User Interfaces with Object-Oriented Programming in APL2

Manuel Alfonseca

Escuela de Ingenieria Informatica,Universidad Autonoma de Madrid
Ciudad Universitaria de Cantoblanco, 28049 Madrid SPAIN
Telephone: (34) 1 397 4467; FAX: (34) 1 397 5277
Manuel.Alfonseca@ii.uam.es

The power of general arrays is used to provide APL2 with object-oriented capabilities, which are used to generate user interface object classes such as windows, menus, dialog boxes and messages, among others, all of which can be created as persistent objects. This makes very straightforward the development of user interfaces for real applications.

The two OOP paradigms

According to Peter Wegner [Weg 95], there are three steps towards Object-Oriented Programming (OOP in short):

- **Objects**: the ability to represent data and programs as chunked pieces isolated from their environment.
- **Classes**: the capacity to classify objects into sets. Mathematically, this corresponds to the **belong** relationship.
- **Inheritance**: the fact that classes may be divided into subclasses that inherit the behavior and properties of their superclass. Mathematically, this corresponds to the **subset** relationship.

Systems that only provide the first step are called **object-based**. Those that also provide the second step are called **class-based**. Inheritance is required for an object-oriented system. Thus, the Microsoft **COM** standard is object-based, while the **CORBA** standard is object-oriented.

OOP systems provide three important characteristics:

- **Encapsulation**: complete isolation of objects from their environment, except for a subset of its behavior, which constitutes its public interface.
- **Polimorphism**: the fact that different objects may have behaviors (programs or **methods**) with the same name, but with different code, according to the class of the object.
- **Message passing**: Objects collaborate with each other by means of messages with the following structure:

```
Object Method Parameters
```

If messages are the only means for object collaboration, we have **pure OOP** or **programming without call**.

There are currently two different Object-Oriented paradigms:

- The class-instance paradigm, which differentiates explicitly between the classes and the objects that belong to them (instances). Objects inherit the methods and properties of their class or its superclasses, but not necessarily the values of the properties, which should be inicialized by a suitable **constructor**. This paradigm has two variants:
    - The declarative approach: classes are declarations; objects are data structures declared to belong to a class. In this variant, no class is an object. This is the C++ approach.
    - The all-object approach: classes are special cases of objects, and as such they also belong to a

metaclass. In this variant, every class is an object, but not all objects are classes. This is the Smalltalk approach.

- The prototype-instance paradigm, which does not differentiate between classes and the objects that belong to them (instances). Every object may be used as a **prototype** for the construction of new objects, and as such may be considered as a class. Therefore, in this paradigm, every object is a class. Objects inherit, not only the methods and properties of their prototype, but also the values of the properties, which prevents the need of initialization. This special kind of inheritance is usually called **delegation**. OOP systems that work according to this paradigm are CMU **Amulet** [McD 95], based in C++, and OOPI [Alf 92], based in C and developed by the author.

A prototype-instance OOP system based in APL2

The system proposed here applies the prototype-instance paradigm to the construction of user interfaces in APL2, and is a complete remake of a previous work by the author [Alf 89].

Objects are represented in APL2 by matrices of two columns. The first column defines the name of a **slot**, the second contains its value. There are four kinds of slots:

- **PARENT** slot: its value is the name of the object that served as a prototype for this object.
- **CHILD** slot: its value is a vector of the names of the objects that have been generated using this object as a prototype.
- **Method** slots: their value is the string "METHOD". The code for each method is an APL2 function whose name is the name of the object, catenated with an underline and with the name of the method slot.
- **Attribute** slots: they may have any value.

The first two slots (which are unique) allow us to reconstruct the object hierarchy, whose root is a special object, named **OBJECT**, whose **PARENT** slot has an empty value, and which contains 10 methods for general use, since they will be automatically inherited by every object:

- **METHOD**: creates a new slot method for the object that receives this message.
- **CREATE**: creates a new object using as a prototype the object that receives this message.
- **ERASE**: erases the object that receives this message.
- **PARENTS**: returns the list of names of the prototypes of the object that receives this message, up to the top of the hierarchy.
- **CHILDREN**: returns the list of names of those objects for which the object that receives this message is a prototype.
- **METHODS**: returns the list of methods available for the object that receives this message, both those defined for the object itself, and those inherited from its prototypes.
- **PROPERTIES**: returns the list of attributes available for the object that receives this message, both those defined for the object itself, and those inherited from its prototypes.
- **VALUE**: returns the value of an attribute of the object that receives this message, either a slot in the object itself, or inherited. This method has a parameter: the name of the desired attribute.
- **MOVE**: changes the prototype (PARENT) of the object that receives this message. This method has a parameter: the name of the new prototype.
- **LIST**: returns the names of all the objects available in the workspace.

Four additional APL2 functions are applicable to every object:

- **MESSAGE**: A monadic function whose right argument is a single string or a higher depth vector. In the first case, the string should contain the name of the object which will receive the

method, followed by the name of the method to be executed and the parameters (which must be literal), separated by spaces. Example:

```
MESSAGE 'OBJECT VALUE MUSIC'
```

In the second case, the vector should contain the same information as individual elements. Example:

```
MESSAGE 'OBJECT' 'VALUE' 'MUSIC'
```

- **ASSIGN**: A monadic function whose right argument is a vector of three elements: the name of one object, the name of an attribute, and a value for the attribute. It assigns the value to the attribute. A slot for the attribute is created if needed. Example:

```
ASSIGN 'OBJECT' 'MUSIC' 'CDEFGAB'
```

- **INSERT**: A monadic function whose right argument is a vector of three elements: the name of one object, the name of an attribute, and a value for the attribute. It adds the value to the attribute. A slot for the attribute is created if needed.

- **ERASE**: A monadic function whose right argument is a vector of two or three elements: the name of one object, the name of an attribute, and (optionally) a value for the attribute. If the value is given, the attribute is assumed to be a list, and the value is erased from the list. If the value is not given, the attribute slot is erased.

A message compiler

The syntax of the **MESSAGE** and related functions is somewhat cumbersome. Besides, **MESSAGE** acts as a message interpreter, locating the requested method either in the object that receives the message or in one of its ancestors. This interpretation slows the process, since it has to be executed on top of another interpreter. In general, a message compilation would accelerate the applications, at the same time allowing us to simplify the syntax.

For this reason, a message compiler has been developed that translates simplified messages into the appropriate APL2 function call. The only restriction introduced is the fact that a message must be the first operation to the left of a function line. Its general syntax is:

```
[label:] [variable assign]
   MESSAGE object method
    [parameters]
```

If the compiler has enough information about the objects to deduce the appropriate function call, the preceding line is replaced by

```
[label:]
   [variable assign | quadWA assign
]
   object            object_method
[parameters]
```

where **object_method** is the name of the APL2 function, which may be defined for the same object or for one of its ancestors.

If the compiler cannot resolve the message into a function call, it generates a call to the message interpreter (function **MESSAGE**).

The compiler also accepts instructions with the following syntax:

```
[label:] [variable assign]
   ASSIGN object attribute value
```

and converts them appropriately.

Finally, the compiler accepts two kinds of simplified conditional instructions and replaces them by their APL2 equivalents. Those two instructions are of the form:

- Conditional GOTO:

```
→label IF condition
```

which is compiled into:

```
→(condition)/label
```

- Conditional execution:

```
instruction IF condition
```

which is compiled into:

```
⍎(condition)/'instruction'
```

The compiler replaces the function being compiled by a new copy where it has included the appropriate changes, as indicated above. A decompiler makes it possible to recover the original function. In fact, a full-screen editor is provided (a modification of function **EDIT** in the **EDIT** workspace, which comes with APL2/PC for DOS), that automatically decompiles a function when invoked and compiles it again at the end, so that the user may forget about the existence of the compiler and write OOP functions with the simplified syntax.

Persistent objects

In many applications it is important to have persistent objects, i.e. objects that remain in existence after the application is closed and/or the machine is shutdown. In this system, persistent objects are stored inside apl2 object files (managed by the AP211 auxiliary processor) and can be created using any object as a prototype, by means of the general use method **CREATE**, whose syntax is:

```
MESSAGE 'OBJECT1 CREATE OBJECT2
       [FILE filename]'
```

where **OBJECT1** is the object to be used as a prototype and **OBJECT2** is the object to be created. If the **FILE filename** parameters are omitted, the object is created in the workspace and is not persistent. Otherwise, the object is persistent and is added to the indicated object file. If the file did not exist, it is created.

Two additional functions are required to make available one or more sets of persistent objects to the application. They are:

- **FILE_USE**: A dyadic function whose right argument is the name of a file of persistent objects. The workspace is made "conscious" of the existence of the objects stored in the file, which will be copied into the workspace the first time they are used. If the left argument is not given, the file is opened for read-only use. In this case, the objects copied into the workspace can be modified, but the changes are not persistent. If the left argument exists, the file is opened for read/write use. In this case, any changes to the objects copied into the workspace will be automatically saved in the file.
- **FILE_RELEASE**: A monadic function whose right argument is the name of a file of persistent objects. The workspace "forgets" about the existence of the persistent objects in the file.

User interface objects

The system has been used to define different object prototypes that may be used to develop user interfaces. These objects are:

WINDOW: defines a general text window object. This object owns two methods and five attributes. The methods are:

- **OPEN**: draws a window at the screen.
- **ACTIVATE**: draws a window at the screen and activates it for user action. Returns information about the key pressed and the cursor position.
- **CLOSE**: removes a window from the screen, recovering what appeared below.

The attributes are:

- **POSITION** in character rows and columns from the top-left corner of the screen.
- **SIZE** in rows and columns.
- **COLOR** for text and background.
- **TEXT** to be shown in the window, either as a vector of strings or a character matrix.
- **OPTIONS**: a vector of strings including 'R/W' if the user can modify the text on the window, and 'FRAME' if the window must be drawn with a frame around it.

MENU: defines a menu or a list box. This object owns two methods and five attributes. The methods are:

- **OPEN**: activates the menu for user action. Returns the option selected and the key pressed.
- **CLOSE**: removes the menu from the screen.

The attributes are:

- **POSITION** in character rows and columns from the top-left corner of the screen.
- **SIZE** in rows and columns.
- **COLOR** for text and background.
- **LIST**: The list of options to be selected.
- **OPTIONS**: a vector of strings including 'NOFRAME' if the menu must be drawn without a frame around it.

MENUBAR: defines a horizontal menu or action bar. This object owns two methods and three attributes. The methods are:

- **OPEN**: activates the action bar for user action. Returns the option selected and the key pressed.
- **CLOSE**: removes the action bar from the screen.

The attributes are:

- **POSITION** in character rows and columns from the top-left corner of the screen.
- **COLOR** for text and background.
- **LIST**: The list of options to be selected.

PANEL: defines a panel or a dialog box. This object owns one method and six attributes. The method is:

- **OPEN**: activates a panel for user action. Returns the texts typed by the user in the panel fields.

The attributes are:

- **POSITION** in character rows and columns from the top-left corner of the screen.
- **SIZE** in rows and columns.
- **COLOR** for text and background of the panel itself.
- **ACOLOR** for text and background of the active field.
- **TEXT**: The text to appear in the panel, outside the fields.
- **FIELDS**: a matrix of six columns defining the text entry fields. The first two columns give the field position, the next is the field size, the next two are the field colors (text and background) when not active, the sixth is the field type (0: read/write, 1: numeric, 3: read/only).

BUTTONBOX: defines a special type of panel: a message box with buttons. This object has **PANEL** as a prototype and owns one method and six attributes. The method is:

- **OPEN**: activates a button box for user action. Returns the number of the button selected.

The attributes are:

- **MESSAGE**: The text to appear in the box, outside the buttons.
- **BUTTONS**: a vector of strings with the text to appear at each button.

As a spectacular proof of the power of OOP for the development of reusable, extensible, easily adjustable and modifiable applications, two complete different interface systems have been developed:

- A text screen system, based on the AP124 auxiliary processor.
- A graphic screen system, based on the AP207 auxiliary processor.

Both systems are completely equivalent and allow the same application to work with a text or a graphic user interface with no change at all. The interesting thing is that

only four methods (APL functions) had to be reprogrammed to perform such a drastic change. Those methods were:

- **OPEN**, **ACTIVATE** and **CLOSE** for the **WINDOW** object prototype.

- **OPEN** for the **PANEL** object prototype.

All other methods, including those that belong to the remaining user interface objects, are exactly the same for both the text and the graphic versions. This is due to the fact that menus and button boxes (and, to a certain extent, panels too), make use of windows to represent themselves on the screen, which makes them screen mode independent.

A sample application

The application coded in figure 1 has been written in the simplified OOP syntax described in this paper. This application is an object editor that uses the following user interface objects:

- **BG**: a green background window.
- **TW**: a title/help window at the bottom of the screen.
- **NW**: a window at the bottom of the screen, superimposed on the preceding window, where the name of the object currently being edited is shown.
- **MW1**: a window that acts as the background for the method list box.
- **MW**: a listbox (menu) without a frame, where the methods available for the object being edited are shown. Both the methods owned by the object and those inherited from its ancestors appear here. A method may be executed by pressing the ENTER key on top of its name.
- **PW1**: a dialog box to request the parameters for the execution of a method.
- **RW**: a window where the result of a method execution is shown. If the ENTER key is pressed on a given line of the result, the line in question is assumed to be the name of an object. If this is the case, that object becomes the current object being edited. Figures 2, 3, and 4 show different instants during the execution of the application, in the case of the text interface. Figures 5, 6 and 7 show the equivalent situations for the graphic interface. Figure 8 is a different application (an educational program in Chemistry to teach the periodic table). The screen in the figure contains 112 different windows (one per element, plus the background and two information windows).

Conclusion

The prototype-instance object-oriented approach is easy to implement in APL2, as demonstrated by the application described in the paper. The performance impact caused by double interpretation may be compensated by means of a partial precompilation.

Object-orientation has proved to be a proper way of handling reusable user interfaces. Our APL2 approach makes it very easy to program them in such a way that the same interface may be used for both text and graphic applications.

References

- [Alf 89] Alfonseca, M.: "Frames, semantic networks and object-oriented programming in APL2", IBM J. of Res. and Devel., Vol. 33:5, p. 502-510, Sep. 1989. Interface and Integrated Development Environment", 3rd European Symposium on Object-Oriented Software Development, B”blingen, Oct. 1992.

- [Alf 92] Alfonseca, M.: "OOPI/OOL/OODE: An Object-Oriented Program Interface and Integrated Development Environment", 3rd European Symposium on Object-Oriented Software Development, Böblingen, Oct. 1992.

- [McD 95] McDaniel, R.; Myers, B.A.: "A Dynamic and Flexible Prototype-Instance Object and Constraint System in C++", Carnegie-Mellon University Technical Report, 1995.

- [Weg 95] Wegner, P.: "Interactive Foundations of Object-Based Programming", Computer, pp.70-72, Oct. 1995.

Figure 1

```
VIEW;NAME;RESULT;METHODS;VALUE;PARM
NAME←'OBJECT'
FILE_USE 'VIEW.OOP'
MESSAGE BG OPEN
MESSAGE TW OPEN
MESSAGE MW1 OPEN
LOOP:NAME←'OBJECT' IF~EXIST NAME
ASSIGN NW TEXT NAME
MESSAGE NW OPEN
METHODS←MESSAGE NAME METHODS
ASSIGN MW LIST METHODS
RESULT←MESSAGE MW OPEN
→END IF 0=ρRESULT
→LOOP IF(↑RESULT)>ρMETHODS
PARM←MESSAGE PW1 OPEN
VALUE←MESSAGE NAME((↑RESULT)⊃METHODS)(↑PARM)
→NEXT IF 0∊ρVALUE
VALUE←⊃VALUE IF 1<≡VALUE
ASSIGN RW TEXT VALUE
RESULT←MESSAGE RW ACTIVATE
MESSAGE RW CLOSE 'NR'
NEXT:MESSAGE MW CLOSE 'NR'
MESSAGE NW CLOSE
→LOOP IF 0∊ρVALUE
→LOOP IF 4 1≡2↑RESULT
→LOOP IF RESULT[4]>↑ρVALUE
VALUE←(VALUE≠' ')/VALUE←VALUE[RESULT[4];]
→LOOP IF~EXIST VALUE
NAME←VALUE
→LOOP
END:MESSAGE MW CLOSE 'NR'
MESSAGE NW CLOSE 'NR'
MESSAGE MW1 CLOSE 'NR'
MESSAGE TW CLOSE 'NR'
MESSAGE BG CLOSE
FILE_RELEASE 'VIEW.OOP'
```

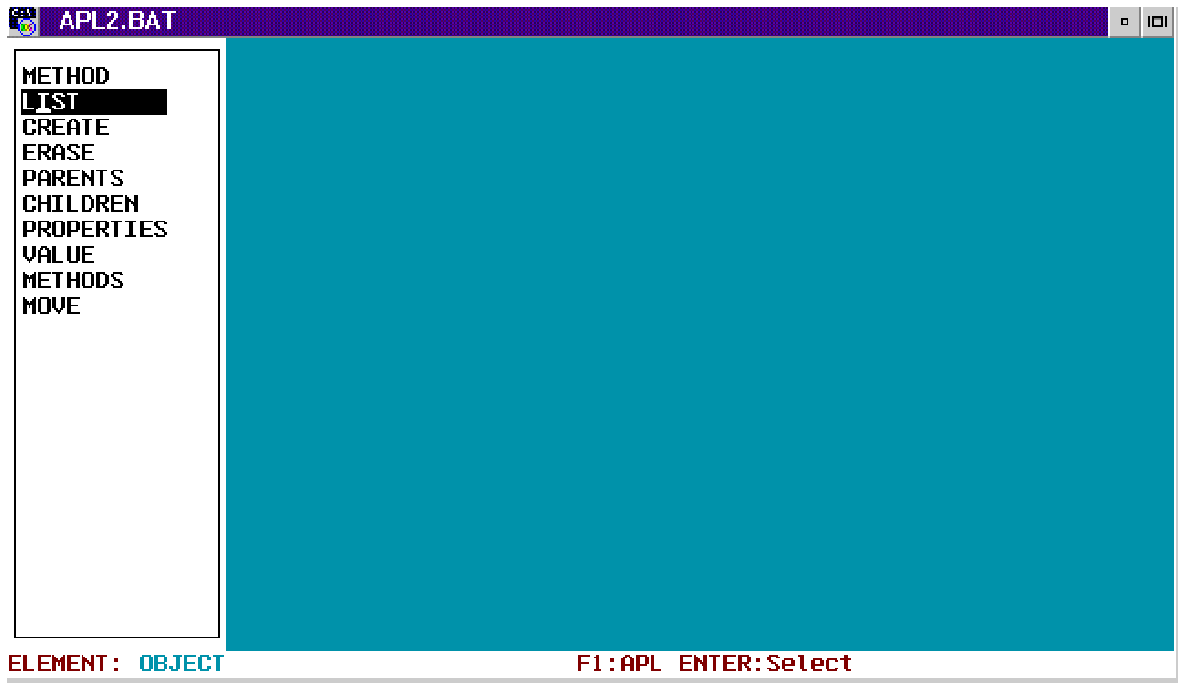

Figure 2

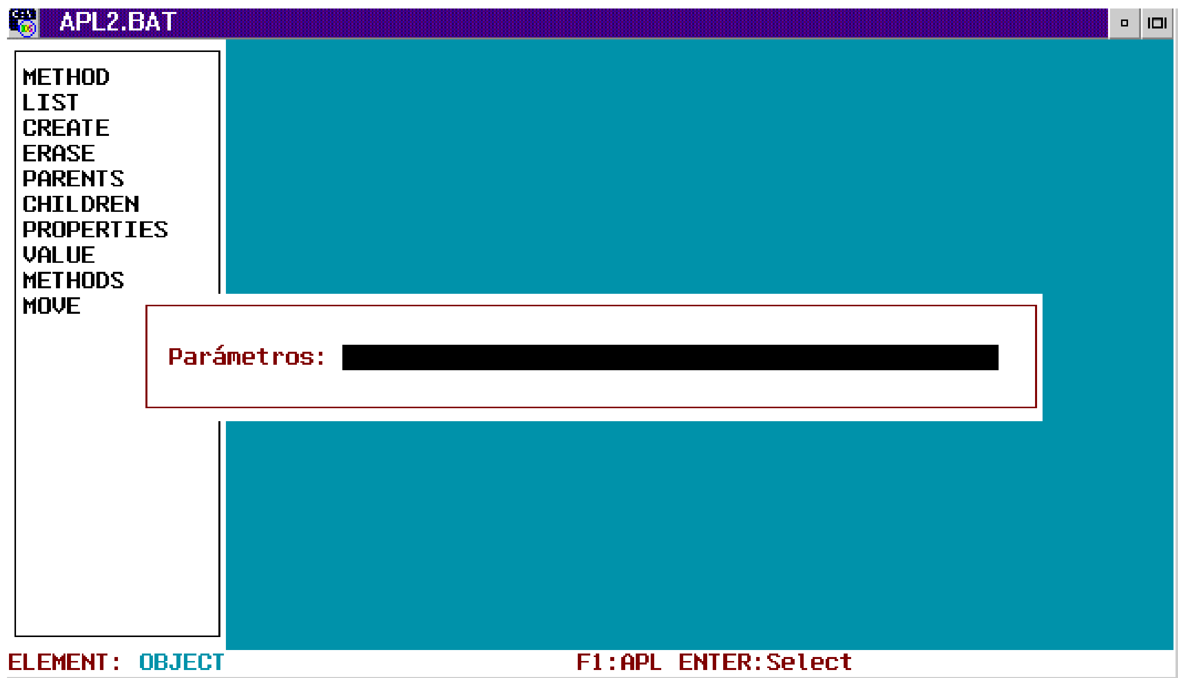

Figure 3

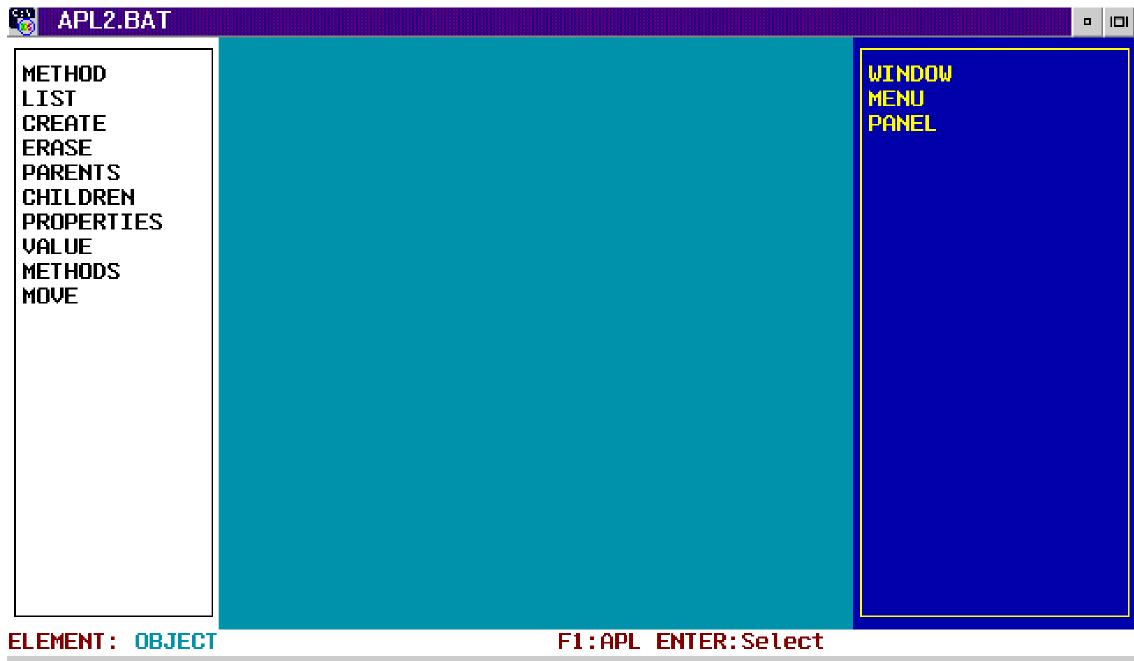

Figure 4

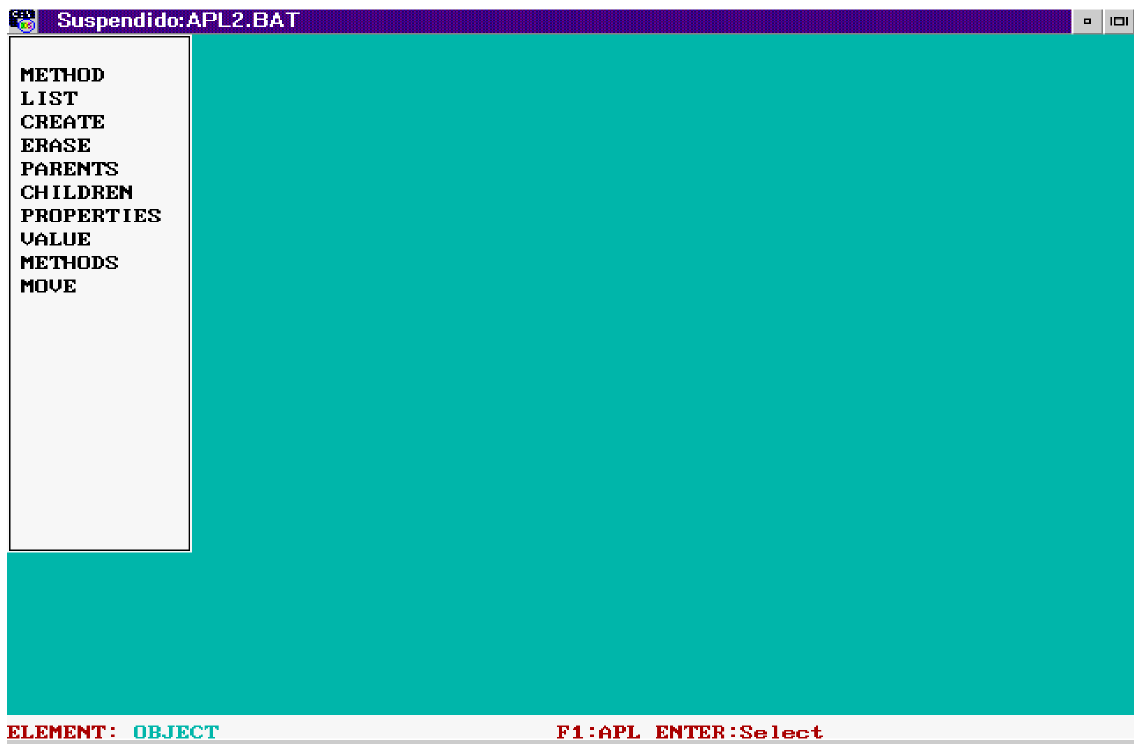

Figure 5

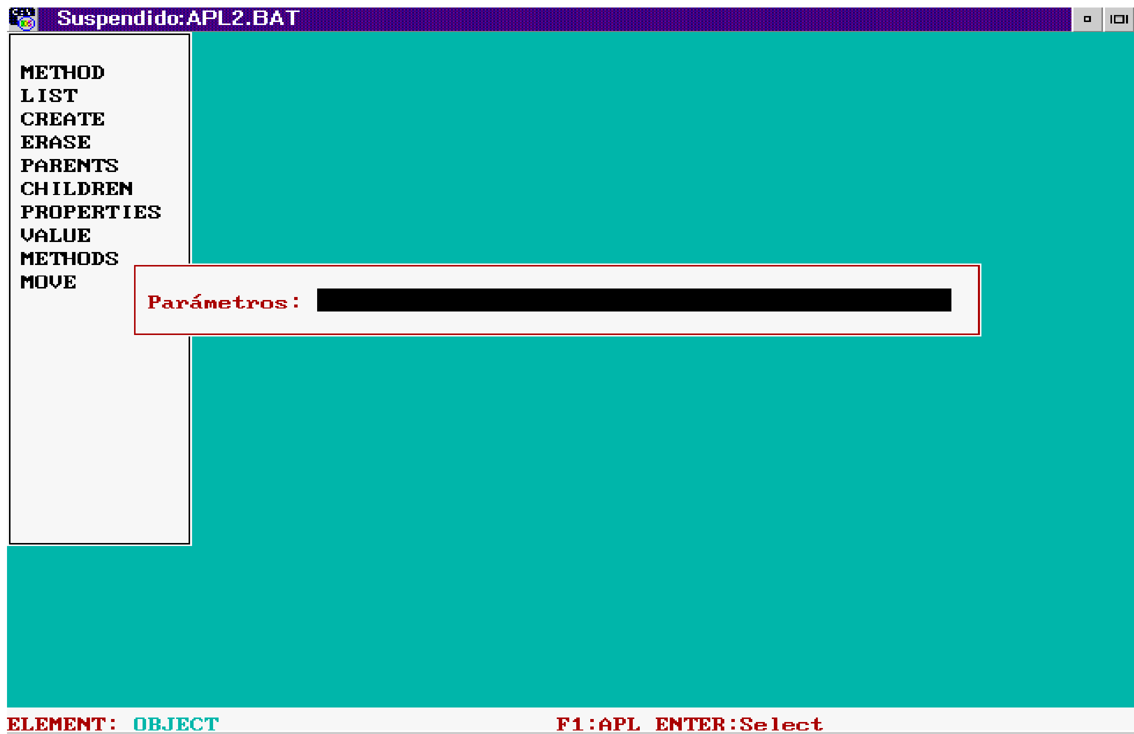

Figure 6

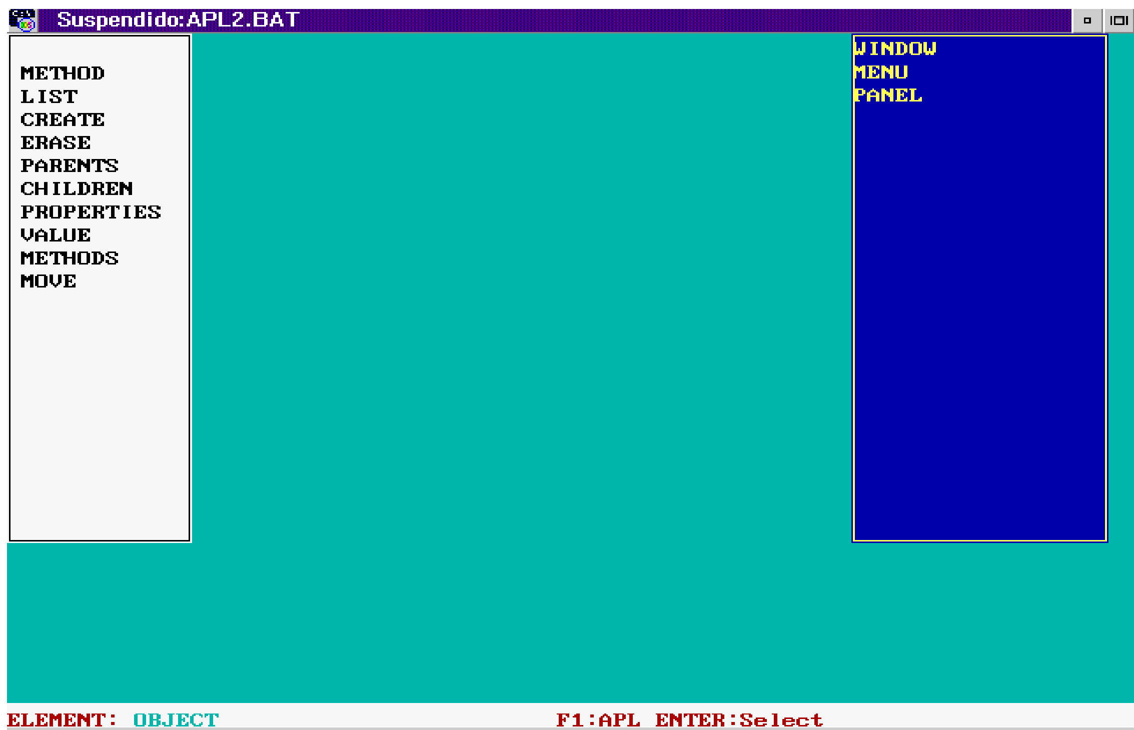

Figure 7

APL2.BAT

GRAFICOS | BUSCAR | AYUDA | SALIR

Sistema Periódico de los Elementos

Azul: líquido
Rojo: gas
Rosa: artificial

IA	IIA	IIIB	IVB	VB	VIB	VIIB	--VIIIB--			IB	IIB	IIIA	IVA	VA	VIA	VIIA	VIIIA
H 1																	He 2
Li 3	Be 4											B 5	C 6	N 7	O 8	F 9	Ne 10
Na 11	Mg 12											Al 13	Si 14	P 15	S 16	Cl 17	Ar 18
K 19	Ca 20	Sc 21	Ti 22	V 23	Cr 24	Mn 25	Fe 26	Co 27	Ni 28	Cu 29	Zn 30	Ga 31	Ge 32	As 33	Se 34	Br 35	Kr 36
Rb 37	Sr 38	Y 39	Zr 40	Nb 41	Mo 42	Tc 43	Ru 44	Rh 45	Pd 46	Ag 47	Cd 48	In 49	Sn 50	Sb 51	Te 52	I 53	Xe 54
Cs 55	Ba 56	La 57	Hf 72	Ta 73	W 74	Re 75	Os 76	Ir 77	Pt 78	Au 79	Hg 80	Tl 81	Pb 82	Bi 83	Po 84	At 85	Rn 86
Fr 87	Ra 88	Ac 89	Db 104	Jl 105	Rf 106	Bh 107	Hn 108	Mt 109									

Ce 58	Pr 59	Nd 60	Pm 61	Sm 62	Eu 63	Gd 64	Tb 65	Dy 66	Ho 67	Er 68	Tm 69	Yb 70	Lu 71
Th 90	Pa 91	U 92	Np 93	Pu 94	Am 95	Cm 96	Bk 97	Cf 98	Es 99	Fm 100	Md 101	No 102	Lw 103

Elemento:Molibdeno
Valencias: 2 3 4 5 6
P.Fusión: 2610
P.Ebullición: 5560
Densidad: 10.2
Masa Atómica: 95.94

Estruc. electrónica: 1s2 2s2 2p6 3s2 3p6 3d10 4s2 4p6 4d5 5s1

Figure 8