INTEGRATED INTERACTIVE COMPUTING SYSTEMS
P. Degano and E. Sandewall, editors
North-Holland Publishing Company / © ECICS, 1983

TOWARDS A TWO-DIMENSIONAL PROGRAMMING ENVIRONMENT

Bertrand Meyer
Electricité de France, Direction des Etudes et Recherches
Service Informatique et Mathématiques Appliquée
1 avenue du Général de Gaulle 92141 Clamart
France

ABSTRACT

The use of modern video display terminals for communication with a computer has a profound effect on the nature of the resulting dialogs. Screen-oriented interactive programs require a new set of tools, techniques and methods. We report on studies on these topics performed in a computing environment based on standard commercial hardware. The paper describes some of the tools which we have used and the ones we have designed ; it then discusses the methodological issues involved in designing two-dimensional dialogs, and shows the kind of program modularity which is required in this framework. Object-oriented programming appears to provide the right basis ; we have applied this methodology using the class concept of the Simula 67 language and the associated prefixing mechanism.

1 - INTRODUCTION

Interactive facilities play an ever increasing part in all the application areas of computers. Today, this evolution does not only imply that the traditional "batch" mode of submitting programs to computers yields more and more to conversational execution ; it also impacts the very form of such executions : whereas dialogs on typewriter-like terminals and the first CRT devices would proceed in a "line by line" fashion, current terminal technology makes it possible to use the full contents of a screen as the basic unit of communication with the computer, giving rise to the so-called "full-screen" or "full-page" mode of interaction.

One of the best-known applications of this technique is the preparation of documents on a computer using one of the "full-screen editors" now available on many computer systems, most notably mainframes and word-processing systems. Users of such tools unanimously appreciate their power and ease of use, to the extent that going back to a line-oriented editor is resented as a painful experience. Full-screen facilities also find applications in many other domains ; examples are software development and maintenance aids, application programs designed to be used by non-specialist users under the guidance of successive "menus", business data processing (where many "transactional systems" are being developed) and Computer-Aided Instruction. In these and many other areas, programmers in ever growing numbers would like to be able to provide full-screen dialogs for the execution of their own programs.

The construction of such dialogs implies that the texts to be exchanged between the programs and their users are two-dimensional ; this requirement adds a new set of difficulties to the general problems of conversational programming, which are themselves far from being fully mastered (in particular as regards the human engineering, or ergonomic, aspect of dialogs). This paper studies some of these problems, and describes some of the solutions which have been implemented at the Direction des Etudes et Recherches of Electricité de France (EDF), laying the basis for what may be called a two-dimensional programming environment. The discussion focuses on three of the basic issues of software engineering, as applied to two-dimensional interactive programming : tools, methods and languages. The ergonomics of dialog systems, which is another important topic, is touched upon only briefly.

In some respects, it may be felt that the discussion below lags behind the current "frontier" technology in hardware and software. In particular, we limit ourselves to the manipulation of text objects, even though considerable experience has been gained in recent years in two neighbouring domains, namely graphics systems and Computer-Aided Design, where more complicated visual objects are processed. It is clear, on the other hand, that some research laboratories have developed two-dimensional environments which are more sophisticated than the one described here ; two examples worth noting are the set of tools built around LISP /14, 15/ and the Xerox PARC SMALLTALK system /3/, which utilizes special-purpose terminals and a dedicated operating system.

On the other hand, the tools which are described in this paper do not appear to be so commonly available in the most widely used environments, whether in industry or universities ; neither do the underlying ideas. It is quite interesting in this respect to study two recent papers in the Communications of the ACM on the subject of interactive programming /4, 10/ ; although quite different from one another, they both discuss how successive questions should be asked from users, how mnemonics and keywords should be designed, how errors should be dealt with, etc. ; both implicitly assume that the dialog considered proceeds in a completely sequential, line-by-line fashion, without even considering that there may exist other cases. Much of the discussion in these papers becomes pointless when one goes to a two-dimensional environment.

Furthermore, an important characteristics of the tools described below should be emphasized, namely the fact that they were developed and are being used in a standard "production" environment rather than in a computer science laboratory. The computing center at the Direction des Etudes et Recherches of EDF is based on IBM hardware (3081, 3033, 370-168, 4341, etc.) under the MVS-SP operating system. The time-sharing system is TSO ; full-screen terminals are of the IBM 3270 or compatible series ; most of them are 3278, 3279-2B and 3279-3B models (the latter having seven colors, semi-graphic possibilities and various other options). Most application programs are written in Fortran. This environment (which also includes a Cray-1 and many other computers) is quite representative of many large classical computing centers.

2 - THE CHARACTERISTICS OF TWO-DIMENSIONAL DIALOGS

The usefulness of two-dimensional dialogs stems from the combination of three properties :

- The second dimension as such, which provides the program user with an overview of a full page of text, rather than just a single line ;
- The use of a page as unit of communication with the computer, which allows the user to design first an overall sketch and then look back on his decisions, correct errors, reverse some choices, before he sends a page of information to the system ;
- The default facility, which makes it possible for the program to fill some zones where user response is expected by predetermined values, so that the user will only have to write the answers if they are different from these values, but not if the questions are unneeded in his particular case, or call for the same answer as in the previous use of the system (one of the criticisms heard most frequently from users of non-page-oriented interactive programs is that one must answer a whole bunch of seemingly useless questions every time one starts using the system).

It should be noted here that a good page-mode interactive program should keep a profile of every user, so that the default answer suggested for each question will be the one chosen by the user during the last execution of the program, rather than a fixed value assumed to suit all users.

Below is an example of a full-page dialog. It is extracted from the FORTRAN command procedure in our AL library (see section 3) and shows the first three screens to be filled when running a Fortran program : the user types in the names of the files containing source and object code, the destination of printouts, the compiling options, the libraries used, etc. It is easy to imagine how many successive questions would have to be answered in an equivalent line-by-line dialog ; most answers would be indentical from one use of the procedure to the next. If full-screen is not available, the designer of such a dialog constantly faces the contradictory demands of two categories of users : the sophisticated ones, who would like to use many advanced features and thus request many options, i.e. many questions ; and the more numerous "vulgar" users, who use standard options and want short dialog sessions.

Worth noting is the presence of an option called "same as last time" which allows the user, from then on, to remain entirely silent, and directs the system not to ask any more questions. This option is particularly useful in a repetitive task such as the test of a given module.

```
--------------------------------------------------------------------------------


      HELLO BERTRAND
      WELCOME TO THE AL FORTRAN EXECUTION SYSTEM


      PLEASE CHECK THE APPROPRIATE BOX :

           SAME AS LAST TIME                           ===) / /

           COMPILATION, LINK-EDIT, EXECUTION           ===) / /

           LINK-EDIT, EXECUTION                        ===) / /

           EXECUTION                                   ===) / /


--------------------------------------------------------------------------------


                         COMPILATION, FORTRAN IV EXTENDED

NAME OF THE FILE CONTAINING SOURCE CODE                ===) tryit.fort(first)

COMPILATION LISTING DESTINATION                        ===) prt
      (TER, PRT, LOC, DMY, SYS=x or file name)

CLASS (only if SYS=C, R, S or U)                       ===)

NAME OF THE FILE FOR OBJECT CODE                       ===) tryit.obj(first)

COMPILER OPTIONS :
                            OPTIMIZATION LEVEL         ===) 2
                            GENERATED CODE LIST        ===) no
```

```
--------------------------------------------------------------------------------

                         --- COMPILATION WAS OK ---

                                LINK-EDIT

NAME OF THE FILE CONTAINING OBJECT CODE ===) tryit.obj(first)

LIBRARIES TO BE INCLUDED
      You may request a library by giving either :
           - a keyword (FORTLIB, GENERALE, IMSL, LINPACK, BENSON, ATELBIB...)
           - the actual name of a file containing the library in load module
             form.

      ===) fortlib

      ===) 'edf.myownlib.load'

      ===) 'edf.peterslib.load'

      ===)

      ===)

      ===)

      ===)

      ===) generale
--------------------------------------------------------------------------------
```

It may be said without overstating the argument that, for the programmer who writes systems having this kind of interaction with their users, the leap from traditional, line-by-line conversational programs to page-oriented ones is as big as the leap from non-interactive "batch" programming to line-oriented interactive programming. The new discipline may (perhaps emphatically) be called "two-dimensional programming" ; the second, vertical dimension introduced by screen dialogs raises many important issues with respect to the methods, techniques and tools of interactivity.

3 - COMMAND PROCEDURES : THE DIALOG MANAGER AND THE AL LIBRARY

The first tool which is available to our users is one which is distributed by the manufacturer. IBM has recently released /8/ a new version of SPF (System Productivity Facility, previously known as Structured Programming Facility), a subsystem of TSO, the basic interactive system under MVS. The main characteristics of SPF, which make it rather nice to use for such functions as text editing or file management, are the following :

- the use of two-dimensional dialogs ;
- the presence of "user profiles" which keep useful information from one interactive session to the next ;
- a particular technique for error processing.

The main improvement brought about by the new version of SPF is the set of functions called the "Dialog Manager" /9/. Thanks to this facility, any programmer writing command procedures in the command language of TSO may use some of the internal tools and techniques of SPF, thus being able to take full advantage of the three properties mentioned above.

The dialog manager may be called through special functions which have been added to the TSO command language. It is not, however, easy to use for novice or occasional users ; neither is it readily interfaced with application programs (in particular those written in Fortran). Its main use in our environment so far has been the implementation of a general-purpose command procedure library, called AL (Atelier Logiciel).

The AL library currently contains some forty procedures which encompass a wide spectrum of tools : access to compilers of the various available languages (Fortran, Cobol, assembly, Algol W, Pascal, Simula 67, Reduce), file manipulation and management, use of specialized programs, access to on-line documentation, etc. Until recently, all were line-oriented conversational procedures, suffering from the drawbacks mentioned above. It is interesting to note that our desire to keep the dialogs simple, and thence to limit the number of available options, had resulted in the proliferation of "customized" versions of the more popular procedures : programmers would copy and modify them, thus hampering our efforts to maintain and improve them.

With the development of two-dimensional versions, these problems have disappeared : we may now afford to include many options, since the user's choices are remembered from one session to the next and he will usually change few of them each time ; no more tedious recoding of the same values is required. During the first use of a procedure, default standard values are pre-filled by the system.

Currently available two-dimensional procedures in AL include Fortran IV (of which the dialog in section 2 was an example), Fortran VS (offering access to the IBM version of Fortran 77), Simula 67, Pascal, Algol W, Cobol, Apothèce (a system for the management of program libraries). The entire library will be progressively adapted.

4 - TOOLS FOR TWO-DIMENSIONAL APPLICATION PROGRAMMING : GESCRAN

Once one has discovered the delights of two-dimensional interactivity, perhaps through the use of SPF and AL, one is often tempted to apply the same techniques to one's own application programs. One available IBM product makes this possible : GDDM (Graphical Data Display Manager /7/), a very powerful tool which also includes semi-graphic facilities. GDDM is also, however, rather complex and heavy, and closely tied to IBM hardware and systems. We thus felt it necessary to design a product which, albeit much less ambitious, would cater for simple uses while remaining rigorous in its definition and more portable.

The result of this effort is a package called Gescran (for "Gestion d'écrans", screen management) /1/. Gescran is a set of Fortran subroutines, designed according to the methodological principles expounded in /13/ ; it allows the programmer to describe and manipulate objects called "screens", to create rectangular "windows" in these screens, to define and change the attributes of these windows (such as associated text, color, brilliance, protection, etc.), and to visualize all or part of a screen on the available terminal. It is important to note that screens and windows are in no way bound to the display hardware : they are purely <u>abstract objects</u>, known to the program solely through a name, which in Fortran is implemented as an integer variable, used internally to contain an address and control flags ; the only operation which may be applied to such a variable is its use as an actual argument in a call to one of the Gescran subroutines. Association with a physical screen occurs only when a visualisation subroutine is called.

Gescran works on the IBM 3270 series of screen terminals, but was designed so as to be adaptable to any terminals offering similar capabilities. The construction and manipulation of the data structures representing screens and their windows are entirely independent from the physical I/O operations.

Among the current developments, we shall mention a study aimed at interfacing Gescran with a graphics package, so that the programmer will have the possibility of describing a Gescran window as graphical and use the graphics package rather than Gescran to manipulate this particular window, provided of course the terminal used provides the corresponding facilities.

<u>5 - COMPUTER-AIDED SCREEN DESIGN : CONSCRAN</u>

An important tool for the efficient use of Gescran, called Conscran, provides a higher-level interface for the design of screens as defined above.

The requirement for Conscran stemmed from a problem which had been met by all Gescran users : before being able to write the sequence of subprogram calls which describes a set of screens and windows, one must <u>design</u> each screen by defining the position of its various windows, the parts they play in the interaction, their contents, color, protection, special features (e.g. blinking, reverse video), etc. Until Conscran became available, the best available technique for this phase was to use a sheet of paper and draw a picture of the screen. Such a medium and method appear rather primitive when compared with the aim pursued.

Conscran relies explicitly on concepts taken from Computer-Aided Design to improve the screen design process. It allows the programmer to perform such design in a two-dimensional interactive fashion : the screens will be "drawn" at the terminal, with all the resulting flexibility ; various designs may be tried, observed, modified. Conscran automatically generates the Fortran subroutine containing the calls to Gescran subroutines which are necessary for the construction of the corresponding screens, thus freeing the programmer from a rather tedious task. Conscran stores the resulting screen designs in a data base, thus allowing for later retrieval and modification. It also generates a paper "map" of the screen, showing the position of the various windows, and a "legend" giving their attributes.

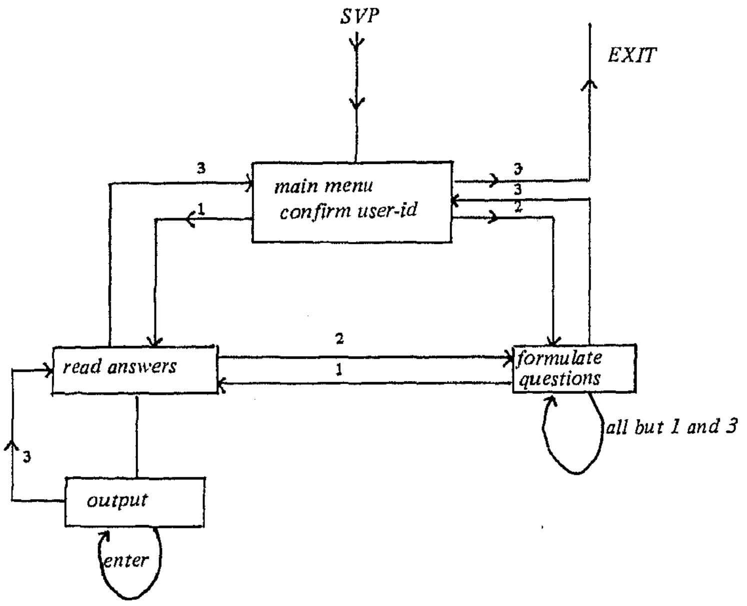
Our current efforts go towards extending Conscran to a system allowing for the design not only of individual screens, but of entire applications as well, using the same underlying principles.

Conscran itself is a two-dimensional interactive program, written in Gescran. Its aim is what may be called "Computer-Aided Screen Design".

6 - THE STRUCTURE OF DIALOG PROGRAMS

Even with the world's best tools, two-dimensional programming raises several difficult issues. One of the most delicate ones is the structure of dialog programs. The behaviour of such programs may usually be quite faithfully modeled by a state transition diagram : one execution of the program will correspond to a path in the associated graph.;

Below is an example of such a graph ; this is one of the applications which we have written with Gescran, the SVP system /5/, which allows users to ask (non-urgent) questions and get answers from the programming assistance service on their terminal. Only the "user" part is shown.



Except for its small size, this example is quite representative of the structure of page-oriented, menu-driven interactive programs. At every step in the execution, associated with one of the states in the diagram, the program outputs a screen ; certain zones are then filled by the user ; after having checked the validity of the answers, the program will perform some action (usually reading or updating a data base). The next step depends on the user's choice, often expressed by his pressing some function key on the terminal. The labels of the edges in the graph correspond to these possible choices.

In a straightforward realization of this scheme, the program for an interactive, menu-driven application will consist of a number of "paragraphs", one per state, each looking somewhat like the following :

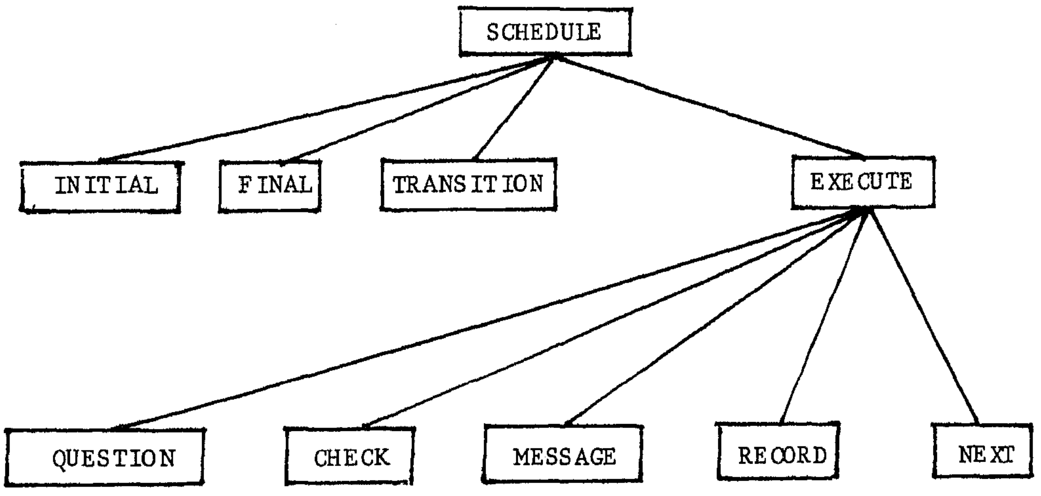
```
state x :
     output screen for state x ;
     repeat
          read user's answers and his choice c for the next step ;
          if error in answer then
               output message
     until no error in answer ;
     record answer ;

     case c in
          $c_1$ : proceed to state $x_1$,
          $c_2$ : proceed to state $x_2$,
          ..........................,
          $c_n$ : proceed to state $x_n$
```

Using such a scheme for the actual programming will result in programs with an intricate branching structure, belonging to the well-known "bowl of spaghettis" type. It has been argued /2/ that such a structure should be avoided in the first place, by applying to the state graphs of menu-driven systems such restricting rules as are imposed by modern programming methodology upon the control structures of programs. We think that the analogy is wrong : designing the internal structure of an engineering product such as a program is really not at all the same as designing the external structure of a process involving humans, such as the dialog with a machine. In our opinion, the structural intricacy of the state graph of many interactive systems is an inherent property of these systems, and artificial "structuring" rules are pointless in this domain. The complexity of the graph may stem from various reasons : there may be temporary detours (corresponding e.g. to "help" keys), shortcuts (which were introduced at some point because a user requested, quite legitimately, the possibility to go directly from a certain state to another one, whereas he previously had to backtrack first to the initial menu), and multi-level exits (corresponding to "escape" keys or "quit" commands). Note that these requirements will defeat any effort to implement menu-driven systems by straightforward application of "structured programming" in its naive form.

Some authors have introduced special-purpose control structures to solve this problem ; one example is the language PLAIN /16/, which uses "exceptions" as in Ada, CLU or PL/I. The use of such constructs seems only marginally preferable to that of ordinary jumps.

A much better solution, as it seems to us, is to completely disconnect the description of the overall structure of the dialog, i.e. the traversal of the graph, from the description of what happens at every step, i.e. the operations performed while in a given state. The latter may be treated with ordinary programming constructs ; for the former, the finite automaton, as used in compilation or real-time applications, is a helpful model. It will be quite useful (although not compulsory) to implement the systems in a table-driven fashion, i.e. represent the state transition diagram by a data structure (usually an array) rather than a function subprogram ; using this technique, the changes in the scheduling of states, which are quite common as projects evolve and users request new facilities, will be easy to accomodate.

More precisely, we shall make use of ten program units on three hierarchical levels :



SCHEDULE only defines the traversal of the transition graph ; it knows nothing about the particular screens of a given application, and should be identical for all applications :

```
SCHEDULE :
    var current : STATE, label : CHOICE ;
    current := INITIAL ;
    repeat
        EXECUTE (current, label) ;
        current := TRANSITION (current, label)
    until FINAL (current)
```

TRANSITION is the function which describes the state diagram : TRANSITION (s, l) is the new state reached when leaving state s by the branch labeled l. As mentioned above, TRANSITION may be represented either by a function subprogram or by a two-dimensional array, the latter leading to a more easily adaptable program.

EXECUTE does what is required in a given state : ask the right question, check the answer, perform the necessary action and return the choice c for the next step :

```
EXECUTE (in s : STATE ; out c : CHOICE) :
    var c : CHOICE, a : ANSWER ;
    repeat
        a := QUESTION (s) ;
        correct := CHECK (a, s) ;
        if not correct then
            MESSAGE (a, s)
    until correct ;
    RECORD (a, s) ;
    c := NEXT (a, s)
```

QUESTION, CHECK, MESSAGE, RECORD and NEXT, on the other hand, are application-specific. The call QUESTION (s) will output the screen associated with state s and read the user's answers :

```
QUESTION (in s : STATE) :
    output the screen for state s ;
    read and return the answer a
```

CHECK (a, s) will return true or false depending on whether answer a is acceptable or not in state s ; MESSAGE (a, s) outputs the error message corresponding to answer a in state s, where CHECK (a, s) is false ; RECORD (a, s) records answer a in state s, where CHECK (a, s) is true ; NEXT (a, s) determines from the user's answer a the exit label which was chosen for leaving state s.

It is natural to look for tools which may help in the construction of interactive systems described in the above framework. Some of the "author languages" in Computer-Aided Instruction (CDC's Plato or IBM's IMG for example) pursue similar goals. Can one use the above scheme to build general-purpose tools for helping in the design of interactive, full-screen applications ? As mentioned before, this is our aim in the current extensions to Conscran.

It is soon realized that this scheme cannot reasonably be implemented as presented above if what is sought is a modular, easily extendible system. A simple remark should convince the reader of this impossibility : if procedures such as RECORD, CHECK, MESSAGE, QUESTION or NEXT were to be put in a library, so as to be re-usable for various applications, then a closer look at the above design shows that these procedures must include among their parameters the state (s), but also the precise interactive application to which this state belongs. In other words, any such general-purpose should know about and discriminate amongst all states of all available applications using them ! This is clearly incompatible with any attempt at modularity.

As it is often the case which such problems, a proper solution may be found by going from procedure-oriented to object-oriented programming, i.e. by basing the structure of the program on the main data structures rather than on the functions to be performed. This is the direction that we have taken ; we have been greatly helped in this effort by the availability in our computing center of one of the few generally available modular, object-oriented languages : Simula 67.

7 - USING A MODULAR , OBJECT-ORIENTED LANGUAGE : SIMULA

Simula 67 /6/ appears particularly well-suited for the practical application of the methodological principles introduced above. The main concepts are those which have been emphasized in /12/ : abstract data types, top-down program and data structure design, genericity. Similar techniques could be applied to a descendant of Simula, Smalltalk /3/.

We will only outline part of the system design. In order to implement the above scheme, it is particularly useful to be able to use a structure corresponding to the abstract notion of a "state". The following characteristics are associated with every state s :

- attributes of s : state number, screen to be output when s is reached ;
- operations which may be requested when the system is in state s : QUESTION, CHECK, MESSAGE, RECORD ;
- actions to be performed when s is reached : EXECUTE.

Such characteristics correspond closely to what may be included in the basic program structure of Simula, the class, which is the implementation of an abstract data type : variables representing the attributes of each state, procedures (subprograms) representing the admissible operations, and statements representing the initial actions. One is thus quite naturally led to the design of a class STATE.

A fundamental property of Simula which will be used here is known as class prefixing : a class may be used as "parent" of other classes, which will inherit its characteristics, to which they will add their own refinements. Procedures may be specified at the level of the parent class, their realizations being given in the descendants ; usually these will not be the same in every descendant. Such procedures are declared as virtual in the parent class. Class prefixing and virtual procedures together form one of the best-known systems for the authentic top-down design of both program and data structures. Here they will allow us to define the class STATE with the following structure :

```
class STATE ;
     comment operations :     ;
          virtual :
               ref (answer) procedure QUESTION ;
               boolean procedure CHECK ;
               procedure MESSAGE ;
               procedure RECORD ;
               ref (choice) procedure NEXT ;
          begin
          procedure EXECUTE (c) ; ref (choice) c ;
               begin    boolean correct ;
               correct := false ;
               while not correct do
                    begin     ref (answer) a ;
                    a := QUESTION ;
                    correct := CHECK (a) ;
                    if not correct then
                         MESSAGE (a)
                    end validation ;
               RECORD (a) ;
               c := NEXT (a)
               end EXECUTE ;
     comment attributes :     ;
               integer screen ;  comment Recall   that   Gescran   uses
                                         integers to denote screens ;
          end STATE
```

Class STATE defines the general properties of a screen. Procedure EXECUTE has now become part of this class ; the same is true for procedures QUESTION, MESSAGE, CHECK, RECORD and NEXT. Note that all these procedures have lost their "STATE" parameter (s in the procedure-oriented version). There is an important difference between EXECUTE and the other five : at the level of class STATE, the latter, while needed, cannot be refined, since their precise implementation may only be known for a given STATE. They are thus defined at the STATE level as "virtual", i.e. only the procedure headings (partial specification) is given. In contrast, procedure EXECUTE is the same for all STATEs ; thus both its heading and body (which uses calls to the five virtuals) may be given at the level of class STATE.

For any given application, there will be a certain number of instances of class STATE, corresponding to the various states of the application. This instantiation concept is readily implemented by the prefixing mechanism :

```
STATE class INITIAL_MENU ; begin  ...  end ;
STATE class COMPILATION_OPTIONS ; begin  ...  end ;
etc.
```

The body of each of these subclasses will include the corresponding body for the procedures QUESTION, CHECK, MESSAGE, RECORD and NEXT.

One of the main benefits of this method is that it allows a truly modular construction of interactive applications, the general-purpose and application-dependent parts being programmed separately. All problems pertaining to a certain state (formulation of the question, treatment of errors, recording of answers, etc.) are dealt with in the module (class) for that state, and there only ; on the other hand, the module for a state does not know anything about its connections with the rest of the application's graph. Thus it becomes possible to add or change states, transitions between states etc. without disturbing anything in any module other than the ones associated with the states directly involved in the modification. Apart from its elegance, such a modular, object-oriented programming yields software products on which modifications and extensions are much easier to perform than with programs structured in a more conventional, procedure-oriented fashion.

8 - CONCLUSION

We hope to have shown that the two-dimensional aspect of screen dialogs has important effects on the structure and use of interactive systems. We hope that the ambitious ongoing developments in the area of integrated software environments will take into consideration the key issues which arise in the design of systems for successful communication between man and machine.

BIBLIOGRAPHY

/1/ E. Audin, G. Brisson, B. Meyer : Gescran ; EDF Report, Atelier Logiciel n° 22, December 1980 (version 4, December 1981).

/2/ J.W. Brown : Controlling the Complexity of Menu Networks ; Communications of the ACM, 25, 7, pp. 412-418, July 1982.

/3/ BYTE Magazine : Special issue on SMALLTALK, August 1981.

/4/ B. Dwyer : A User-Friendly Algorithm ; Communications of the ACM, 24, 9, pp. 556-561, September 1981.

/5/ E. de Drouas : Manuel d'Utilisation de SVP ; EDF Report, Atelier Logiciel n° 32, October 1981.

/6/ O. J. Dahl and K. Nygaard : Simula 67 Common Base Language ; Norsk Regnesentral (Norwegian Computing Center), Oslo, 1970.

/7/ IBM : Graphical Data Display Manager - Release 2 ; order no. SC33-0101-1, October 1981.

/8/ IBM : System Productivity Facility for MVS - Program Reference ; order no. SC34-2038-0, December 1980.

/9/ IBM : System Productivity Facility for MVS - Dialog Management Services ; order no. SC34-2036-1, March 1981.

/10/ H. Ledgard, J.A. Whiteside, A. Singer, W. Seymour : The Natural Language of Interactive Systems ; Communications of the ACM, 23, 10, pp. 556-563, October 1980.

/11/ B. Logez, M.-P. Nardy : Conscran, manuel d'utilisation ; EDF report, Atelier logiciel n° 38, 1982.

/12/ B. Meyer : Quelques Concepts des Langages de Programmation modernes, et leur Application à SIMULA 67 ; Bulletin AFCET-GROPLAN n° 9, 1979.

/13/ B. Meyer : Principles of Package Design ; Communications of the ACM, 25, 7, pp. 419-428, July 1982.

/14/ E. Sandewall : Programming in the Interactive Environment : The LISP Experience, ACM Comp. Surv., 10, 1, March 1978, pp. 35-72).

/15/ W. Teitelman : A Display Oriented Programmer's Assistant, in Proc. 5th Int. Jt. Conf. on Artificial Intelligence, Dpt. Comp. Sc., Carnegie-Mellon Univ., Pittsburgh, 1977, pp. 905-915

/16/ T. Wasserman : PLAIN : An Algorithmic Language for Interactive Information Systems ; in Algorithmic Languages, de Bakker and van Vliet (Eds.), North-Holland, 1981, pp. 29-47.