The Construction of User Interfaces and the Object Paradigm

Joëlle Coutaz
Laboratoire de Génie Informatique (IMAG)
BP 68, 38402 St-Martin-d'Hères, France
UUCP ...!mcvax!inria!imag!joelle

Abstract : This article concerns generic tools for the construction of user interfaces: Application Frameworks and User Interface Management Systems (UIMS's). In this article, we propose a new taxonomy for these tools, identify their limitations and show how the object paradigm can be exploited to overcome the current deficiencies such as support for generality, context and distribution. This taxonomy is the result of our own experience in designing MOUSE, an object-oriented UIMS based on a general model, PAC, that can be applied at any level of abstraction of a user interface. PAC structures any component of an interactive application into three parts: the Presentation which defines the external behaviour, the Abstraction which corresponds to internal concepts and the Control which bridges the gap between the syntax and the semantics.

Keywords: User Interface, User Interface Management System, Object-Oriented Programming.

1. Introduction

Issues pertinent to user interface design can be discussed from two points of view: that of the application programmer and that of the end user. Here, we are concerned with the problems of the programmer, concentrating on principles and tools that facilitate the construction of user interfaces. One fundamental principle is the separation of the functions of an interactive application from its user interface. The functions of an application define the tasks that it is able to accomplish whereas the user interface mediates between the functions and the user. In reality, this division which is the direct application of the principle of modularity, is not easy to achieve. Indeed, mediation implies communication, communication implies I/O and I/O is naturally distributed throughout the application. It is not surprising then that the code which handles the interaction is intermixed with the code which implements the functions of the application. In these conditions, it is difficult or even impossible to put into practice a second important principle in user interface design, the iterative refinement of the user interface. In this paper, we are concerned with tools that encourage the separation between the functions of an application and its user interface: generic systems.

Generic systems, such as user interface management systems, automatically provide a standard user interface framework upon which the functional components of applications can be encrafted. The next section proposes a classification that identifies the advantages and the limitations of these building tools. Section 3 shows how the object-oriented paradigm can circumvent some of the unsolved problems in generic systems. Section 4 describes how the notion of object has been exploited in MOUSE, our own User Interface Management System.

2. Generic Systems

Generic systems include two types of tools: application frameworks and User Interface Management Systems (UIMS's). These categories are examined in the following two paragraphs.

2.1 Application Frameworks

Application frameworks like MacApp [Schmucker 86a], EZWin [Lieberman 85] and Apex [Coutaz 86] package the code that implements most of the user interface into a reusable and extensible skeleton. The implementor's task consists in filling the blanks of the skeleton, adding new functions or overriding parts that do not fit the application domain. Code extensibility, reusability and overriding are naturally supported by object-oriented programming techniques as MacApp, EZWin and Apex demonstrate.

2.1.1 MacApp and EZWin

MacApp is implemented in Object Pascal, an object-oriented extension of Pascal [Schmucker 86b]. It models an interactive application as an instance of the class TApplication which is responsible for tasks that are common to all of the applications such as launching and opening documents. A document is also an instance of a class, TDocument, with methods for manipulating files. The class Tview takes care of the content of a document displayed in a window for things such as selection and refreshing.

EZWin is implemented on a Symbolics 3600, using ZetaLisp with the object-oriented Flavors package. It models an application as an editor for presentation objects. A presentation object is a visual representation of a domain entity. It takes care of both input actions from the end user and output feedback. EZWin is based on three types of objects that Lieberman observed as common in structure and behaviour across applications. These are: the presentation objects, the command objects which represent the end user's actions, and the EZWin object corresponding to the entire user interface to an application. The system maintains a set of presentation objects that can be created, deleted or modified through command objects.

2.1.2 Apex

Apex is implemented in C on top of the Macintosh toolbox. As shown in the figure 1, the architecture of an Apex interactive application is comprised of two components: the Application part which implements the functions (i.e. the semantics of the dialog) and the Interface part which implements the presentation (i.e. the syntax of the dialog). The Application contains the code that is specific to the domain whereas the Interface is domain independent and contains the reusable code.

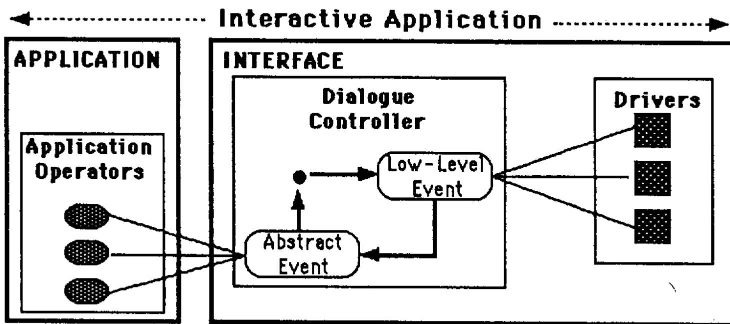

Figure 1: An Interactive Application built with Apex. (Thick arrows illustrate the control flow, thin lines represent procedural calls and • indicates the entry point of the cycle)

The reusable code is organized around two classes of objects: the dialog controller which contains the top level loop and acts essentially as a message dispatcher, and drivers such as window, menu and form drivers, each specialized in the processing of a predefined set of messages. A message may contain either a low level event generated by the underlying system, or an abstract event exchanged between a driver and the Application. A low level event, such as a window update or a scrolling request, is handled by a driver object as far as possible. If the driver cannot fully process a low level event, (for exemple, a mouse click in a window area that is not handled by the window driver) it generates an abstract event for further semantic processing. This abstract event, which is an enriched version of the low level event, is then sent to the Application (for example, the mouse event mentioned above would generate an "InContent" abstract event). The Application behaves like a semantic server with predefined operators such as "inContent", and may send messages to manipulate drivers such as create a window or read/write on the serial port.

Apex and EZWin are very similar in their goal of separating what is domain dependent from what is common to applications. In MacApp, this distinction is not made clear. Domain dependent notions tend to be diluted across various classes. As a result, the MacApp user needs to pay attention to the definitions of the various classes instead of concentrating his attention on a single piece of code (as is the case with the Apex Application).

2.1.3 Motivation for Application Frameworks and Evaluation

Application frameworks have been introduced to overcome problems with the use of toolboxes. Toolboxes generally provide the programmer with a wide range of powerful abstractions for implementing user interfaces. This diversity of services makes toolboxes flexible but hard to use. It requires the programmer to discover the right glue for assembling the pieces from the toolbox. Not only may this task be a tremendous technical barrier for the first time developer but also it must be carried out for each application. It is not surprising then that a strong interest has recently emerged for a tool that provides the implementor with both an implementation model and reusable code.

The existence of an implementation model and the availability of reusable code have several related side-effects: the implementor can concentrate on the functions of the application, user interface consistency is improved and application development time can be reduced by a factor of four or five [Schmucker 86a]. The experience described in [Garret 86] about the use of MacApp for the construction of a large scale application is an eloquent testimony to the advantages of application frameworks.

Our own experience with Apex leads us to believe that an application framework is not expensive to develop on top of a powerful toolbox. Our first version of Apex which included only drivers for windows and menus, has been developed in three months by a graduate student who had no previous knowledge of the Macintosh toolbox. The delicate part for him was to apply the object-oriented paradigm of which he had book-knowledge only. A more important and fundamental problem was to determine which abstractions should be implemented and how to organize them into a consistent model.

To solve this problem, we first defined five practical guidelines: first, any domain dependent code must be outside of Apex and made accessible through standard entry points; second, any domain independent code must be in Apex and structured according to functional criteria; third, closely related functions must constitute a unit (i.e. a driver); fourth, each unit must augment the functionality and hide the complexity of the services provided by the toolbox; fifth, Apex must be expandable in order to enrich and add services as required by a potentially wide range of applications.

When putting these principles into practice, it became natural to define the Application as an object whose methods implement a predefined set of semantic operators; in the Interface, each driver is a class whose methods implement a predefined set of operators dealing with a particular aspect of the user interface; although Apex is not a pure object-oriented system (there is no way to create instances of the class Application nor is it possible to create new classes) we have made extensive use of the principle of modularity along with the message passing mechanism promoted by the object-oriented programming. By doing so, it has been possible to implement a first simple version of Apex with a minimum of standard drivers and to then extend it easily as new applications were constructed. Clearly, the object-oriented approach is vital to the design of Apex.

In spite of their advantages, application frameworks still require the application implementor to deal with the underlying toolbox for appending code or modifying subroutines. Unlike UIMS's presented in the next paragraph, application frameworks impose a low level programming interface on the implementor.

2.2 User Interface Management Systems (UIMS's)

UIMS's, automatically generate an executable interactive application from a specification provided by the implementor. For doing so, a UIMS is comprised of a run time kernel whose architecture is similar to an application framework and a tool for processing the specification. This specification is usually a textual description using a special purpose language [Hayes 83] or an extended notation of an existing formalism such as petri-nets [Barthet 86], BNF [Bleser82, Olsen 83] or transition networks [Wasserman 85]. The specification is made easier when it is carried out through an interactive tool as in Flair [Wong 82], Menulay [Buxton 83], SOS [Hullot 86] and Peridot [Myers 86].

The UIMS model has served as the basis for a number of tools for dialogue specification and dialogue generation. The next section proposes a classification for these tools, indicates which design choices are desirable as well as identifies deficiencies in current UIMS's. It will also show how some of these limitations can be overcome with the object-oriented paradigm.

3. Taxonomy for UIMS's and Benefits from the Object Paradigm

UIMS's have been classified according to three design choices [Hayes 85]: the level of abstraction at which the specification is performed, the localization of the control of the dialogue and the expression of the ordering of events. In the next paragraphs, we propose a taxonomy with five additional criteria: adaptiveness, support for concurrency, context, distribution and generality.

3.1 Level of Abstraction

The level of abstraction defines the unit of exchange between the application and the UIMS. This unit can range from low-level tokens such as keystrokes, to the level of abstraction manipulated by the application such as complete and syntactically correct commands. Low-level abstractions concern the events at the user-visible interface whereas high-level abstractions are application-dependent.

The level of abstraction defines the effectiveness of the separation between the semantics and the syntax, i.e between the functions and the presentation of the application. Consequently, it determines the extent to which the implementor is able to concentrate on the logical functioning of the application. In order to facilitate the implementor's task, it is desirable that the information exchanged between the application and the UIMS be application concepts. The object-oriented paradigm provides a natural mechanism for representing information at the desired level of abstraction.

3.2 Localization of the Control

The control of the dialogue may be internal, external or mixed [Tanner 83]. The control is internal when it resides in the application. It is external when it is maintained by the UIMS. It is mixed when it is alternatively handled by the application and the UIMS. Internal control presents two serious drawbacks: first, it may lead to situations where an application traps the user in a kind of local mode by forcing him to reply right away to a question without allowing a new request that would help in the choice of the correct alternative. Second, by being rooted in the application, internal control does not enforce a clean separation between functions and presentation.

Conversely, when the UIMS maintains the control, the application is viewed as a semantic server. This technique entails a clean separation between functions and presentation. It opens the way to more modular programs although it may impose a style of programming on the implementor. We believe that the external control of the dialogue should lead to fewer arbitrary constraints on the user, constraints that may be unnecessarily imposed by an application implementor unaware of the basic "do's and don'ts" in user interface design. At the opposite of the internal control, external control operates in accordance with a user-driven style.

Between the two extremes, mixed control allows the implementor to switch freely between internal and external modes. This flexibility is useful for applications that need to notify the UIMS of asynchronous events (such as the ringing of the alarm clock) or for functions which dynamically require the user to specify some parameter before processing can continue (such as print commands in "more" mode, or expert systems which typically ask the user to provide additional information to resolve conflicts). However, mixed control may induce software maintenance problems (just like the goto statement). External control with hooks for mixed control seems to be a reasonable choice. In section 4.4, we indicate how the object paradigm can be applied to implement these hooks.

3.3 Ordering of Events

Ordering of events may be frozen in a declarative manner as with BNF and ATN based specification tools. The number of allowed actions (for example, the description of the free ordering of command parameters) and exceptional conditions must be anticipated exhaustively. This task may become difficult if the specification has to be performed at a low level of abstraction such as keystroke or mouse level. In addition, it is not clear how these tools account for parallel events.

The difficulty with ATN and BNF tools comes from the mismatch between the way they model the user and the behaviour of the real user. They model the user as a parsable sequential input file whereas the user is essentially unpredictable. A reasonable solution to this problem is the combination of an external control with high level abstractions which moves the problem of event ordering outside of the implementor's view. Paragraph 4.2 shows how the object paradigm is appropriate for the management of the user's unpredictable behaviour.

3.4 Adaptiveness

Adaptiveness is the abiliy to adjust to new or modified surroundings. For a computer system, adaptiveness is concerned with the multiple sources of variation in the environment: hardware, operating systems, languages and users. Here, we are concerned with the adaptiveness of the user interface to the user.

A user interface may be adaptable or adaptive. An adaptable user interface may be customized by hand on an explicit intervention of either the implementor or the user. In most cases, a modification to the user interface requires the recompilation of whole code. In the best case, as in the Macintosh, the user is able to perform cosmetic changes without recompilation. These changes, although superficial, are made possible on account of the object-oriented paradigm.

At the other end of the spectrum, an adaptive user interface behaves like an intelligent observer who automatically adapts the user interface to the habits and expertise of the user without being too intrusive. Experiments have been attempted in this direction, in particular for tutorials, but none have been undertaken in the framework of UIMS's. Given that knowledge can be modelled as a network of concepts, it seems reasonable to distribute the knowledge about the user across Loops-like objects [Bobrow 83] where rules would allow the object to alter his behaviour according to the user's actions.

3.5 Concurrency

Broadly speaking, concurrency refers to the existence of multiple activities. In the area of user interface, concurrency has three complementary facets: simultaneous use of multiple input and output devices, simultaneous interaction between a user and multiple applications, and simultaneous interaction between multiple users (e.g. teleconferencing),

Interaction between multiple users introduces a new dimension in the area of user interface: the social dimension which, so far, has not been the subject of significant study. The first two types of concurrency is becoming common on workstations supporting multitasking. In particular, the switchboard model [Tanner 86], based on rapid intertask message passing, provides UIMS's with a powerful tool for supporting parallel input. Interestingly, like Smalltalk objects, the switchboard tasks are small entities communicating through synchronous messages.

3.6 Context and Distribution

A context is a data structure that determines the meaning of a situation. This notion has been widely exploited in software engineering in particular for operating systems and programming languages. In the area of user interface, it is considered as an essential notion for helping the user to determine "where he is, where he can go and where he has come from". Although important, this notion is still not well integrated in current UIMS's.

One possible explanation of the situation is that context is inherently distributed and that current UIMS's do not support distribution well. Every component of a computer system manages a context. Unfortunately, context items are still defined without paying attention to the need of the end-user or are not made available to other components. As a result, the end-user is not well informed of the current state of the system nor is he well informed of his semantic errors. We believe that the object-oriented paradigm is the appropriate vehicule for distributing and communicating contextual information across objects and computers. However, the object paradigm does not solve the important problem of defining what is useful to the end-user!

3.7 Generality

Generality refers to the range of applications that a particular UIMS is able to support. The variety of applications results in a tremendous diversity of requirements. To simplify the situation, one can distinguish applications that require a fine grained control over I/O devices such as document formatters or graphic systems as opposed to applications for which information exchange does not require sophisticated services.

No UIMS claims to be fully general. Either it supports a specific class of applications as does Cousin [Hayes 83], or it provides the implementor with hooks for accessing the underlying toolbox as does ADM [Schulert 85]. A new trend is to provide application implementors with an object-oriented environment containing an extensible set of abstractions for the expression of the user interface. The difficulty is to define the appropriate basic set, determine the right organization through the composition and/or inheritance mechanism, and nearly as important, make the environment easy to use and learn.

(Larry Tesler in [Tesler 86] reports that most of the learning time of such an environment may go to mastering the extensive class library)

To summarize, the desirable criteria for a UIMS are the following: high level abstraction, external control with hooks for mixed control, implicit event ordering, adaptability, support for concurrency, contextual information and distribution. The next section shows how some of these design criteria have been integrated in MOUSE.

4. MOUSE, an Object Oriented UIMS

The architecture of MOUSE is based on a general object-oriented model, called PAC, that can be applied at any level of abstraction in a user interface. The following paragraphs describe PAC, identify its advantages and compare it with other object based models.

4.1 PAC, the Design Model

In MOUSE, an interactive application is comprised of three parts: Presentation, Abstraction and Control.

The Presentation defines the concrete syntax of the application, i.e. the input and output behaviour of the application as perceived by the user. The Abstraction part corresponds to the semantics of the application. It implements the functions that the application is able to perform. The Control part, called the MOUSE controller, maintains the mapping and the consistency between the abstract entities involved in the interaction and implemented in the Abstract part, and their presentation to the user. It embodies the boundary between semantics and syntax.

For example, the application "Clock" implements and involves two abstract entities in the dialogue : the data structure "Time" and the function "SetTime". "Time" may be presented as a digital or a dial clock, SetTime may be explicitly presented as a button or implicitly presented through the direct manipulation of the needles of the dial clock. The job of the Control part is to invoke SetTime on specific user's actions and provoke the update of the dial clock when the application (i.e the Abstract part) makes a request.

The Presentation of an application is implemented with a set of entities, called interactive objects, specialized in man-machine communication. As for applications, an interactive object is organized according to the PAC model. Consider for example the pie chart shown in the figure 2.

1. The Presentation is comprised of:
    - for output, a circular shape and a colour for each piece of the pie;
    - for input, the mouse actions that the user can perform to change the relative size of the pieces.
2. The Abstraction is comprised of an integer value within the range of two integer limits.
3. The Control maintains the consistency between the Presentation and the Abstraction. For example, if the user modifies the size of one piece, Control provokes the update of the integer value. Conversely, if the application or another interactive object modifies the value of the integer, the size of the pieces is automatically adjusted.

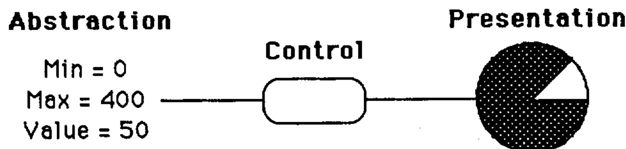

Figure 2: An elementary Interactive Object

Compound objects can be built from elementary interactive objects. They also adhere to the PAC model. Consider, for example, the super pie chart shown in the figure 3. It is made from two elementary objects: the pie chart described above and a numerical string which shows the current abstract value of the pie chart. If Control C receives a message notifying him of the modification of the abstract value, it notifies both C1 and C2 of the alteration. Conversely, if the user changes the size of a piece of the pie with the mouse, C1 reflects the modification to C who, in turn notifies C2. A sketch of the code for these objects is given in the appendix.

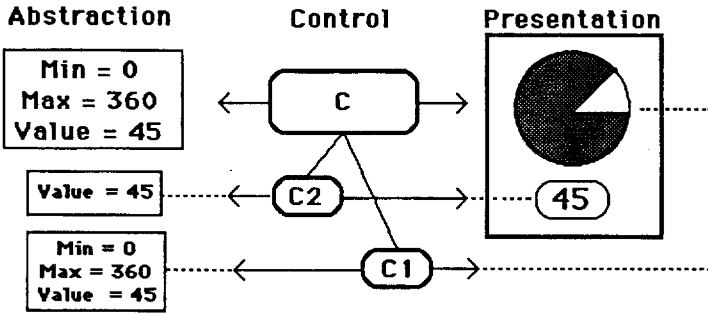

Figure 3: A Compound Interactive Object composed of an object of the class Pie Chart illustrated in the figure 2 and of an object of the class string.

In summary, as shown in the figure 4, everything in MOUSE is a PAC object from the elementary interactive object to the whole application. This recursive object-oriented organization presents some advantages which are described in the following paragraphs.

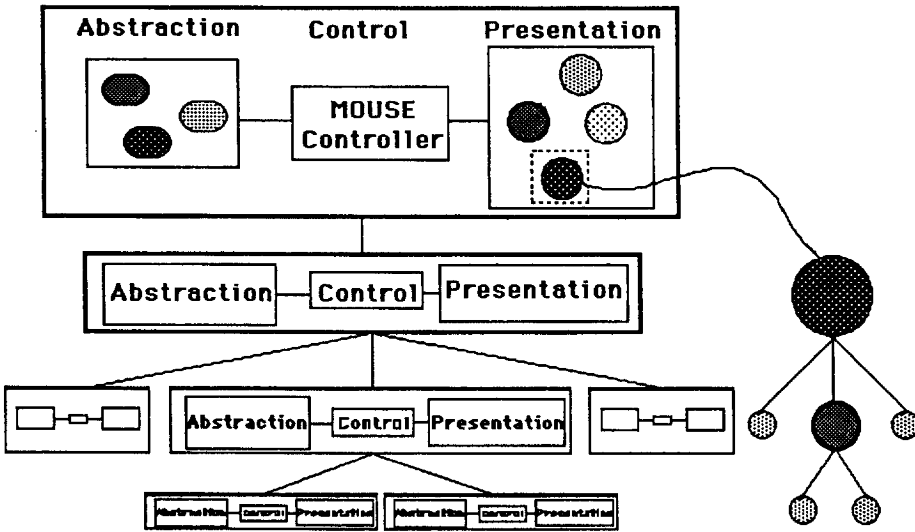

Figure 4: The Design Model. The upper rectangle represents the whole interactive application as a PAC entity. Its Abstraction involves three concepts in the dialogue. The MOUSE Controller bridges the gap between the Abstraction and the Presentation. The Presentation is made of 4 interactive objects. The second lower rectangle shows the PAC structure of the compound interactive object represented as a black circle. This object is built from two elementary PAC objects and one compound object which, in turn, is composed of two elementary PAC objects.

4.2 The Interest of the PAC Model

The interest of the PAC model is three-fold:

1. It defines a consistent framework for the construction of user interfaces that is applicable at any level of abstraction. As a direct consequence, the units of exchange between the application (i.e. the Abstract part) and the UIMS (i.e the MOUSE controller) are application concepts, not low-level details semantically irrelevant to the application.

2. It cleanly distinguishes functional notions from presentation policies but introduces the control part to bridge the gap between the abstract and the concrete worlds. The role of the control part may be extended from consistency maintenance between the two worlds, to the management of local contextual information that may be useful for help, error explanation and automatic adaptation to the user.

3. It takes full advantage of the object-oriented paradigm with the notion of interactive object. Let us develop this last point.

An *interactive object is an active entity*. It evolves, communicates and maintains relationships with other objects. Such activity, parallelism and communication are automatically performed by the Object Machine, the generic class of the interactive objects. The Object Machine defines the general functioning that is made common to all of the interactive objects by means of the inheritance mechanism. In particular, each object owns a private finite state automaton for maintaining its current dialogue state. On receipt of a message, an object is thus able to determine which actions to undertake according to its current state. In particular, The MOUSE controller at the top of the hierarchy of controllers, maintains the global state of the dialogue with the application.

Interactive objects implement the dialogue in a distributed way. This feature can serve as a basis for the implemention of facilities related to the notion of context. It also provides the necessary grounds for concurrent multiple I/O in the following way. The set of automata (one automaton per interactive object) defines the global state of the interaction between the user and the application. The control of the interaction is therefore distributed in an evolutive network of interactive objects. Dialogue control is not handled by a unique monolithic dialogue manager difficult to maintain, extend and implement, in particular when one wants a pure user-driven style of interaction. Conversely, since interactive objects are able to maintain their own state, it is easy to let the user switch between objects in any order. Thus, an object-oriented approach provides for free the maintenance of the user's arbitrary manipulations.

Interactive objects are easily customizable. Object-oriented programming languages support data abstraction which makes it possible to change underlying implementations without changing the calling programs. In the present case, this principle allows the internal modification of an interactive object without changing its presentation and abstract interfaces. Interestingly, it also allows the modification of one interface without any side-effect on the other interface. For example, one can modify the presentation of an interactive object (such as attaching a different key translation table to an interactive object of type string) without reflecting on its abstract behaviour. This property makes it possible fine grained dynamic adjustments of the user interface without questioning the presentation of the whole application.

The application of the object-oriented paradigm to the construction of user interfaces is rather new but not specific to MOUSE. The following paragraph presents similar works.

4.3 Related Work

Other models than PAC rely on the notion of object and make the distinction between functions and presentation. These are the Smalltalk MVC model [Golberg 84], Ciccarelli's PPS model [Ciccarelli 84], EZWin [Lieberman 85] and GWUIMS [Sibert 86].

Model, View and Controller are three Smalltalk classes. A Model implements some functions, a View is the output rendering of a Model and a Controller is the input driver of a Model. PAC and MVC differ in two points:

1. PAC distinguishes but encapsulates functions and presentation into an object. A local controller encompasses the boundary between local semantics and local syntax. At the opposite, MVC makes an explicit use of three Smalltalk objects which must maintain their consistency through message passing. In MVC, the notion of control is diluted across three related objects whereas it is explicitly centralized in PAC.
2. PAC combines input and output behaviour into one component whereas MVC distributes them across two objects. The distribution has the advantage of flexibility (one can change the input syntax without disturbing the component dealing with the output syntax). Unfortunately, it is often the case that, at the fine grained level of the interaction, input events are strongly related to immediate output feedbacks.

PPS is organized around a network of Presenters and Recognizers that manipulate two types of data bases. An Application data base defines some functions and a Presentation data base presents the Application data base. A Presenter builds a Presentation data base from an Application data base. The user modifies a Presentation data base, the associated Recognizer interprets the user's actions and

modifies the corresponding Application data base. As in MVC, PPS makes a dichotomy between input and output behaviour but, at the opposite of MVC, these behaviours are not carried out by objects. PPS objects are passive entities manipulated by Presenters and Recognizers.

EZWin objects are close to PAC interactive objects: both combine input and output policies. The originality of PAC objects is the existence of the Control part which lays the ground for the implementation of distributed notions related to the notion of control. In particular, the Control part can maintain a context (e.g. current status, validity, defaults) that can be usefully exploited during interaction to provide the end user with dynamically tuned informative feedback.

GWUIMS revolves around five types of objects: A-objects embody the semantics of the application; I-Objects bridge the gap between A-Objects and R-objects. R-objects control the presentation. They are at the boundary between the lexical and syntactic levels of a UIMS. Lexical input is carried out by T-objects whereas lexical output is performed by G-objects. An elementary PAC object is functionally equivalent to the combination of a R-Object with a T-Object and a G-Object, and the MOUSE Controller is functionally equivalent to I-Objects.Clearly, PAC and GWUIMS are closely related in their overall organization although GWUIMS splits the presentation into input and output objetcs.

4.3 Current State of the Project and Future Evolution

MOUSE is currently under development in C on top of the Macintosh toolbox. The whole code of an interactive application is executed in a single Macintosh process. The MOUSE run time kernel is being extended to support coroutines in order to provide the application implementor with hooks for mixed control. The MOUSE controller will then be able to process READ messages issued by the Abstract part without forcing the end-user to directly satisfy the request. Consequently, the user may invoke several functions of the application (which may in turn issue READ requests) before replying to the initial request. For doing so, on reception of a READ message, the MOUSE Controller creates a request object and activates the interactive objects that represent the abstract entities referenced in the READ message. The request object watches the interaction between the user and the interactive objects and detects when the user has satisfied or cancelled the READ. When the READ operator is completed, control automatically returns to the instruction following the READ.

Once the run time kernel has been fully implemented and a reasonable data base of interactive objects been constructed, we need to build a programming environment for the construction of user interfaces. This environment will include an editor and a manager for automatically handling versions of user interfaces for a particular application. The editor will allow the implementor to interactively build new interactive objects and to define the presentation of the application.

5. Conclusion

The object paradigm allows for the construction of highly interactive user interfaces. It naturally leads the way to the definition of UIMS's where the user interface is implemented as small manageable units rather than as a single monolithic system. In addition, it makes it possible to distribute the semantic, syntactic and lexical levels of a user interface at various levels of abstraction rather than defining a rigid unique boundary. This distribution allows a UIMS to handle the user's arbitrary manipulations and parallel I/O more easily, to provide the user with useful explanation about his semantic errors and to facilitate fine grained dynamic customization.

It is only recently that object oriented programming has been applied to user interface design. As a result, a standard technique for organizing user interface software in terms of objects has not yet evolved. This is illustrated by the fact that each of the tools presented above has its own method for exploiting the object paradigm. More experience is needed in order for a general model to emerge.

Acknowlegment. This work is part of the project GUIDE, an object oriented distributed system, which is being designed at the Laboratoire de Génie Informatique (IMAG, University of Grenoble). It sets the foundation of the user interface for the GUIDE workstations.

Appendix

Some operators for the object Pie. A C-like notation is being used. Method names are in bold.The prefix **pie** is used for message selectors. Methods prefixed with **pie** are gathered in a single module that implements the Control part of a piechart. The prefixes pieAbs and piePres are used for methods that constitute the Abstract and the Presentation parts. In principle, they are callable from the Control part only.The prefix **obj** is used to denote methods of the class Object.

pieSetVal (whichPie, val, fromWhom)	*to change the current value*
{pieAbsSetVal (whichPie, val);	*a method of the Abstract part to change the abstract value*
valtoratio (val, ratio);	*a local procedure to translate the value to a ratio for the image*

```
if (fromWhom != MYSELF)                  image is updated only if the request doesn't come from the object itself
    piePresSetRatio (whichPie, ratio);}a method of the Presentation part to update the image according to the ratio
pieEvent (whichPie, theEvent)            to handle events from devices
  {case (theEvent.what)
     MOUSE: mouse (whichPie, theEvent); .... }
mouse (whichPie, theEvent) a local procedure to process mouse events
  {case (wheremouse = mouseloc(whichPie, theEvent.where))
     INCONTENT:     a mouse event inside the circle signals a dynamic displacement
       if (objIsValid(whichPie,MOVE)) to determine whether the object currently accepts to be moved around
          pieMove(whichPie, theEvent.where);
     INNEEDDLE:     a mouse event inside the needle signals a dynamic displacement of the needle
       if (objIsValid(whichPie,MOVENEEDLE)) piePresTrackneedle(theEvent.where, ratio);
       if (piePreshaschanged(whichPie, ratio))
          {ratiotoval(ratio, val);       if the needle has moved, then the ratio has changed and the abstract
          pieAbsSetVal (whichPie, val, MYSELF)} value needs to be updated.
       if (objIsTalkative(whichPie)      returns true if the object has to signal ahanges of its abstract interface.
          { ... ;                        build msg to notify the ancestor of a change in the abstract interface
          ObjSendAncestor(whichPie, msg)};given a message and an object, call the appropriate method of
                                         the object's ancestor ... }
```

<u>One operator for the super pie chart described in 4.1</u>

```
spieSetVal (whichSpie, val, fromWhom)    to set the current abstract value of a super pie chart
  {case (fromWhom)
     ANCESTOR:                           the request comes from the ancestor
       ... ;                             build appropriate message to update the value of son1 ( i.e. the pie chart)
       objSendSon(objSon(whichSpie, son1), msg);
       ... ;                             build appropriate message to update the value of son2 (i.e. the string)
       objSendSon(objSon(whichSpie, son2), msg);
     SON:                                the request comes from a son
       objSendSon(theotherson, msg);
       if (objIsTalkative) objSendAncestor(whichSpie, msg)}
```

References

[Barthet 86] M. F. Barthet; C. Sibertin-Blanc: La modélisation d'applications interactives adaptées aux utilisateurs par de réseaux de Petri à structure de donnée; Actes du 3eme Colloque-Exposition de Génie Logiciel, Versailles, mai 1986, 117-136

[Bleser 82] T. Bleser, J.D. Foley: Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces; Huma Factors in Computer Systems Proceedings, march, 1982.

[Bobrow 83] D.G. Bobrow, M. Stefik: The Loops Manual; Tech. report KB-VLSI-81-13, Knowledge Systems Area, Xerox, Pal Alto Research Center, 1981.

[Buxton 83] W. Buxton, M.R. Lamb, D. Sherman, K.C. Smith: Towards a Comprehensive User Interface Management Systen Computer Graphics 17(3), July 1983, 35-42.

[Ciccarelli 84] E.C. Ciccarelli: Presentation Based User Interfaces, Technical Report 794, Artificial Intelligence Laboratory Massachusetts Intelligence Laboratory, August, 1984.

[Coutaz 86] J. Coutaz: La Construction d'Interfaces Homme-Machine, Rapport de Recherche IMAG-LGI, RR 635-I, nov. 1986.

[Garret 86] N. L. Garret, K. E. Smith: Building a Timeline editor from Prefabs Parts: The Architecture of an Object-Oriente Application; Proceedings OOPSLA'86, Sigplan Notices 11(21), November 1986, 202-213.

[Goldberg 84] A. Goldberg, D. Robson : Smalltalk-80 : The Interactive Programming Environment; Addison-Wesley Publ 1984.

[Hayes 83] P.J. Hayes, P. Szekely: Graceful Interaction through the Cousin Command Interface; International Journal of Ma Machine Studies 19(3), September 1983, 285-305.

[Hayes 85] P.J. Hayes, P. Szekely, R. Lerner: Design Alternatives for User Interface Management Systems Based on Experienc with Cousin; Proceedings of the CHI'85 Conference, The Association for Computing Machinery Publ., April 1985,169-175.

[Hullot 86] J.M. Hullot: SOS Interface, un Générateur d'Interfaces Homme-Machine, Actes des Journées Afcet-Informatique su les Langages Orientés Objet, Bigre+Globule, 48, Publ. IRISA, Campus de Beaulieu, 35042 Rennes, janvier, 1986, 69-78.

[Lieberman 85] H. Lieberman: There's More to Menu Systems than Meets the Screen; SIGGRAPH'85, 19(3), 181-189.

[Myers 86] B. Myers, W. Buxton: Creating Highly-Interactive and Graphical User Interfaces by Demonstration; SIGGRAPH'8(20(4), 1986, 249-257.

[Olsen 83] D.R Olsen, E.P Dempsey: Syngraph: A Graphical User Interface Generator; Computer Graphics, July 1983, 43-50.

[Schmucker 86a] K. Schmucker: MacApp: An Application Framework; Byte 11(8), 1986, 189-193.

[Schmucker 86b] K. Schmucker: Object-Oriented Languages for the Macintosh; Byte 11(8), 1986, 177-188.

[Schulert 85] A.J. Schulert, G.T. Rogers, J.A. Hamilton : ADM - A Dialog Manager; Proceedings of the CHI'85 Conferenc The Association for Computing Machinery Publ., April 1985, 177-183.

[Sibert 86] J.L. Sibert, W.D. Hurley, T.W. Bleser; An Object Oriented User Interface Management System; SIGGRAPH'8(20(4), 1986, 259-268.

[Tanner 83] P. Tanner, W. Buxton: Some Issues in Future User Interface Management Systems (UIMS) Development. IFI Working Group 5.2 Workshop on User Interface Management, Seeheim, West Germany, November 1983.

[Tanner 86] P. Tanner, S.A. Mackay, D. A. Stewart: A Multitasking Switchboard Approach to User Interface Managemen SIGGRAPH, 20(4), 1986, 241-248.

[Tesler 86] L. Tesler: Programming Experiences; Byte, 11(8), August 1986, 195-206.

[Wasserman 85] A. Wasserman: Extending State Transition Diagrams for the Specification of Human-Computer Interactior IEEE Transactions on Software Engineering, 11(8), August 1985.

[Wong 82] P.C.S. Wong, E.R. Reid: Flair- User Interface Design Tool; ACM Computer Graphics, 16(3), July 1982, 87-98.