A Programming Model for User Interface Compositions

A.B. POTENGY[1,2]
C.J.P. LUCENA[2]
D.D. COWAN[3]

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

[2]Depto. de Informática
Pontifícia Universidade Católica
Rio de Janeiro, 22460, RJ, Brazil
lucena@inf.pub-rio.br

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

Abstract. Current improvements in software development techniques are changing the concept of reusability to a more general approach. In this context, the Object Oriented Model has been providing great advances to the programming process. One advance is the reuse of objects by abstract operations like aggregation/decomposition and by generalization/specialization, in contrast with the old fashioned "cut and paste then modify" paradigm of Structured Programming. The Abstract Data View (ADV) model is a clean alternative for reusing Abstract Data Types and their Graphics User Interfaces. This article discusses mechanisms for reusing user interfaces by compositions, as well as development environments designed to support construction of GUIs. An ADV-based graphics user interface development environment is described, providing a basic framework for experiments. We are interested in identifying the benefits of applying the ADV model to a programming process.

1 Introduction

A Graphics User Interface Development Environment (GUIDE) is a set of tools designed to improve programmer's performance in creating high quality interactive applications. Such an environment (see figure 1) typically receives a GUI design and an application code as input and returns a well behaved interactive application as output.

Although command-driven applications can be considered to be interactive, once they establish some level of dialog with the user, we are concerned with user-friendly window applications. Most GUIDE systems available today were created to solve a set of problems raised by the increase of user needs. This set of problems typically involves the integration of multiple media (text, graphics, images, video and audio) with underlying applications (such as DBMS and spreadsheets) into a single tool, allowing both user input and media output. We describe below simple examples of what we call composite interactive documents (integrating only text, graphics and underlying applications). These problems were addressed in [8], where solutions are presented based on the Abstract Data View model described later.

1.1 Composite Interactive Documents

1.1.1 Example 1

Consider a simple composite interactive document requiring a picture in the middle of text. We open a window of a drawing program inside the text area and proceed to produce a picture using the usual draw commands. The window containing the text document and the text document are called the *enclosing window* and the *enclosing document*, respectively; similarly, the drawing window and the drawing document are called the *enclosing window* and *enclosing document*. Of course, these definitions ap-

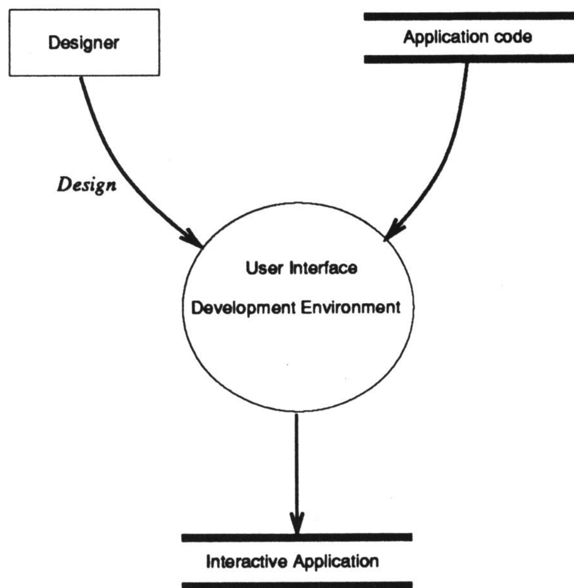

Figure 1: User Interface Development Environment

ply recursively; an enclosed document can become an enclosing document, since it can contain an enclosed window. The enclosed window can be manipulated in the usual way within the enclosing document, that is it can be opened, closed, moved and resized.

In this example the enclosed document displays the user interface of the drawing package. The enclosing and enclosed documents are independent, as they do not need to communicate their contents to each other: the drawing does not affect the text, and the text does not affect the drawing. However, the enclosing document and the enclosing window must communicate some kinds of controls, since both can be affected by commands to open, close, resize, move or print the enclosed window.

1.1.2 Example 2

Consider a document where a table representing the result of an SQL database is to be placed in an enclosed window within the enclosed document. We open an SQL query window on the screen for input (not in the enclosing document), where we type the query, and also an enclosed window in the document for the result of the query.

This example requires that both an input and output window be opened for the SQL query, the output window being an enclosed window. The output from the SQL query must be directed to the enclosed window.

Example 3

In this example the enclosing document is a spreadsheet with one cell containing a SQL database query that produces a single numerical result. Computing numbers by a SQL server and placing it in the spreadsheet data structure could cause a recalculation of the contents of several other cells in the spreadsheet. The cell containing the query is both the enclosed window and the enclosed document.

When the cell with the query is recognized as containing a foreign element, it is passed to the appropriate application for computation. Upon completion of the calculation, the results of the query are returned to the spreadsheet data structure. The interaction is strictly among the SQL server, the spreadsheet data structure, and the program; the SQL server does not link to the spreadsheet user interface. The solution to this problem is already solved in several commercial software systems.

1.2 Multi-level Graph Editor

Another application of interest is what we call a multi-level graph editor. A simple graph editor allows users to build graph diagrams interactively by grouping simple graphics primitives representing nodes and links. In a multi-level graph, nodes are actually anchors to enclosed graphs. For example, if the user clicks on a node, a new window containing the graph associated with that node appears on the screen. Another way to activate an enclosed graph is by continuously zooming on a node until the associated graph becomes visible.

This example can be used as a reusable component to build applications like Petri-Nets editors and simulators, Data-Flow-Diagrams Editors, etc., as multi-level graph editor subclasses. There are many graph-representable models in our environment that can be implemented by changing the constraints over graph editor subclasses. Some systems allow this capability, although they do not necessarily address this problem as a starting point in their development.

1.3 3D Graphics Viewers

The current state-of-the-art GUIDEs enforce the separation between user interface objects and application objects. Such a feature makes it possible, for example, to associate multiple instances of different UI objects to the very same instance of application object. Some objects act on mutually exclusive subsets of the application object members, while others act on the same data or call the same methods. Having multiple instances of user interface objects acting

on the same instance of application object provides great flexibility. However, the development of applications using that kind of facility requires careful design.

Think of a view as a window where a rendering algorithm output is displayed. Consider, for example, a *wire-frame view* and a *ray-tracing view* representing a model. Suppose the user is able to pick a polygon in the model by generating an appropriate event in the view where it is displayed. If the user performs some operations on this polygon (like changing its color), there should be a automatic mechanism to propagate the effect of the operation to other views that are associated with the same model. However, if the user is able to change light source parameters in the ray-tracing view, these changes do not need to be propagated to wire-frame views, whereas it is probably necessary to update other ray-tracing views.

It is possible to solve these problems with most systems available today. However, the amount of work to be spent on the solutions and the quality of these solutions depend on the quality of the concepts these systems implement. In this article we are concerned with establishing some concepts for GUI construction by composition, concentrating our attention on reuse aspects.

1.4 Specifying GUIs

Most current systems offer an integrated environment providing interactive GUI editors, GUI description languages, code translators, and toolkit libraries. In the following subsections we present an overview of some systems that offer GUI development facilities.

The programmer can use the GUI editor to generate a GUI specification file, which is generally in a declarative description language, or create it directly by text editing. The specification code in this file is translated into a specification in a procedural language or into an intermediate code by a translator. This new code is bound to the toolkit library and the application code to form an executable application (see figure 2). This process is repeated every time it is necessary to change the user interface. The toolkit library must provide routines for the interpretation of the intermediate code generated by the GUI description translator.

Specifying user interfaces using a declarative language directly is an approach that many programmers prefer over interactive editors. Sometimes programmers feel more comfortable in dealing with textual information, which gives the illusion of greater control over the objects.

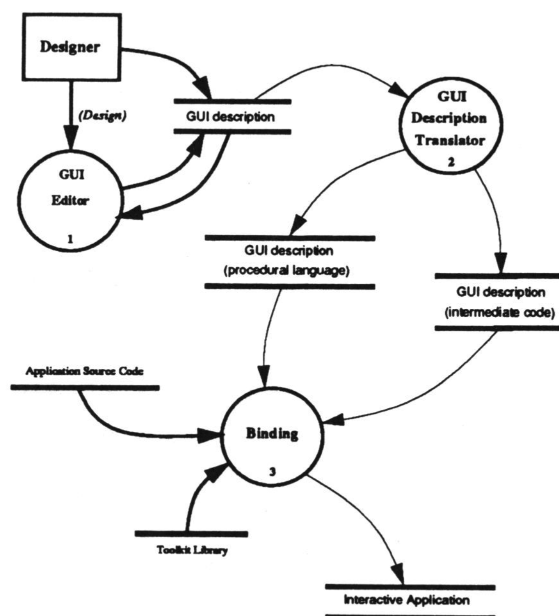

Figure 2: Integrated GUIDE

Translating the UI description into an intermediate code for interpretation has the advantage that users are able to redefine the UI without having to recompile any application code. Another advantage is that the binding process between the application code and the user interface code is much simpler than if the programmer were using a programming language code generator.

1.5 Motif Overview

1.5.1 UIL and MRM

The User Interface Language (UIL) is a specification language for describing the initial state of a user interface for interactive applications. The specification describes the objects (such as menus, form boxes, labels, and pushbuttons) used in the interface and specifies the functions called when the interface changes state as a result of user interaction. The Resource Manager (MRM) consists of library subroutines that access UIL at run time and create a user interface.

A widget, as defined by the Xt Intrinsics, is a user-interface component consisting of data structures and of a set of procedures that act on those data structures. The UIL specifies the widgets, gadgets, and compound objects (objects created from widgets and gadgets) that make up the interface. It also identifies the subroutines to be called whenever the interface changes state as a result of user interaction. The MRM creates widgets based on definitions

obtained from one or more User Interface Definition (UID) files generated by the UIL Compiler.

The MRM is able to create multiple instances of widget classes defined in the UID files. This is an important feature that makes it possible to implement a clean solution for the 3D Graphics Viewer problem, for example. Unfortunately the UIL does not offer nice inheritance mechanisms. But it may not be too expensive to enhance the language to provide such capability.

1.5.2 AIXwindows Interface Composer (AIC)

The AIXwindows Interface Composer is an editor allowing developers to interactively create and test Motif graphics user interfaces. This GUI editor offers C and UIL code generation. The Xt intrinsics enables defining a new widget class as a subclass of an existing widget class. AIC can be extended to support any new widget classes.

1.6 InterViews

Interviews [7] is a software system for window-based applications. Like most user environments, InterViews is object-oriented in that components such as windows, buttons, menus, and documents are active elements with inherited behavior. The name InterViews comes from the idea of a user interface object presenting an interactive view of some data. For example, a text editor implements an interactive view of the contents of a text file. The goal of InterViews is to facilitate the composition of user interfaces from reusable components.

1.6.1 Interface Builder

InterViews contains Ibuild, a tool for interactively building a user interface. Ibuild allows the user to arrange and connect interactors and scenes, generate C++ code for the interface, compile the code and execute the resulting mini-application. To complete the application, the generated code defines a base class from which subclasses can be written. This approach allows the interface to be modified later without affecting the subclasses.

1.7 Serpent

Serpent [13] is a user interface management system (UIMS) being developed at the Software Engineering Institute (SEI). Serpent supports the development and implementation of user interfaces, providing an editor to specify the user interface and a runtime system enabling communication between the application and the user.

Serpent supports the incremental development of the user interface from prototyping through production and maintenance. This is done by providing an interactive layout editor for prototyping, by integrating the layout editor with a dynamic specification language for production and maintenance, and by having an open architecture so that new user interface functionality can be added during the maintenance phase.

1.8 Andrew Toolkit

The Andrew Toolkit [10] (ATK) is an object-oriented system designed to provide a foundation on which a large number of diverse user-interface applications can be developed. With the Toolkit, the programmer can piece together components such as text, buttons, and scroll bars to form more complex components. It also allows for the embedding of components inside other components, such as tables inside text or drawings inside of tables. Some of the components included in the Toolkit are multi-font text, tables, spreadsheets, drawings, equations, rasters, and simple animations. The Toolkit is written in C, using a simple preprocessor to provide an object-oriented environment.

The original concept of the Andrew Toolkit came from discussions about the development of a text editor allowing the user to embed and edit other components, such as tables, drawings, and rasters. This is the composite editor problem exemplified above. To solve this problem, ATK offers two abstractions: data objects and views. These concepts are very similar to the abstract data types and abstract data views we present in this document.

1.8.1 The Basic Toolkit Objects: Data Objects and Views

Data objects and views are the Toolkit's basic object types; a Toolkit component is usually made of a view/data object pair. While the data object contains information to be displayed, the view contains information about how data is to be displayed and how the user is to manipulate the data object. For example, the text data object contains the actual characters, style information, and pointers to embedded data objects, as well as ways to alter the data, such as inserting and deleting characters. The text view consists of information such as the location of the text, the currently visible portion of the text, and what piece of text is currently selected. The text view provides ways to draw the text, handle various input events, and manipulate the visual representation of the text.

The view/data object distinction exists to provide a system where multiple views can simultaneously display information contained in a single data object. There may also be two different types of views displaying information contained in a single data object.

1.9 OpenWindows Developer's Guide

The OpenWindows Developers's Guide is a GUIDE designed to improve GUI development in the OpenWindows system. It consists of an interactive editor (guide), a Graphical Interface Language (GIL), and Language translators (gxv and gxv++ - the former translates GIL into C code and the latter translates GIL into C++ code).

The *guide* interactive editor enables programmers to create, modify and test user interfaces very easily, and generate GIL code. The language processors translate the GIL code output from the editor into C or C++ code calling the XView library. The OpenWindows Developer's Guide encourages separation between application objects and user interface objects. By using the C++ code generator feature, programmers can build UI objects inheriting the classes created by gxv++. As a consequence, it is possible to change the user interface without any changes to the existing application code. The gxv++ creates a class for each window in the user interface. Each of these classes contain XView objects as members.

1.10 GUIIZER

The GUIIZER is a GUIDE designed to offer some facilities in GUI development work and system customization. The first version of GUIIZER, developed by A. Potengy, D. Marchesin, and B. Plohr at the State University of New York at Stony Brook, is informally specified in [11]. We have been applying this tool to nontrivial applications since work on its beta version was completed in February, 1990. The initial application was a set of simulation tools for petroleum recovery, large applications created following the structured model of programming.

Since this model did not provide enough global control and evolvability it was decided to adopt the Object Oriented model [16] [17] [9], trying to preserve the existing tools (like the GUIIZER) as much as possible. Experience showed that although the GUIIZER gives a lot of power in GUI development and maintainability when applied to traditional structured fashion "good-apps", it is not general enough to offer full support to object orientation, even though its basic components were object oriented. We needed to adopt another GUIDE for our purposes.

The basic reason we had problems applying the GUIIZER in OOP is that it was an object-oriented tool designed to support structured programming. In fact, the whole architecture of this tool reflects the way "structured good-apps" should behave in general. For the GUIIZER, an application is a unique object, and the only way it can be accessed is through certain standard public member functions. The GUI for an application is specified in a GUISPEC file (GUISPEC is a specification language for GUIs). During the application initialization, the tool reads this file and builds its GUI.

This approach is fine if one considers applications as finite sets of routines and their GUIs communicate with them by calling these routines. But this is not the case with object oriented "good-apps", when an application is defined as a set of abstract data typed objects and the GUI communicates through methods. Actually one wants more. A GUI specification should exist for each object class to which the user is allowed access, and the specification should be flexible enough to reflect the way the class was defined. If a class was defined using inheritance, composition, referencing or whatever, the GUI for that class should be specified using a very similar mechanism. Finally, one does not want to be forced to locate that specification in a separate single file, away from the class definition.

This is what *Abstract Data Views* (ADV) [1] is all about. Our goal was then to bring the GUIIZER into the ADV model described in the next section and to identify the long-term benefits of doing it. The main idea was to build a simple ADV-based GUIDE, reusing the objects of GUIIZER, and to experiment with user interface design by compositions. We did not simply adopt other available GUIDEs (like the ones above) because we needed to keep a clear separation between the concepts around the design and the implementation. After establishing the appropriate concepts, we would be able to choose a system to use, or to develop one if needed.

The new GUIDE developed contains the classes present in the GUIIZER and a set of new classes to support the ADV model. The classes present in the GUIIZER implement the Open Look hierarchy using the XView toolkit [4] on X-Windows [5] [14] [12] [18](called GUIObjects) and some other related to GUISPEC management (Parser, Scanner, Definition, Guispec, etc.). The ability to deal with GUI specifications independently of any programming language is still a useful feature and was preserved. The new classes are composed of general definitions of ADV and Interactive abstract classes.

The ADV based GUIDE was implemented using the C++ language [15] [3] [2] and the XView toolkit. A simple example is provided to show its basic capabilities. Current object oriented "good-apps" should be adapted to the ADV model with very low cost. It would be interesting to make experiments to determine the ADV model behavior in adapting general object oriented applications to its context.

2 The ADV concept

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

An Abstract Data View [1] (ADV) might be called a visual realization of an abstract data type (ADT) because it has many of the properties of an ADT. However, the ADV is 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".

Moreover, ADVs can be nested. The nesting capability shows itself on the screen, where each ADV is drawn inside its parent region. This feature is a generalization of the concept of subwindow. Among other advantages over the "old" mechanism ADVs 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 by another ADV. This allows, for instance, the insertion of images generated by a draw-package inside text being manipulated in a text editor, in a way the user can actually interact with the images.

ADVs can also be specialized. An existing ADV for a ADT can be used as a basis for defining another ADV for that ADT, or ADVs for the ADT's subclasses. A GUIDE supporting ADVs should encourage 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 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 ADV represents all user interface aspects of ADTs.

Each ADV may or may not be connected to an object of an application. The correspondence is specially useful in WYSIWYG[1] 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 3 illustrates the ADV x ADT relation and a clear separation between visual and application objects.

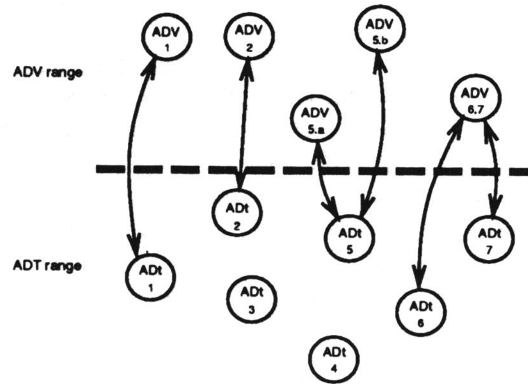

Figure 3: ADV X ADT

3 Description of an ADV/GUIDE

This section describes the basic structure of a GUI toolkit library designed to support the ADV concept. Most objects in the toolkit are based on the Open Look [6] standard. These objects inherited their basic funcionality from the ones allready available in the XView toolkit. Some additional features were added to make the new objects suitable for the composition mechanisms like inheritance and aggregation. This library is composed by a set of objects that realize the user interface. Inheritance mechanisms are widely used to provide nice opportunities for extensions. Figure 4 shows the inheritance tree for the some relevant objects. An example of how to use this library is shown in the next section. The objects we used in the example (see Figure 6) are:

3.1 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 single purpose of establishing graphics interaction be-

[1]WYSIWYG stands for What You See Is What You Get

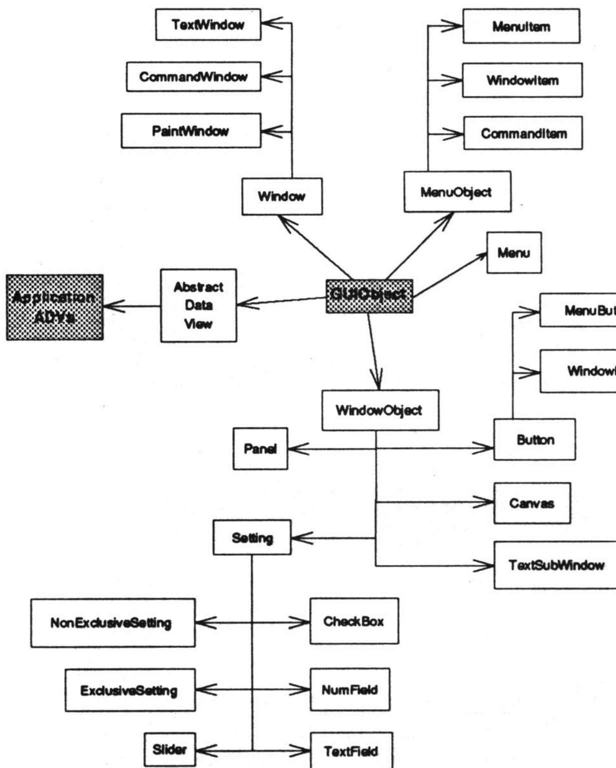

Figure 4: GUI Objects Inheritance Structure

tween users and applications will be called a GUIObject. For example, *ADVs*, as well as *Menus*, are *GUIObjects*

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

3.3 Buttons

A *Button* is a *WindowObject* associated with an "action"; it performs an operation according to an event. When the user "clicks" a *Button*, a notify procedure is called (let us use the name *action* to mean a notify procedure). Like other *GUIObjects* it is able to describe itself in GUISPEC language. The *MenuButton* and the *WindowButton* are *Buttons* too. The *MenuButton* has a reference to a *Menu* which displayed when the mouse menu button is pressed, and starts a default action if "clicked". The *WindowButton* makes a *Window* appear when "clicked".

3.4 Settings

There is a whole class of *WindowObjects* called *Settings*. Those objects are typically associated with object attributes, in contrast to *Buttons* which are always associated with object methods. Typically, a *Setting* has a value associated with it; this value is directly given by the user. For example, the *NumField* and the *TextField* have their values typed in. And the *Slider* has its value defined by dragging a slider cursor.

3.5 ADV Objects

The ADV-model related classes are simple abstract classes application programmers can use by plugging their classes in (here plugging A into B means making A a subclass of B). Each class, in the *Abstract Data Type* (ADT) range, 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 this class is also an Interactive object. The ADV subclass is specified reflecting the Interactive subclass structure, obviously providing access only to its public members. Figure 5 illustrates the concept.

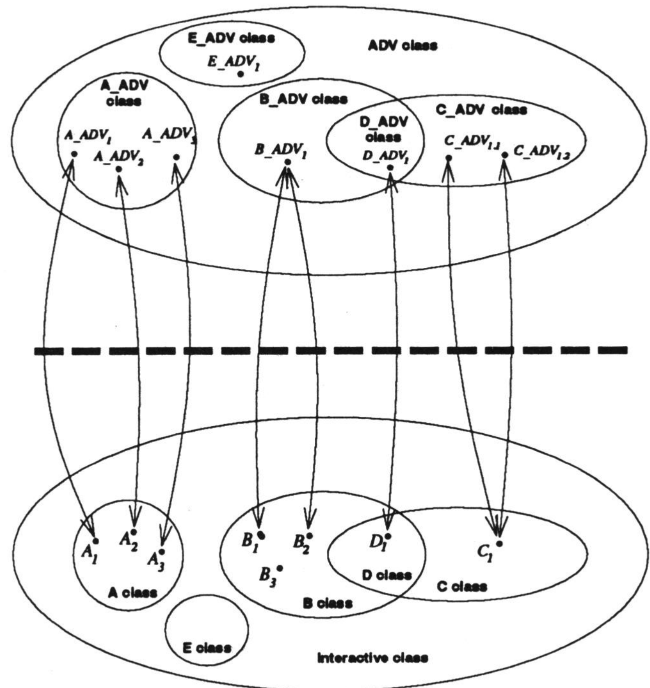

Figure 5: ADV x ADT revised

The ADV class contains at least a reference to a default enclosing Window (a Window is a GUIObject subclass, present in the current GUIIZER version), and a reference to an Interactive object. The ADV is

a free form enclosed interface object (as discussed in the introduction examples - section 1). Each instance of 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 5, 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.

There is an ADV method to update the GUIObjects contained in the corresponding ADV object when it is asked to. 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 on 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. A great improvement is that an ADV subclass is as general as the class it is associated with, providing nice reusability possibilities by applying on ADVs the same abstract operations applied on ADTs. For example, as in Figure 5, class D is a subclass of B and C, since its correspondent 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

This section presents a very simple example using the ADV objects described above (see Figure 6). Figure 7 illustrates a view of a trivial class called A. A is an interactive subclass and is composed by a single integer component called *Aattr*1. The only method for this class is *Print*.

This class can be easely implemented in C++ by making use of 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 by a *Slider* subclass (for A's integer attribute) and a *Button* subclass (for A's Print method). These two *GUIObject* subclasses are defined as *A_ADV_Members*.

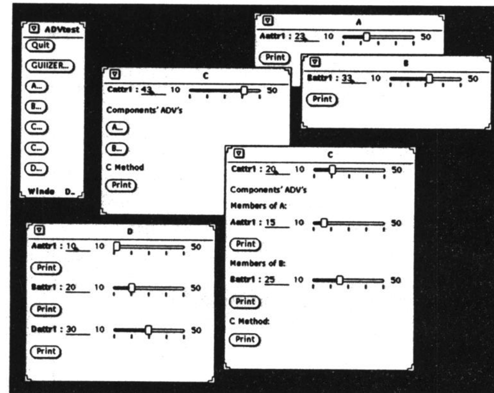

Figure 6: An ADV based example

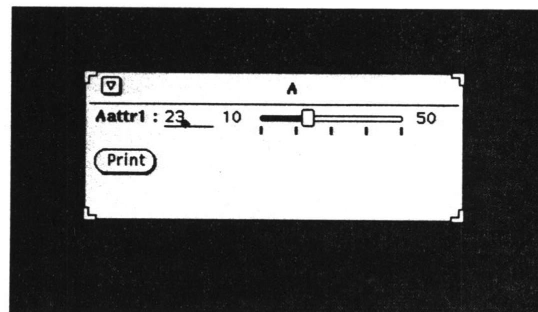

Figure 7: An A_ADV instance

The operations on these classes are very simple, since they inherited the GUIIZER objects 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".

Figure 8 shows a view of another trivial class called B. B was implemented by mimicing the definition of A. *B_ADV* (Figure 8) was also built this way.

Having the two trivial classes and their ADVs shown above, we are able to perform many kinds of composition experiments. For example, C is an aggregate with components A and B. Its abstract Data View (*C_ADV*) is also defined as an aggregate with components *A_ADV* and *B_ADV* (see Figure 9).

Note that to define the ADV for class C we **were 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 graphics

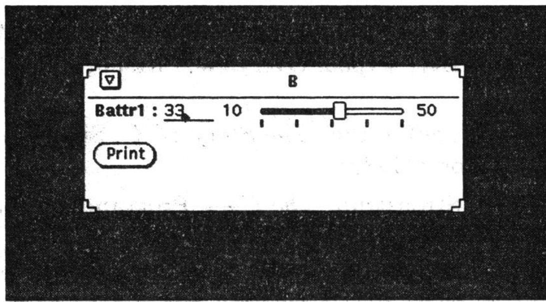

Figure 8: A B_ADV instance

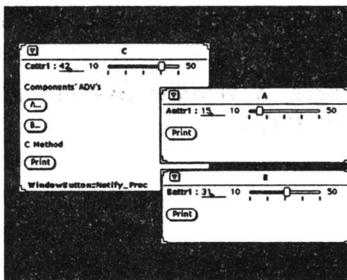

Figure 9: A C_ADV instance

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 (see Figure 10).

Now consider a multiple inheritance composition of *A* and *B* to build a class called *D*, and with one more attribute *Dattr*1 and another method *Print*. The class *D_ADV* is implemented the very same way, by inheriting *A_ADV* and *B_ADV*. Figure 11 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 main idea here 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 it is time to build its visual representation, the GUI. 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.

5 Conclusion

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

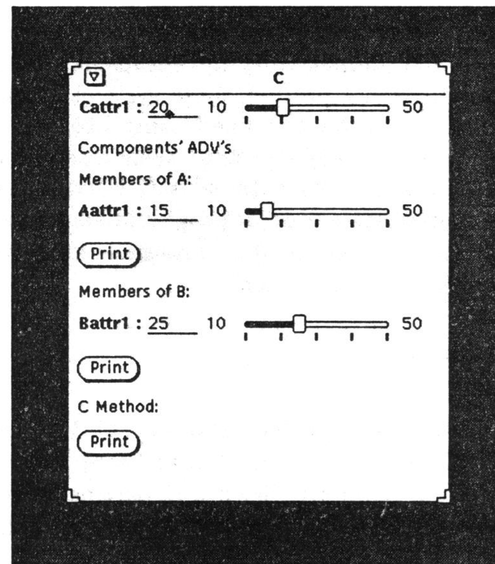

Figure 10: A C_ADV subclass instance

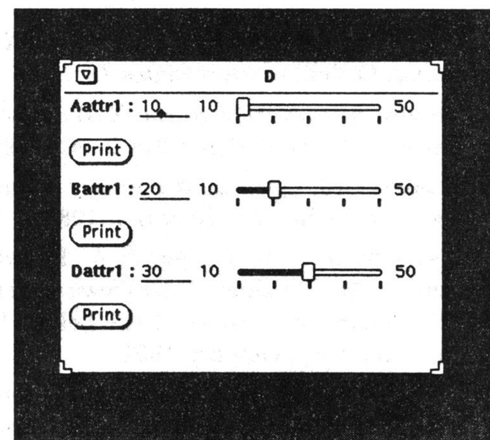

Figure 11: A D_ADV instance

The example 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 having consistent graphics user interfaces for them. 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 very same kinds of combinations we used before for the ADTs.

Two other important results from the use of ADVs is 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 potentially an apropriate tool for programmers whose goal is the fast development of nice reusable user friendly object oriented applications.

References

1. D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena and T. M. Stepien, *Abstract Data Views*, University of Waterloo, Computer Science Department, Technical Report , Waterloo, Ontario, February 1992.

2. S. C. Dewhurst and K. T. Stark, *Programming in C++*, Prentice Hall, 1989.

3. B. Eckel, *Using C++*, Osborne/McGraw-Hill, 1989.

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

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

6. J. Kannegaard, *Open Look Industry / Outlook / Overview*, Sun Technology (Autumn 1988).

7. M. A. Linton, P. R. Calder, J. A. Interrante, S. Tang and J. M. Vlissides, *InterViews Reference Manual*, The Board of Trustees of the Leland Stanford Junior University, October 1991.

8. C. J. P. Lucena, D. D. Cowan, R. Ierusalimschy and T. M. Stepien, *Application Integration: Constructing Composite Applications from Interactive Components*, University of Waterloo, Computer Science Department, Technical Report , Waterloo, Ontario, March 1992.

9. M. Mullin, *Object Oriented Program Design With Examples in C++*, Addison-Wesley, 1989.

10. A. J. Palay, W. J. Hansen, M. L. Kazar, M. Sherman, M. G. Wadlow, T. P. Neuendorffer, Z. Stern, M. Bader and T. Peters, *The Andrew Toolkit: An Overview*, Information Technology Center, Carnegie Mellon University, Pittsburg, PA.

11. A. B. Potengy, *GUIIZER: Um Ambiente de Desenvolvimento de Interfaces Gráficas com Usuário*, Instituto Militar de Engenharia, Seção de Engenharia de Sistemas, Projeto de Fim de Curso, Rio de Janeiro, October 1991.

12. D. Pountain, *The X Window System* , Byte (January 1989).

13. Software Engineering Institute, Carnegie Mellon University, *Serpent Overview*, Pittsburg, PA, May 1991.

14. A. Southerton, *The Story of X* , Supplement to UnixWorld (1989).

15. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

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

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

18. T. Yager, *X Window System on the March*, Byte (October 1989).