PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 30/92

Program Design Using Abstract Data Views - An Illustrative Example

D. D. Cowan
Luis Fernando Barbosa
Roberto Ierusalimschy
Carlos J. P. Lucena
Simone B. de Oliveira

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 30/92
Editor: Carlos J. P. Lucena Novembro, 1992

Program Design Using Abstract Data Views - An Illustrative Example*

D. D. Cowan
Luis Fernando Barbosa
Roberto Ierusalimschy
Carlos J. P. Lucena
Simone B. de Oliveira

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br
techrep@inf.puc-rio.br (for publications only)

Program Design Using Abstract Data Views – An Illustrative Example

D.D. Cowan *L.F. Barbosa R. Ierusalimschy C.J.P. Lucena S.B. de Oliveira

Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro, Brasil
lucena@inf.puc-rio.br

September 25, 1992

Abstract

Creating new applications by integrating user interface and application components is a relatively new idea which is currently of wide interest. A significant part of this problem is clearly defining the separation between user interface and application components. This paper uses an example to illustrate a new design methodology based on the concept of an abstract data view (ADV), a structuring method which cleanly defines this separation. Both the design and a Smalltalk implementation of the example are described. Prototypes of this example and several others which use the ADV concept are currently running in our laboratory.

Resumo

A criação de novas aplicações pela integração dos componentes da interface com o usuário com os componentes da aplicação e uma idéia nova que comeca a ganhar aceitação ampla. Uma parte substancial do problema é a definição clara da separação entre os componentes da interface dos da aplicação. Este artigo usa um exemplo para ilustrar uma nova metodologia de design baseada no conceito de visões abstratas de dados (ADVs), um método de estruturação de programas que define esta separação claramente. O design da solução é uma implementação em Smalltalk são descritos. Protótipos deste exemplo e diversos outros que usam o conceito de ADV estão operacionais em nossos laboratorios.

Key-words: programming techniques, object oriented programming, tools and techniques, software design, abstract data types, abstract data views, interactive applications.

Palavras-chave: técnicas de programação, programação orientada a objetos, ferramentas e técnicas, design de software, tipos abstratos de dados, visões abstratas de dados, aplicações interativas.

*D.D. Cowan is with the Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

1 Introduction

Composing new applications by integrating user interface and application components is a relatively new idea which is currently of topical interest, and various aspects of this problem have been described in the literature ([SG86, Nye90, Mye90, BBG+89, Fol89, KF90, KP88, Har89]). A significant part of this problem is clearly defining the separation between the user interface and the application components so that both of them can be reused in a broad range of applications. A design methodology which clearly addresses this aspect of reuse has the potential to lead to a disciplined approach to application development. This paper illustrates a new design methodology which cleanly separates the user interface from the underlying application server. A key component of this methodology is the notion of an abstract data view (ADV), a general design paradigm for the user interface component. The generality of the ADV approach is illustrated through the design and implementation of a specific example, namely an editor for linear graphs. A formal description of the ADV approach is presented in [CILS92].

2 An Application - A Generic Graph Package

The research project on abstract data views is examining many interactive applications in order to determine the generality of this design approach. One such application, a generic linear-graph package which supports a graphical user interface for editing graphs and allows the implementation of many different graph algorithms, was chosen to thoroughly test the concepts. This graph package must also permit nesting of graphs[1] since many applications such as data-flow diagrams and finite-state machines could use this nesting facility.

Graphs are used to represent many different types of structures and each application area often has a specific way of viewing the graph. For example, electric circuits, process diagrams, maps and Petri nets represent four different methods for viewing graphs. Moreover, some views require quite complex "viewing" algorithms in order to avoid problems such as the intersection of arcs. Thus, the generic graph package and its user interface should be easily separated so that different application-dependent user-interfaces can be used with the same package.

The generic graph package used an object-oriented design and the nodes and arcs were implemented as objects[2]. The initial design tried to follow the Smalltalk Model-View-Controller (MVC) paradigm [KP88] by creating a "graph viewer" that would concentrate all algorithms and data structures related to graphical presentation in the View, and place the algorithms relating to the graph structure in the Model. As the design progressed some disadvantages of the MVC model became evident. Ideally the "view" only needs information about the view or screen positions of individual nodes and arcs, all other information about relationships among the nodes and arcs can be held in the model data structure. However, a direct application of the MVC model needs to store large tables with information for all graph elements, and to link each piece of visual information with its associated object.

[1] That is, each node can be decomposed and presented as a subgraph.

[2] Other representations are also possible.

To solve the problem of duplicating information in the view and to maintain the separation between interface and application, the object model was used not only inside the application, but inside the interface manager as well. Instead of a monolithic "graph viewer", "node viewers" and "arc viewers" were created, where a node viewer is an object that only stores information and algorithms about presentation of and interaction with a node. Therefore, a node viewer does not have data about adjacent arcs or nodes, or anything related to the graph topology. That information belongs to the application, and is stored in the original node and arc objects. The viewer objects are called *Abstract Data Views* (ADVs).

Even though the interface does not store the graph topology, access is often required to this information. To allow this connection, each viewer object has a special variable, called "owner", that refers to the corresponding object in the application.

The design still had to handle nesting, since that feature was required by the initial problem specification, which allowed nodes to be decomposed into subgraphs. Actually, the system already had a restricted form of nesting: ADVs for arcs and nodes can not exist by themselves, floating on the screen. There must be an encapsulation, a visual margin to delimit them. With the nesting capability, this external frame was promoted to the status of an ADV, whose owner is the graph. The fact that the ADVs for nodes and arcs are nested inside this "frame" ADV, implies that they can only be displayed inside this area. Moreover, their position is always interpreted as relative to their external ADV. Any movement or scrolling of the graph is accompanied by movement of the nested ADVs.

From the previous description, it is clear that there is a strong similarity between the concept of nested ADVs and the concept of *subwindows* in window systems. However, there are also several differences. Subwindows are always rectangular areas, while ADVs have their own display methods, and so can have any shape (e.g. the shape of an arc). Subwindows have their position defined relative to the external window. When that is scrolled, they do not scroll together. Finally, there is a difference in the way they are used. ADVs are intended to be created and destroyed much more frequently than subwindows (like nodes and arcs during an editing session), and therefore need a different implementation[3]. Based on that analogy, ADVs are often called *light windows*.

3 An Editor for Linear Graphs

The design approach using ADVs is illustrated with a simplified version of the generic linear-graph package presented in Section 2. Although the design has been reduced in complexity the essential features of ADVs are still required for implementation. This section contains a brief description of the package and subsequent sections outline the design approach and implementation.

An editor for linear graphs consists of a graphical interface supported by appropriate data structures that allows the user to draw a graph by interactively creating and removing its nodes and arcs, where these components are usually represented in a window by circles and line segments. In this example, the graph editor interface is composed of two menus and a working window where the actual drawing appears. There is an element menu which allows the user to choose the type

[3]A good comparison is with processes in Unix systems. Because they are somehow "heavy", many systems implement internal processes, usually called *threads* or *light processes*.

of element (node or arc) and an action menu to specify whether the element is to be created or removed. The elements and actions are selected from the respective menu with a pointing device. Once the element and the action are selected the cursor is positioned in the working window to establish a position to draw or remove a node or an arc.

If the user chooses to create a node, a dialogue will appear requesting the name of the node, the node will then be added to the data structure for the graph and its visual representation will be drawn in the window. A node is removed by selecting the remove command from the menu and then indicating the node with the pointing device. Any arcs which are incident on that node are also removed. Arcs are added by indicating the two nodes to which the arc will be connected. If there is already an arc connecting the two selected nodes, or if the nodes are the same, the action will be ignored. Arcs are removed by selecting the remove command followed by the arc.

4 The Design

This section presents an informal specification of the linear-graph package called GraphEditor and a corresponding user interface called VisualGraphEditor, and illustrates the clean separation between them. There are four basic types in this specification: nodes, arcs, events and sets. The visual representation of nodes and arcs cannot be placed on the screen in the same position, so position is the key used to locate either of these elements in both the user interface and indirectly in the data structures supporting the graph package.

A user action is initiated with the mouse or keyboard and is sent to a server which creates an event. An event contains several components including: the window in which the user action occurred, the exact position in that window and the event type (which mouse button, or which key from the keyboard). The visual graph editor or user interface uses the event and a previous selection from the Action and Element menus to determine whether to create or remove a node or arc.

The design of the graph editor follows the ADV approach and clearly separates the nodes and arcs and their visual representation. Nodes and arcs are represented by the types ADTNode and ADTArc, respectively. Another type called GraphEditor contains the definition of the types ADTNode and ADTArc and the collections of those elements in two sets ADT_NODES and ADT_ARCS. These sets are initially empty.

An ADTNode contains the name of the node, and an ADTArc contains the name of the two nodes to which it is connected. There are four basic functions which manipulate elements of the types ADTNode and ADTArc: CreateADTNode, RemoveADTNode, CreateADTArc, and RemoveADTArc. The function CreateADTNode receives as argument the name of the node to be created, and returns the newly created node. The function RemoveADTNode receives as argument the name of the node to be removed. The function CreateADTArc receives as arguments the names of the nodes to which the new arc should be connected, and returns the newly created arc. The function RemoveADTArc receives as arguments the names of the nodes to which the arc to be removed is connected. An outline of the GraphEditor is shown in Figure 1. The four functions are defined by an informal statement of their pre- and post-conditions.

The visual representation of the nodes and arcs are represented by the types ADVNode and ADVArc, and their corresponding elements are stored in the sets ADV_NODES and ADV_ARCS.

Name = Sequence of characters

Type *GraphEditor*

Declaration: ADT_NODES: $ADTNode$-set
ADT_ARCS: $ADTArc$-set

init mk-GraphEditor(ADT_NODES,ADT_ARCS) $\triangleq$
ADT_NODES = { } $\wedge$ ADT_ARCS = { }

Type *ADTNode*

Declaration: *node_name*: *Name*

Function *CreateADTNode* (*node*: *Name*) *adtnode*: *ADTNode*
external wr ADT_NODES
pre: There is no node with this name in the ADT_NODES set
post: Create and include node in the ADT_NODES set

Function *RemoveADTNode* (*node*: *Name*)
external wr ADT_NODES
pre: There is one node with this name in the ADT_NODES set
post: Remove the node from the ADT_NODES set

End *ADTNode*

Type *ADTArc*

Declaration: *from_node*: *Name*
to_node: *Name*

Function *CreateADTArc* (*from*, *to*: *Name*) *adtarc*: *ADTArc*
external wr ADT_ARCS
pre: There is no arc between the *from* node and the *to* node
and these nodes are not identical
post: Create an arc between the *from* node and the *to* node
and include it in the ADT_ARCS set

Function *RemoveADTArc* (*from*, *to*: *Name*)
external wr ADT_ARCS
pre: There is an arc between these nodes
post: Remove the node between the nodes and from the ADT_ARCS set

End *ADTArc*

End *GraphEditor*

Figure 1: The ADT Graph Editor

$Event_Type = ...$

$Window_ID = ...$

$$
\begin{aligned}
Position :: pos_x &: \{0, ..., 640\} \\
pos_y &: \{0, ..., 200\}
\end{aligned}
$$

$$
\begin{aligned}
Event :: \quad type &: Event_Type \\
window &: Window_ID \\
position &: Position
\end{aligned}
$$

$Action_Type = CREATE\ or\ REMOVE$

$GraphElement_Type = NODE\ or\ ARC$

ADV *VisualGraphEditor* **For Type** *GraphEditor*

Declaration: *Action*: *Action_Type*
GraphElement: *GraphElement_Type*

init mk-GraphEditor(Action, GraphElement) $\triangleq$ Action = CREATE $\wedge$ GraphElement = NODE

ADV *ADVNode* **For Type** *ADTNode*

Declaration: *position*: *Position*
owner: *ADTNode*
ADV_NODES: *ADVNode*-set
ADV_ARCS: *ADVArc*-set

init mk-Graph(ADV_NODES) $\triangleq$ ADV_NODES = { } $\wedge$ ADV_ARCS = { }

Function *CreateADVNode* (*p*: *Position*, *n*: *Name*)

Function *RemoveADVNode* (*p*: *Position*)

Function *RemoveRelatedArcs* (*n*: *Name*)

End *ADVNode*

ADV *ADVArc* **For Type** *ADTArc*

Declaration: *position*: *Position*
owner: *ADTArc*

Function *CreateADVArc* (*p*: *Position*, *from*, *to*: *Name*)

Function *RemoveADVArc* (*p*: *Position*)

End *ADVArc*

Function *DispatchEvent* (*event*: *Event*; *name*1, *name*2: *Name*)

End *GraphEditor*

Figure 2: Type and Prototype Declarations for the Graph Editor ADV

```
Function DispatchEvent (event: Event; name1, name2: Name)
external wr Action
         wr GraphElement
pre:     true
post:    cases event.Window_ID of
                       CREATEID → Action = CREATE
                       REMOVEID → Action = REMOVE
                         NODEID → GraphElement = NODE
                          ARCID → GraphElement = ARC
               WORKINGWindowID → cases Action of
                                   [CREATE] → cases GraphElement of
                                                 [NODE] → CreateADVNode(event.position, name1)
                                                  [ARC] → CreateADVArc(event.position, name1, name2)
                                              end

                                   [REMOVE] → cases GraphElement of
                                                 [NODE] → RemoveADVNode(event.position)
                                                  [ARC] → RemoveADVArc(event.position)
                                              end

                                 end

         end
```

Figure 3: The Function Display for the ADV

Both the types ADVNode and ADVArc and the sets ADV_NODES and ADV_ARCS belong to the type VisualGraphEditor. This type also contains the interface aspects of the application, such as the element and action menus. The type VisualGraphEditor appears in Figure 2, and shows the nested types ADVNode and ADVArc and their corresponding function prototypes.

Assuming the server has a callback mechanism and knows the function corresponding to an event, the design does not need to associate windows directly with event handlers. The screen has five regions, and they are separated into two basic categories: menu windows, and a working window. When an event occurs, a function called DispatchEvent defined in the ADV GraphEditor and shown as a prototype in Figure 2 handles the event. The complete definition of DispatchEvent is shown in Figure 3.

When the mouse is activated in a menu window, the corresponding choice will be noted. The element menu provides the choices NODE and ARC, and the action menu provides the actions CREATE and REMOVE. The element menu is initialized with the NODE selection, and the action

menu with the CREATE selection. It is assumed that exactly one element and one action will be active at one time.

When the mouse is activated within the working window, the appropriate action is performed, based on the last noted menu selection. The type Event contains the event that has occurred, and the position and window where it has occurred. The variable `name1` holds the name of a node and the pair of variables `name1` and `name2` are used to represent the to_node and the from_node of an arc.

Each element of type ADVNode contains the position of the node and its owner (the corresponding ADTNode), and similarly each element of type ADVArc contains the position of the arc and its owner (the corresponding ADTArc). The ADV_NODES and ADV_ARCS collections can be represented by sets, and are initially empty.

There are four basic functions which manipulate the ADV elements: CreateADVNode, RemoveADVNode, CreateADVArc, and RemoveADVArc. There is also an auxiliary function called RemoveRelatedArcs, which removes any arcs connected to a node which is to be removed. The function CreateADVNode receives as arguments the position and the name of the node to be created. The function RemoveADVNode receives as arguments the position of the node to be removed. The function RemoveRelatedArcs receives as argument the name of the node that is going to be removed. The function CreateADVArc receives as arguments the position and the names of the source and target nodes of the arc to be created. The function RemoveADVArc receives as argument the position of the arc to be removed. To remove a visual representation of an arc, the selected position must correspond to an arc. The actual arc (ADTArc) is removed first, and then its visual representation is removed from the collection ADT_ARCS. The four functions are defined in Figure 4 by an informal statement of their pre- and post-conditions.

5 The Implementation

The graph editor was implemented using Smalltalk. The subclasses which involve the concepts of ADVs and ADTs are: VisualGraphEditor, ADVNode, ADVArc, GraphEditor, ADTNode, and ADTArc.

The subclass VisualGraphEditor is responsible for the graph editor interface, for the visual representations of the graph elements (the ADVNodes and ADVArcs), and for the operations upon these visual representations. This subclass also supports the element (node/arc) and the action (create/remove) selection menus. When the user changes the selection on the two menus, the state of both menus is altered. When the user selects a point within the workspace area, the appropriate action is taken: nodes and arcs are either created or removed.

The visual representations of the graph elements (nodes and arcs) consist of the ADVNode and ADVArc classes. The VisualGraphEditor contains collections of those elements stored in Smalltalk dictionaries labelled advNodeDic and advArcDic. The VisualGraphEditor also contains the functions that manipulate these elements. Figure 5 illustrates this implementation.

As described previously, ADVNode contains the node position in the working window, and the owner corresponding to that position, namely a reference to the ADTNode it represents. Similarly the ADVArc contains the arc position and its owner, a reference to its corresponding ADTArc.

The GraphEditor manipulates the graph elements, namely the ADTNode and ADTArc classes.

Function *CreateADVNode* (p: *Position*, n: *Name*)
external wr *ADV_NODES*
pre: There is no node at this position and with this name
post: Insert the node in the ADV_NODES set and draw the node
and ask **CreateADTNode** to add the node

Function *RemoveADVNode* (p: *Position*)
external wr *ADV_NODES*
pre: There is a node at this position
post: Remove it from the ADV_NODES set and call **RemoveRelatedArcs** Function
and remove the node from the screen and so ask the **RemoveADTNode**
to remove the ADTNode

Function *RemoveRelatedArcs* (n: *Name*)
external rd *ADV_ARCS*
pre: true
post: For all arcs do:
If the arc contains the node as a to_node or a from_node
ask the **RemoveADVArc** Function to remove it

Function *CreateADVArc* (p: *Position*, *from*, *to*: *Name*)
external wr *ADV_ARCS*
pre: There is not any arc at this position
post: Include it in the ADV_ARCS set and draw it on the screen
and ask **CreateADTArc** to add the arc

Function *RemoveADVArc* (p: *Position*)
external wr *ADV_ARCS*
pre: There is an arc at this position
post: Remove it from the ADV_ARCS set and remove it from the screen and
ask the **RemoveADTArc** Function to remove the ADTArc

Figure 4: The functions for the Graph Editor ADV

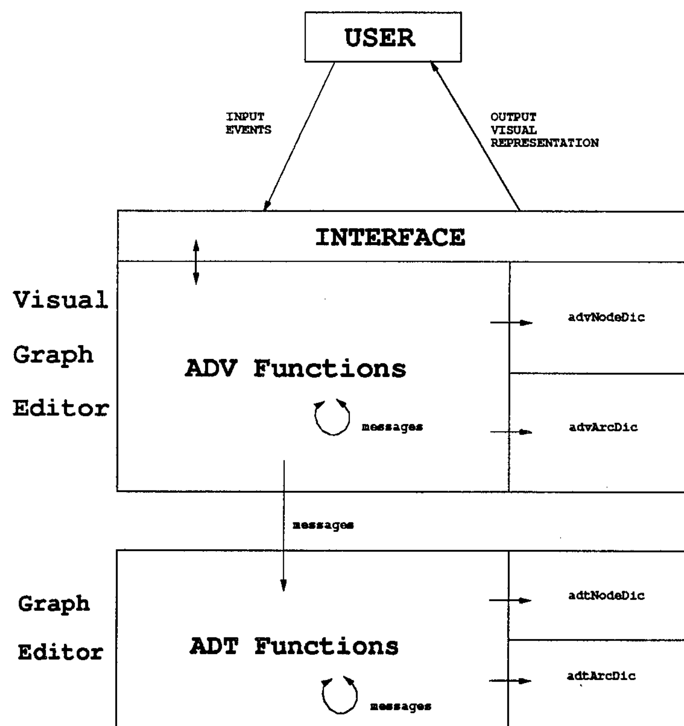

Figure 5: The Smalltalk Implementation

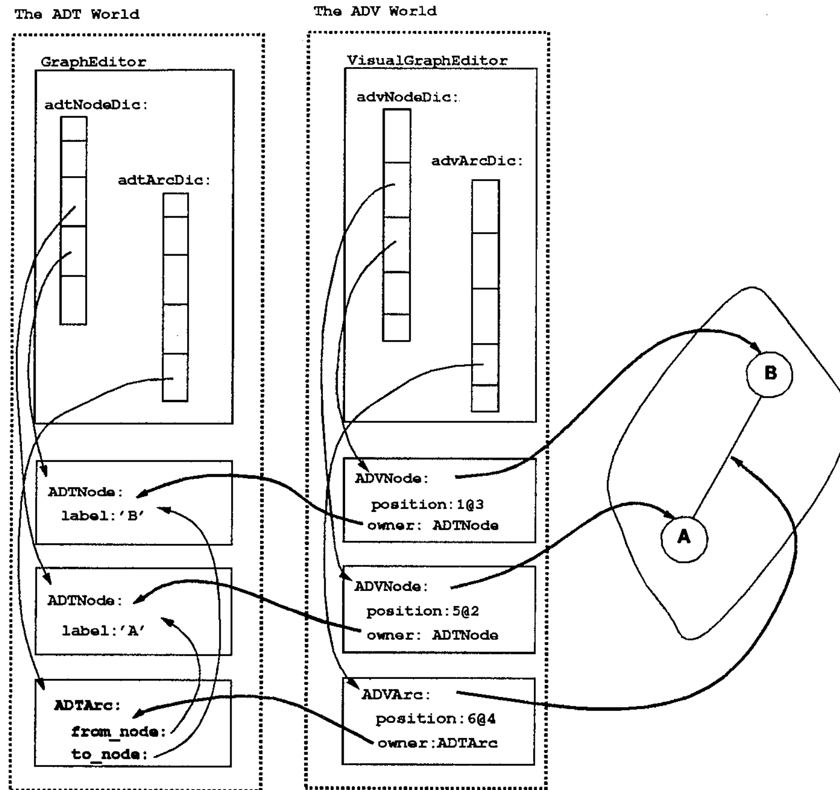

Figure 6: Nodes and Arc

It contains the collections of those elements, in two dictionaries adtNodeDic and adtArcDic, and the functions which manipulate the nodes and arcs. ADTNode contains the node label, and the ADTArc contains the labels of the two nodes to which it is connected.

Figure 6 represents how two nodes and one arc that connects the two nodes, are represented in the Smalltalk implementation. In Figure 6, each dictionary entry has a reference to the corresponding element and the arrows in the Figure indicate that access is from the ADV World to the ADT World.

The reader should notice that there is a clear separation between the ADT World and the ADV World. Moreover, the ADTs have no knowledge whatsoever of the existing ADVs. However, the ADVs must have knowledge of the ADTs they represent and so each one contains a reference to the respective ADT by means of the *owner* variable. This separation makes it possible to create different types of ADVs, thus allowing for the creation of different visual representations for a single collection of ADTs.

Figure 6 also illustrates that there is a visual representation on the screen associated with each ADV. An ADVNode is represented by a circle, and an ADVArc is represented on the screen by a line connecting two nodes.

6 Conclusions

This paper has illustrated by an example, a design method based on Abstract Data Views (ADVs), which clearly separates the application components from the user interface. Thus, different representations of an application component can be presented by connecting them to a different user interface through the owner variable. Hence, this design approach allows both the application components and user interfaces to be reused easily in a wide variety of interactive applications. The feasibility of the Abstract Data View approach has also been demonstrated through an actual implementation in Smalltalk. Although this paper has shown the efficacy of the ADV approach there is still substantial research to be accomplished. Work on formal specification of the concept and programming language-constructs and environments are three areas of significant interest in our current research program.

References

[BBG+89] S. J. Boies, Wm. E. Bennett, J. D. Gould, S. L. Greene, and C. Wiecha. The Interactive Transaction System (ITS): Tools for Application Development. Computer Science RC 14694 (65829), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York, September 1989.

[CILS92] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. Technical Report 92–07, University of Waterloo, Computer Science Department, Waterloo, Ontario, February 1992.

[Fol89] James Foley. Defining Interfaces at a High Level of Abstraction. *IEEE Software*, 6(1):25–32, January 1989.

[Har89] Rex Hartson. User-Interface Management Control and Communication. *IEEE Software*, 26(1):62–70, January 1989.

[KF90] Won Chul Kim and James D. Foley. DON: User Interface Presentation Design Assistant. In *UIST '90 Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 10–20, Snowbird, Utah, USA, October 1990. ACM Press.

[KP88] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP*, pages 26–49, August September 1988.

[Mye90] Brad A. Myers, (editor). The Garnet Compendium: Collected Papers, 1989-1990. Technical Report CMU-CS-90-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1990.

[Nye90] Adrian Nye. *Xlib Reference Manual.* O'Reilly & Associates, 1990.

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.