University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada

Technical Report Series

CS-93-16

Program Design & Implementation With Abstract Data Views

by

A.B. Potengy C.J.P. Lucena D.D. Cowan R. Ierusalimschy

March, 1993

Program Design & Implementation With Abstract Data Views

A.B. Potengy[1,3] C.J.P. Lucena[1] D.D. Cowan[2] R. Ierusalimschy[1]

[1]Depto. de Informática
Pontifícia Universidade Católica
Rio de Janeiro, 22453-900, RJ, Brazil
[lucena,roberto]@inf.puc-rio.br

[2]Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
dcowan@csg.uwaterloo.ca

[3]Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina 110, Rio de Janeiro
22460, RJ, Brazil
potengy@visgraf.impa.br

March 1993

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 simple examples to illustrate a new design and implementation approach based on the concept of an abstract data view (ADV), a structuring method which cleanly defines this separation.

Categories and Subject Descriptors: D.1.5 [**Software**]: Programming Techniques – *Object-oriented Programming*; D.2.2 [**Software**]: Software Engineering – *Tools and Techniques*; D.2.10 [**Software**]: Software Engineering – *Design*; D.2.m [**Software**]: Software Engineering – *Miscellaneous*;

General Terms: Abstract Data Types, Interactive Applications, Programming, User Interfaces

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.

Current improvements in software development techniques such as the Object Oriented Model [Takahashi90, RBL90, Mullin89], are generalizing the concept of reusability through abstract operations such as aggregation/decomposition and generalization/specialization. A design methodology which clearly addresses these aspects of reuse has the potential to lead to a disciplined approach to application development. A key component of one such methodology is the notion of an abstract data view (ADV) [CILS92, CILS93b], a general design paradigm for the user interface component, which allows reuse of Abstract Data Types and their Graphical User Interfaces. Extensive experiments with ADVs have shown that in general they can be mapped into working efficient programs. Besides the examples in this paper, ADVs have been used to interconnect modules in a user interface design system (UIDS), to support concurrency in a cooperative drawing tool, and to design a ray tracer which was then implemented in a distributed environment. This last program operates in a client-server configuration.

This article discusses a design approach using Abstract Data Views (ADVs) that permits the reuse of user interface objects by clearly separating them from their corresponding application objects. The generality of the ADV approach is illustrated through the design and implementation of a number of examples, including an editor for linear graphs.

2 The ADV concept

All the systems mentioned in the previous section have two fundamental ideas in common: the separation of application objects from their graphical user interfaces, and the presence of some kind of object/view pairs. These ideas are combined in the ADV Model.

An Abstract Data View (ADV) [CILS92] might be called a visual realization of an abstract data type (ADT) because it has many of the properties of the latter. However, *views* are restricted to the user interface aspects of applications. An ADV is essentially a "visual object", that is, an object which has a graphical representation in a window, an entity with which a user can interact through devices such as the mouse and the keyboard. A visual object represents graphically an object of an application which, otherwise, has no "external representation". Thus, an ADV separates the application objects from their graphical user interfaces in terms of object/view pairs.

Moreover, abstract data views can be nested. The nesting capability shows itself on the screen, where each view is drawn inside its parent region. This feature is a generalization of the concept of subwindow, since views are form-free, that is, they do not necessarily have the conventional rectangular shape.

Nesting also allows for an external ADV (associated with a different application) to be used as a component of another ADV. This allows, for instance, the insertion of images generated by a draw-package inside text being manipulated in a text editor, allowing the user to interact with the images [CILS93b].

ADVs can also be specialized. An existing view for an object can be used as a basis for defining other views for object or other ADT subclasses. ADVs should support inheritance abstractions for this kind of composition. This characteristic allows the programmer to apply to the ADVs the same abstract composition operations applied to the application ADTs.

An ADV is similar to an ADT in that an ADV also possesses an internal state and a set of operations. The set of operations of an ADV are the ones found in general-purpose GUIs: operations to manage input events such as mouse movements and clicks, and to perform output activities such as drawing, resizing and scrolling. The ADV approach assumes that the application ADTs never include operations related to user interface aspects of the applications. The ADVs represents all user interface aspects of ADTs.

Each view may or may not be connected to an object of an application. The correspondence is specially useful in WYSIWYG interfaces, because it allows the user to see and manipulate every individual object inside a program. ADTs also may be disconnected from a view. Figure 1 illustrates the ADV x ADT relation and a clear separation between visual and application objects.

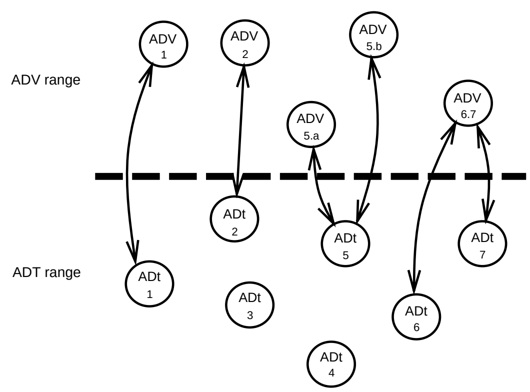

Figure 1: ADV X ADT

3 A GUI Toolkit Library for ADVs

This section describes the structure of a GUI toolkit library designed to support the ADV concept. Most objects in the toolkit are based on the Open Look [Kannegaard88] standard. These objects inherited their basic functionality from the ones already available in the XView toolkit [HDN, Jones89]. Some additional features were added to make the new objects suitable for composition mechanisms such as inheritance and aggregation. This library is composed of a set of objects that constitute the user interface. Inheritance mechanisms are used to provide extensions. Figure 2 shows the inheritance tree for some relevant objects. An example of how to use this library is shown in the next section. The objects used in the example shown in Figure 5 are:

GUI Objects. The highest level class in the GUI system is *GUIObject*. There are no instances of pure *GUIObjects*, since it is an abstract class. Every class with the purpose of establishing graphics interaction between users and applications will be called a GUIObject. For example, *ADVs* as well as *Menus*, are *GUIObjects*

Window Objects. *WindowObjects* are objects which can be placed in *Windows*. These objects are listed in the *WindowBody*. Every *WindowObject* is associated with a *Window*, i.e.,

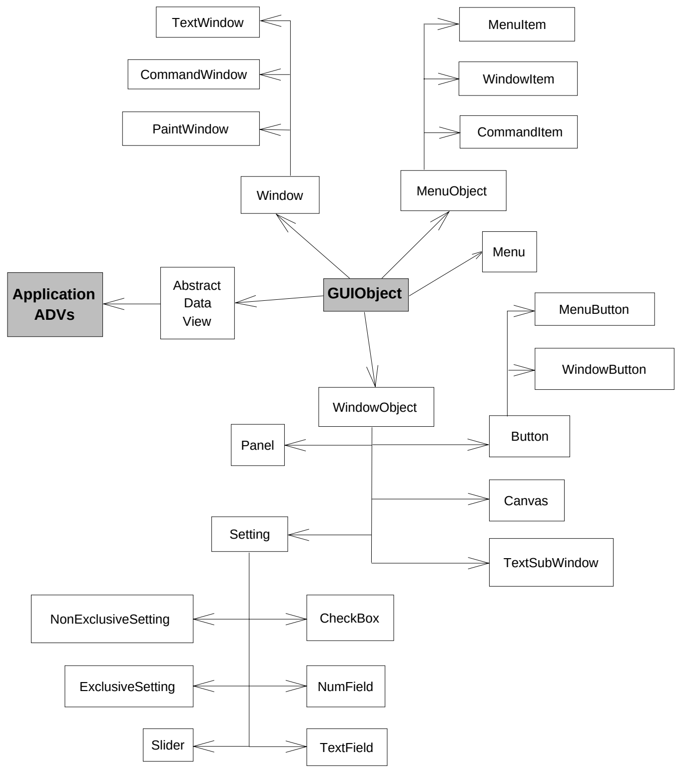

Figure 2: GUI Objects Inheritance Structure

belongs to a *WindowBody* set of elements. Every *Window* has a *WindowBody*, and when a *WindowObject* is created it is included in a *Window*.

Buttons. A *Button* is a *WindowObject* associated with an "action"; it performs an operation corresponding to an event. When the user "clicks" a *Button*, a notify procedure or action is called.

Settings. There is a whole class of *WindowObjects* called *Settings* that are associated with object attributes. Typically, a *Setting* such as the *NumField*, the *TextField* and the *Slider* have an associated value; this value is directly supplied by the user.

3.1 ADV Objects

The ADV-model related classes are simple abstract classes that application programmers can use by plugging one class into another (here plugging A into B means making A a subclass of B). Each *Abstract Data Type* (ADT) class, which is supposed to communicate with the user via ADVs inherits the properties of the general Interactive class. Thus, we say that an instance of an ADT class which communicates with a user is also an Interactive object. The ADV subclass is specified reflecting the Interactive subclass structure, and providing access only to its public members. Figure 4 illustrates the concept.

Specification *ADV*
  Subtype of *GUIObject*

  declaration *main_window*: *Window*
               intobj: *Interactive*

  Constructor *CreateADV* (*iobj*: *Interactive*)

  Operation *UpDate* ()
  post // ADV subclass dependent procedure
End *ADV*

Specification *Interactive*

  declaration *class_name*: $Char^*$
               avds: ADV^*

  Constructor *CreateInteractive* (*name*: $Char^*$)

  Operation *UpDate* ()
  post for $i \leftarrow 1$ `to` len *advs*
      do *advs*[*i*]. *UpDate*()
      end
End *Interactive*

Figure 3: The specifications of the ADV and Interactive classes

The ADV class in Figure 3 contains at least a reference to a default enclosing Window and a reference to an Interactive object. The ADV is a free form enclosed interface object. Each instance of an ADV is related only to one Interactive object at a time, but many ADV objects can be related to the same Interactive object at the same time. For example, in Figure 4, the A_ADV instance A_ADV_1 is associated with the instance A_1 of A. The instance B_ADV_1 sometimes is linked to B_1 and sometimes to B_2. There are also two instances of C_ADV class associated with the same instance C_1 of C. Of course there must be a mechanism to tell the ADV to which instance it is related. However, because the way instances are stored is application dependent, the application designer will be in charge of specifying the association mechanism. The specifications in Figure 3 describe the ADV and Interactive classes in a VDM-like notation [Ier91a] similar to a programming language, so no familiarity with VDM should be required of the reader.

There is an ADV method that updates the GUIObjects contained in the corresponding ADV object on request. When an Interactive object is changed by an ADV, by itself, or by any object internal to the application, it will send a message to all associated ADVs, requesting that they update themselves. To do this, the Interactive class contains a list of references to ADV objects.

Our experience also tells us that ADVs should be able to send messages to each other. This only makes sense with ADVs that are related to the same object (directly or indirectly), so that the best object to manage the message traffic is the Interactive object. An ADV subclass that is as general as its associated class provides reusability possibilities by applying to ADVs the same abstract operations as are applied on ADTs. For example, in Figure 4, class D is a subclass of B and C, since its corresponding ADV is built in the same way, the class D_ADV is a subclass of B_ADV and C_ADV.

4 Writing ADV-Based Applications

In this section we illustrate the basic methods of constructing ADVs by mimicking the structure of ADTs and the operations of aggregation, multiple inheritance and composition. These methods are then used in a generic graph package for constructing and visualizing linear graphs.

4.1 The Basic Construction Approach

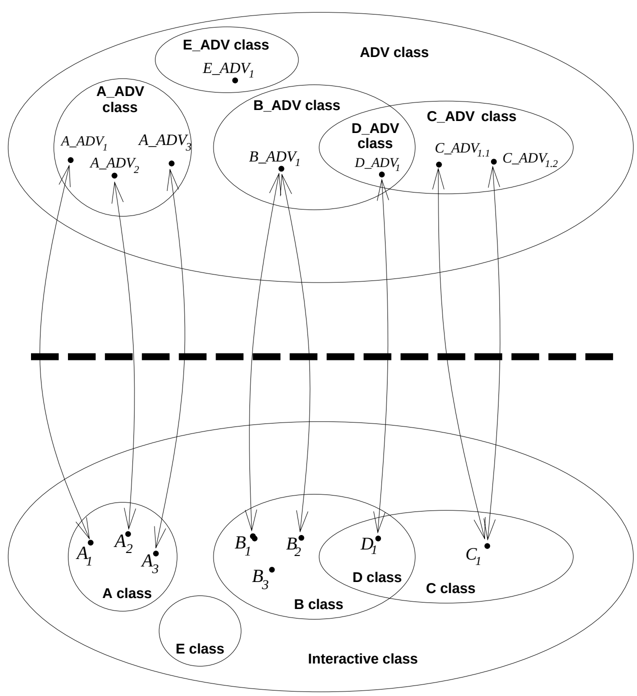

Figure 4: ADV x ADT revised

This section presents a simple example using the ADV objects described in Figure 5. Figure 6 illustrates a view of a trivial class called A. The class A is an interactive subclass and is composed of a single integer component called $Aattr1$. The only method for this class is *Print*. The specification for the class A is shown in Figure 7.

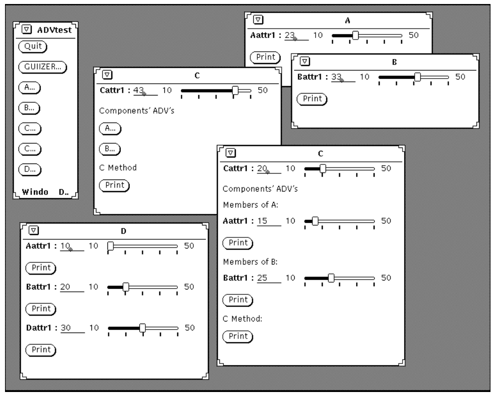

Figure 5: An ADV based example

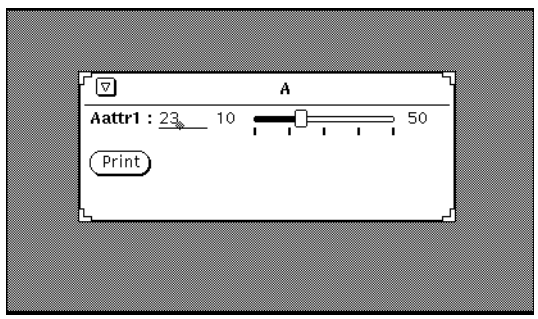

Figure 6: An A_ADV instance

This specification in Figure 7 can be easily implemented in C++ [Stroustrup90] by using inheritance. Once class A is defined as an *Interactive* subclass, the invariant properties of the latter are preserved. The operation *Print* can be implemented as an A method. Doing this, the *Print* method does not need to receive arguments because it is already associated with an instance of class A.

The Abstract Data View for class A is composed of a *Slider* subclass (for A's integer attribute) and a *Button* subclass (for A's Print method). These two *GUIObject* subclasses are defined as members of *A_ADV*. The operations on these classes are very simple, since they inherited the GUIObject's functionality. If we were using a programming language without inheritance and polymorphism, interface coding would be necessary to simulate these mechanisms. In fact the Slider is an ADV for a "class Integer" and the Button is an ADV for a "class Method". These two ADVs (not Windows) are aggregated to form the ADV for class A, both being enclosed in a single Window with label "A" as described in the specification in Figure 8.

Figure 9 shows a view of another trivial class called B. B and its visual representation (B_ADV) shown in Figure 10 were implemented by mimicking the definition of A and A_ADV.

Having defined the two trivial classes A and B and their corresponding ADVs, we are able to perform many kinds of composition experiments. The specification for C as an aggregate with components A and B, is shown in Figure 13. Its Abstract Data View (C_ADV) where the specification is shown in Figure 14, is also defined as an aggregate with components A_ADV and B_ADV. A

Specification *A*
  Subtype of *Interactive*

  declaration $Aattr1$: *Integer*

  Constructor *CreateA* (*attr*: *Integer*)
  post $Aattr1 = attr \wedge CreateInteractive("A")$

  Operation *Print* ()
  pre true
  post Send description of instance to std output
End *A*

Figure 7: The specification of the subtype A

visual representation of an instance of (C_ADV) is shown in Figure 11.

Note from Figure 14 that to define the ADV for class C we **are not concerned** with the contents of *A_ADV* and *B_ADV*. The definition of *C_ADV* is a simple composition just like C. Again, the basic structure of the abstract data type is preserved in its graphical representation (or abstract data view). Another interesting fact is that C++ provides a nice syntax tool to implement ADVs. Once we defined the class C_ADV it was trivial to create a new visual representation for class C by just defining a C_ADV subclass with a different visual implementation as shown in Figure 12.

Now consider a multiple inheritance composition of A and B to build a class called D, and with one more attribute $Dattr1$ and another method *Print*. The class D_ADV is implemented the same way, by inheriting A_ADV and B_ADV. Figure 15 shows the resulting ADV. This is another powerful tool the ADV concept provides. Almost no additional code is needed to implement an inherited ADV. The specifications for the design of class D and its ADV is described in Figure 17.

The main idea is to show that programming within the ADV model may be considered as a two-step activity. First, define application objects without paying attention to their GUIs. Once there is a well defined environment model we can build its visual representation. If the model is composed of a set of composite abstract data types, its visual representation will be a set of composite abstract data views consistent with the underlying model.

4.2 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 test the concepts thoroughly. This graph package must also permit nesting of graphs[1] since many applications such as data-flow diagrams and finite-

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

Specification A_ADV
  Subtype of ADV
    redefine $UpDate$

declaration as: $Aattr1Slider$
  ab: $APrintButton$
  $owner$: A

Specification $Aattr1Slider$
  Subtype of $Slider$
    redefine $SaveValue$

Operation $CreateAattr1Slider$ ()
ext rd $main\_window$
  wr $owner$
post $CreateSlider(main\_window, "Aattr1:") \wedge Value = owner.Aattr1 \wedge$
  $min = 10 \wedge max = 50$

Operation $SaveValue$ ()
ext wr $owner$
post $owner.Aattr1 = Value$

End $Aattr1Slider$

Specification $APrintButton$
  Subtype of $Button$
    redefine $Action$

Operation $CreateAPrintButton$ ()
ext rd $main\_window$
  wr $owner$
post $CreateButton(main\_window, "print")$

Operation $Action$ ()
ext rd $owner$
post $owner.Print()$

End $APrintButton$

Constructor $CreateA\_ADV$ $(a: A)$
post $owner = a \wedge CreateADV(a) \wedge$
  $as = CreateAattr1Slider() \wedge ab = CreateAPrintButton()$

Operation $UpDate$ ()
post $as.SetValue(Aattr1)$

End A_ADV

Figure 8: The specification of the ADV for class A

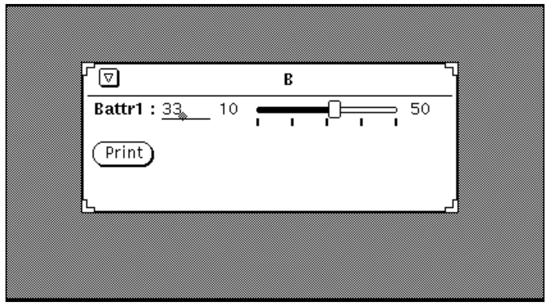

Figure 9: A B_ADV instance

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 a 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. 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 in the view, and to link each piece of visual information with its associated object.

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 *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

Specification *B*
  Subtype of *Interactive*
  declaration *Battr*1: *Integer*
  Constructor *CreateB* (*attr*: *Integer*)
  Operation *Print* ()
End *B*

Specification *B_ADV*
  Subtype of *ADV*
    redefine *UpDate*
  declaration *bs*: *Battr*1*Slider*
    bb: *BPrintButton*
    owner: *B*

  Specification *Battr*1*Slider*
    Subtype of *Slider*
      redefine *SaveValue*
    Operation *CreateBattr*1*Slider* ()
    Operation *SaveValue* ()
  End *Battr*1*Slider*

  Specification *BPrintButton*
    Subtype of *Button*
      redefine *Action*
    Operation *CreateBPrintButton* ()
    Operation *Action* ()
  End *BPrintButton*
  Constructor *CreateB_ADV* (*b*: *B*)
  Operation *UpDate* ()
End *B_ADV*

Figure 10: The specification of the ADV for class B

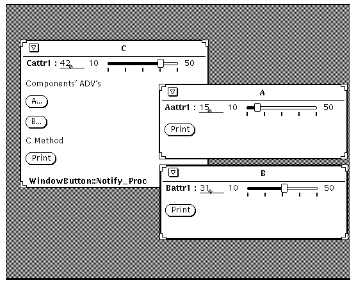

Figure 11: A C_ADV instance

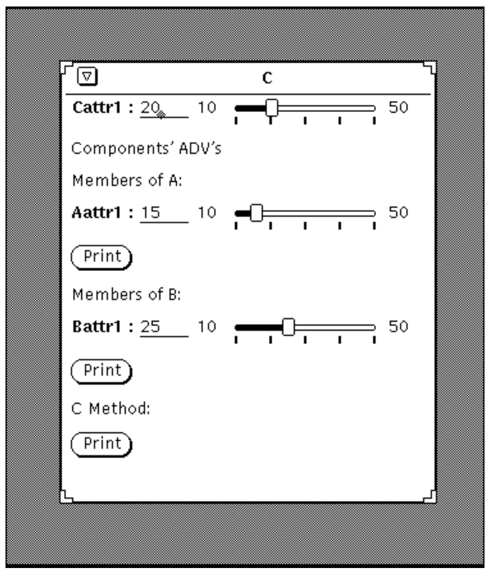

Figure 12: A C_ADV subclass instance

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[2].

[2]Based on that analogy, ADVs are often called *light windows*. A good comparison is with processes in Unix

Specification *C*
Subtype of *Interactive*

declaration *Cattr*1: *Integer*
a: *A*
b: *B*

Constructor *CreateC* (*attr*: *Integer*)

Operation *Print* ()

End *C*

Figure 13: The specification of class C

4.3 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.

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 Node and Arc, respectively. Another type called GraphEditor contains the definition of the types Node and Arc and the collections of those elements in two sets NODES and ARCS. These sets are initially empty.

A Node contains the name of the node, and an Arc contains references to the two nodes to which it is connected. There are four basic functions which manipulate elements of the types Node and Arc: CreateNode, RemoveNode, CreateArc, and RemoveArc. The function CreateNode receives as argument the name of the node to be created, and returns the newly created node. The function RemoveNode receives as argument the name of the node to be removed. The function CreateArc receives as arguments the names of the nodes to which the new arc should be connected, and returns the newly created arc. The function RemoveArc receives as arguments the names of the nodes to which the arc to be removed is connected. An outline of the GraphEditor with the four function prototypes is described in Figure 18. The four functions used in the GraphEditor are defined by an informal statement of their pre- and post-conditions in Figure 19.

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. 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 and the specification of these elements is shown in Figure 20. The type VisualGraphEditor appears in the specification in Figure 21, and shows the nested types ADVNode and ADVArc and their corresponding function prototypes.

---

systems. Because they are somehow "heavy", many systems implement internal processes, usually called *threads* or *light processes*.

Specification C_ADV
  Subtype of ADV
    redefine $UpDate$

  declaration cs: $Cattr1Slider$
    cb: $CPrintButton$
    a_adv: A_ADV
    b_adv: B_ADV
    $owner$: C

  Specification $Cattr1Slider$
    Subtype of $Slider$
      redefine $SaveValue$

    Constructor $CreateCattr1Slider\ ()$

    Operation $SaveValue\ ()$

  End $Cattr1Slider$

  Specification $CPrintButton$
    Subtype of $Button$
      redefine $Action$

    Constructor $CreateCPrintButton\ ()$

    Operation $Action\ ()$

  End $CPrintButton$

  Constructor $CreateC\_ADV\ (c{:}\ C)$
  post $owner = c \wedge CreateADV(c) \wedge$
    $cs = CreateCattr1Slider() \wedge cb = CreateCPrintButton() \wedge$
    $a\_adv = CreateA\_ADV(c.a) \wedge b\_adv = CreateB\_ADV(c.b)$

  Operation $UpDate\ ()$
  post $cs.SetValue(Cattr1) \wedge a\_adv.UpDate() \wedge b\_adv.UpDate()$

End C_ADV

Figure 14: The specification of the ADV for class C

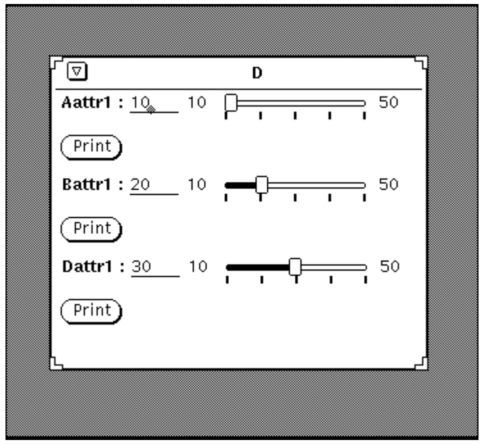

Figure 15: A D_ADV instance

Specification *D*
  Subtype of A, B
  declaration *Dattr*1: *Integer*
  Constructor *CreateD* (*attr*: *Integer*)
  Operation *Print* ()
End *D*

Figure 16: The specification of class D

Each element of type ADVNode contains the position of the node and its owner (the corresponding Node), and similarly each element of type ADVArc contains the position of the arc and its owner (the corresponding Arc). 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 (Arc) is removed first, and then its visual representation is removed from the collection ARCS. The specification for the four functions

Specification D_ADV

Subtype of ADV

redefine $UpDate$

Subtype of A_ADV

rename $UpDate$ as $AUpDate$

Subtype of B_ADV

rename $UpDate$ as $BUpDate$

declaration $ds\colon Dattr1Slider$
$db\colon DPrintButton$
$owner\colon D$

Specification $Dattr1Slider$

Subtype of $Slider$

redefine $SaveValue$

Constructor $CreateDattr1Slider$ ()

Operation $SaveValue$ ()

End $Dattr1Slider$

Specification $DPrintButton$

Subtype of $Dutton$

redefine $Action$

Constructor $CreateDPrintButton$ ()

Operation $Action$ ()

End $DPrintButton$

Constructor $CreateD\_ADV$ $(d\colon D)$

post $owner = d \wedge CreateA\_ADV(d) \wedge CreateB\_ADV(d)$

End D_ADV

Operation $UpDate$ ()

post $ds.SetValue(Dattr1) \wedge AUpDate() \wedge BUpDate()$

Figure 17: The specification of the ADV for class D

Specification *GraphEditor*
  Subtype of *Interactive*

  declaration $NODES$:Node-set
               $ARCS$:Arc-set

  init mk-GraphEditor(NODES,ARCS) $\triangleq$
    NODES = { } $\wedge$ ARCS = { }

  Specification *Node*
    Subtype of *Interactive*

    declaration *node_name*: *Name*

    Constructor *CreateNode* (*node*: *Name*)

    Destructor *RemoveNode* (*node*: *name*)
  End *Node*

  Specification *Arc*
    Subtype of *Interactive*

    declaration *from_node*: *Name*
                 to_node: *Name*

    Constructor *CreateArc* (*from*, *to*: *Name*)

    Destructor *RemoveArc* (*from*, *to*: *Name*)
  End *Arc*
End *GraphEditor*

Figure 18: The specification of the Graph Editor

Constructor *CreateNode* (*node*: *Name*)
ext wr *NODES*
pre There is no node with this name in the NODES set
post Create and include node in the NODES set

Destructor *RemoveNode* (*node*: *name*)
ext wr *NODES*
pre There is one node with this name in the NODES set
post Remove the node from the NODES set

Constructor *CreateArc* (*from*, *to*: *Name*)
ext wr *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 ARCS set

Destructor *RemoveArc* (*from*, *to*: *Name*)
ext wr *ARCS*
pre There is an arc between these nodes
post Remove the node between the nodes and from the ARCS set

Figure 19: The specification of the four functions for the Graph Editor

$Event\_Type = ...$

$Window\_ID = ...$

$$\begin{aligned} Position :: \; pos\_x &: \{0, ..., 640\} \\ pos\_y &: \{0, ..., 200\} \end{aligned}$$

$$\begin{aligned} Event :: \; type &: Event\_Type \\ window &: Window\_ID \\ position &: Position \end{aligned}$$

$Action\_Type = CREATE \; or \; REMOVE$

$GraphElement\_Type = NODE \; or \; ARC$

Figure 20: The specification of the element and action menus in the ADV for the Graph Editor

Specification *VisualGraphEditor*
Subtype of *ADV*

declaration *Action*: *Action_Type*
GraphElement: *GraphElement_Type*
vge_owner: *GraphEditor*
init mk-GraphEditor(Action, GraphElement) $\triangleq$ Action = CREATE $\wedge$ GraphElement = NODE

Specification *ADVNode*
Subtype of *ADV*

declaration *position*: *Position*
owner: *Node*
ADV_NODES:ADVNode-set
ADV_ARCS:ADVArc-set
init mk-Graph(ADV_NODES) $\triangleq$ ADV_NODES = { } $\wedge$ ADV_ARCS = { }

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

Destructor *RemoveADVNode* (*p*: *Position*)

Operation *RemoveRelatedArcs* (*n*: *Name*)

End *ADVNode*

Specification *ADVArc*
Subtype of *ADV*

declaration *position*: *Position*
owner: *Arc*

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

Destructor *RemoveADVArc* (*p*: *Position*)

End *ADVArc*

DispatchEvent (*event*: *Event*; *name*1, *name*2: *Name*) $\triangleq$

End *VisualGraphEditor*

Figure 21: The specification of the ADV for the Graph Editor

with informal statements about their pre- and post-conditions are described in Figure 22.

4.4 The Implementation

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

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 labeled advNodeDic and advArcDic. The VisualGraphEditor also contains the functions that manipulate these elements [C+92].

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

The GraphEditor manipulates the graph elements, namely the Node and Arc classes. It contains the collections of those elements, in two dictionaries adtNodeDic and adtArcDic, and the functions which manipulate the nodes and arcs. Node contains the node label, and the Arc contains the labels of the two nodes to which it is connected.

Figure 23 represents how two nodes and one arc that connects the two nodes, are represented in the Smalltalk implementation. In Figure 23, 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 23 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.

5 Conclusions

This paper has illustrated by examples, 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.

Constructor *CreateADVNode* (p: *Position*, n: *Name*)
ext 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 **CreateNode** to add the node

Destructor *RemoveADVNode* (p: *Position*)
ext 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 **RemoveNode**
to remove the Node

Operation *RemoveRelatedArcs* (n: *Name*)
ext 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

Constructor *CreateADVArc* (p: *Position*, *from*, *to*: *Name*)
ext 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 **CreateArc** to add the arc

Destructor *RemoveADVArc* (p: *Position*)
ext 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 **RemoveArc** Function to remove the Arc

Figure 22: The specification for the four ADV functions

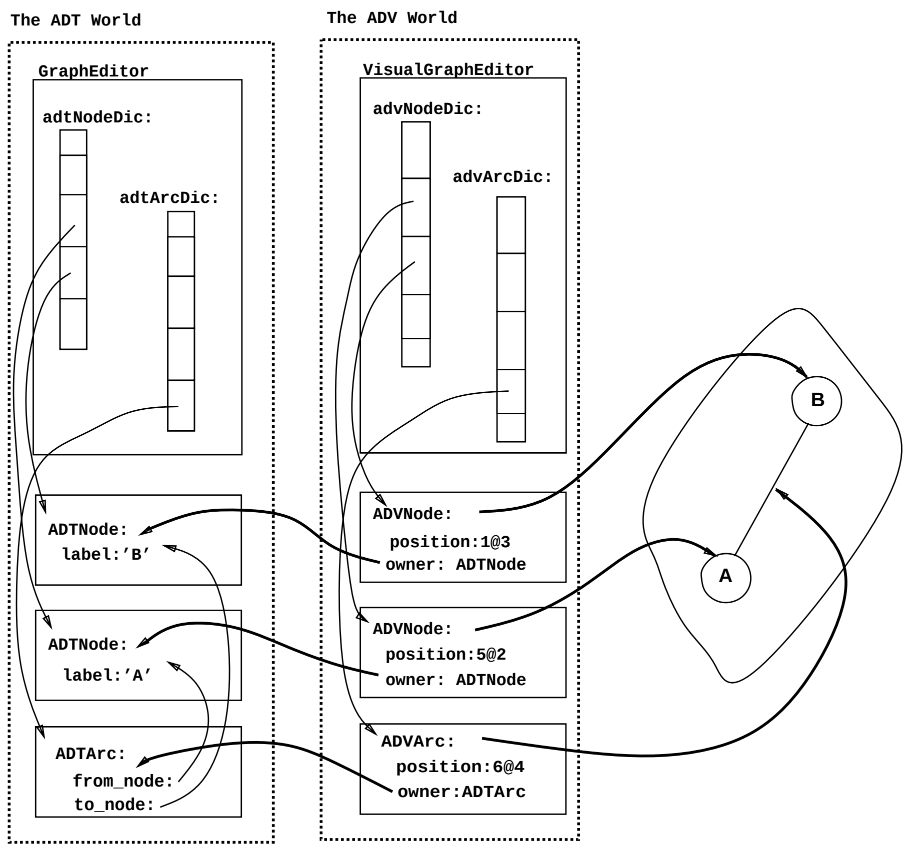

Figure 23: Nodes and Arc

The feasibility of the Abstract Data View approach has also been demonstrated through actual implementations in Smalltalk and C++. 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 programming environments are three areas of significant interest in our current research program.

The ADV model was presented as a standard way of building user interfaces by intensive use of compositions. Such operations (like aggregation and inheritance) offer reusability, keeping the GUI structure consistent with the associated application objects.

The examples presented here made use of very simple classes. This approach is appropriate for studying composition possibilities. Now suppose there is a very large system containing thousands of complex classes without consistent graphical user interfaces. How much effort would be spent in developing non-trivial user friendly applications? In the ADV model, when a class is constructed, so is its GUI. After having combined the classes in the application, one can build the application GUI using the same combinations we used for the ADTs.

Two other important results from the use of ADVs are GUI compatibility and encapsulation. Within this programming process, large systems (including their GUIs) can be combined using abstract operations without worrying about their internal structures. We believe that an

ADV/GUIDE is an appropriate tool for programmers whose goal is the fast development of reusable user-friendly object-oriented applications.

Extensive experimentation with the ADV concept for the design of man-machine interfaces led us to believe that the concept could be generalized and extended to model internal module or object interconnection interfaces. Thus, we are examining ADVs in this context. Further experimentation with generalized ADVs has illustrated its use in the design of concurrent and distributed software systems as well as sequential systems.

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.

[C+92] D. D. Cowan et al. Program Design Using Abstract Data Views–An Illustrative Example. Technical Report 92–54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[CCLP92] M. Coffin, D. D. Cowan, C. J. P. Lucena, and A. B. Potengy. Distributed Abstract Data Views: Design and Implementation. Technical Report 93–16, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, November 1992.

[CILS92] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. *Abstract Data Views. Structured Programming*, 14(1):1–13, January 1993.

[CILS93b] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.

[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.

[HDN] Heller, Dougherty and Nyr. *Overview of XView Programming.* O'Reilly & Associates Inc..

[Ier91a] Roberto Ierusalimschy. A Method for Object-Oriented Specifications with VDM. Technical report, Monografias em Ciência da Computação, PUC-Rio, February 1991.

[Jones89] O. Jones. *Introduction to the X Window System.* Prentice-Hall International Editions, 1989.

[Kannegaard88] J. Kannegaard. *Open Look Industry / Outlook / Overview.* Sun Technology, pp 58–62, Autumn 1988.

[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.

[Mullin89] M. Mullin. *Object Oriented Program Design With Examples in C++.* Addison-Wesley 1989.

[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.

[RBL90] R. Wirfs-Brock, B. Wilkerson and L. Wiener *Designing Object-Oriented Software.* Prentice Hall, New Jersey, 1990.

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

[Stroustrup90] B. Stroustrup and M. Ellis. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Takahashi90] T. Takahashi and H. K. E. Liesenberg. *Programação Orientada a Objetos. VII Escola de Computação*, São Paulo, 1990.