Graph Abstractions
as the basis of an Extensible Graph Editing Tool

Niels C. Larsen, Martin K. Molz and Kurt Nørmark
Aalborg University
Denmark*

March 10, 1998

Abstract

A wide variety of computer tools are based on graphs. In many modern applications, graphs also play an important role in the user interfaces of the tools. Although such tools, in a foundational sense, are based on the mathematical graph concept, it is frequently the case that a richer and more practical graph concept is needed for user interface purposes. The richness comes from several sources, but the variation in the ways the edges are rendered is of particular importance. In this paper we will demonstrate that a variety of relationships among nodes (one-to-one through many-to-many) can be understood as composite edges, the parts of which are particular kinds of subgraphs of the overall graph. We will also demonstrate how composite edges can be structured and managed in terms of two design patterns known from the area of object-oriented design. The paper is organized in two parts, the first of which lays the mathematical foundation of the latter.

1 Introduction and Motivation

Graphs play an important role in many different computer usages. Flow charts, state machines, entity relationship diagrams, call trees, and inheritance hierarchies are just a few examples of graphs used in the area of computer science. Similar examples could be drawn from a wide variety of other domains.

More or less specialized graph manipulation facilities are part of a great number of modern computer tools. One of the main ideas behind the work in this paper is to envision a *generic graph processing tool*, which can be used whenever there is a need to deal with graphs on a computer. A similar idea has found widespread use in the area of text processing, where Emacs [10, 11] is widely used for nearly any kind of plain text manipulation on UNIX systems.

Emacs is indeed a successful example of a generic text editing tool. In this work we present the key ideas behind Ginger[1] [5], an interactive graph processing tool which has been designed with some

* Department of Computer Science, Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark. Contact email: normark@cs.auc.dk, WWW: http://www.cs.auc.dk/∼normark/Ginger

[1] Ginger is an acronym for the circular definition "Ginger is an INteractive Graph EditoR".

inspiration from Emacs. The development of the Ginger prototype is the experimental foundation of the ideas presented in this paper.

It is not straightforward to design a generic graph processing tool which can be used whenever the need of manipulating a graph exists. It is not the ambition of this paper to come up with a complete design of such a tool, but we think our contributions can help solve part of the problem.

The following aspects seem to be important for a generic graph processing tool:

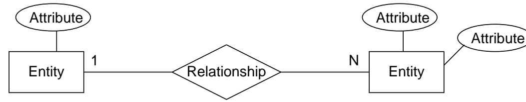
- **Internal graph conceptualization**: Support of an open-ended set of graph concepts, including node and edge concepts.
- **External graph conceptualization**: Flexible mechanisms to support variations in the visual appearance of nodes and edges on the screen and other output media.
- **Algorithmic extensibility**: Extensible functionality at the level of graph algorithms.
- **Standard interfacing**: Well-developed interfaces both towards the human user (interactive interface) and towards other programs (external graph representation, programmatic interface) which have the need to present and manipulate information represented as a graph.

Although the plain graph concepts are simple and straightforward, there exists a great number of variations of graph concepts, which necessarily have a profound influence on the supporting computer tools: Directed and undirected graphs (cyclic and acyclic), weighted graphs, and hypergraphs are all examples of such variations. Moreover and even more important, it seems to be the case that specialized graph concepts are created, depending on the context of use. As examples of the latter we may mention layered graphs (graphs in which nodes are graphs) and hypertext graphs (graphs in which edges are anchored somehow in the nodes interior).

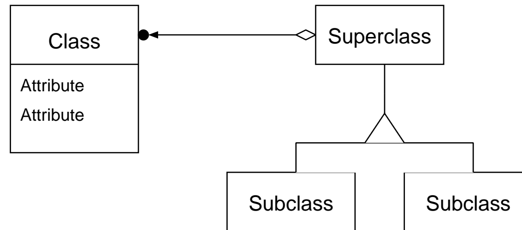
It would be attractive to support a graph concept (including node and edge concepts) which can be specialized to subsume the examples from above, and others as well. Concept formation, along the lines of object-oriented modelling, seems to be well-suited for this purpose. This has, indeed, been proposed by several other authors in the area [4, 7].

The visual properties of graphs cannot be underestimated. It is undoubtedly fair to say that the basic appeal of graphs stems from the traditional, visual presentation of a graph in terms of circles or rectangles representing nodes, and lines or arrows representing edges. The mathematical graph model serves as the foundation, but graphs are most often identified with their visual presentation. In particular, this is true when it comes to the more practical applications of graphs as supported by computer tools.

The number of variations of different renderings of nodes and edges are overwhelming. Figure 1 shows some typical examples drawn from the area of software engineering. The relationship between the two entities in figure 1(a) can be seen in at least two different ways when interpreted in terms of nodes and edges: The diamond can be seen as a node connected by two edges to the entity nodes, or the diamond can be considered simply as a decoration on the edge between the two entities. We prefer the latter view, because it is coherent with the logical content of the diagram, ie. the existence of one logical relationship between the two entities. Similar considerations apply to the edges (lines) connecting the nodes (class icons) in the OMT diagram.



(a) Prototypical nodes and edges in an entity/relationship diagram.



(b) Nodes and edges representing classes and their relationships in the OMT notation [9]

Figure 1: Example nodes and edges from two different graphical notations.

As a simple approach one may attempt to deal with the numerous variations in node and edge decorations via a number of node and edge attributes such as shape, size, color, labels, line style etc. The problem with this idea is that the list of desireable decorations and gadgets is never-ending.

In this paper we will focus on the subproblem of edge rendering, exploring an alternative approach. The basic idea promoted in this paper is to consider complicated edges as a kind of subgraphs or composite edges, in which, for instance, bending points are invisible nodes, and in which decorations such as the diamond mentioned above are visible nodes of the subgraphs. A central part of the design of composite edges concerns the interaction and coordination between the whole and its parts. We present a solution to this problem, which is inspired from the design patterns known as Composite and Observer [3].

Given appropriate graph concepts and ways to visualize instances of these leads us quite naturally to the question of the functionality on the graphs. It should be obvious that flexible extensibility at the algorithmic level is a much wanted property of a graph manipulation component. An *extension language* on top of an efficient kernel (centered around an object-oriented realization of the basic graph concepts) is probably an attractive solution. In fact, this is the most concrete inspiration from Emacs on our work with Ginger. In Emacs, a Lisp dialect is the extension language on top of a kernel implemented in a more conventional and efficient programming language (C). In section 4 of this paper we will briefly touch on our experience with Scheme [1] and an object-oriented package on top of Scheme, Meroon [8], for this purpose.

In the rest of the paper we will focus on graph concepts, both at the internal representation level and with respect to the external appearance of the graphs in the user interface. In section 2 we discuss graph abstraction in general. This is used as a foundation of graph presention with substantial variations on the rendering of edges. In section 2 we end up with a distinction between two categories of abstractions, called logical abstraction and visual abstraction. In section 3 we propose a concrete way

of implementing the abstractions from the earlier section. Finally, in section 4 we describe the Ginger graph tool, in which we have tested the ideas described in this paper.

2 Graphs and graph abstractions

In this section we will develop a number of graph abstractions that turn out to be useful in dealing with graphs like the ones in figure 1. Thus, our main interest in this section is graphs with edges that are more complicated than just a tuple of two nodes and illustrated as an arrow or a straight line segment. In addition our developments are useful for understanding the graph compacting facilities found in several graph editing tools, such as daVinci [2]. We start with a brief presentation of the mathematical graph and hypergraph concepts.

2.1 Mathematical Graphs and Hypergraphs

In mathematical graph theory, a graph is defined as follows:

Graph. A graph G is a pair (X, U) where X is a set of nodes and U is a family[2] of edges $e = (x, y)$ in $X \times X$.

If $G = (X, E)$ is a graph and $e = (x, y)$ is an edge in E, x will be called the *initial ending point*, and y the *terminal ending point* of e. These terms directly support the notion of a directed graph; undirected graphs may be seen as a special case in which we disregard the roles of the ending points.

In many practical situations we deal with graphs in which some edges are attached to more than two nodes. The fork-shaped edge in the inheritance relationship between the superclass and its two subclasses in figure 1(b) is an example of such an edge. Often, and also in this particular example, such graphs can be seen as a notational convenience, where an edge with more than two ending points stands for two or more conventional edges. The mathematical counterpart to graphs like these is called hypergraphs.

Hypergraph. A hypergraph G is a pair (X, U) where X is a set of nodes and U is a family of tuples from the family of cartesian products $X \times X \times ... \times X$.[3] An element from U is called a hyperedge.

Hypergraphs can be seen as a generalization of graphs.

2.2 An application-oriented graph concept

We wish to capture the mathematical hypergraph concept in a practical graph concept which will found the base of our later work with a graph drawing tool. Since hypergraphs are a generalization of

[2] We use the term “family” in the meaning of a set in which duplication of elements is allowed.

[3] This definition implies that the tuples in U may have different degree.

"ordinary" graphs, the latter will be automatically supported when the practical graph concept covers hypergraphs. Thus, when we use the term "edge" and "graph" in the following, we implicitly refer to hyperedges and hypergraphs, respectively.

It turns out to be primarily the edge concept that needs special attention in the development of a practical graph concept; hence we will focus on edges in this section. From a computer science perspective it is natural to regard edges as objects with their own identity. In other words, an edge becomes a first class object, contrary to the mathematical definition above, in which edges and hyperedges were defined as tuples of nodes. The understanding of an edge as a first class object is important in a practical graph tool because – among other reasons – it allows us to equip edges with attributes and operations.

Edge objects have the ability to connect a number of node objects. This ability can be captured by letting each edge object hold two lists of references to node objects: a list of initial endpoints and a list of terminal endpoints. With this understanding of edges we can talk about a *directed hypergraph*. Note that since a reference to a particular node may appear in both lists, a node may play the role as both an initial and a terminal endpoint of the edge. The *degree of an edge* is defined as the total number of endpoints of an edge, initial as well as terminal.

The mathematical graph concept is supported by letting edge objects hold exactly one node reference in the list of initial endpoints and one node reference in the list of terminal endpoints. We refer to such edges as *conventional edges*. A conventional edge is always of degree two. The mathematical (undirected) hypergraph concept is supported by disregarding the separation of endpoint references in two lists, i.e. by forming the concatenation of the two endpoint lists.

The edge concept presented in this section places no restrictions on the number of node references in the endpoint lists of an edge. That freedom leaves room for a number of special cases, some of which proves to be useful later on. One of these special cases is an edge holding two empty endpoint lists; we refer to such an edge as a *separate edge*. Another special case is an edge with node references in only one of the endpoint lists, i.e. an edge with only initial or only terminal endpoints. We refer to such an edge as an *unconnected edge*. These kinds of edges are useful as temporary representations of edges, which we change in an interactive editing process. In addition they play an important role in section 2.4 on graph abstractions.

It is convenient to define a textual as well as a graphical notation for edge objects. We use the notation $e = ([x_1, x_2, ..., x_n], [x_{n+1}, x_{n+2}, ..., x_m])$ as a textual representation of the edge e in which the initial endpoint list contains the node references $x_1, x_2, ..., x_n$, while the terminal endpoint list contains $x_{n+1}, x_{n+2}, ..., x_m$.

Conventional edges are usually drawn as arrows pointing from one node to another. We can extend this basic graphical notation to support arbitrary edges by allowing separate and unconnected edges to be drawn as edges with "dangling" endpoints and by graphically composing (hyper)edges of a number of conventional edges. Figure 2 illustrates a simple, canonical graphical notation[4] of (hyper)graphs based on that idea. Often we want to settle on a richer or more specialized notation, however. This will be the theme of section 3 of this paper.

[4]This figure has been produced using the Ginger tool. As such, the figure represents a kind of a meta application of the ideas presented in this paper. In section 3.4 of this paper we explain in details, how such edges are produced and maintained using Ginger. Subsequent figures produced directly by Ginger are marked with (Figure: Ginger).

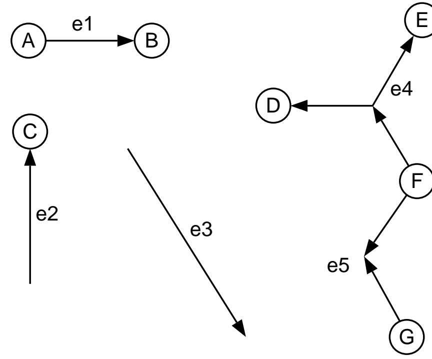


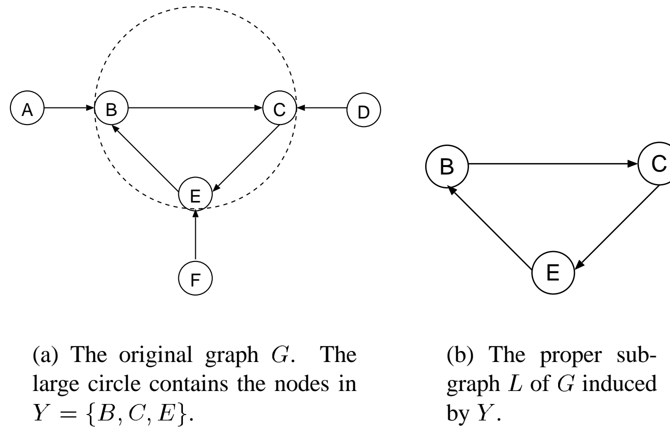
Figure 2: Visualization of a graph G, with a set of nodes $X = \{A, B, C, D, E, F, G\}$ and edges $U = \{e_1, e_2, e_3 e_4, e_5\}$, where $e_1 = ([A], [B]), e_2 = ([\,], [C]), e_3 = ([\,], [\,]), e_4 = ([F], [D, E])$, and $e_5 = ([F, G], [\,])$. (Figure: Ginger).

2.3 Subgraphs

In order to identify "a subset of a graph", which is a candidate for graph abstraction, we need the concept of a subgraph. The definition applies to graphs as well as hypergraphs.

The proper subgraph induced by a set of nodes Y. Let Y be a subset of X in the (hyper)graph $G = (X, U)$. $L = (Y, V)$ is a proper subgraph of G if V is the family of (hyper)edges in U for which *all* ending points (initial as well as terminal) belong to Y.

Thus, a subgraph L of G disregards all edges that involve nodes outside the set of nodes, which induces L. Figure 3 gives an example of a proper subgraph.



(a) The original graph G. The large circle contains the nodes in $Y = \{B, C, E\}$.

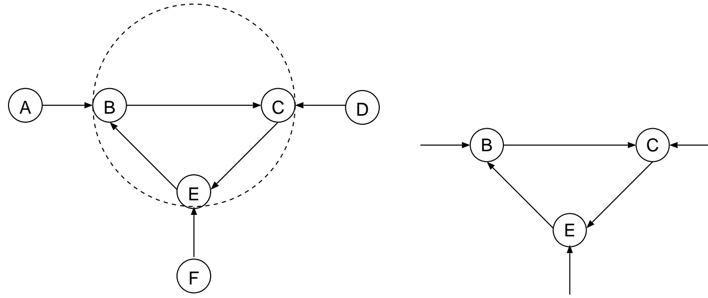
(b) The proper subgraph L of G induced by Y.

Figure 3: A graph G and a proper subgraph induced by $\{B, C, E\}$.

It is also possible to define a subgraph in which unconnected edges are allowed. We refer to this kind of subgraph as a *relaxed subgraph.*

The relaxed subgraph induced by a set of nodes Y. Let Y be a subset of X in the (hyper)graph $G = (X, U)$. The graph $L = (Y, V)$ is a relaxed subgraph of G if V is the largest subset of U in which every edge v has *at least one* ending point in Y.

If the nodes in Y are connected to the nodes in X (via one or more edge in U) there will be at least one unconnected edge in V. Figure 4 illustrates the concept of a relaxed subgraph.



(a) The original graph G. The large circle contains the nodes in $Y = \{B, C, E\}$.

(b) The relaxed subgraph L of G induced by Y. Notice the three edges of degree 1 which distinguish the relaxed subgraph from the proper subgraph shown in figure 3(b).

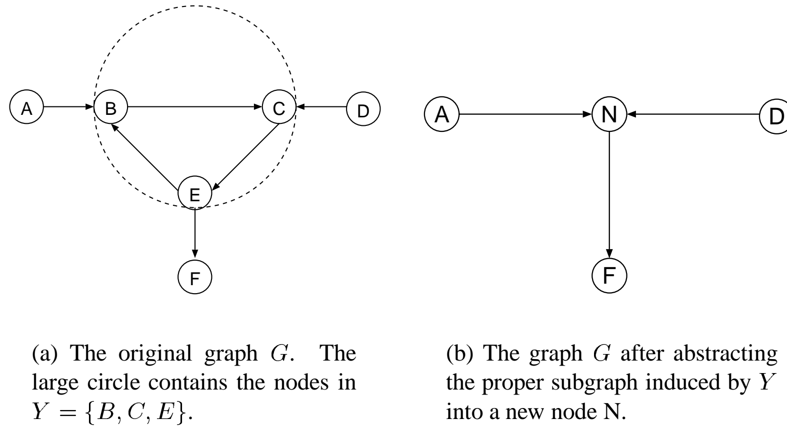
Figure 4: A graph G and a relaxed subgraph L of G.

2.4 Subgraph abstractions

It is possible to abstract a proper subgraph $L = (Y, V)$ of $G = (X, U)$ into a new node N of G. This is illustrated in figure 5. From a transformation point of view the subgraph L is substituted by the node N. The edges in $U \setminus V$ with an ending point in Y get N as the ending point in the abstracted graph. Thus, every edge entering or leaving a node in the subgraph L is connected to the new node N in the abstracted graph.

In a similar way it is possible to abstract a relaxed subgraph $L = (Y, V)$ of $G = (X, U)$ to a new edge e of G. In general, e becomes a hyperedge with ending points in $X \setminus Y$. The degree of e depends on the number of nodes in $X \setminus Y$ connected[5] to nodes in Y via edges in U. Figure 6 illustrate a typical abstraction process in which e becomes a hyperedge of degree 3. In figure figures 7 we illustrate a less typical abstraction process in which the degree of the resulting edge is one. In case there are no edges in U which connect nodes in X and Y e becomes an edge of zero degree.

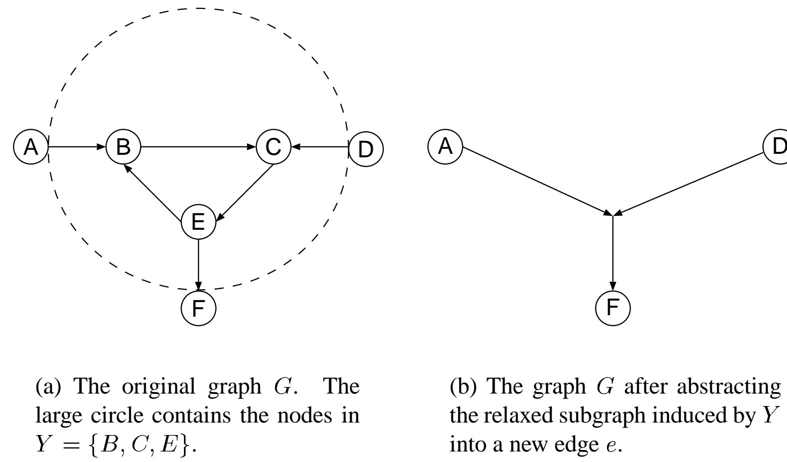
[5]A node x is *connected* to y if there is an edge v in which x is an initial ending point of v and y is a terminal ending point of v, or vice versa.



(a) The original graph G. The large circle contains the nodes in $Y = \{B, C, E\}$.

(b) The graph G after abstracting the proper subgraph induced by Y into a new node N.

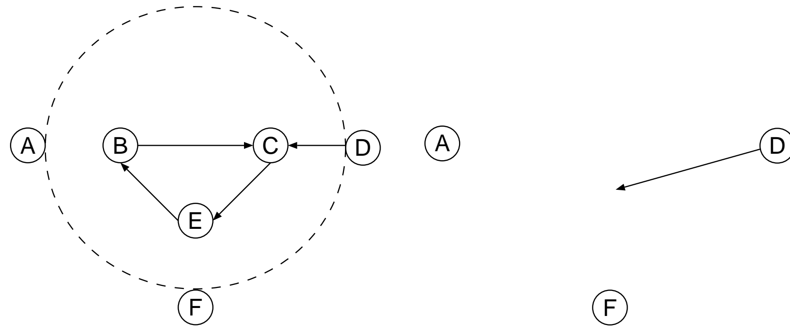
Figure 5: Node abstraction of a proper subgraph of G.



(a) The original graph G. The large circle contains the nodes in $Y = \{B, C, E\}$.

(b) The graph G after abstracting the relaxed subgraph induced by Y into a new edge e.

Figure 6: Edge abstraction of a relaxed subgraph of G.



(a) The original graph G. The large circle contains the nodes in $Y = \{B, C, E\}$.

(b) The graph G after abstracting the relaxed subgraph induced by Y into a new edge e.

Figure 7: Edge abstraction of a relaxed subgraph.

2.5 Graph abstractions

We are now in a position where an arbitrary subset Y of nodes in a (hyper)graph $G = (X, U)$, where Y is a subset of X, induces two new graphs G' and G'' that are different abstractions of G:

- **The node abstracted graph induced by Y.** The proper subgraph generated by Y can be abstracted to a new node, and as such G is abstracted to G'. In G, all nodes in Y, and all edges entirely connected to nodes in Y, are collapsed to a single node.
- **The edge abstracted graph induced by Y.** The relaxed subgraph $L = (Y, V)$ generated by Y can be abstracted to a new edge, and as such G is abstracted to G''. In G, all nodes in Y, and all edges with at least one ending point in Y, are collapsed to a single edge e.

It is possible to distinguish between several different kinds of situations where these abstractions can be applied.

- **Logical abstraction.** A subgraph is substituted by a node or edge, and hereby the graph is simplified. This affects the representation of the graph. For any graph algorithmic purpose, the subgraph is thus considered as a single node or edge. From a visual point of view, however, the abstracted node or edge is shown as the original subgraph.
- **Visual abstraction.** The subgraph is collapsed to a single node or edge in the visualization of the graph, but no changes have occured in the underlying graph representation. Only from a visual point of view the graph has been simplified. This kind of abstraction characterizes the visual compaction facilities found in graph editors such as Edge [7] and daVinci [2].
- **Logical and visual abstraction.** The subgraph is substituted by a single node or edge, both in the internal representation and in the visualization of the graph. This kind of abstraction may be seen as a proper graph transformation in the direction towards a more simple graph.

In the next section we will see how the ideas behind logical, edge abstracted graphs can provide for rich and versatile variations of edge renderings in a practical graph editing tool.

3 Composite graph concepts and hierarchies

In the previous section, we described a number of general graph *theoretical concepts*, specifically proper and relaxed subgraphs and the graph abstractions derived from these. In this section we will explain how to shape a number of similar *practical concepts* which we use as the basis for a graph editing tool. The concepts are implemented in Scheme and Meroon. We will start by introducing composite nodes and composite edges in terms of the results from the previous section. Following that we introduce a hierarchy of graph concepts, which include composite nodes and composite edges. Next, we address the problem of handling the dependencies among the constituents of a composite object and the composite object itself. It turns out that the structures and the interaction among objects can be handled by instances of the well-known design patterns called Composite and Observer. Finally, a number of concrete examples are discussed.

3.1 Composite graph elements

As a consequence of section 2.3 we can represent a many-to-many relationship among a number of nodes as a relaxed subgraph. The relaxed subgraph can be abstracted to a single hyperedge by means of a logical abstraction process. This simplifies the graph from a logical and topological point of view. However, from a visual point of view the hyperedge is presented in terms of the relaxed subgraph.

We are now interested in forming a concept which captures the mathematical idea of a hyperedge representing the relaxed subgraph. In order to emphasize that the hyperedge is formed in terms of more primitive graph constituents, we call the new concept a *composite edge*. We could in a similar way introduce a *composite node* which is the result of abstracting a proper subgraph, but we do not use composite nodes in the remaining parts of this paper.

A fundamental requirement to the design of composite graph elements is that they appear as first class members, in the same way as simple nodes and simple edges. This “first-classness” can be expressed as the following two properties that must hold for composite graph elements:

- **Homogeneity**: Programatically, they can be used just like simple (non-composite) elements by clients.
- **Unity**: In response to interactive manipulation, they appear as a unit to the user.

The homogeneity property implies that composite elements may be nested to arbitrary depth. Thus, a composite edge can, together with other nodes and edges, be abstracted further on to another composite graph element.

The unity property has as a consequence that interaction on the parts of a composite graph elements is interpreted as interaction on the whole. For that purpose we need to care about structures and mech-

anisms which allow us to capture the necessary interaction and cooperation between the composite objects and its parts.

3.2 The hierarchy of node and edge concepts

The concepts of simple nodes and edges together with the composite graph concepts introduced above need to be classified together in a single concept hierarchy. Using the object-oriented paradigm this has led us to the class hierarchy of "built-in graph elements" in figure 8. A table, which explains the responsibilities of the individual classes, is shown in figure 9.

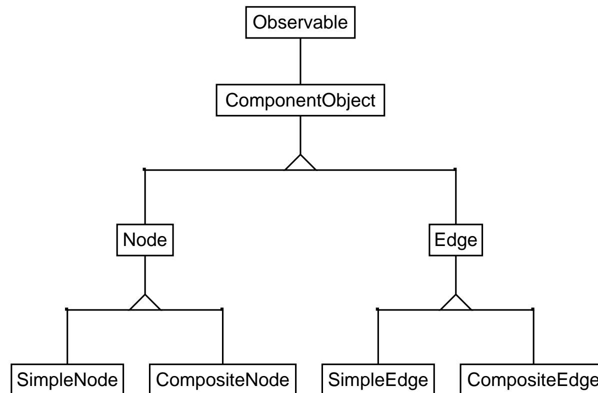


Figure 8: The hierarchy of graph element types. (Figure: Ginger)

The hierarchy of built-in types is open, and intended, for further extension by addition of user-defined subclasses, in particular specializations of the `CompositeNode` and `CompositeEdge` classes. We will now explain the elements of the hierarchy in figure 8 and their relationships.

All classes, except `SimpleNode` and `SimpleEdge` are abstract. The two classes at the top of the hierarchy together with `CompositeNode` and `CompositeEdge` contain methods which are necessary in the design patterns, which we use in the next section.

The abstract class `Edge` is responsible for building the structure of a graph. Thus, `Edge` contains methods for linking node and edge objects. The `Edge` class represents hyperedges as described in section 2 since it places no restrictions on the number of nodes it may connect.

The concrete classes `SimpleNode` and `SimpleEdge` extend their superclasses with a number of methods that control the visual appearence (attributes) of nodes and edges. A `SimpleEdge` is furthermore restricted to connecting two nodes. Objects of these two types can be displayed and manipulated interactively in the graph editing tool without further programming efforts.

Finally, the abstract classes `CompositeNode` and `CompositeEdge` contain methods which – in cooperation with those in `Observable` and `ComponentObject` – support the implementation of composite graph elements. As mentioned above, user-defined composite graph elements will typically be implemented as concrete specializations of `CompositeEdge`. We will now describe the realization of composite graph elements using these classes and the design patterns in which they are involved.

Class *Observable*

registerObserver(obs,asp,meth)	make obs observe aspect asp using meth as update method
notifyObservers(asp, val)	notify all obsers that aspect asp has changed to val

Class *ComponentObject*

display	displays the object on the screen
undisplay	removes object from the screen
select	make object the selected one
unselect	make sure the object is not selected

Class *Node*

Class *Edge*

to(Node)	connect Node at terminal ending point of this edge
from(Node)	connect Node at initial ending point of this edge
unlinkTo(Node)	unconnect Node from terminal ending point
unlinkFrom(Node)	unconnect Node from initial ending point
inpointingNodes	return the set of nodes of initial ending point
outpointingNodes	return the set of nodes of terminal ending point

Class SimpleNode

x(Int)	set the x-coordinate of this node
y(Int)	set the y-coordinate of this node
width(Int)	set the width of this node
height(Int)	set the height of this node
label(String)	set the label of this node
bitmap(File)	assign a bitmap to this node
display	displays the object on the screen
undisplay	removes object from the screen
select	make object the selected one
unselect	make sure the object is not selected

Class SimpleEdge

direction(Dir)	assign dir as the direction of this edge
to(Node)	connect Node at the terminal ending point
from(Node)	connect Node at the initial ending point
display	displays the object on the screen
undisplay	removes object from the screen
select	make object the selected one
unselect	make sure the object is not selected

Class *CompositeNode*

addComponent(Comp)	add component to this node
deleteComponent (Comp)	delete component from this node

Class *CompositeEdge*

addComponent(Comp)	add component Comp to this edge
deleteComponent(Comp)	delete component Comp from this edge

Figure 9: The responsibilities of the classes from figure 8. Classes and responsibilities in *italic* are abstract.

3.3 Realizing dependencies in composite graph elements

We handle the relationships between a composite graph element and its parts via use of the design pattern called Composite. In general, a Composite pattern is used to compose objects into tree structures of parts and wholes, and to allow for the same treatment of composite as well as non-composite objects. The diagram in figure 10 illustrates the static structure of the Composite pattern as it is used in Ginger.

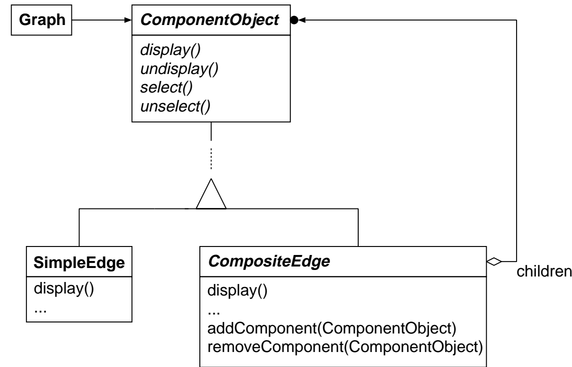


Figure 10: The Composite design pattern as it is used in Ginger.

As it can be seen in the figure the class `ComponentObject` is responsible for display and selection (together with the reverse actions). The display and selection methods are all abstract, and consequently they need to be elaborated at more specific levels in the class hierarchy.

The class `CompositeEdge` contains the basic data structures and methods that allow the construction of a composite object by adding component objects to the composite. Removal of objects is also supported.

When a new user-defined composite node or edge type is implemented as a specialization of `CompositeNode` or `CompositeEdge`, one of the tasks of the implementor is to specify concrete versions of the display and selection methods inherited from `ComponentObject`. In most cases this is a simple matter of redirecting method calls to the corresponding method in each of the components.

The unity property, introduced in section 3.1 requires some kind of coordination between the whole (the composite object) and its parts (the component objects). For instance, if the user selects a component object, this should typically result in the entire composite object being selected. In addition, a composite should be able to react on changes in its surroundings. As an example of this, we will later study a composite “orthogonal” edge consisting of a horizontal and a vertical line segment; in order to keep the line segments horizontal and vertical, respectively, the composite edge has to detect and react appropriately whenever one of the nodes which it connects is moved.

In both cases, the composite object needs to be informed of certain events that occur in its components or in its context. The Observer design pattern [3] describes a proven way to arrange exactly this kind of collaboration between observers and subjects. In this context, the possible subjects are the components of a composite edge, or the nodes connected to a composite edge, and the observer is the composite object. The use of the Observer pattern in Ginger gives rise to the class `Observable` in the top of

the class hierarchy shown in figure 8. This abstract class contains methods to register an object of any subtype of `ComponentObject` as observer to the `Observable` object. The observer is registered with interest in some particular *aspect* of the `Observable` object. The registered observers are notified whenever the observed object (the subject) changes state with respect to the given aspects. The notification includes information about the relevant aspect.

As an alternative to the use of the Observer pattern, the dependencies among components and composits could be expressed in a system of constraints and then be handled by a constraint solver. That solution has some benefits, for instance ease of expression. The cost, however, is the relative complexity of the constraint solver compared to the simple application of the Observer design pattern. As we will see in the next subsection, the current design also allows quite easy and elegant expression of the dependencies among graph elements.

3.4 Examples of composite graph elements

In this subsection, we present three examples of composite graph elements all of which are implemented as subclasses of the `CompositeEdge` class. The extended class hierarchy is shown in figure 11.

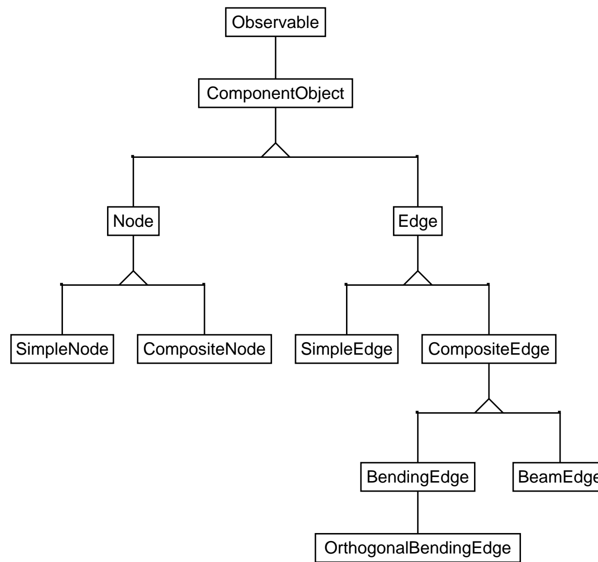


Figure 11: Ginger's hierarchy of graph element types, extended with subclasses of `CompositeEdge`. (Figure: Ginger)

The first and most simple example we present is the composite edge `BendingEdge`. It is an edge equipped with a bending point which allows the edge to bend in an arbitrary angle, see figure 12. The bending point can be moved interactively by dragging it using a pointing device.

A BendingEdge is composed of two SimpleEdge objects (one in the "head" of the edge and one in the "tail") and a very small SimpleNode object in the bending point. These objects are private to the BendingEdge and are created during construction of BendingEdge objects. The class `BendingEdge`

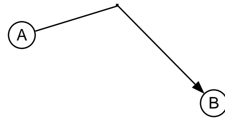


Figure 12: BendingEdge object connecting two SimpleNode objects. (Figure: Ginger)

redefines the methods *to*, *from*, *display*, and *select*. The redefined *to* method simply connects the "head" component edge to the node given as argument, by calling the *to* method on the component edge. The *from* method is implemented in a similar way. The display and selection methods have equally straightforward implementations since they just have to call the corresponding method on all the component objects.

Whenever one of the components of the BendingEdge is selected interactively, the other components must also be marked as selected, otherwise the BendingEdge does not succeed in giving the user the impression that it is a logical unit. This is accomplished by using the Observer pattern described above. The BendingEdge registers itself as observer of each of its components with respect to the aspect "selection", and each time a component changes selection state, the BendingEdge thus receives control and can update the state of all other components accordingly.

The power and elegance of composite graph elements based on the Observer pattern is further illustrated by our next example, the `OrthogonalBendingEdge`. This specialization of `BendingEdge` automatically places the bending point so that the edge always bends orthogonally, see figure 13.

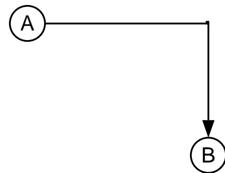


Figure 13: OrthogonalBendingEdge connecting two SimpleNodes. (Figure: Ginger)

OrthogonalBendingEdge is identical to BendingEdge except that it redefines the *to* and *from* methods. The challenge for the OrthogonalBendingEdge is to stay orthogonal even when the nodes it connects are moved. However, the OrthogonalBendingEdge can handle this by registering itself as observer of each of the nodes it connects, with respect to the position of the nodes. This registration is handled in the redefined *to* and *from* methods. When one of the nodes is moved, the OrthogonalBendingEdge is notified and it can then update the position of the bending point to reflect the new situation. Each of the redefined *to* and *from* methods consist of just a few lines of Meroon code. Figure 14 serves as a side bar which explains the redefined *to* method of OrthogonalBendingEdge.

As a final example, we present the BeamEdge, shown in figure 15.

Compared with the previous examples, the BeamEdge has two interesting characteristics:

- It uses nested composite graph elements.
- It is a hyper edge.

```
1. (define-method (to! (obe OrthogonalBendingEdge) (n SimpleNode) . option)
2    (call-next-method)
3    (x! (BendingEdge-bendNode obe) (SimpleNode-x n))
4.   (registerObserver n obe 'x orthogonalBendingEdgeNotifyEndpointChange))
```

The following items explain the example line by line:

Line 1: A method `to!` is to be defined. The exclamation mark signals by convention that the method changes the state of the object. The method has two parameters, `obe` and `n`, of types `OrthogonalBendingEdge` and `SimpleNode` respectively. Finally, the `option` parameter is a placeholder for additional information not used in this example.

Line 2: The first procedure call in the `to!` method, `call-next-method`, is a Meroon primitive which activates the `to!` method in the superclass, `BendingEdge`. This method takes care of connecting the node to the edge's terminal endpoint.

Line 3: The x coordinate of the simple node in the bending point is initialized with the value of the x coordinate of the node `n` being connected.

Line 4: The OrthogonalBendingEdge object, `obe`, is registered as an observer of the node `n` with respect to the "`x` aspect", such that the update method `orthogonalBendingEdgeNotifyEndpointChange` will be called each time the node `n` changes horizontal position (x coordinate). When called, the method updates the x coordinate of the node in the `OrthogonalBendingEdge`'s bending point in the same manner as in line three of this example.

Figure 14: An example of a method in the class OrthogonalBendingEdge, which participates in the Observer design pattern.

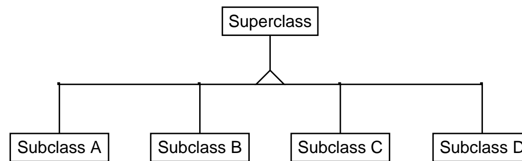


Figure 15: A BeamEdge representing a specialization relationship between a superclass and four subclasses. (Figur: Ginger)

The internal composition of a BeamEdge is illustrated in figure 16, where the parts are separated from each other to make clear what the constituents are. The edge consists of a SimpleEdge, a SimpleNode (later denoted the *center-node*) and, in this example, four OrthogonalBendingEdge objects (slightly overlapping) as nested components of the BeamEdge.

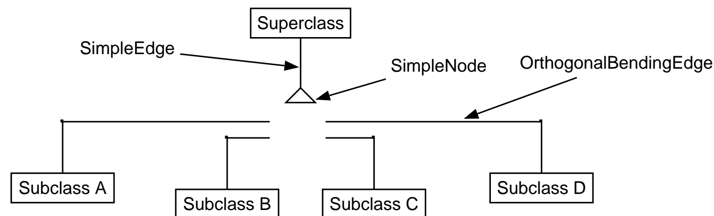


Figure 16: Components of a BeamEdge.

The visual appearence of the BeamEdge is controlled by the position of the nodes which it connects, via dependencies realized using the Observer pattern. Naturally, each of the component Orthogonal-BendingEdges keeps itself orthogonal. Therefore, all the BeamEdge has to do in order to maintain a visual appearence as shown in figure 15, is to register itself as an observer of the "superclass" node, and thereby – when necessary – update the position of the center-node to keep this node in a fixed distance straight below the superclass node.

A BeamEdge is a hyperedge since it may connect more than two nodes. The BeamEdge has one designated initial endpoint which is the node corresponding to the superclass node above, and any number of terminal endpoints. Nodes – corresponding to subclasses in the example above – can be connected to the BeamEdge's terminal endpoints by calling the BeamEdge's *to* method once for each node to be connected.

This concludes our series of example composite graph elements. In the following section, we present the Ginger graph editor which implements the ideas presented above.

4 The Ginger graph editor

The incremental and experimental development of the Ginger graph editor has been of central importance to the development of the ideas presented in this paper. This section covers some basic design issues and illustrates the user interface of the editor.

A number of requirements was posed on the graph editor. First of all, we needed an interactive graph editor which could be used as a base for our experimentation with the more interesting aspects of graph manipulation. But we did not want to create just a "quick and dirty" prototype; rather, we wanted the basic, interactive graph editor to be useful in its own right as an easy to use, efficient and portable tool for simple graph manipulation tasks.

This led to a design based on a "kernel" implemented in the C programming language and using Scheme[6] as a dynamic extension language.

The kernel implements a basic graph datatype and handles the visual presentation and interactive manipulation of this graph datatype via the graphical user interface implemented using the X toolkit and -library.

The extension language allows the user to define new graph elements like the examples given in the previous section. As mentioned earlier, Ginger uses Meroon, which is an object-oriented extension to Scheme, to implement the hierarchy of graph element types shown in figure 8 and to let the user define new subtypes.

A snapshot of Ginger's graphical user interface is shown in figure 17.

The larger part of the main window is the graph area where the current graph may be edited by direct manipulation (selection and dragging of nodes and edges using a pointing device). The menu buttons at the top of the main window offer file operations (load/save graphs, export to PostScript file etc.), selection control (node and edge selection modes and operations), layout algorithms (array layout,

[6]Ginger uses the Elk [6] implementation of the Scheme programming language. Elk is well suited for the purpose because it supports easy linking of the Scheme interpreter into a C program.

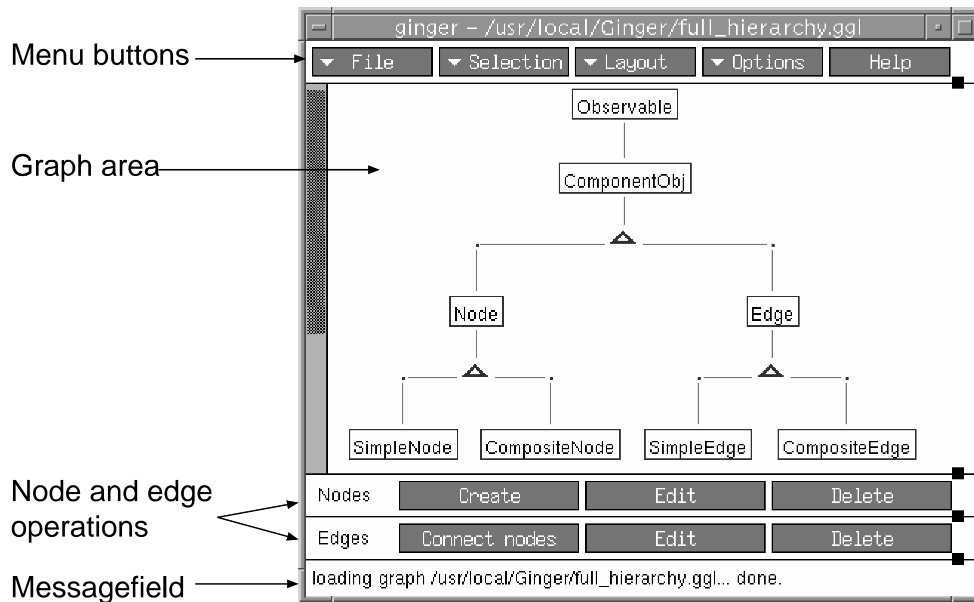


Figure 17: Ginger's main window

circular layout and hierarchical layout can by applied to any selection of the graph), and various program options. The help button opens the editor's hypertext help window.

Ginger's extension language is accessed via a command prompt in a pop-up window which can be opened from the file menu. Here, the user has programmatic access to the built-in graph datatypes described above and to the full functionality of Elk Scheme.

To complete the example of "intelligent" composite graph elements, we will show how the BeamEdge presented in the previous section handles interactive relocation of one of the nodes connected to it. The BeamEdge was first used in the figure showing the hierarchy of node and edge concepts; the figure is repeated here as figure 18.

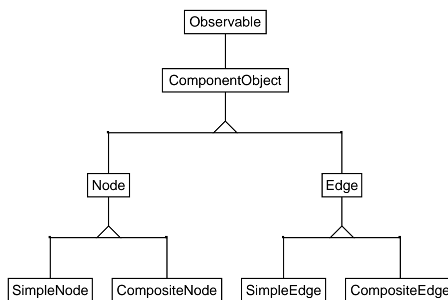


Figure 18: The hierarchy of graph element types. (Figure: Ginger)

Now imagine that a user interactively drags the `Edge` node slightly downwards and to the right. This action affects the "upper" BeamEdge connecting `Edge` to its superclass `ComponentObject` and the "lower" BeamEdge connecting `Edge` to its two subclasses, `SimpleEdge` and `Composi-`

teEdge. The new situation is illustrated in figure 19.

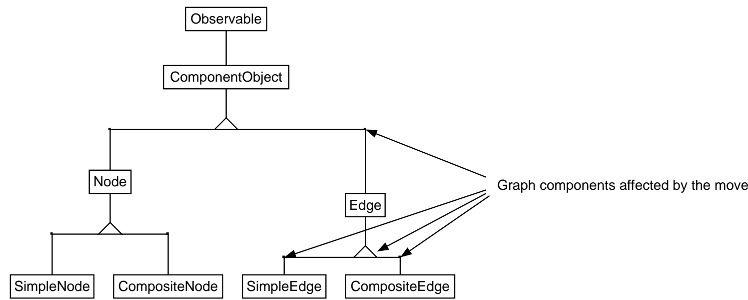


Figure 19: The hierarchy of graph element types.

With respect to the upper BeamEdge, the change is handled by the BeamEdge's OrthogonalBendingEdge-component, so the BeamEdge itself needs not take any action. Since a BeamEdge must keep its center-node (the triangle-shaped gadget) in a fixed distance straight below the superclass node, the lower BeamEdge adjusts the position of its center-node in repsonse to the interactive move of the `Edge` node. That adjustment, in turn, triggers an update action in each of the two OrthogonalBendingEdge-components of the lower BeamEdge; hence each of these components adjust the position of their bending points in order to remain orthogonal.

The above figures only show static snapshots of the editor. The Ginger graph editor is available for downloading[7] for a more dynamic and interactive experience.

5 Conclusions

In this paper we have addressed the problem of making a graph tool, which supports many different edge and hyperedge shapes. The important observation is that many complicated edges in "real life graphs" can be regarded as some kind of a subgraph aggreated by a number of more primitive edges and nodes.

At the theoretical level the main contribution is the definition of a relaxed subgraph, which can be abstracted to an edge relative to the surrounding graph. We think of the graph abstraction as a logical abstraction, because it simplifies the topology of the graph. At the visual level, however, the relaxed subgraph serves as a presentation of the abstracted edges in terms of primitive edges and nodes. From a mathematical point of view, a relaxed subgraph may be a confusing concept, because it involves edges of degree less than two. From a more practical computer science point of view, however, it makes sense to work with edges as first class objects, even in the case where one or more ending point are (temporarily) disconnected from the nodes of the graph.

At the practical level, the main contribution of the paper is the organization of a relaxed subgraph in a composite edge together with the realization of dependency preserving mechanisms between the the composite edge and its surround. The dependencies are realized by instances of the design patterns known as Composite and Observer.

[7] Downloading can be done via the WWW Ginger homepage: http://www.cs.auc.dk/∼normark/Ginger

It is our conclusion that the insight from this paper may turn out to be a small but significant contribution to a general graph editor component, which we envision as an important building block of many future tools in a wide variety of application areas.

References

[1] William Clinger and Jonathan Rees. *Revised4 Report on the Algorithmic Language Scheme*, 1991.

[2] Michael Fröhlich and Mattias Werner. *DaVinci V1.4 User Manual*. Universität Bremen, e-mail: daVinci@Informatik.Uni-Bremen.DE, 1994.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 0-201-63361-2.

[4] P. D. Karp, J. D. Lowrance, T. M. Strat, and D. E. Wilkins. The Grasper-CL Graph Management System. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA 94025, 1994.

[5] Niels C. Larsen and Martin K. Molz. Ginger is an interactive graph editor. Master's thesis, Departmennt of Computer Science, Aalborg University, 1996. In Danish.

[6] Oliver Laumann. *Building Extensible Applications with Elk - C/C++ Programmer's Manual*. http://www-rn.informatik.uni-bremen.de/software/elk/split/cprog/cprog.html.

[7] Frances N. Paulisch. *The Design of an Extendible Graph Editor*. Springer-Verlag, 1993. ISBN 0-201-56953-1.

[8] Christian Queinnec. *MEROON V3: A Small, Efficient and Enhanced Object System*. file://ftp.inria.fr/INRIA/Projects/icsla/WWW/Meroon.html.

[9] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall International, 1991.

[10] Richard Stallman. *GNU Emacs manual*. The Free Software Fundation Inc, 1985.

[11] Richard M. Stallman. EMACS: The Extensible, Customizable, Self-Documenting Display Editor. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, chapter 15, page 300. McGraw-Hill, 1984. ISBN 0-07-003885-6.