Montage - an ActiveX Container for Dynamic Interfaces

Gordon Woodhull and Stephen C. North
AT&T Laboratories

Montage - an ActiveX Container for Dynamic Interfaces

Gordon Woodhull
nodrog@ix.netcom.com
Stephen C. North
north@research.att.com
Information Visualization Research.
AT&T Laboratories
180 Park Ave.
Florham Park, NJ, USA

Abstract

Montage is a customizable, embeddable ActiveX container. Its client objects may be positioned dynamically by an external layout agent. *Montage* manages toolbars and user interface modes, integrating disparate components into a single, consistent interface. An important part of this task is supporting "group repositories" of related objects for data transfer operations such as cut-and-paste, drag-and-drop, save and load. *Montage* does not rely on large external libraries such as the Microsoft Foundation Classes, and thus is relatively lightweight. The prototypical *Montage* application is an embeddable display for dynamic networks (abstract graphs).

Introduction

An important trend in interactive computing is toward better integration of tools and services. ActiveX[1] is a protocol or set of interfaces for tool interconnection, enabling sharing of data and the user interface. ActiveX makes it possible, for example, to embed a live Excel spreadsheet in a WordPerfect document, or to place a third-party display widget on a Visual Basic canvas. ActiveX originated as an inter-client cut-and-paste protocol, but has grown to include many advanced services, including object naming, in-place editing, canvas event management (*e.g.* resize, drag-and-drop), toolbar and menu sharing, and loading and saving of persistent state. It employs the Component Object Model (COM) for communication between components. See Brockschmidt [2] for further background discussion on ActiveX and COM.

ActiveX is as difficult to program as it is feature-rich. It has dozens of interfaces, with hundreds of methods to call or implement. Although standard development tools simplify the design of contained objects (*controls*), we found insufficient support for designing new *containers* with a full complement of ActiveX features. We wrote *Montage* to provide this support in the form of a general container. We envisioned a small, efficient container object that would integrate with common desktop tools, having all aspects of its user interface and canvas layout determined by the components that run inside it.

The motivation for creating *Montage* arose from the Microsoft Windows port of *dotty* [5], a graph editor.[2] *Dotty* was written for Unix X Windows and is similar to other well-known Unix/X graph editors, such as GraphEd [7], Edge [9], and Graphlet [8]. *Dotty* was converted to MS Windows by re-coding its file selector, text-entry and other widgets in win32 graphics. This approach made *dotty* into an isolated island of X among MS Windows programs. Functioning almost identically to the Unix version, the port lacks even basic MS Windows compatibility such as access to native print drivers. Users would like a much higher level of tool integration, such as the ability to embed graph diagrams in text documents, or to incorporate multimedia controls in diagrams. Additionally, we encountered basic limitations in *dotty*'s architecture when our research in algorithms moved into dynamic on-line layouts – *dotty* assumes batch layout. All this suggested a fresh architectural approach.

In this context, we were interested to learn how much could be gained by wholesale adoption of ActiveX and win32 in a graph editor well adapted to MS Windows. As we developed a graph control, it became apparent that the layout and containment mechanisms could be cleanly separated. Our win32 graph editor became a general-purpose container, *Montage*, automated by a set of dynamic layout engines.

[1] Also known as Object Linking and Embedding or OLE.

[2] "Graph" always refers to an abstract graph or network in this paper.

A General ActiveX Container

In 1996, Microsoft drafted a new specification for what were then OLE Controls, OCX96, which among other things allowed for non-rectangular and transparent controls. Central to OCX96 was the definition of a *windowless control*. Before this time, all OLE objects used windows, which are overlapped rectangular regions that receive messages directly from the operating system. Windows were essential to early OLE because they meant that a visual region within one document could be owned by an application in an entirely different process. OCX96 defined new interfaces so that the container could ask the contained object mouse hit-testing and region opacity questions, and forward it the messages it would receive had it a window. Also, the contained object could ask the container for services, such as drawing handles and mouse capture, that normally require a window.

The facts of graph layout made OCX96 very appealing to us: nodes are usually non-rectangular, and it would be impractical to put opaque rectangles around edges. If *Dynagraph* were a windowless control container, both nodes and edges could be represented by ActiveX controls. But the first two versions of the Windows *Dynagraph* were written with the Microsoft Foundation Classes (MFC), which do not support containment of windowless controls. The Active Template Library (ATL) added efficient support for writing windowless controls in early 1997, but still there was no library support for containers.

Since it was clear that we would have to write any support for windowless controls from scratch, it seemed like an opportunity to abandon the cumbersome MFC entirely. Another aspect of OCX96 made this especially appealing: the facility for transparent controls would make it possible to extract all *Dynagraph*-specific behavior out of the containment functionality, leaving a general ActiveX container and a fairly simple graph application. The main design question became, "**If ActiveX defines a containment relation, what is the abstract data type for that relation?**" The *Montage* answer is to factor out all interaction, layout, coordinate transformation (zoom, etc.), and persistence, as for example a function for ordering objects is factored out of an ADT for dictionaries. What is left is a z-ordered list of positioned content objects at various levels of activation.

Montage is not the only solution to this problem. Dynamic HTML (which was developed concurrently) also moves layout policy out of the core container and into client programs. Yet browsers do not have persistence models: they are *presentation* oriented, not *document* oriented. Visual Basic and almost all other OLE applications with scripting languages also function as general containers, but developers may find themselves fighting with the layout and other peculiarities of the document data type. In contrast, the *Montage* document type is simply an ordered list of controls; we model the familiar OLE GUI in auxiliary controls and other components.

Automation

ActiveX Automation is the name for the invoking program functionality through public COM interfaces. An **automation object** implements and publishes interfaces; an **automator** (or "automation controller" or "client") calls methods in these interfaces. Wherever possible, *Montage* has been factored into components that communicate over such public interfaces. The layout engines, modes, and storage system are considered external clients, with no privileged access to *Montage* data. This design allows the easy replacement of most non-core parts of *Montage*, and also allows fine control over its functionality from any Automation-compatible language, including C++, Java, and in the future, Visual Basic and Web scripting languages such as JScript and VBScript. Through Automation, a *Montage* diagram can be connected to a dynamic process, for instance to show diagrams of a computer network. If the system can then react to diagram events, *Montage* becomes a "transparent" user interface to manipulate external objects.

Dynamic Layout Engines

A key requirement for *Montage* is to support dynamic network diagrams. The elements of network diagrams are nodes and edges. These are represented in *Montage* by ActiveX controls. For convenience we supply a "shape node" control specifically for graph diagrams, but other ActiveX documents or controls have equal status. Similarly, edges are ActiveX controls that draw generalized curves; we supply a simple non-interactive control to draw polylines and Bezier splines, but the component architecture leaves the door open for interactive edges, varying edge styles, etc. *Montage* supports both windowed and windowless controls. We use windowless controls for both edges and shape nodes; nodes having a naturally rectangular aspect, as do most ActiveX documents, may be either type.

Layout in *Montage* (and in all of ActiveX) centers on the **site** object, which represents the relationship between the container and contained object. The site's properties include position, a reference to the contained object/control, and ambient properties such as default colors and fonts. However, standard ActiveX does not define any interfaces to change these properties. In other words, container functionality can be divided into fetches, requests, and commands, but ActiveX only defines interfaces for the first two. Standard ActiveX does offer ways to give hints to containers, but policies about layout and activation are generally hidden.

In contrast, *Montage* pushes policy decisions out of the container. It defines interfaces for clients to directly change site properties. In this model, clients make hints or issue requests by treating them as events sent to sites. In the case of *Dynagraph*, to support layout control by a central server, the *Montage* architecture specifies that automators such as modes (described below) that need to change a diagram do this by generating events on the affected objects' sites rather than making changes directly. If a sink picks up an event, it changes the layout; otherwise an automator is free to make the change itself. In effect, the site has an **owner** (which may be the control itself) that interprets all requests made on the site, usually forwarding them to a layout engine (see Figure 1).

In our main application, the *Dynagraph* ActiveX control, the *Dynagraph* library maintains network diagrams. *Dynagraph* is a portable C library that defines an interface for incremental graph layout services. Clients can open and close diagrams and edit their contents by sending layout events to an engine. Events refer to operations on individual nodes and edges referenced by client-side descriptors, for example, "insert node *v* at (x,y)", "move edge *e* to (x_0 ,y_0, x_1, y_1, x_2, y_2,…)" or "delete node *v*". To update live diagram displays, layout managers send events of the same type via callbacks. Note that a single request event may yield many diagram display update events, and it is also possible for an update request to be denied by a layout engine (say, if a request is inconsistent with diagram-specific constraints). Compound updates are handled by input event queues in layout engines. The library currently has managers for dynamic hierarchical layout [10], virtual physical ("spring") models, and incremental orthogonal layout.

In order to abstract the layout engines into a separate module, we created a COM wrapper for the *Dynagraph* library. The C structures are manipulable through COM interfaces, and the callbacks get broadcast through COM connection points. Then a separate module, called DGM (*Dynagraph for Montage*), manages the translation from *Montage* events to *Dynagraph* events, and from *Dynagraph* events to commands for *Montage*. This design helps ensure that neither is dependent on the details of the other.

Modes and Toolbars

Because *Montage* is a general control container, it has no built-in user interface. On its own it does not respond to mouse or keyboard events, so the only objects that can respond are controls activated within *Montage*. The user interface of a *Montage* application consists mainly of the controls plugged in as **modes** and as **toolbars**. Modes provide the main interface. These are transparent, windowless controls placed in front of all inactive objects but behind all active objects. Here they receive all mouse events that the active contained objects do not process on their own.

For example, a left mouse button drag consists of the Windows messages WM_LBUTTONDOWN, multiple WM_MOUSEMOVEs, and a WM_LBUTTONUP. If an active windowed control is under the mouse, Windows routes these messages directly to it, for instance selecting text in a Word document. Otherwise the messages go to the *Montage* window, which searches the Z order for the control under the mouse and forwards the messages. If there is no active object in front of it, the active mode will then receive the messages through the OCX96 interface IOleInPlaceObjectWindowless. If the mode interprets the drag as a move, it then asks *Montage* what control is below it at the coordinates of the WM_LBUTTONDOWN, and generates the event IMCCSiteOwner::Move on the site, or calls IMCCSite::put_Rect itself if there are no sinks for that event.

The design of modes as contained controls is a strategy to ensure that the *Montage* architecture is as open as possible. Even though two modes are compiled into the same dynamic link library (DLL) as *Montage*, all use only public interfaces to manipulate *Montage*. Accordingly, some of *Montage*'s internal calculations, such as "find control at point," are exposed as public methods, and *Montage* provides utility objects to simplify some of the common, cumbersome operations of modes, such as data transfer.

The two basic modes included with *Montage*, View and Edit, should be somewhat familiar from other ActiveX containers. **View mode** activates all contained objects so that they can be edited but does not allow any "structural" operations, such as moving, deleting, copying or pasting. **Edit mode** allows structural changes, as well as in-place activation with a double-click. Both modes should be applicable in many layout/UI applications. In fact, together they define a container with purely manual layout.

Unlike the generic View and Edit modes, the **Draw mode** of *Dynagraph* is entirely application-specific. Drawing (placing new objects) is a function of the user interface that will usually be specific to a class of applications, because the contained objects have various types that are presumably selected by various user-interface operations. In *Dynagraph* Draw mode, clicking on the canvas creates a new node, and dragging between two nodes creates a new edge.

In a typical application, multiple modes are active concurrently. ActiveX defines two levels of activation: **in-place** and **UI**. In-place activation only allows an object to receive mouse messages; a mouse message "falls through" any mode (or other windowless control) that does not catch it. UI Activation provides for toolbar, menu, and keyboard sharing, but unfortunately only defines negotiation between a container and *one* contained object. Presumably it would be much harder to negotiate between three or more graphic interfaces. But in the typical *Montage* case, we can assume that the concurrently active objects have been designed compatibly. So *Montage* defines a protocol for **Cooperative UI** (**coUI**) **Activation**. In coUI Activation:

- Keyboard messages fall through interested objects in the same way as mouse messages.
- Objects get a chance to add to the menu *Montage* negotiates with its container, and register to receive callbacks from their additions.
- The toolbar is exposed as a resource for all *Montage*-aware objects.

Additional services could ensure that the clients' uses of these resources do not collide.

To support toolbars, *Montage* can be activated "into" another window, typically the frame window of its container. Two auxiliary controls round out toolbar functionality. The **toolbar** control provides familiar grab handles around any ActiveX control, and can be floated. (To float, it removes the control and toolbar from the bar area, and re-activates the control into a new, independent floating window.) The **toolback** control provides the blank space on which toolbars can be arranged. Toolbars use the same layout architecture as graphs: moves initiated with the mouse get translated into requests to a **toolbar layout engine**, which ensures that toolbars don't pile up, while keeping them fairly close to where the user placed them.

Persistence and Data Transfer

Persistence is a key feature that must be implemented by ActiveX clients and servers, especially because data transfer actions such as cut-and-paste rely on the same mechanisms as the simpler save and load. Persistence of a collection of linked heterogeneous components requires some way to save the references between objects. The hierarchical persistence protocols provided by ActiveX make it possible for objects to create and initialize other objects, but there is no general way to connect objects not in a hierarchy. The problem may arise earlier: it is often only possible to create a live object by using a factory, yet the creator does not necessarily own (or know about) the factory.

Montage defines a new storage type called a **group** for this purpose. A group holds a set of named and anonymous objects, and supplies monikers for them. Monikers are standard ActiveX objects that can be "bound on" one object to get another; *Montage* uses "item monikers," which are the standard way to bind string names on a live object. A group's monikers, especially those identifying anonymous objects, are valid only within that group, so any group item that holds monikers must have them translated if the item is added to a new group. Because it would be inefficient and possibly incorrect to copy most live objects, the group model defines the **persistor**, a separate object that holds the monikers to reconnect an object on loading. The persistor gets copied and translated for each group, whereas its subject, the connected object, does not. A persistor may also hold a moniker to its subject's factory; in this way the persistor can be created without special knowledge although its subject cannot.

Persistors implement the interface IGroupItem, which allows them to control the way they are copied from group to group and to make sure any connected objects are copied as well. Since the persistors pull objects from group to group as necessary, drag-and-drop and cut-and-paste are fairly simple operations: the initiator of the transfer (usually a mode) creates a new group and pastes all objects to be transferred into the new group with IGroup::Paste. Then it saves the group to a storage object, and wraps the storage object in an ActiveX-compatible data object for the operating system. The destination of the transfer retrieves the group and pastes the objects into its own group to complete the transfer.

Data transfer between *Montage* and another application is also straightforward, as the ActiveX format negotiation protocol defines a standard for embedded objects. If items are dragged out of *Montage* and into another application, that application can embed them as a new instance of *Montage* because *Montage* supports the standard "embedded" format. Instead of transferring names (that the client will know nothing about), the client loads the data as a complete object. Similarly, if the user drags from a non-*Montage* object onto a *Montage* canvas, modes can accept this drop in the embedded object format as a new contained item, though they know nothing about the source application. (For example, Graph Draw Mode embeds an object as a new layout node.)

The *Dynagraph* Application

Figure 2 shows *Montage*, embedded in Microsoft Word, running *Dynagraph*. The session actually consists of three instances of *Montage*: the graph view, the toolbar area, and the node palette in the toolbar. Additional diagrams and views would involve more instances. Nodes may be added to a diagram by dragging items from the palette or another application to the canvas, or by clicking on the canvas to insert the object selected in the palette. Similarly, items can be added to the palette by dragging them from the canvas or from another application. It is satisfying that this capability arises naturally from the *Montage* architecture, as it took considerable programming effort to put similar features into *dotty*.

Figure 3 shows sample automator code that inserts a node into a graph. It demonstrates most of the common *Montage* operations: the manipulation of sites and layers, the use of transforms and connectors, and the activation model.

Unfortunately, the details of coding in COM may make the code somewhat cryptic to the uninitiated. The syntax of the code requires some explanation:

- All COM interfaces (which by convention start with the letter "I") extend the interface IUnknown, which provides methods for reference counting and querying for interfaces on the object. ATL's CComPtr and CComQIPtr template classes ("qi" is a macro based on CComQIPtr) abbreviate the use of these methods by automatically calling the IUnknown methods as necessary.

- All COM methods return error codes that should not be ignored, so the RUN macro echoes errors by returning them, effectively treating them as exceptions.

- Since COM only supports methods and not properties, it recommends that get and set methods be prefixed with "get_" and "put_" respectively. ("new_" is an analogous convention in *Montage* code for methods that create objects that share memory with their parents.)

- All classes that do not share memory have 128-bit IDs (whose constants start with "CLSID_"), and are created with the system call CoCreateInstance.

The code starts with the creation of a new site in the *Montage* container. Sites are the only way to refer to contained objects; they represent the work that *Montage* is doing to keep a contained object afloat. Next it places the site in the main layer, which is the lowest layer in the Z order in the standard configuration of *Montage*.

The call to CoCreateInstance instantiates the control which will provide the visual part of the node. This control may also come from a data transfer triggered by a drop or paste operation. (In Graph Draw Mode, the data source is the active selection in the palette.) Next the code tells the control its size through the standard ActiveX interface IOleObject. The control does not have to accept this size (*i.e.* SetExtent may return an error code), so code within the connector asks the size through GetExtent rather than trusting the argument value. This illustrates a recurrent design theme in ActiveX component design: just about every method call is a request that can be interpreted or ignored, a callback or future query works better than remembered state.

As described before, the *Dynagraph* application of *Montage* has three levels of objects. The next section of code creates both *Dynagraph* and DGM objects. The DGM object is made aware of the two sides it is mediating through the interfaces INeedOnePointer (of the Group model) and IObjectWithSite (of standard ActiveX). It is unfortunate that such non-specific methods must be used to set up the connector, but using generic interfaces makes persistence simpler. Specifically, INeedOnePointer makes it simple to define a generic persistor for the common case where the persisted object needs to be connected to exactly one other object when it is loaded, or in order to load. And IObjectWithSite makes it possible for the site persistor to tell the owners their property, without knowing what the site means to them. When it is told its site, the connector sets itself up as a sink of the IMCCSite-Owner event interface, so that it handles all layout events.

Now that the structure is in place, it is possible to actually begin the layout. The code here tells the *Dynagraph* node object directly what the position is; it could instead use the event interface on the site, but it can be pretty certain at this point that the *Dynagraph* node object will handle the event. First it must translate the coordinates, though: the engine expects canvas coordinates, expressed in units of .01mm (HIMETRIC), not the window coordinates the mode is working from. So this code fetches the transform object (another modular component) from *Montage*, and asks that to do the translation.

Finally the node can be shown. The code here uses the IMCCSiteOwner event interface on the site to issue a show request. We omit the code for clarity, but the function intend() enumerates the sinks of that interface on the site, and calls IMCCSiteOwner::Intend() on each.

Summary

Montage is a convenient framework for ActiveX container applications and controls. It does not replace ActiveX, but handles most of a container's busywork, while leaving actual interface policy decisions to the modes and layout engines. An interesting issue is whether *Montage* ought to have a more abstract interface, instead of one so closely tied to Microsoft Windows and COM. Whether this is worth much effort is moot without an alternative application embedding platform to target.

Our work list includes support for object linking (besides embedding), dispatch interfaces for compatibility with Visual Basic and scripting languages, DCOM, further development of coUI Activation, and replacement of some high-level C++ code (such as that in the example) with scripts. A more ambitious goal on the layout side is to support compound nodes and edges. Complications arise in managing layouts where edges are allowed to connect nodes at variable levels of nesting, as in Harel's Statecharts [6]. Because there is yet no automated Statechart drawing algorithm or tool, this is an interesting practical problem.

Acknowledgments

The authors thank Ted Kowalski from the AT&T Labs Software Practices and Technology Center for technical and financial support. We also thank Pierre Sierra for programming the first version of *Dynagraph*. We also gratefully acknowledge contributions by David Dobkin, Emden Gansner, Eleftherios Koutsofios and Phong Vo to the *Dynagraph* system.

Montage may be obtained under a non-commercial license at www.research.att.com/sw/tools/*Montage*.

References

[1] Barghouti, Naser, John Mocenigo and Wenke Lee. "Grappa: A Graph Package in Java", *Proc. Graph Drawing '97,* Lecture Notes in Computer Science vol. 1353, pp. 336-343, Berlin: Springer-Verlag, 1998. See also www.research.att.com/~john/Grappa

[2] Brockschmidt, Kraig. *Inside Ole, Second Edition*. Redmond: Microsoft Press, 1995.

[3] Cooper, Alan. *About Face: The Essentials of User Interface Design*. Foster City, CA: IDG Books, 1995.

[4] Ellson, John and Stephen North, "TclDG – a Tcl Extension for Dynamic Graphs", *Proc. 4th USENIX Tcl/Tk Workshop*, pp. 37-48, July 1996.

[5] Gansner, Emden and Stephen North. "An Open Graph Visualization System and its Application to Software Engineering," submitted, Nov. 1997.

[6] Harel, David. "On Visual Formalisms", Comm. ACM 31:5, pp. 514-530, May 1988.

[7] Himsolt, Michael. "GraphEd: An Interactive Graph Editor", *Proc. STACS '89*, Lecture Notes in Computer Science vol. 239, pp. 532-33.

[8] Himsolt, Michael. "The Graphlet System", *Proc. Graph Drawing '96,* Lecture Notes in Computer Science vol. 1190, pp. 233-240. See also www.uni-passau.de/~himsolt/Graphlet

[9] Newbery-Paulish, Frances and Walter F. Tichy. "EDGE: An Extendible Graph Editor"in *Software – Practice and Experience* 20:S1, pp. 64-88, 1990.

[10] North, Stephen. "Incremental Layouts in DynaDAG" in *Proc. Graph Drawing '95,* Lecture Notes in Computer Science vol. 1027, pp. 409-418. Berlin: Springer-Verlag, 1996.

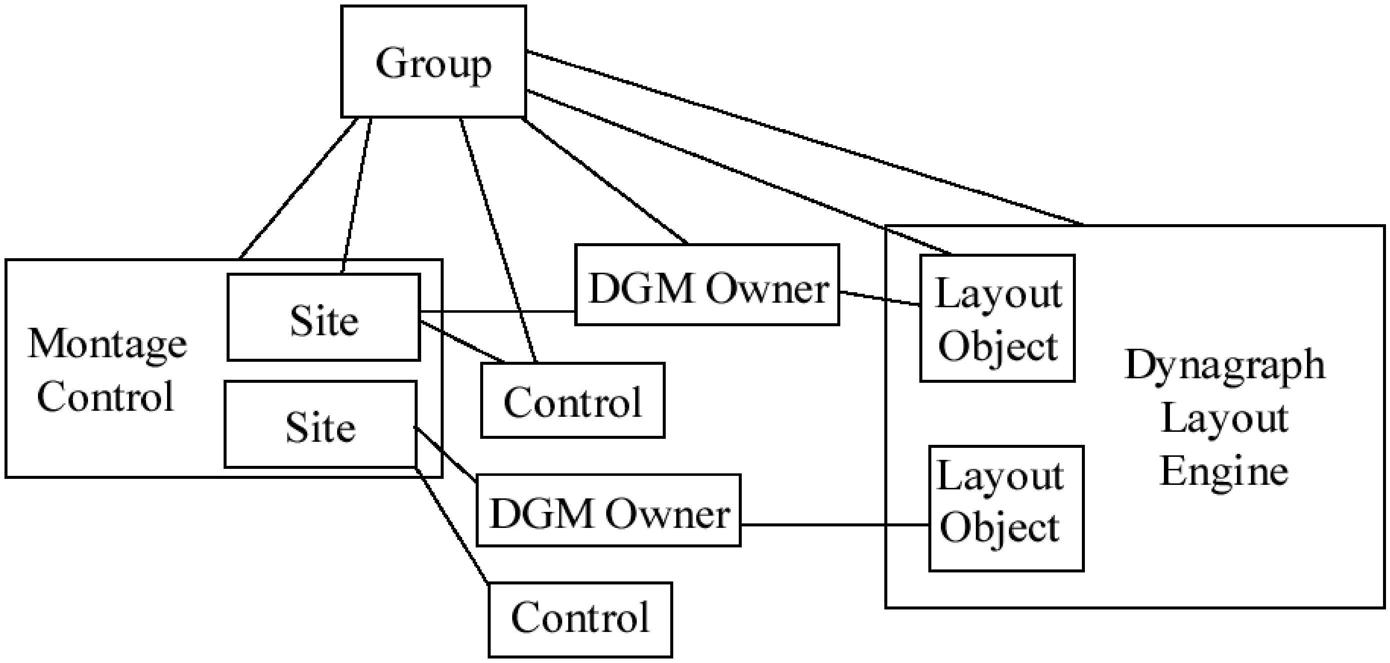
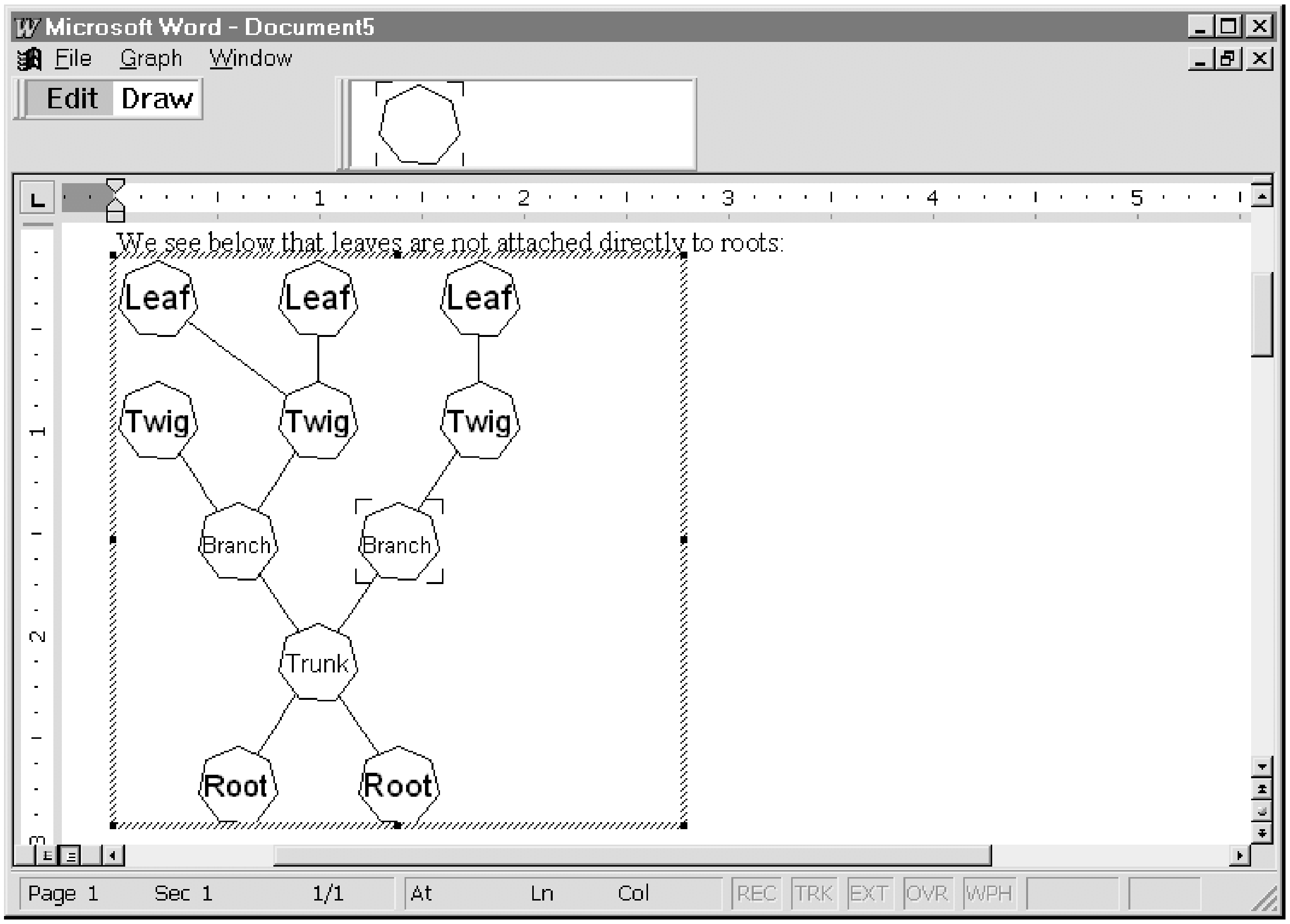


Figure 1: *Montage Dynagraph* object model. Contained boxes represent objects that share memory; lines represent communication over COM interfaces.



We see below that leaves are not attached directly to roots:

Figure 2: A *Dynagraph* diagram embedded in Word

```
CComPtr<IUnknown> pControl;
CComPtr<IMCCSite> pSite;
// create site in main layer
RUN(m_cont->new_Site(&pSite));
RUN(pSite->put_Layer(m_mainl));

// create control
RUN(CoCreateInstance(CLSID_CShapeNodeCtl,0,CLSCTX_ALL,
      IID_IUnknown,(void**)&pControl));

// size it.
if(hasSize)
      if(qi(IOleObject) oo = pControl)
            hr = oo->SetExtent(DVASPECT_CONTENT,&size);
// create layout node
CComPtr<IDGNode> pNode;
RUN(m_eng->new_Node(&pNode));
// Create connector & init.
CComPtr<IDGMConnector> pConnect;
RUN(CoCreateInstance(CLSID_DGMNodeConnector,NULL,CLSCTX_INPROC_SERVER,
      IID_IDGMConnector,(void **)&pConnect));
RUN(qi(INeedOnePointer)(pConnect)->TakePointer(pNode));
RUN(qi(IObjectWithSite)(pConnect)->SetSite(site));
// set position.
CComPtr<IMCCTransform> t;
RUN(m_cont->get_Transform(&t));
POINTL ptVirt;
RUN(t->W2CP(pt,&ptVirt));
RUN(pNode->put_Pos(pointf(ptVirt)));
// prob. just connector is sinked by now, but you never know.
RUN(intend(site,MCCS_SHOWN));
```

Figure 3: The code required to insert a *Dynagraph* node