

A New Approach for Visualizing UML Class Diagrams

Carsten Gutwenger
Research Center caesar
Friedensplatz 16
D-53111 Bonn, Germany
gutwenger@caesar.de

Joachim Kupke
Research Center caesar
Friedensplatz 16
D-53111 Bonn, Germany
kupke@caesar.de

Michael Jünger
University of Cologne
Pohligstraße 1
D-50969 Köln, Germany
mjuenger@informatik.uni-koeln.de

Sebastian Leipert
Research Center caesar
Friedensplatz 16
D-53111 Bonn, Germany
leipert@caesar.de

Karsten Klein
Research Center caesar
Friedensplatz 16
D-53111 Bonn, Germany
kklein@caesar.de

Petra Mutzel
Vienna University of Technology
Favoritenstraße 9–11 E186
A-1040 Wien, Austria
mutzel@ads.tuwien.ac.at

Abstract

UML diagrams have become increasingly important in the engineering and reengineering processes for software systems. Of particular interest are UML class diagrams whose purpose is to display class hierarchies (generalizations), associations, aggregations, and compositions in one picture. The combination of hierarchical and non-hierarchical relations poses a special challenge to a graph layout tool. Existing layout tools treat hierarchical and non-hierarchical relations either alike or as separate tasks in a two-phase process as in, e.g., [Seemann 1997]. We suggest a new approach for visualizing UML class diagrams leading to a balanced mixture of the following aesthetic criteria: Crossing minimization, bend minimization, uniform direction within each class hierarchy, no nesting of one class hierarchy within another, orthogonal layout, merging of multiple inheritance edges, and good edge labelling. We have realized our approach within the graph drawing library GoVisual. Experiments show the superiority to state-of-the-art and industrial standard layouts.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—graphics data structures and data types; D.2.6 [Software Engineering]: Programming Environments—graphical environments; G.2.2 [Discrete Mathematics]: Graph Theory—graph algorithms

Keywords: Software Visualization, Graph Drawing, Graph Layout, UML Class Diagrams

1 Introduction

The Unified Modeling Language (UML) by [Booch et al. 1999] provides a mainly graphical notation to represent the artifacts of a software system. The notation has been rapidly adopted as the

accepted notation for object-oriented analysis and design. UML incorporates notations to describe systems at various levels of abstraction. UML diagrams can be used to model requirements, designs, implementations and tests. Since these diagrams are means of communication between customers, developers and others involved in the software engineering and reengineering process, it is critical that the diagrams present information clearly. Appropriate layout of these diagrams can assist in achieving this goal (see [Purchase et al. 2001]).

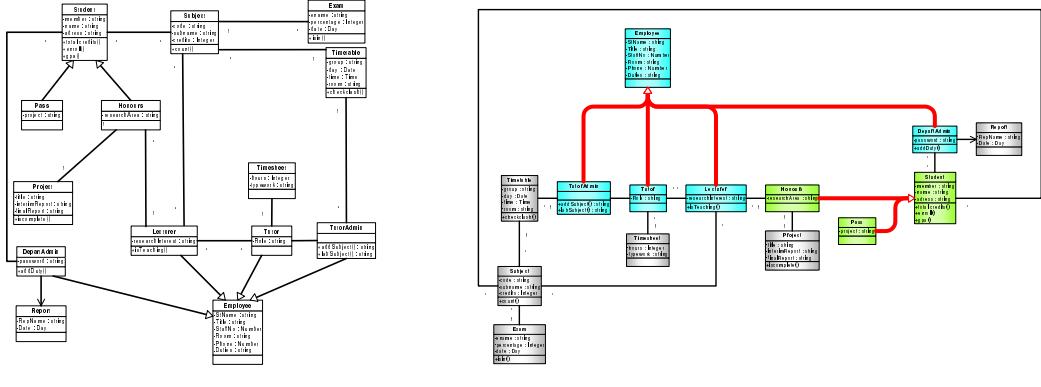
Of particular interest are *UML class diagrams* consisting of classes represented by rectangular regions containing the class name, attributes and operations of the class, and different kinds of relationships between classes that are represented as lines. We distinguish two kinds of relationships: *Generalizations* representing inheritance in class hierarchies and *associations* including *aggregations* and *compositions*. In UML class diagrams, the generalizations are typically drawn strictly hierarchically displaying the inheritance structure of the classes; the associations are the non-hierarchical elements of the diagram. A UML class diagram therefore can be modelled as a graph $G = (V, A, E)$ consisting of two kinds of connections: arcs representing the generalizations in the set A , and edges representing the associations in the set E .

1.1 Aesthetics

Figure 1 shows a small example of a UML class diagram taken from [Purchase et al. 2001] containing two hierarchies consisting of two and four classes, respectively. The authors have used this diagram to perform preference experiments assessing the effect of individual aesthetics in the application domain of UML class diagrams, resulting in a priority listing of aesthetics in the application domain of UML diagrams. According to [Purchase et al. 2001], the most important aesthetic preferences for UML class diagrams are

- crossing minimization,
- bend minimization,
- orthogonality,
- horizontal labels,
- and joined inheritance arcs.

These aesthetic preferences are easily met in the layout of the same class diagram shown in Figure 1(b) which has been automatically created by our new approach. Moreover, our new approach supports the visual perception of the human reader by



(a) Original Layout.

(b) GoVisual UML Layout.

Figure 1: UML class diagram used in [Purchase et al. 2001] for preference experiments.

- drawing generalizations in the same class hierarchy always in the same direction,
- avoiding nesting of class hierarchies,
- highlighting the various class hierarchies by different colors,
- and highlighting the generalizations by color.

In this paper, we suggest a unique technique which visualizes lucidly arranged, orthogonal diagrams featuring hierarchical and non-hierarchical elements in such a way that the directed edges of a component all follow the same direction. Transferred to UML class diagrams, a layout is created which represents each inheritance hierarchy in an aligned fashion. The computed orthogonal layout is compact and the number of edge crossings is small in practice.

Complex class diagrams often use aggregation/composition hierarchies in addition to generalization hierarchies. These aggregations/compositions may describe a second hierarchical dimension that users wish to emphasize in a layout. While we focus throughout this paper on generalization hierarchies, the same approach can be used to visualize hierarchies different from generalizations. This can be done for any kind of association. It is even possible to visualize all hierarchical dimensions within one diagram. However, such an approach does not reveal enough analytical information, since in this case the graph is usually fully directed, including directed cycles. Thus a layout based on such a directed graph does not emphasize any hierarchical dimensions.

1.2 State-of-the-Art and Industrial Standard

The combination of hierarchical and non-hierarchical relations poses a special challenge to a graph layout tool. Commercial software typically uses purely hierarchical methods, see, e.g., [Rational Software Corporation 2002] and [TogetherSoft Corporation 2002] that cannot properly distinguish between hierarchical and non-hierarchical relations.

Figures 2 and 3 show such typical industrial layouts. Both diagrams have been taken from different industrial model-driven development tools and show an object-oriented software project.

The examples in Figures 2 and 3 show clearly the disadvantages of a hierarchical layout style. While Figure 2 shows technical problems such as unnecessary bends and edges passing through class boxes, Figure 3 provides some non-hierarchical elements improving the layout. However, in both examples it is not possible to

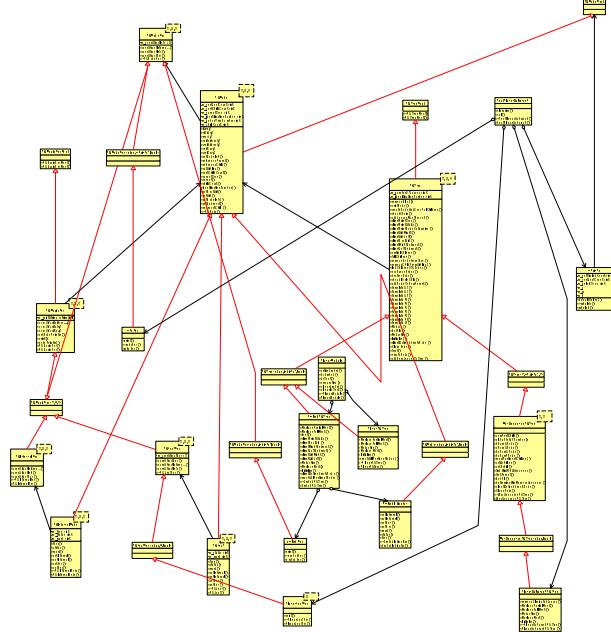


Figure 2: Industrial layout in hierarchical style.

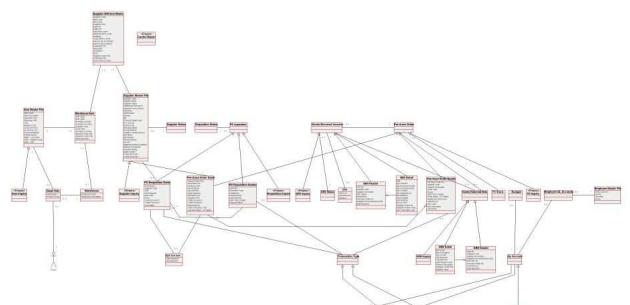


Figure 3: Industrial layout in hierarchical style.

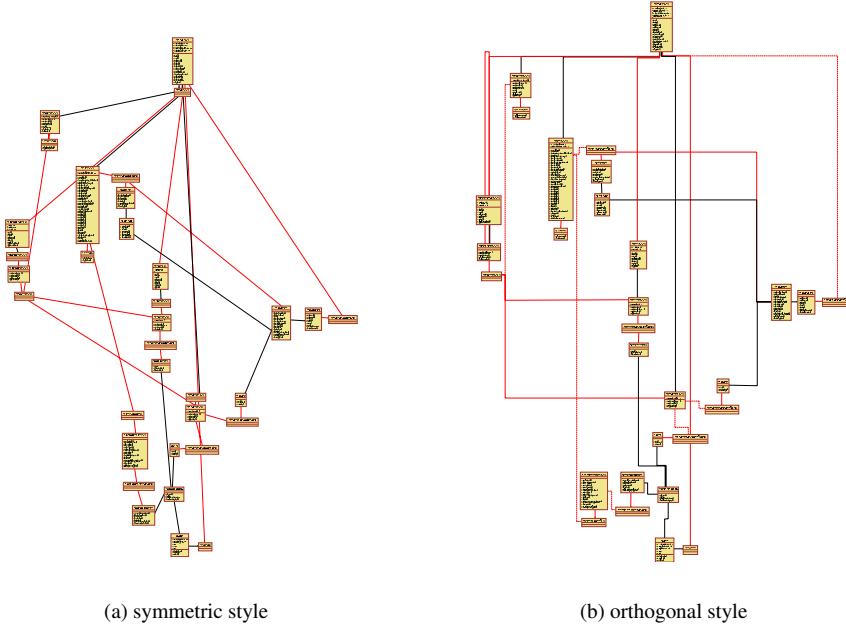


Figure 4: Industrial layout in different styles. Same diagram as in Figure 2.

clearly identify for every edge the connected endpoints or to identify the inheritance hierarchies. We leave it to the reader to find out how many inheritance hierarchies are contained in both diagrams. In Section 5, Figure 17 and Figure 18, the same diagrams are given with GoVisual layout in which the number of inheritance hierarchies is clearly visible.

Figures 4(a) and 4(b) demonstrate that pure non-hierarchical visualization must fail for class diagrams. Both Figures show the same graph as in Figure 2 and have been created by one of the leading software development tools. The layout given in Figure 4(a) is a typical symmetric layout that is the result of a low budget solution and leaves the human reader uninformed about the structure of the software project. The orthogonal layout of the same diagram demonstrates, apart from the obvious technical weaknesses, that the missing hierarchical information still does not reveal clear information on the project.

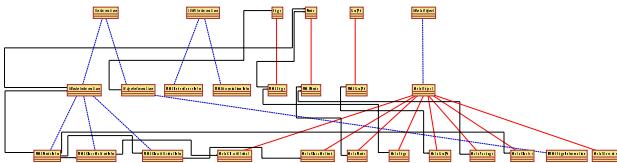


Figure 5: Seemann style layout

The weakness of strictly hierarchical or strictly non-hierarchical layout algorithms for mixed-hierarchical graphs has been recognized early by [Seemann 1997], presenting a two phase method combining Sugiyama layout [Sugiyama et al. 1981] for the inheritance hierarchies and a routing method for the undirected edges. The approach has been enhanced by [Eichelberger 2002]. The drawback of this method is immediate: routing edges within a given drawing results in hard combinatorial problems that are attacked via local optimization methods. With an increasing number of undirected edges, this approach leads to diagrams which are difficult to

read. Figure 5 shows a layout of a diagram with one inheritance hierarchy produced by this approach given in [Eichelberger 1999].

One combinatorial subproblem that occurs when drawing mixed-hierarchical graphs, namely mixed upward planarization that we shall discuss in Section 3.3, has also been considered in [Eiglsperger and Kaufmann 2001].

1.3 The GoVisual Approach

Several aspects are important when drawing class diagrams: The generalizations induce hierarchical components (*hierarchies*) of the graph, thus each hierarchy must be drawn such that all arcs run in the same direction; there are no restrictions on how the edges in the set E must be drawn. We call a drawing of G that satisfies these requirements a *mixed-upward* drawing. Moreover, the number of crossings between relationships should be small, generalizations belonging to different inheritance hierarchies should never cross, one hierarchy should not enclose another hierarchy, and the area covered by the drawing should be small.

For a clear visualization of the specific combination of hierarchical and non-hierarchical components in UML class diagrams, we put special emphasis on meeting a balanced mixture of the following aesthetic criteria:

- *Crossing minimization*
- *Bend minimization*
- *Orthogonal layout*
- *Uniform direction within each class hierarchy*: Arcs of a class hierarchy should point in a consistent direction.
- *No nesting of one class hierarchy within another*: A class hierarchy is not enclosed by a circle (in the undirected sense) of arcs of a different hierarchy.

- *Merging of multiple inheritance edges*: lines representing generalizations join prior to reaching the super class, rather than being presented as separate arcs.
- *Good edge labelling*: Labels are placed at predefined positions (beginning, end or midpoint of an edge) minimizing the overlap area.

Following these guidelines, the GoVisual approach consists of two steps. In the first step, a *mixed-upward planarized representation* is computed, i.e. a planar representation P_G of G in which edge crossings are replaced by dummy vertices of degree 4 (the two crossing edges are split and the two vertices created by the split operations are identified resulting in a vertex with four adjacent edges), and that allows for a planar drawing that is also mixed-upward. In the second step, a mixed-upward planar drawing of P_G is constructed, and the dummy vertices are replaced by edge crossings in order to obtain a drawing of G .

The rest of this paper is organized as follows. Section 2 introduces the required graph-theoretical background and Section 3 describes the GoVisual drawing algorithm. The available GoVisual diagramming software is presented in Section 4 and finally, Section 5 shows several layouts of UML class diagrams automatically generated by the GoVisual software.

2 Preliminaries

In this section, we introduce basic terms and results from graph theory and graph drawing. For a detailed discussion, please refer to the textbooks [Harary 1972] on general graph theory and [Di Battista et al. 1998] on graph drawing.

2.1 Planar Graphs and Embeddings

A graph $G = (V, E)$ is generally visualized by a *drawing* in the plane with the vertices drawn as points and the edges drawn as closed Jordan curves, connecting their incident vertices. An intersection of two edges in a drawing is called a *crossing*.

A graph $G = (V, E)$ is called *planar* if it can be drawn in the plane such that no two edges cross each other except at common endpoints. *Combinatorial* or *planar embeddings* are equivalence classes of planar drawings which can be defined by the ordering of the incident edges around each vertex. Two drawings of the same graph are called equivalent if the clockwise circular ordering of the incident edges around each vertex is the same. We say that they realize the same combinatorial embedding. A graph with a given fixed planar embedding is also called a *plane graph*. A plane graph divides the plane into regions called *faces*. A face of a plane graph is uniquely described by its surrounding edges. The one unbounded face of a plane graph is called the *exterior face*. All other faces are called *interior faces*.

Planarity of a graph $G = (V, E)$ can be tested in $O(|V|)$ time by, e.g., an approach of [Lempel et al. 1967] using the *PQ-tree* data structure presented by [Booth and Lueker 1976]. The algorithm can be extended to output a planar embedding [Chiba et al. 1985].

2.2 Upward Planar Digraphs

A *directed graph* or *digraph* G is a pair (V, A) , where V is a finite set of *vertices* and A is a finite set of *arcs*, where each arc $a \in A$ consists of an ordered pair of vertices $u, v \in V$. Ignoring for every arc the order of its vertices, we get an undirected graph that is called the *underlying graph* of G . If $a = (u, v)$ then a is an *outgoing arc* of u and an *incoming arc* of v . A *source* is a vertex with no incoming arcs and a *sink* is a vertex with no outgoing arcs. An acyclic digraph with exactly one source is called a *single source* graph. Consequently, an

acyclic digraph with exactly one sink is called a *single sink* graph. A directed acyclic graph with exactly one source s and exactly one sink t and an arc (s, t) is called an *st-digraph*.

An *upward drawing* of a directed acyclic graph G is a drawing of G such that each edge is drawn as a curve monotonically increasing in vertical direction. An *upward embedding* Γ is a representation of G that consists of the clockwise orderings of the incoming edges for every vertex with respect to an upward drawing. Usually, any drawing of a directed acyclic graph G is called upward if the edges follow a common direction. A directed acyclic graph $G = (V, A)$ is called *upward planar* if it has an upward drawing that is planar with respect to the underlying graph. An *upward planar embedding* is an upward embedding with respect to an upward planar drawing.

Upward planarity of a directed acyclic graph $G = (V, A)$ with a single source can be tested in $O(|V|)$ time by, e.g., an approach by [Bertolazzi et al. 1998] using the *SPQR-tree* data structure. The algorithm is also able to output an upward planar embedding.

2.3 Cluster Graphs and Cluster Planarity

Cluster graphs are graphs with recursive clustering structures over the vertices. A *cluster graph* $C = (G, T)$ consists of an undirected graph G and a rooted tree T such that the leaves of T are exactly the vertices of G . Each node v of T represents a *cluster* $V(v)$ of the vertices of G that are leaves of the subtree rooted at v . The tree T describes an inclusion relation between clusters and is called the *inclusion tree* of C . The graph G is called the *underlying graph* of C . $T(v)$ represents the subtree of T rooted at the node v , and $G(v)$ denotes the subgraph of G induced by the vertices in the cluster associated with node v . We define $C(v) = (G(v), T(v))$ to be the *subcluster graph* associated with node v . An edge (v, w) in which v is a vertex in $G(v)$ and w is a vertex in $G \setminus G(v)$ is said to be *incident* to cluster v .

In a *drawing* of a cluster graph $C = (G, T)$, the graph G is drawn as points and curves as usual. For each node v of T , the cluster is drawn as a simple closed region R (i.e., a region without holes) that contains the drawing of $G(v)$, such that the following three conditions hold.

- (i) The regions for all subclusters of v are completely contained in the interior of R .
- (ii) The regions for all other clusters are completely contained in the exterior of R .
- (iii) If there is an edge e between two vertices of $V(v)$ then the drawing of e is completely contained in R .

A cluster graph $C = (G, T)$ is a *c-connected cluster graph* if each cluster induces a connected subgraph of G .

The drawing of an edge e and a region R has an *edge-region crossing* if the drawing of e crosses the boundary more than once. A drawing of a cluster graph is *c-planar* if there are no edge crossings or edge-region crossings. A graph having a *c-planar* drawing is called *c-planar*. Notice that the planarity of the underlying graph does not imply the existence of a *c-planar* drawing of a cluster graph as shown in the example of Figure 6.

According to [Feng et al. 1995], *c*-planarity can be tested for *c*-connected cluster graphs in $O(|V|^2)$ time using the *PQ-tree* data structure. This result has recently been extended to a wider class of graphs in [Gutwenger et al. 2002]. Both algorithms are also able to output a *c*-planar embedding. So far, no polynomial time algorithm is known to test *c*-planarity for non *c*-connected cluster graphs.

2.4 Planarization

Crossing minimization is among the most important aesthetic criteria in graph layout. Unfortunately, no polynomial time algorithm

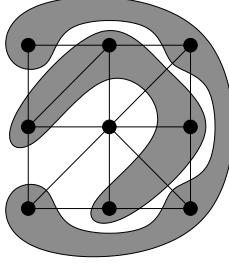


Figure 6: A planar cluster graph that is not c -planar.

is known for finding a drawing with the minimum number of crossings. The best method known for crossing minimization in practice is the planarization approach. It works in two steps:

In a first step, a planar subgraph is computed, and then, in a second step, the remaining edges are subsequently re-inserted into a combinatorial embedding of G . For a fixed combinatorial embedding, an edge can be inserted with the minimum number of crossings by computing a shortest path in the extended dual graph. Recently, [Gutwenger et al. 2001] have presented an algorithm which solves the one-edge insertion problem optimally over the set of all embeddings of G in linear time. This leads to a topological representation of the graph in which the crossings are already fixed. In order to keep the topology for the remaining steps of the drawing algorithm, the crossings are replaced by artificial vertices of degree four. The result is a so-called *planarized representation* P of G which consists of a planar graph and its planar embedding. In order to draw P , any planar drawing algorithm can be used. Planar orthogonal methods that minimize the number of bends in the final drawing typically lead to nice and readable layouts. In practice, this method usually leads to drawings with few crossings, few edge bends, and small area.

Further restrictions of planarity, such as upward or c -planarity lead to similar planarization methods. Under the restriction that upward planarity and c -planarity can only be tested for a subset of directed and clustered graphs, respectively, approaches for *upward planarization* and *cluster planarization* can be developed by using or modifying the corresponding testing algorithm.

3 The Algorithm

A mixed-hierarchical graph G is a graph $G = (V, A, E)$ consisting of an arc set A of *generalizations* and an edge set E of *associations*, such that the digraph $G_A = (V, A)$ induced by the generalizations of G is acyclic.

We generalize the notion of planarity, upward planarity and planarization to mixed-hierarchical graphs as follows: Let $G = (V, A, E)$ be a mixed-hierarchical graph. A *mixed-upward planar* drawing of G is a planar drawing of G in which the implied drawing of each connected component of G_A has the property that all arcs are drawn following the same direction. The components of G_A are called the *hierarchies* of G . An arc $(w, v) \in A$ is directed from w to v . In UML notation, the vertex v is said to be the *super class* of w and w is the *child* of v that *inherits* from v . Let $v_1, v_2, \dots, v_k \in V$, $k \geq 2$ be super classes of a vertex w . Then w *multiply inherits* from v_1, v_2, \dots, v_k and such a constellation is called *multiple inheritance*. A hierarchy that does not contain multiple inheritance is called an *inheritance tree*.

G is called *mixed-upward planar* if it admits a mixed-upward planar drawing. An *embedding* of G is a combinatorial embedding of G with a fixed exterior face. Notice that, in contrast to simple planarity, it is not possible to choose an arbitrary face as exterior face. An embedding that is realized by a mixed-upward planar drawing

is called *mixed-upward*. A planarized representation of G (seen as undirected graph) which is also mixed-upward planar is called a *mixed-upward planarized representation* of G .

Our drawing model for generalizations is shown in Figure 7. Consider a vertex v in an inheritance hierarchy with children w_1, \dots, w_k , $k \geq 2$. All arcs leading to v are joined in a single point from which a line with arrow head leads to v . Altogether, only two bends are required. The UML specifications allow to use this model of representation in UML class diagrams and many users prefer it.

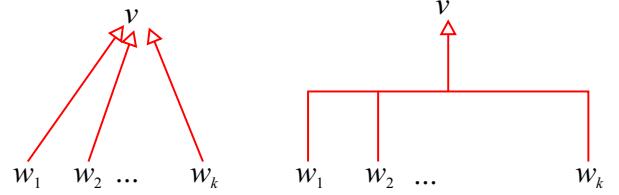


Figure 7: The drawing model for generalizations.

Let $v_1, v_2, \dots, v_k \in V$, $k \geq 2$, be super classes of a vertex w . Then different to the previous case the edges entering vertex w do not join before entering w but are kept separate. Figure 8 illustrates this situation. For each edge, at most two bends are required.

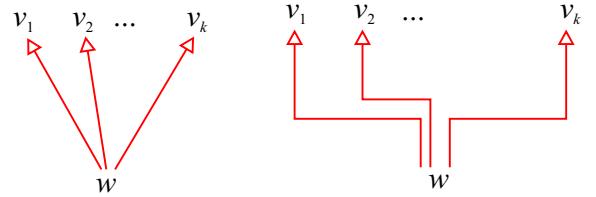


Figure 8: The drawing model for multiple inheritance generalizations.

Two hierarchies H_1 and H_2 in an embedding Γ of a planarized representation P_G of G might be nested in the following sense. Let \tilde{H}_1 denote the subgraph of P_G induced by all arcs and edges in P_G whose corresponding arc in G is incident only to vertices in H_1 , and define \tilde{H}_2 analogously for H_2 . We say that H_2 is *nested* in H_1 in Γ if there is an undirected cycle c formed by arcs of \tilde{H}_1 such that \tilde{H}_2 is in the interior of c in Γ .

3.1 GoVisual Drawing Framework for UML Class Diagrams

In this section, we first give an overview on the strategy of the layout algorithm for UML class diagrams. Consider a mixed-hierarchical graph $G = (V, A, E)$, that represents a UML class diagram. The GoVisual drawing framework follows a strategy as it is described in Section 2.4: It first planarizes a graph G , computing a mixed-upward planarized representation and then, in a second step, computes the orthogonal layout of the mixed-hierarchical graph G .

The computation of the mixed-upward planarized representation is the more interesting part of the layout algorithm. The second step performing the computation of the orthogonal layout is more tedious due to complex implementational details, and we therefore give more room to the first part. The computation of the mixed-upward planarized representation is sketched in the algorithm *Topological Embed*, see Algorithm 1.

The algorithm *Topological Embed* performs three main tasks. First, the graph G is preprocessed in order to transform it into a mixed-hierarchical graph G' that meets our requirements for the drawing model. This step is described in detail in Section 3.2. In

Algorithm 1: Topological Embed

Input : Mixed-Hierarchical Graph $G = (V, A, E)$

Output : Mixed Upward Embedding Γ of G

Preprocess G resulting in graph G' ;

for every hierarchy $H = (V_H, A_H)$ **do**

 Compute an upward planarized representation P_H of H ;

 Construct *st*-graph P_H^{st} from P_H ;

 Add all remaining associations between vertices of V_H to P_H^{st} , constructing the final upward planarized representation of H ;

end

Compute a cluster planarized representation Γ of G' using P_H^{st} for every hierarchy H , guaranteeing that the cluster planarized representation induces an upward planarized representation for every hierarchy;

Remove all dummy edges from the planarized representation Γ of G' ;

return Γ ;

the second and third main steps of Topological Embed, a mixed-upward planarized representation of G' is computed. Apart from the minimization of the edge crossings, this computation must meet two requirements: The hierarchies need to be directed in uniform direction and nesting of hierarchies is not allowed. Thus GoVisual subdivides the planarization of the graph G' into two steps. First, GoVisual computes for every hierarchy an upward planarized embedding to ensure a uniform direction. Second, the graph is clustered, gathering every hierarchy in its own cluster, in order to ensure that there will be no nesting of hierarchies in the subsequent planarization step. Each of the two steps is described in detail in Sections 3.3 and 3.4.

3.2 Preprocessing

The drawing model presented at the beginning of this section alters the graph that has to be drawn. We can easily find an example, where its application leads to a crossing although the embedding of the graph is mixed-upward planar (see Figure 9(a)). This means that the mixed-upward planarized representation produced in the planarization step does not define the number of crossings in the final drawing, further crossings might occur.

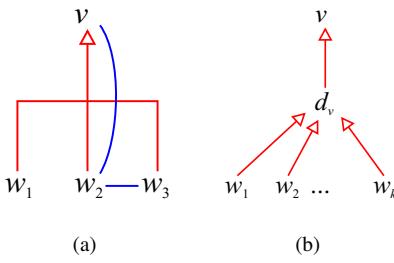


Figure 9: Drawbacks of the drawing model for generalizations and the modifications made in the preprocessing step. Red edges indicate generalizations, and blue edges associations.

Therefore, we also alter the graph accordingly in a preprocessing step such that the number of crossing vertices in the planarized representation is in fact the number of crossings in the final drawing (compare Figure 9(b)). For each vertex v in a class hierarchy with at least 2 children w_1, \dots, w_k , we introduce a new vertex d_v

representing the point where the edges from the children to v are joined. Each edge (w_i, v) is replaced by a new edge (w_i, d_v) , and a new edge (d_v, v) is added. We call the inserted vertices d_v *generalization mergers* and the graph resulting from the preprocessing step $G' = (V', A', E)$.

3.3 Upward Planarization

Upward planarization is performed by Topological Embed on every hierarchy to ensure a uniform direction for this hierarchy. Each hierarchy $H = (V_H, A_H)$ of G represents a directed graph. In order to compute an upward planarized representation P_H of H , several techniques can be applied.

If H has only a single sink (base class), the upward planarity test for directed acyclic graphs by [Bertolazzi et al. 1998] can be applied. If H is upward planar, the algorithm also computes an upward planar embedding of H in time $O(|V_H|)$. In this case, no crossings are necessary at all. A straightforward extension of the test for graphs with several sinks is to introduce a super-sink t and to add arcs from all sinks of H to t . This is equivalent to demand that all sinks must be on the exterior face, which makes sense for UML class diagrams. On the other hand, the general problem of upward planarity testing is NP-complete (see [Garg and Tamassia 1995]). Figure 10 shows a hierarchy H represented by solid red edges. The hierarchy H has a single sink and is obviously upward planar, thus H and P_H are identical.

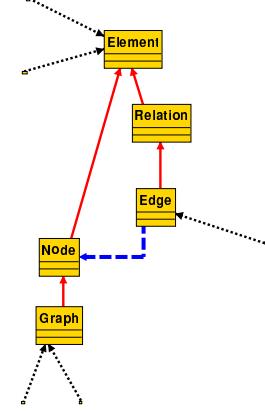


Figure 10: The hierarchy H is shown by red edges, associations between nodes of V_H shown as dashed blue lines. Associations between V_H and $V \setminus V_H$ are shown as dotted black lines. H and P_H are identical.

If the upward planarity test fails, techniques have to be applied that replace edge crossings with dummy vertices. Possible approaches are to adapt the crossing minimization step of the Sugiyama algorithm [Sugiyama et al. 1981; Eades and Kelly 1986; Gansner et al. 1993; Jünger and Mutzel 1996], or to use the technique described in [Eiglsperger and Kaufmann 2001].

Since every upward planar embedding is a subgraph of a planar *st*-digraph, we can augment the resulting upward planarized representation P_H to a planar *st*-digraph P_H^{st} . Finally, we reinsert the remaining associations between vertices in V_H using a standard technique as described in [Battista et al. 1999] or the SPQR-tree based optimal algorithm in [Gutwenger et al. 2001].

Figure 11 shows P_H^{st} for the example of Figure 10. The *st*-graph P_H^{st} is presented by blue and solid green edges. The green edges are the original edges from H . To construct the *st*-graph we use either existing associations between nodes of V_H shown as dashed blue lines or add extra edges shown as dotted blue lines. Here we use one association shown as blue edge between the nodes "Node" and

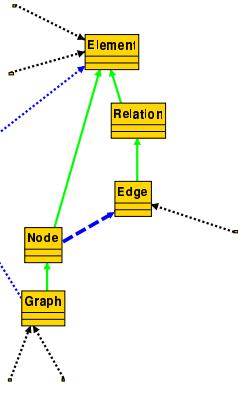


Figure 11: P_H^st for the example of Figure 10. P_H^st is shown by green and blue edges.

"Edge", and add an extra edge between "Element" and "Graph". Until the computation of the mixed upward planar embedding is finished, the association ("Node", "Edge") is considered to be directed.

3.4 Computation of Cluster Planarized Representations

In order to avoid nesting of hierarchies, we place each hierarchy into a cluster and apply the cluster planarization algorithm. By the previous step described in Section 3.3 every cluster is already planar. Figure 12 shows the example of Figure 11 with P_H^st defining a cluster.

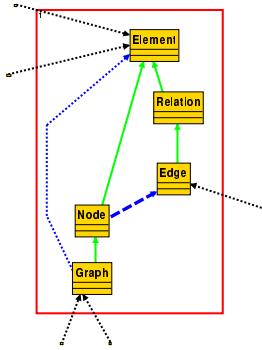


Figure 12: P_H^st induces a cluster.

For each cluster, we compute the planar embedding of the corresponding P_H^st and construct the wheel graph which represents all possible permutations of the vertices on the exterior face of a cluster (see [Feng et al. 1995]). We make sure that this wheel graph corresponds to an upward planar embedding of P_H^st . These wheel graphs are connected with the remaining edges between vertices of different clusters, and a planar subgraph of the resulting graph is computed which contains all edges in the wheel graphs and a set F of edges connecting vertices in different clusters. Since it is essential to have all wheel graph edges in the planar subgraph, we cannot apply the PQ-based algorithm (see [Jünger et al. 1998]). Instead, we use the algorithm based on iterative planarity testing. In the next step, we construct a cluster planar embedding of the graph consisting of all the graphs C_i and the edges in F .

Finally, we re-insert the edges between vertices in different clusters that are not contained in F . This can be done as described

in [Di Battista et al. 2002]. The resulting cluster planarized graph has been constructed such that all hierarchies are embedded upward planarized and nesting of hierarchies is avoided. It remains to remove the dummy edges that do not correspond to associations and generalizations which have been introduced to construct st -graphs of the upward planarized hierarchies.

3.5 The Orthogonal Drawing Step

For simplicity, we restrict ourselves to single-inheritance hierarchies. We apply the *topology-shape-metrics* approach as proposed in [Tamassia et al. 1988]. The previous steps described in Sections 3.1–3.4 give us the topology of the drawing, i.e., an embedded mixed-upward planarized representation Γ of graph G' . Since each hierarchy is a tree, no two generalizations cross and all the generalizations from a super class v to its children w_1, \dots, w_k can be drawn with at most two bends as shown in Figure 7.

The shape of the drawing is determined as follows. First, Γ is transformed into a 4-graph Γ' by replacing each high-degree vertex v with a face f_v (that we call a *cage*) such that $\deg(v) = \deg(f_v)$ as described in [Klau and Mutzel 1998] (see also Figure 13). Then we apply the bend-minimization algorithm of Tamassia [Tamassia 1987] as described in [Klau and Mutzel 1998] or the Giotto algorithm as described in [Tamassia et al. 1988] with certain degree and bend constraints. These constraints guarantee that generalizations are always drawn as in the model described at the beginning of this section and that cages have rectangular shape.

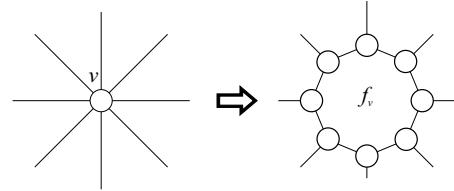


Figure 13: Transformation of high-degree vertices.

The compaction phase computes the metrics of the drawing, i.e., the final coordinates. Constructive flow-based compaction (see [Klau et al. 2001]) is used to get a first drawing of the expanded graph Γ' . In this step, we have to make sure that each cage is large enough so that the real vertex can be placed into it. Then, Γ' is re-transformed into Γ by placing each high-degree vertex into its cage and routing the adjacent edges in an orthogonal fashion within the cage, see Figure 14. Finally, flow-based improvement compaction is applied in order to reduce the size of the drawing and to remove unnecessary bends introduced during the edge routing step.

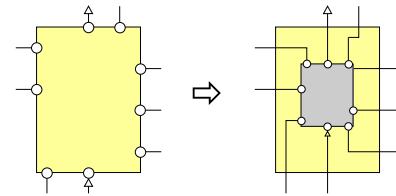


Figure 14: Placement of high-degree vertices and routing of adjacent edges.

4 The Software

The orthogonal UML layout algorithm presented in this chapter is integrated into a large framework of layout algorithms and data

structures for the automatic layout of diagrams, called GoVisual®. GoVisual is an object-oriented C++ class library.

The GoVisual C++ class library is an independent library based on its own fast and efficient data structures and combinatorial algorithms. It currently provides the layout and labelling styles as listed below. All layout styles provide a set of drawing and optimization algorithms that can be combined to a layout algorithm that provides the best results for the user within the chosen style. Furthermore a variant of parameters can be manipulated within each style.

- Orthogonal Layout for UML Class Diagrams as described in this paper.
- Orthogonal Cluster Layout providing an orthogonal layout for cluster graphs that focuses on the crossing minimization between edges and the minimization of the number of bends of the edges (see e.g. [Di Battista et al. 2002]).
- Orthogonal Layout having a focus on the minimization of crossings between edges and the minimization of the number of bends of the edges.
- Tree Layout for the visualization of non-circular structures.
- Hierarchical Layout visualizing data that is meant to reflect the existence of a prioritised ranking system within the different elements.
- Symmetric Layout for displaying hidden symmetries within a diagram.
- Circular Layout for network visualization.
- Advanced Labelling offering a method to automatically place labels of edges, respecting the size and the type of the labels as well as the desired position at the edge and maximizing the overlap free area.
- Labelling placing labels of connectors either at one of the two ends of the connector or at its midpoint.

Contact and further information about GoVisual Software can be found at:

<http://www.oreas.com>

4.1 GoVisual API

The GoVisual API enables the user to access the layout algorithms within a C++, Java (using Java Native Interface JNI) or Microsoft .NET environment and to integrate them into their own applications. To access a layout algorithm, the user only needs to create a data structure that contains information on the relationships of the diagram. After successfully applying a layout algorithm, the layout information of the diagram is stored in the data structure.

The Software is currently available for the platforms Windows 95, 98, ME, NT, 2000, XP. Versions for Unix and Linux systems are in preparation.

4.2 GoVisual Plug-Ins

GoVisual is also available as Plug-In for Borland® Together® ControlCenter™, Gentleware Poseidon for UML, and Microsoft® Visio® 2000/2002. Both the ControlCenter and Poseidon are model-driven development tools for object oriented software projects.

After installation of a GoVisual Plug-In, a new GoVisual menu entry appears within the application that allows to apply an appropriate layout algorithm for laying out the current diagram. As in the GoVisual API, the user is able to combine drawing and optimization

algorithms in order to achieve the best results. We decided to use sliders that allow to choose between different optimization goals since users tend to ignore complex explanations on advantages and drawbacks on the different methods.

Figure 15 shows the layout menu as well as the options menu with a typical user friendly slide bar to choose between different optimization algorithms.

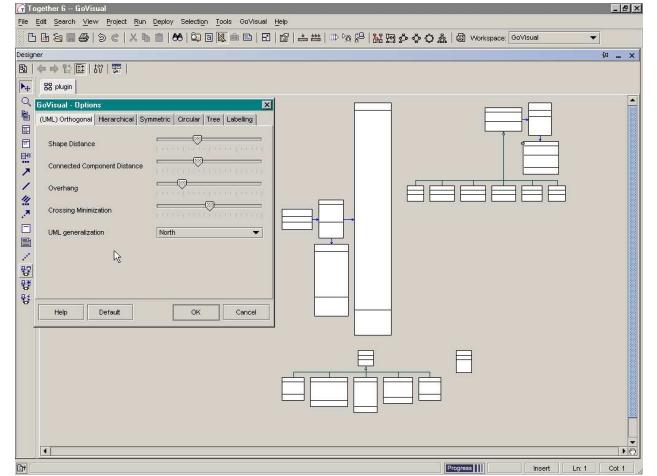


Figure 15: GoVisual within Borland Together ControlCenter.

5 Examples

This section shows different UML diagrams that have been laid out using GoVisual. Each layout has been computed within less than a second on a standard PC.

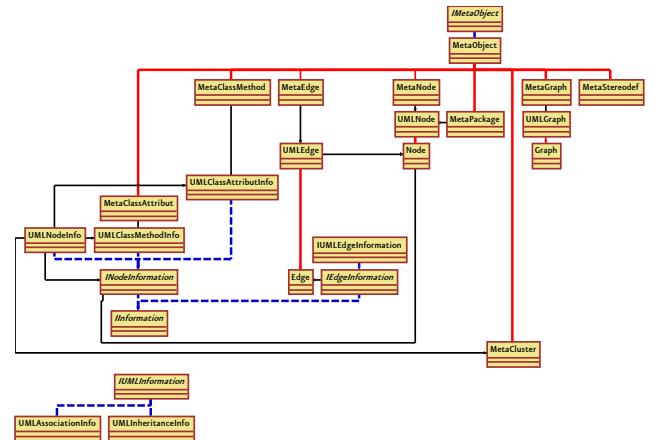


Figure 16: GoVisual UML Layout of the example shown in Figure 5.

Figure 16 gives a GoVisual layout of the diagram that was presented in Figure 5 to demonstrate the state of the art layout by [Seemann 1997]. The GoVisual layout contains no crossings at all (compared to 59 crossings in Figure 5) and seven bends in associations (compared to 46 bends in associations in Figure 5).

Figure 17 gives a GoVisual Layout of a UML class diagram that we already showed in Figures 2, 4(a), and 4(b). While it was impossible to understand the structure of the software project presented

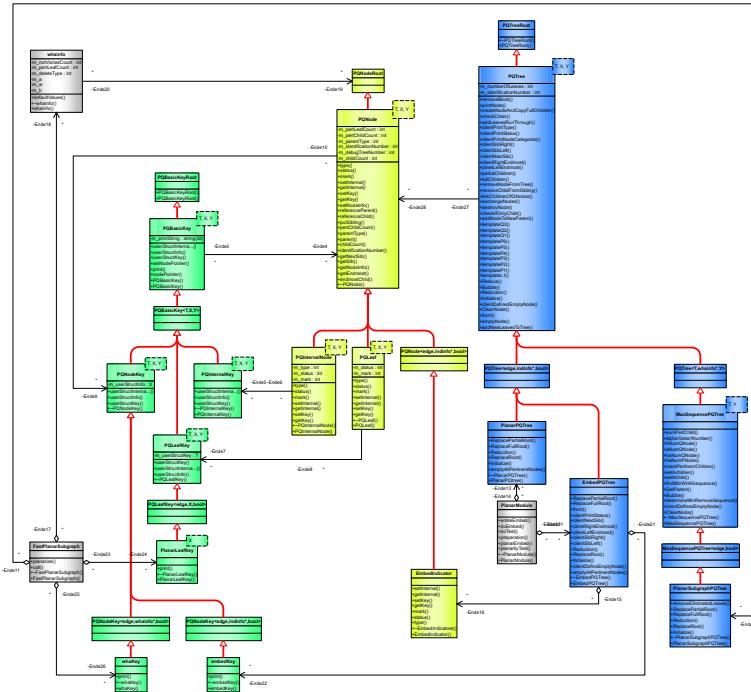


Figure 17: GoVisual UML Layout of the example shown in Figure 2, 4(a), and 4(b).

by the diagram in any of the industrial layouts, it becomes immediately clear by the GoVisual layout. The human reader immediately recognizes three inheritance hierarchies and the understanding of the interactions between these hierarchies is supported by just one crossing.

Both examples in Figures 17 and 18 have been drawn with the GoVisual Plug-In for Microsoft Visio. Figure 18 shows the example displayed within the Visio environment.

References

- BATTISTA, G. D., EADES, P., TAMASSIA, R., AND TOLLIS, I. 1999. *Graph Drawing*. Prentice Hall.

BERTOLAZZI, P., DI BATTISTA, G., MANNINO, C., AND TAMASSIA, R. 1998. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing* 27, 1, 132–169.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *Unified Modeling Language User Guide*. Addison Wesley Longman.

BOOTH, K., AND LUEKER, G. 1976. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences* 13, 335–379.

CHIBA, N., NISHIZEKI, T., ABE, S., AND OZAWA, T. 1985. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences* 30, 54–76.

DI BATTISTA, G., EADES, P., TAMASSIA, R., AND TOLLIS, I. G. 1998. *Graph Drawing*. Prentice Hall.

DI BATTISTA, G., DIDIMO, W., AND MARCANDALLI, A. 2002. Planarization of clustered graphs. In *Graph Drawing (Proc. GD '01)*, Springer-Verlag, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 60–74.

EADES, P., AND KELLY, D. 1986. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria* 21, A, 89–98.

- EICHELBERGER, H. 1999. *Entwicklung eines Frameworks zum automatischen Zeichnen von Software-Entwurfsdiagrammen*. Master's thesis, Universität Würzburg.
- EICHELBERGER, H. 2002. SugiBib. In *Graph Drawing (Proc. GD '01)*, Springer-Verlag, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265 of *Lecture Notes in Computer Science*, 467–468.
- EIGLSPERGER, M., AND KAUFMANN, M. 2001. An approach for mixed upward planarization. In *Proc. 7th International Workshop on Algorithms and Data Structures (WADS'01)*, Springer Verlag, vol. 2125 of *Lecture Notes in Computer Science*, 352–364.
- FENG, Q.-W., COHEN, R. F., AND EADES, P. 1995. Planarity for clustered graphs. In *Algorithms – ESA '95*, Springer Verlag, P. Spirakis, Ed., vol. 979 of *Lecture Notes in Computer Science*, 213–226.
- GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND VO, K. P. 1993. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.* 19, 3, 214–230.
- GARG, A., AND TAMASSIA, R. 1995. On the computational complexity of upward and rectilinear planarity testing. In *Graph Drawing (Proc. GD '94)*, Springer-Verlag, R. Tamassia and I. G. Tollis, Eds., vol. 894 of *LNCS*, 286–297.
- GUTWENGER, C., MUTZEL, P., AND WEISKIRCHER, R. 2001. Inserting an edge into a planar graph. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*, ACM Press, 246–255.
- GUTWENGER, C., JÜNGER, M., LEIPERT, S., MUTZEL, P., PERCAN, M., AND WEISKIRCHER, R. 2002. Advances in c-planarity testing of clustered graphs. In *Graph Drawing (Proc. 2002)*, Springer-Verlag, M. Goodrich and S. Kobourov, Eds., vol. 2528 of *Lecture Notes in Computer Science*, 220–235.
- HARARY, F. 1972. *Graph Theory*. Addison Wesley.
- JÜNGER, M., AND MUTZEL, P. 1996. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications (JGAA)* (<http://www.cs.brown.edu/publications/jgaa/>) 1, 1, 1–25.
- JÜNGER, M., LEIPERT, S., AND MUTZEL, P. 1998. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design* 17, 7, 609–612.
- KLAU, G. W., AND MUTZEL, P. 1998. Quasi-orthogonal drawing of planar graphs. Tech. Rep. MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken.
- KLAU, G. W., KLEIN, K., AND MUTZEL, P. 2001. An experimental comparison of orthogonal compaction algorithms. In *Graph Drawing (Proc. GD 2000)*, Springer, J. Marks, Ed., vol. 1984 of *LNCS*, 37–51.
- LEMPEL, A., EVEN, S., AND CEDERBAUM, I. 1967. An algorithm for planarity testing of graphs. In *Theory of Graphs: International Symposium: Rome, July 1966*, Gordon and Breach, New York, 215–232.
- PURCHASE, H., ALLDER, J.-A., AND CARRINGTON, D. 2001. User preference of graph layout aesthetics: A UML study. In *Graph Drawing (Proc. GD 2000)*, Springer-Verlag, J. Marks, Ed., vol. 1984 of *LNCS*, 5–18.
- RATIONAL SOFTWARE CORPORATION, 2002. Rational Rose, Rational XDE.
- SEEMANN, J. 1997. Extending the sugiyama algorithm for drawing UML class diagrams. In *Graph Drawing (Proc. GD '97)*, Springer-Verlag, G. Di Battista, Ed., vol. 1353 of *LNCS*, 415–424.
- SUGIYAMA, K., TAGAWA, S., AND TODA, M. 1981. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern. SMC-11*, 2, 109–125.
- TAMASSIA, R., BATTISTA, G. D., AND BATINI, C. 1988. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern. SMC-18*, 1, 61–79.
- TAMASSIA, R. 1987. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.* 16, 3, 421–444.
- TOGETHERSOFT CORPORATION, 2002. Together ControlCenter.