An Object-Oriented Approach to the Design of Graphical User Interface Systems

Fabio Paterno'
CNUCE-C.N.R. - Via S.Maria 36 - 56100 Pisa - Italy
ICSI - 1947 Center Street - Suite 600 - Berkeley, CA 94704

TR-92-046

August 1992

Abstract

In this paper the problems concerning the design of graphical user interface systems composed of a set of interaction objects allowing users to interact with structured graphics are discussed. Each interaction object can have input and output functionality. Here we want to point out the problems and the requirements that are raised in performing such design in an object-oriented environment. For this purpose the importance of task-oriented design hierarchies of interaction objects in order to make the translation from the user task to the system functions easier is addressed. The design of a hierarchy of interaction objects following this approach is proposed. This contrast with the current window systems toolkits design because it is mainly driven by the semantics of the interaction objects rather than their appearance. Finally an example of a common graphical user interface performed by the proposed approach is presented.

Email address: paterno@icnucevm.cnuce.cnr.it - Phone: +39 50 593289

Introduction

This paper presents the issues raise from the performing of the refinement of an abstract description of graphical user interfaces defined by a formal specification into an object oriented language. We use Sather [O91], an object-oriented programming language developed at International Computer Science Institute, as case study for functionality provided by an object-oriented platform. The formal specification was performed by LOTOS [BB87] and is described in [PF92b].

The main features that must be supported by a User Interface System (UIS), the system that manages the communication among user and application, in order to be characterized by interactivity, flexibility and usability are:
- multiple parallel dialogues among users and applications;
- multiple processing levels, that means that the system architecture is organized in different layers;
- multiple feedback levels: each layer can provide a feedback to inform the user on the state of the current interactions;
- multiple views of the same abstract graphical description;
- dynamic activation and deactivation of the interaction objects.

The design of user interface systems providing this kind of functionality can be obtained by a composition of graphical interaction objects. Different changes to the general model of a graphical interaction were performed after analysis of the dynamic behaviour carried out by applying automatic tools on the LOTOS formal specification that detected a few possible inconsistencies. We call *interactor* the resulting abstract description of a general graphical interaction. This means that it encapsulates in a general framework the dynamic behaviour of the wide range of possible graphical interactions. One difference of this proposal with respect to other previous proposals such as MVC[GR83], PAC[C87] and others is that it allows us to clearly identify the relationship among input and output part of a graphical interaction. Moreover, it attempts to integrate concepts deriving from graphics systems and user interface systems in order to design systems that can provide interaction with structured graphics scenes.

Another important goal that we consider is to make the development of user interfaces easier. This implies to simplify the task-to-function translation: after identifying objects and attributes in the task-domain problem space, to transform them as immediate as possible, into functions, objects and attributes supported from the available system.

The general environment for the development of user interfaces that we refer consists in three phases:
- a visual editor where the designer defines the logical structure of the user interface by manipulating graphical representations of the interaction objects and their compositions [PF92a];
- the performed graphical representation can be automatically translated into a formal specification that provides an environment more suitable to verify its dynamic behaviour and properties [PF92b];
- when the designer is satisfied of the specification this can be refined into a programming language in order to obtain the executable system which the user interacts with.

This paper concerns the third phase. We have chosen an object oriented language for implementation because it allows us for reusability, modularity and data hiding and can provide a more suitable environment to design an implementation closer to the conceptual world of the designer. This choice implies different problems, for example, to find the good abstractions for building graphical interfaces, how to encapsulate state and operations of interaction objects into objects of the implementation environment, how to exploit inheritance in order to make extension easy.

In the next paragraph we discuss related works, then a description of the approach to the modelling of graphical interaction and a their design space are presented. After remembering the principal concepts of Sather we show how the abstract model can be implemented in this object-oriented environment. A proposal to the design of a hierarchy of interactor characterized by the supported user task and then, in a second moment, by their appearance is presented. Finally we apply the presented concepts to perform an example of user interface.

Related Work

One of the first serious attempt to address an object-oriented approach to user interface design was [B86], where the first aspects of this approach were located: a hierarchy of graphical objects, the possibility to compose objects in order to consider them as a unique entity and to define dependency among them, the separation of interface and application.

In [LB90] there is a proposal for a layered user interface system (in Procol language) with different element types depending on the layer: physical events, abstract events, interaction tools, panels. It provides a design where an interaction technique consists of three components: the graphics presentation, an event handler and a method part. They claim that the familiar class-inheritance paradigm, adopted by most object-oriented models is too static and inflexible and consider a prototype-instance model more appropriate. Here instances are not created from a class description but from a (possibly initialized) prototype. They use protocols to specify input patterns and sequencing. Because all access to the underlying window system has to go through a window object they create a portable framework for designing and developing interactive systems. The model aims to provide extensibility, adaptability of interaction styles and support for graphical interaction. In this way they want to avoid the low level of programming of existing toolkits. One limitation of this approach is that it does not allow for pick logical input devices.

Hill [H90] presented a 2-D graphics system for multi-user interactive graphics based on a hierarchical display structure of graphical objects and extensive use of constraints to maintain graphical consistency. He identifies six requirements for graphics systems: structural parallelism, composition of graphical objects, communication, responsiveness, hierarchy, abstraction.
Shan presented [S90] another proposal for user interfaces modelling. He starts from the consideration of the rigidity of the Smalltalk approach, then he decomposes a user interface in modes, each one with its semantics (defined as a connection of the underlying application which generates the semantics), its appearance and its interaction component. In [B92] there is an attempt to separate user interface details from widget semantics by defining abstract classes for appearance

and behaviour. In this way it is possible to obtain a unique manipulation for widgets previously separated. This was carried out in C++ and OSF/Motif environment. This approach is still strongly affected by the existing toolkit design.

In [WK90] some requirements for an object oriented second generation graphics standard are provided, among them we can remember: inheritance, integration of geometric modelling in a kernel; multi level part hierarchy with a well defined semantics; dynamic model for part editing (the number of subparts may change dynamically during run-time); more flexible communication patterns (mutual communication among application and graphical user interface); rule intepreters and constraint solvers for supporting new declarative and less imperative styles for graphics programming. They claim that the key point for the definition of the graphic standard is that a clear description of the messages which can be sent by the application programmer to instances or classes of the kernel must be given. Here the problem to integrate graphics system concepts with those of UIS is not addressed.
The problem to try to map user task with interaction techniques was addressed in [FWC84] but they did not consider object-oriented implementation environments.

The Abstract Design of the Graphical Interaction

Modelling of the graphical interaction is a continuing evolving issue. We can identify three tendencies:
- The input model of current graphics systems such as GKS and Phigs, that is about twenty years old but is still used, classifies logical input devices depending on the data type that they return towards the application (a point, an identifier of a graphical object, an integer and so on). Its limitations are well known: the appearance of input devices is implementation-dependent, there is no possibility to connect input and output functionality without application support, it provides a fixed set of data types, it classifies in the same logical input device, systems with very different approach and behaviours;
- The toolkits of window systems such as Athena X toolkit [SW88] and Interviews [LVC89] have a different approach: they model their classes depending on the external appearance (a scrollbar, a menu, a button) and provide functionality to compose them mainly from a layout point of view. In this case the limitations are: too low level interfaces for the programmer, they focus their attention more on appearance and layout issues, rather than on more important semantic issues [J92]; the interaction with the callback procedures has different problems: when a procedure is called it is not possible to activate other parts of the application until it returns; they do not mesh well with application semantic because know nothing about the values that they handles (for example they allow an user to select a label but they returns an identifier that the application may map erroneously to another item);
- Other modelling techniques, such as Small talk and PAC do not provide clear indications about the strict relationship among input and output functionality.

Our approach to the abstract design of graphical interaction has two main goals:
- to integrate the concepts of the toolkits of window systems with those of graphics systems. This problem, as the previous section showed, is not yet deeply investigated;

- to introduce the user point of view in the design of these systems in order to obtain a better usability, either from the UIS designer point of view either from the user point of view. In [NC92} is remembered that general model for interaction objects may be difficult to apply because developers often have problems identifying the objects for their system. For this purpose it is important to have interactors characterized for how they support the user task that means what type of manipulation they allow users to perform. Also allowing to associate different presentations with the same task-oriented interaction should be possible but without making the choice of the presentation the main and unique problem in developing user interfaces.

We can see a graphical interaction (Fig.1) as composed of three components: the *collection*, where an higher level description of the output is provided; the *picture*, that provides a description of the same scene described by the collection but at a lower abstraction level and determines how to represent echoing of input values; the *measure*, that receives input data from the lower levels, it applies its function on them and then delivers the result to the outside.

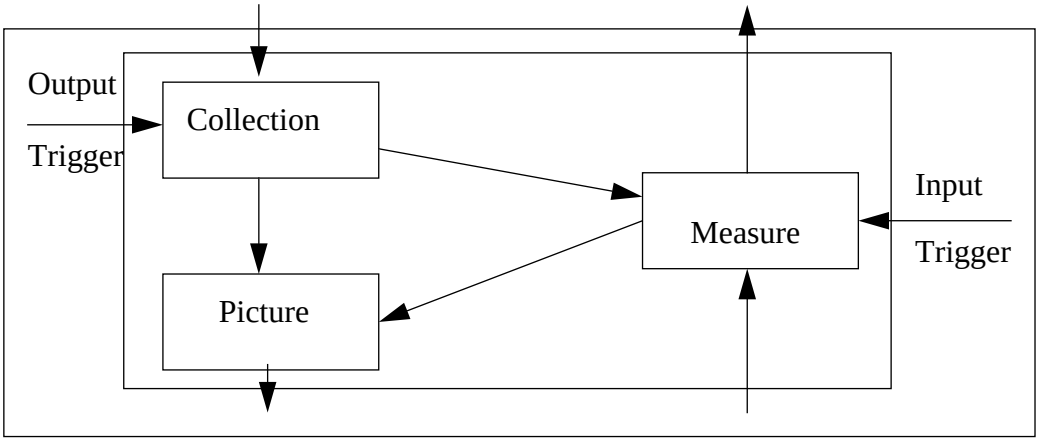

Fig.1 The Architecture of a Graphical Interaction Object

As example of the possible environments performed by the couple collection/picture we can refer the Computer Graphics Reference Model [ISO91] where the graphics functionality were subdivided into five layers: the application-oriented model; a completely geometrically defined scene; a view of the scene; the complete association of attributes to the output primitives; the access to the physical devices. More generally speaking, the transformation in the output part can be of two types:
- common graphical transformations typical of graphic output pipelines for obtaining an image from the application model such as those indicated in the Computer Graphics Reference Model;
- different output representations of the same application data, i.e. a pie chart versus a bar chart for visualizing the same data set;

We can single out in each interaction object two parts: the internal one defining its semantics, which data type the measure produces toward the application (that is different from the application semantic that means what the application performs by the received specific value); the external one defining its appearance that is defined by the collection and the picture. Then we have its

dynamic behaviour defined by enabling/disabling the possibility to receive data from the outside and by the interaction among the input and the output part.

The layout management among interactors appearance is performed by the low level interactors. There exists one for each frame. The frame is an output area which different interactors refers to. Inside the collection of this low level interactors it is possible to define constraints operators among the output area associated with different interactors.

There are two ways to manage the control of interactors:
- when they are active there is an input trigger associated with the measure to indicate when its function has to be applied and an output trigger indicating when the collection has to be updated in order to send new graphical information to the picture and the measure. For example in a pop up menu we can consider the button press (that implies the menu appears) as an output trigger, and the button release (that implies that the current selected element is sent to the application) as an input trigger. Triggers can be considered as boolean functions that when are verified generate the control signal.
- there are control objects, *controllors*, whose main aim is to indicate when graphical objects are active that means when they are enabled to receive data; this decision is performed depending on the events that controllors can receive.

Starting from the design of an interactor we can define a design space for graphical interaction objects (Fig.2): one axis for appearance, the graphical representation produced by the output part; one for semantics, the information transmitted toward the application; and one for input, what is received from the user to perform the interaction. The control is external to this design because from this point of view is not important if the trigger event is performed by pressing a mouse button or by speaking in a voice device. The design space for interactors is different from the proposal contained in [S90] because here the semantics is defined in a more precise way as the data type delivered by the measure to the application or to higher level interactors. If we consider the example of two buttons where one returns the "yes" string and the other returns "no" within this approach they are considered with the same semantics (a string) while in [S90] are different.

By this classification we can compare interactors. We have interaction objects that have the same appearance and semantics such as a menu (where the appearance is a list of similar elements and the semantics is the string associated with the selected element) but are differing for the input (we can have menu where the element of interest is selected by a pointing device but in other cases this can be performed in different ways, for example by typing the word associated with it) otherwise we can have interactors with the same input but different semantics, for example a pick interactor and a locator interactor, both receive a point as input but the former returns a graphical object while the latter returns a point in higher level coordinates, and they also provide different appearances (in the pick case a set of selectable graphics objects and in the locator case a cursor shape).

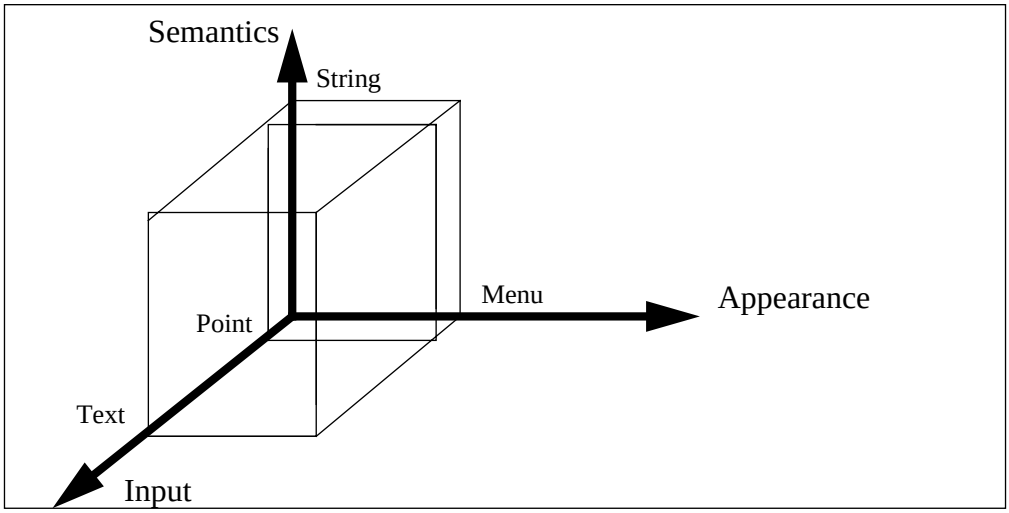

Fig.2 A 3D Space for Interaction Objects Classification

Another goal of our approach is to have a full symmetry among input and output data types. So, for example, it is possible to have as input a typical output primitive such as a polyline. Generally the graphics entities are defined by a code, a geometry and a set of attributes. The type of attributes and geometry depends on the considered abstraction level.

Sather

To perform a prototypal implementation of interaction objects following the previously described design the programming language Sather (it is public available at icsi-ftp.berkeley.edu) was used. In this section we remember the main basic concepts of Sather. Its primary goals are efficiency and reusability. The Sather compiler generates efficient C code that in this way can be considered as a portable assembly language. This language takes some concepts from Eiffel but stresses implementation inheritance, simplicity, and efficiency and its semantics differs from Eiffel in a number of respects (for example the Sather type system allows the programmer to explicitly distinguish between declarations that cause dispatch and declarations that are resolved by the compiler). It is more structured and with more features with respect to C++. In [SO91] there is a comparison among CLOS, Eiffel and Sather.

Sather features are: clean and simple syntax, parameterized classes, object-oriented dispatch, multiple inheritance, strong typing and garbage collection. All code is partitioned into units called classes. The entities defined in a class are called its features. There are four kinds of features: routines, object attributes, shared attributes and constant attributes.

A class A inherits the features of another class B means that the features of B are textually copied into A at the point the name appears. Later features in a class with the same name as earlier ones override the earlier definitions. This means that the order in which classes are inherited can have an effect on the meaning. A class may inherit from many other classes (multiple inheritance). The inheritance relations define an acyclic directed graph. Classes serve both the functions of declaring the structure of objects and encapsulating code into well-defined modules.

There are six kinds of variables in Sather: the shared, constant, and object attributes of classes, and the arguments, local variables, and return values of routines. The four kinds of type specification are: non-parameterized, whose features are completely specified by the class specification; parameterized, where the name of the class is followed by the specification of the type of the parameter; type parameter, where the type of the parameter is specified by a type variable, when a parameterized class is used, the type variable is instantiated and the compiler generates appropriate calls; dispatched, where the type will be dynamically dispatched and it can be a subclass of the given class, there is a runtime mechanism that chooses the routine to be called depending on the runtime type of an object.
Two kinds of type specification can be considered mechanisms for old code to call new code: parameterized classes allow the compiler to optimize such calls; object-oriented dispatch is a run-time mechanisms which gives more flexibility at the expense of some efficiency.
The second one is performed because Sather's typing rules are based on late checking in order to ease rapid prototyping and provide maximal freedom in reusing existing definitions despite the strong-typing approach. This means that inherited definitions need only be type-correct in the descendent context and client calls to actualized parametric classes need only be type-correct in the actualized version of that class. Parameterized classes can be used to model open modules some of whose operations are only defined in combinations with other classes in the different descendents.

The problems that the choice of using Sather as implementation language opens are: how is possible to map the previously defined abstract design of user interfaces into the Sather Language; is Sather suitable to support user interfaces development or there are features of objected oriented languages that would be useful but are not present in Sather?

A Sather Interactor

In order to support reusability by exploiting inheritance we define two types of class hierarchies: one for interactive objects, one for graphical entities. The most immediate approach to map an interactor into an object oriented language is to try to encapsulate the general behaviour of an interactor object in the definition of a class. This is the superclass for all interactors. Then we have a class hierarchy for each of its main components (collection, picture, measure). By multiple inheritance an interactor has the features of one instance of each of these objects. A subclass of an interactor can modify these components.

In the refinement from LOTOS specification into Sather implementation we followed the following rules:
- an interactor that in LOTOS is obtained by a parallel composition of processes in Sather is performed by an object class that is obtained by multiple inheritance of the objects associated with the component processes of the LOTOS parallel composition;
- a LOTOS process is associated with a Sather class;
- the parameters of the processes are associated with the internal data of the classes;
- the behaviour of a process usually is a choice among different behaviours: we associate each possible behaviour choice with a class function;
- for each possible behaviour in the process definition the input commands indicate the parameters

for the corresponding Sather function;
- when a LOTOS process calls itself recursively with a modified status this is reflected in the Sather implementation by an assignment to the internal data of the corresponding class.
Future work will be dedicated to check the validity of the results obtained by making automatic this translation.

If we compare LOTOS, that is a formal specification language, and Sather, that is a programming language, we can notice how Sather provides the possibility of compact and structured code especially with respect to ACT ONE, the notation for specification of algebraic data types included in LOTOS.

An object of class interactor has in its definition three objects belonging to different classes (collection class, picture class, measure class) and four functions, one for each possibility to receive and processing information from the outside. These functions allow to perform the associated interaction depending on the data received from the outside. For each of the component object a class hierarchy is defined: the collection for manipulation of output structured graphics data and its visualization; the picture carries out a lower graphics output and performs feedback of the current element selected by input; the measure performs input processing.

Class	*Protocol*
INTERACTOR	create, input, output, inputrigger, outputrigger
COLLECTION	create, interpret, trav_meas, trav_feed
FEEDBACK	create, output, highlight
MEASURE	create, update, measure, echo

Table 1

The collection has a state composed of a set of graphical entities. Its functions allows it to receive new entities (*interpret*); to traverse them and send the result to the feedback (*trav_feed*); to traverse them and send the result to the measure (*trav_meas*).
The feedback has a state composed of a set of graphical entities (at a lower level with respect to the collection), and a function to update its state depending on the data received from the collection and to transmit the results to the underlying layers (*output*) and another to provide echoing of the new data received from the measure (*highlight*).
Finally, the measure has a state composed of the last input data processed and a set of graphical entities received from the collection that are necessary to compute its function. Here the functions are: to update the measure when receiving new items from the collection (*update*); the *measure* function to apply its processing on the last received element when the input trigger is verified, and the *echo* function that is computed when new input data are received and it returns the current selected element to the feedback object for performing echoing to the user.

Now we consider the specification of the root of the interactors hierarchy. It is obtained by multiple inheritance of the root of measure, collection and feedback hierarchies. An interactor object has associated five functions: create, input trigger, input, output trigger, output. Then the

interactors descendents can define different behaviours with respect to the general one and can compose by multiple inheritance instances of the descendents of its components objects. The variables preceded by the dollar sign can held objects of a descendent of the class which follows the dollar sign in the specification. They are used when it is not possible to know previously the data type that will be used.

When the *input trigger* is verified the measure function is called with the indication of the interactor that will receive its result. This is useful because in this way it is possible to describe UIS whose topology can always dynamically changed. When an input data is received in the *input* function, this is passed to the measure to compute the information necessary for the echo that is delivered to the feedback that will perform the related highlighting. When the *output trigger* is verified then if it is the first time that it occurs the interactor is initialized, that means the related window is created and mapped by the underlying window system, otherwise the current collection is traversed and the related data are transmitted to the feedback and the measure. If output data are received by the *output* function they are added to the current collection.

```
class INTERACTOR is
  -- A General Description of a Graphical Interaction

col: COLLECTION;
feed: FEEDBACK;
mea: MEASURE;
init: BOOL:=false;

create(cl: $COLLECTION; fd: $FEEDBACK; ms: $MEASURE) : SELF_TYPE is
res:=new;
res.col:=cl;
res.feed:=fd;
res.mea:=ms;
end; -- create

inputrigger (bl: BOOL; tin: $INTERACTOR) is
if bl then mea.measure(tin); end;
end; -- inputrigger

input(pin: $ENTITY) is
feed.highlight(mea.eco(pin));
end; -- input

outputrigger (bl: BOOL) is
if not(init) then init:=true; end;
if bl then mea.update(col.trav_meas); feed.output(col.trav_feed);  end;
end; -- outputrigger

output (op: ARRAY{$ENTITY}) is
col.interpret(op);
end; -- output
end; -- class INTERACTOR
```

Now we want to show how a simple common graphical interaction such as a menu can be described within this approach. In the menu collection we can find an array with an element for each item of the menu and all the elements have to belong to the same class. In the case of the rectangle menu each element is associated with a rectangle that defines the area related to the ele-

ment, a string that appears inside the rectangular, an identifier that is defined by the position in the array and the background color and the foreground color (for drawing the rectangle and the string). When the output trigger is verified the collection is interpreted. In this case it means to send to the feedback the same information and to the measure the list of rectangles, strings and the associated identifiers.

The feedback draws to the outside the last picture that received from the collection. When receives a value from the measure it performs echoing for example by inverting foreground and background color of the indicated element.

The measure receives as input a point. It finds out which rectangle the point is internal to by using the information received from the collection and then it transmits the related identifier to the feedback for echoing. If the input trigger is verified the string of the current selected element is delivered to the outside.

The Design of the Refinement of a Hierarchy of Interactors

The traditional approach to object-oriented user interfaces is to associate the graphical objects visible from the user with an underlying object representation. Here we aim to a task-object association. This means that we want to make it possible to design user interface systems by creating instances of objects that are characterized more by their semantics than their appearance. In order to stress the semantic aspects in the design of the hierarchy of the available interaction objects, its first levels depend on their semantics, then we distinguish subclasses by the received input data and finally by the type of output visualization applied. An example of a typical user task that can have different appearance is a choice that can be performed by a wide set of interaction techniques such as a set of button-radio or a menu or a cycle button or a type-in-field. We think that this approach allows for a more immediate development of user interfaces. Refinement of appearance can be left as a secondary aspect. This is performed by exploiting multiple inheritance that allows us to associate different presentations with the same semantics in the definition of interactor objects.

We identify four basic task:
Choice, the selection of an element from a set of visualized objects;
Position, the selection of a specific point in the space defined by the graphic application, we can distinguish subclasses depending on if the point is 2D or 3D and for the echo type;
Quantifier, the definition by a graphic interaction of a specific input value of a particular application domain, its subclasses allow to distinguish among input of a constant value (such as a button), input of a value chosen among a set of available ones, input of a value chosen among an interval of values (such as a scrollbar);
Text, the input of user defined text. Providing text to the application is classified a text task just only if it is passed by an input device such as a keyboard. If the input text is obtained by selecting some text already visualized this is considered a subclass of choice interaction.

Now we want to show more in detail an example of this methodology for designing a hierarchy of interaction techniques. We can define a Choice class as an interaction allowing the user to

identify an object out of a set by providing a point. We have to make a first distinction among single Choice and a Multiple Choice class that allows to select more objects from the available set. This means that the first refinement is performed on the semantics of the interaction. In the interactor hierarchy under both of them there is the same subtree of interaction objects, the only difference is the number of returned elements. Then we can refine the subclasses introducing appearance aspects. If the selectable elements are all of the same type we have a Menu subclass otherwise we have a Pick subclass. A Transformation interaction object returns the selected element after having applied a transformation on it. The Menu can be further refined by only appearance-dependent aspects indicating the type of the menu elements (rectangles, ovals, pixmap and so on). Another refinement can be obtained by the type of echo: inverting foreground and background colors; by adding a symbol such as a cross on the selected element; by providing a 3D echo and so on.

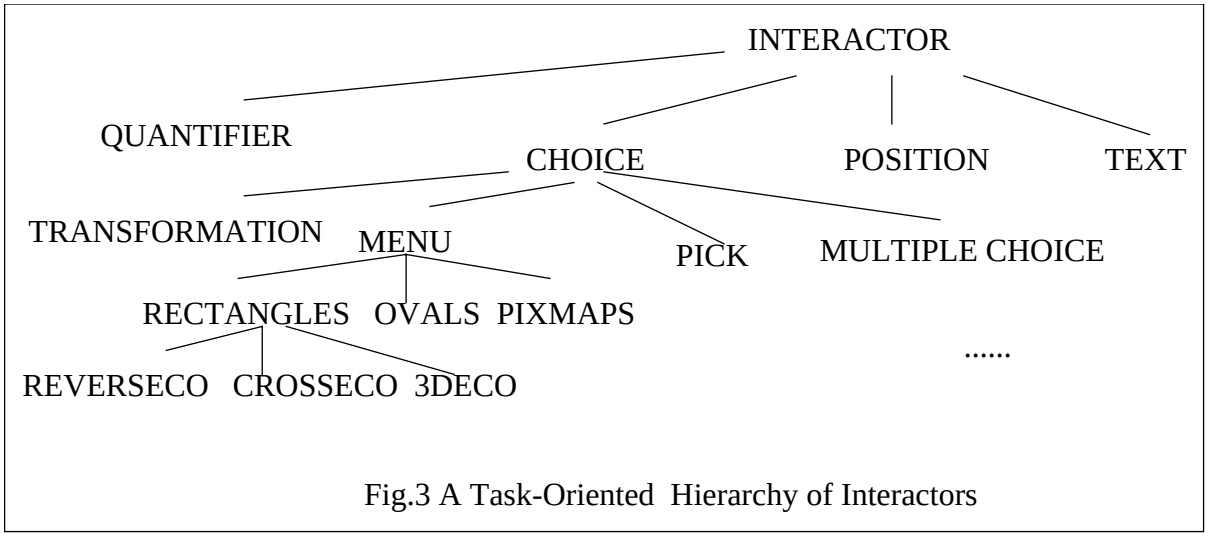

Fig.3 A Task-Oriented Hierarchy of Interactors

If we want to define a Choice object from the root interactor superclass we have to refine a measure object in order to allow it to have a state composed of a point received from the lower level, a set of area associated with menu elements and a measure function that by comparing the point received with the set of area can detect the performed choice. This means that the Choice class differs with respect to the Interactor class only for inheriting a measure_ch object instead of its ancestor, measure. The only difference in this two classes is that the measure function of the measure_ch class allows to select one graphics entity of entity_ch class or of its subclasses depending on the received point.

An Example of User Interface obtained by the presented approach

The architecture of a UIS can be obtained by composing graphical interaction objects in two possible ways: the picture of an interactor is delivered to the collection of another one (for example the first interactor performs some modelling transformation and the second works out a projection of the received 3D scene); the measure of an interactor passes its data as result to the measure of another one (for example one interaction object receives a point in device coordinates, transforms

it in logical coordinates and passes the result to another object that selects one graphical element depending on the position received). At the end we obtain a multi-layered system where the elements of all the layers are defined in an uniform way.

Within this approach it is possible to create systems more powerful with respect existing graphics systems because it is possible to perform a wider set of manipulations of graphical items. This means that in this type of approach the role of the application is more limited. The resulting architecture of a UIS within this approach is a graph with the lower level elements interacting with the user and the higher level elements interacting with the application.

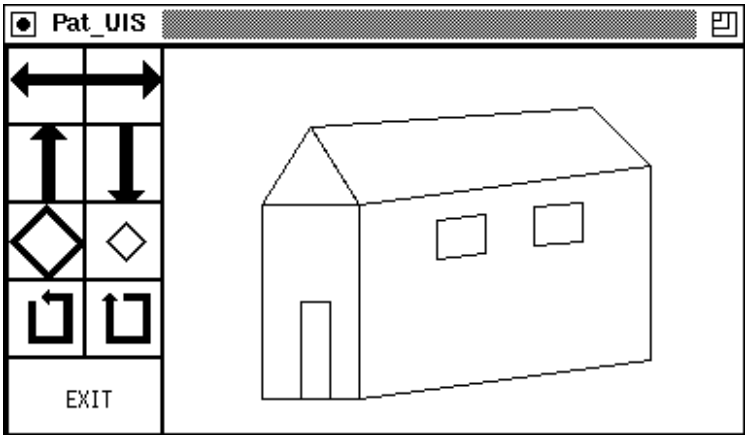

Fig.4 An example of performed graphical user interface

Now we consider a simple example in order to show how it is possible to perform an interaction with a graphics scene within this approach. We have a graphical editor such as that represented in figure 4 where the application visualizes a graphical scene and the user can select one graphical element in the set of defined graphical objects that may compose also a complex scene, and then he can indicate one transformation (there is one button for each possible transformation such as shifting right, left, down, up or rotating clockwise, counterclockwise or scaling bigger, smaller) that is automatically applied with a constant coefficient on the selected graphical object.

To perform this interaction we have: one object of pick class whose output part performs visualization of graphical entities (when the collection sends them to the picture it applies a transformation from user coordinates to window coordinates) while the input part allows the user to select one of them; a set of objects of button class associated with each possible basic transformation that in this case return a string indicating the type of the related transformation; one object of transformation class performing the selected transformation.

There is also a Controller objects that deliver the input trigger events for all the interactors. It communicates with the window system: when the button of the mouse is pressed the window system inform it about which is the interactor whose output area is containing the cursor and it calls the related input trigger function. In the case of the Pick object the controller sends to it before the position of the cursor and then the input trigger signal so it can detect which is the selected element

by the point indicated by the cursor when the button of the mouse was pressed. One Controller can implement different policies to perform graphical interactions depending on when it generates the trigger events: in our example it can impose to have an object-oriented specification of the transformation (the user has to select the object and then the transformation) or a command-oriented specification (before selecting the transformation and then the object to modify).

The Pick object works out the current selected object and sends it to the Transformation Interactor. This is an only input object that means that it has not a collection and the feedback performs only echoing of the measure. In this case returns the transformed graphical object to the Pick Interactor that will visualize it in the new state.

The Transformation object to perform its functionality has to receive data of different types in the input function: graphical primitives from the Pick object, transformation indication from the Button objects. This was made possible exploiting the dispatched objects provided by Sather: the input data type was defined as $entity that means it can be any subclass of entity (that is the root class of all graphical objects) and then when a data is received a test on its type is performed before deciding which type of processing to perform. Figure 5 shows how these objects are related.

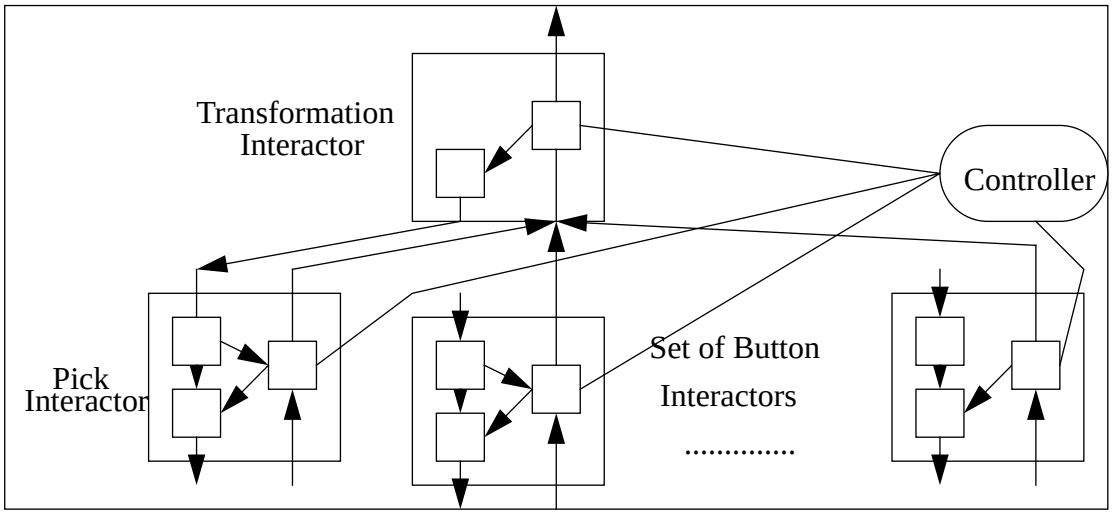

Fig.5 The composition of interacting objects performing the example

The communication between the application and the UIS is performed in the following way: when the application wants to pass data to an interaction objects it has before to provide them by the output function of the object and then, when it wants that the processing of the interaction object is applied, it has to call the output_trigger function.

In the implementation some problems were raised by the controvariant rule that Sather applies. It requires that the arguments of routines in descendents be supertypes of the corresponding arguments in ancestors in the case of arguments of dispatched types. It is imposed to guarantee that the descendent can handle any call made on the parent. This was slightly rigid because sometimes in defining a hierarchy of interaction techniques where we want that subtypes can receive or produce graphics entities that are subclasses of those manipulated by superclass interactors.

Conclusions

We shown how the hierarchies of classes related to interaction objects should be primarily modelled depending on their semantics in order to make the task-to-function translation easier. This contrasts with the current toolkit approach where they are modelled depending on their appearance.

This classes should be obtained by multiple inheritance of objects performing abstract description of the object, its appearance and its input functionality. This allows easy reconfigurability of interactors. For example the same semantics can be easily associated with different appearance just changing the collection and the feedback part that are inherited.

The system that is being developed is highly portable because has been built for using the standard de facto X Window System as underlying window system.

Sather is currently being extended to pSather, an experimental parallel version of the language that adds threads, synchronization, protection and exception handling [FLM91]. Future work will be dedicated to pass the current prototypal implementation in the parallel Sather in order to exploit all the possibilities of the model for UIS that is intrinsically parallel and to apply the system for interacting with a neural networks simulator.

Acknowledgments

This work was carried out whilst Fabio Paterno' was Visiting Scientist at the International Computer Science Institute. Support from Consiglio Nazionale delle Ricerche and International Com puter Science Institute is gratefully acknowledged. We would also like to thank David Bowley for useful discussions on the implementation.

References

[B92] A.Baker. "Designing Reusable Widget Classes with C++ and OSF/MOTIF". The X Resource 2, Spring 1992. pp.106-130.
[B86] P.Barth. "An Object Oriented Approach to Graphical Interfaces". ACM Transaction on Graphics, Vol.5, N.2, April 1986, pp.142-172.
[BB87] T.Bolognesi, H. Brinskma. "Introduction to the ISO Specification Language LOTOS". Computer Networks and ISDN Systems, vol.14, pp.25-59, 1987.
[B91] L.Bass et al. "A Metamodel for the Runtime Architecture of an Interactive System". SIGCHI bulletin, January 1992, pp.32-37.
[C87] J.Coutaz. "PAC, an Object Oriented Model for Dialog Design". Proceedings Interact'87, pp.431-436.
[DH91] D.Duce, F.Hopgood, R.Gomez, J.Lee (Eds.). "Report of the Concepts, Methods, Methodologies Working Group" in "User Interface Management and Design". pp.35-45. Springer Verlag 1991.
[FP90] G.Faconti, F.Paterno'. "An Approach to the Formal Specification of the Components of an

Interaction". Proceedings Eurographics ’90. Montreaux. pp.481-494.
[FLM91] J.A.Feldman, C.C.Lim, F.Mazzanti. "pSather Monitors: Design, Tutorial, Rationale and Implementation". ICSI Technical Report 91 - 031. 1991.
[FWC84] J.Foley, V.Wallace, P.Chan. "The Human Factors of Computer Graphics Interaction Techniques". IEEE Computer Graphics & Application, pp.13-48, November 1984.
[GR83] M.Goldberg, D.Robson. "Smalltalk-80: the Language and its Implementation". Addison Wesley Publishing Company, Reading Ma, 1983.
[H90] R.Hill. "A 2-D Graphics System for Multi-User Interactive Graphics Based on Objects and Constraints". pp.67-91, in Advances in Object Oriented Graphics I. Springer Verlag.
[ISO91] ISO/IEC DIS 11 072. Information Processing Systems. Computer Graphics. Computer Graphics Reference Model. ISO centrals Secretariat, Geneva, 1991.
[J92] J.Johnson. "Selectors: Going Beyond User-Interfaces Widgets". Proceedings of CHI’92 Conference, pp.273-279.
[LVC89] M.A.Linton, J.Vlissides, P.Calder. "Composing User Interfaces with InterViews". IEEE Computer, February 1989, pp.8-22.
[LB90] C.Laffra, J.van den Bos. "A Layered Object-Oriented Model for Interaction", pp.47-61, in Advances in Object Oriented Graphics I. Springer Verlag.
[NC92] V.Normand, J.Coutaz. "Unifying the Design and Implementation of User Interfaces through the Object Paradigm". Proceedings of ECOOP ’92. July 1992.
[O91] S. Omohundro. "The Sather Language", ICSI Internal Report, June 1991.
[PF92a] F.Paterno’, G.Faconti. "A Visual Environment to Define Composition of Interacting Graphical Objects", to be published on the Visual Computer, Springer Verlag. 1992.
[PF92b] F.Paterno', G.Faconti. "On the LOTOS Use to Describe Graphical Interaction". Proceedings HCI Conference 1992. York. Cambridge University Press. In press.
[SO91] H.Schmidt, S.Omohundro. "CLOS, Eiffel and Sather: A Comparison", ICSI Internal Report 91-047, September 1991.
[S90] Y.Shan. "An Object-Oriented Framework for Direct-Manipulation User Interfaces. pp.3-19, in Advances in Object Oriented Graphics I. Springer Verlag.
[SW88] R.Swick, T.Weissman. "X Toolkit Widgets - C Lamguage X Interface". X Version 11, Release 2, MIT Project Athena 1988.
[VL9O] J.Vlissides, M.Linton. "Unidraw: A framework for Building Domain-Specific Graphical Editors", pp.237-268, ACM Transactions on Information Systems, Vol.8 N.3, July 1990.
[WK90] P.Wisskirchen, K.Kansy. "The New Graphics Standard - Object Oriented", pp.199-215, in Advances in Object Oriented Graphics I. Springer Verlag.