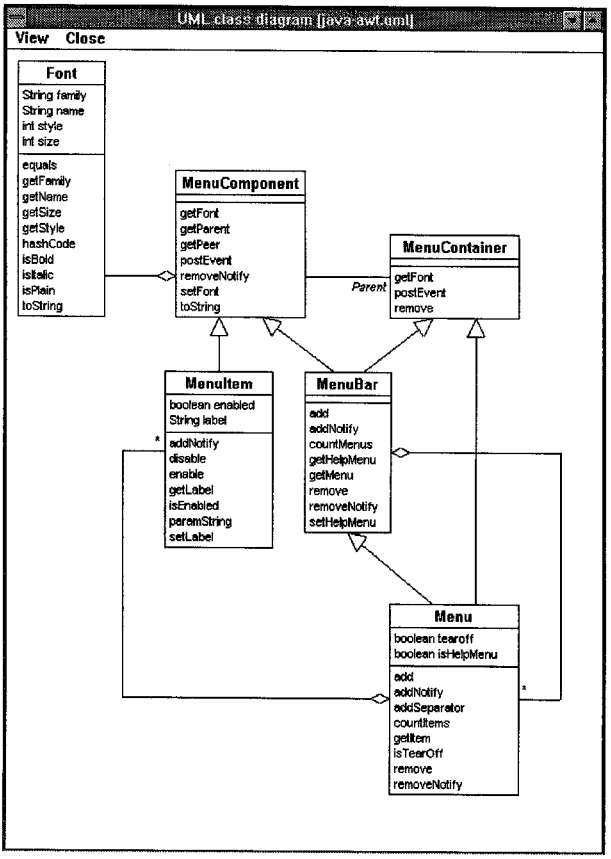

Fig. 6. Example from the AWT class library for Java (© by Sun Microsystems Inc.)

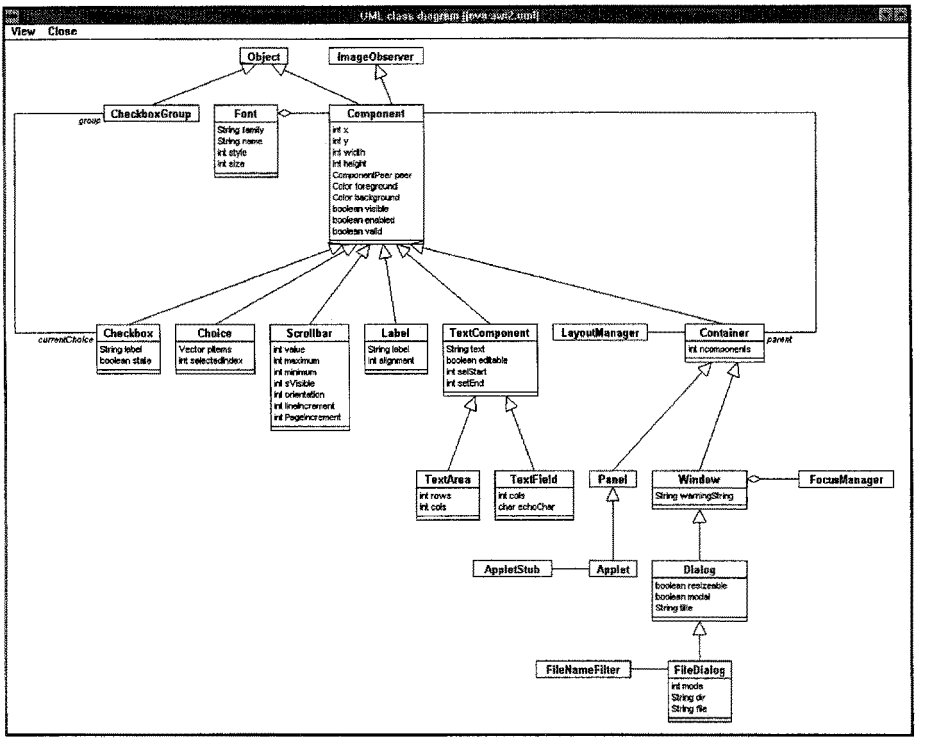

Fig. 7. Example from the AWT class library for Java (© by Sun Microsystems Inc.)