

# Design and Implementation of the Visual DaVinci Language

Raúl Champredonde<sup>1</sup>

Armando De Giusti<sup>2</sup>

Laboratory of Investigation and Development on Computer Sciences<sup>3</sup>

Department of Computer Sciences

Faculty of Exact Sciences

National University of La Plata.

## Abstract

A visual language, called Visual DaVinci is presented in this article, along with its implementation most important aspects.

Visual DaVinci was specially developed for structured programming initial teaching at computer sciences courses of studies.

It uses the control flow paradigm by specifying diagrams similar to the Nassi-Schneiderman one, thus forcing the development of a structured code.

Diagrams are automatically derived to textual code, which can be modified by the programmer.

It also allows a textual development of the code, with a restricted syntax that also requires the generation of a structured code and the keeping of certain rules regarding programming style.

Syntactic verification and execution are based on the code, in order to free language efficiency from its visual nature.

**Key words :** Visual languages - Objects - Compilation and interpretation

---

<sup>1</sup> Semi Full-time Practical Classes Coordinator. LIDI. UNLP  
e-mail : rchampre@info.unlp.edu.ar

<sup>2</sup> Full-time Chair Professor. Director of the LIDI. UNLP  
Principal Researcher of the CONICET  
e-mail : degiusti@info.unlp.edu.ar

<sup>3</sup> LIDI Laboratorio de Investigación y Desarrollo en Informática  
Departamento de Informática. Facultad de Ciencias Exactas. Universidad Nacional de La Plata  
Calle 50 y 115, Primer Piso, 1900 La Plata  
Tel/Fax : 54-21-22-7707  
e-mail : lidi@info.unlp.edu.ar

## Introduction

Some years ago, a group of researchers at the LIDI developed the specifications for an abstract machine (Lubo-I robot) and an associate programming language, which were intended for the teaching of structured programming initial concepts[DeGiusti88] [DeGiusti89].

These specifications are since then being used at the entrance course and at the first year subject Computers Programming of the Course of Studies on Computer Sciences at the National University of La Plata.

The robot can perform a series of actions over the city, which are determined by the execution of a program. The city is one hundred streets by one hundred avenues. Streets are horizontally oriented, while avenues are vertically oriented.

Corners are determined by the intersection of streets and avenues. Each corner is identified by an ordinate pair (Av, St), where Av and St are integers between 1 and 100.

At the beginning, the robot is placed at the (1,1) corner and facing North. Robot orientation can be modified so that it faces one of the four cardinal points.

Each movement carries the robot from one corner to the following one, according to the direction in which it is oriented.

At each corner there may be an arbitrary amount of flowers and papers, which can be collected by the robot. In order to do this the robot carries two bags, one for flowers and the other for papers. Each time it collects a flower, it puts it in its flower-bag; and each time it puts a flower on a corner, it takes it from the flower-bag. The same procedure is repeated with papers and the paper-bag. The robot can put flowers or papers at the corner where it is standing only if the corresponding bag is not empty.

There may also be obstacles preventing the robot to pass through a certain corner; in this case the robot will have to find an alternative way, that is, it will have to modify its normal trajectory. Obstacles may be either simple or barriers. An obstacle is simple if there is no other obstacle at the neighboring corners. A set of obstacles form a barrier if they are placed over a straight line at consecutive corners.

The language used for the robot programming is structured and has the minimal constructions needed for learning this programming paradigm.

Every program written with this language consists of three parts: header, statements, and body. The header simply determines the name of the program. The second part is used for variables and subprograms statements. A variable statement determines its name and type. Subprograms have the same general structure than the program except for the formal parameters. The parameters passage model chosen is Ada [Olsen83] [DoD], since it is the most representative one of those used by most conventional languages.

The body of the main program, the same as the subprograms, is a statements sequence. These statements may be primitive, simple, or composite. Primitive statements are those directly executable by the robot. That is, those telling it to walk from one corner to another, to turn right, to pick or leave a flower or a paper, etc. Simple statements are subprograms allocations and modifications defined by the user or by the system. Composite statements are selection control structures (if / if not), conditional iteration (while), and unconditional iteration (repeat).

Expressions are formed from values, user or system-defined variables, input parameters or input/output parameters, connected with relational, arithmetic or logical operators.

A more detailed description of robot specifications and language syntax and semantics can be found in [DeGiusti88] [DeGiusti89].

Due to several reasons related to the learning process and to the experimental research, the already mentioned specifications were converted into a visual language based on the flow control paradigm [Glinert90a], so that the robot can be programmed highlighting program structure over implementation language syntax. This technology is thus simultaneously presented as a development tool [Chang86] [Burnett94] [Chang90] [Glinert90b] [Shu88] [Golin90a].

## Visual languages

There are a number of definitions of what a visual language is. Some authors speak of visual languages, some others of visual programming, and others of visual programming languages.

The following are some of those definitions:

- ⇒ Visual programming refers to any system allowing the user to specify a program in two or more dimensions. Conventional textual languages are not considered to be two-dimensional because compilers or interpreters process them as a large unidimensional sequence [Myers90].
- ⇒ A visual language handles visual information, or supports visual interaction, or allows programming with visual expressions. This last concept is taken to be the definition of a visual programming language.  
Visual programming languages are better classified, according to the type and scope of the visual expressions used, into icons-based languages, shapes-based languages, and diagrammatic languages (or diagrams-based languages).  
Visual programming environments provide graphic or iconic elements which can be interactively handled by users according to some specific spatial grammar for programs building [Golin90b].
- ⇒ Visual programming is usually defined as the use of visual expressions (such as graphics, drawings, animations or icons) during the programming process. These visual expressions can be used in programming environments as graphic interfaces for textual programming languages; they can be used to form the syntax of new visual programming languages leading to new paradigms such as programming through demonstration; or they can be used in the graphic representation of a program structure or behavior [McIntyre92].

⇒ A visual language is a programming language that mainly uses a graphic notation [Najork94].

A considerable amount of visual languages paradigms has been created in time. The two more interesting ones are control flow and data flow [Chang90] [Shu88].

Control flow paradigm uses a flow-diagram-type visual diagram to describe a program's control flow. Visual languages using this paradigm are based on the same semantic model as procedural languages of the Fortran, Algol and Pascal family.

Data flow paradigm uses boxes to denote directed arches and functions in order to connect the output of certain functions with the input of some others. Languages using this paradigm are based on the semantic model used by functional programming languages.

There are many arguments in favor of visual programming. They are generally centered on the fact that human beings process, due to their nature, more easily and quickly images than text. That is, they acquire more information and in a shorter time if they discover an image graphic relations than if they read a text [Raeder85].

Some of these arguments are here listed:

- ⇒ The text is strictly sequential, whereas images allow a varying access to any of their parts.
- ⇒ Images indistinctly allow a global or detailed vision, depending on what the person expects to find.
- ⇒ The human sensorial system is "designed" for images processing in real time. Therefore, images can be more quickly accessed and decodified.
- ⇒ Text is unidimensional by nature, whereas images are multi-dimensional.
- ⇒ Images provide a much richer language due to their visual properties, such as colors, shapes, sizes, textures. This leads to an information codification much more compact than that of texts.
- ⇒ Images can capture an abstract idea in an easier way.
- ⇒ Texts are sequences of words and punctuation marks. In turn, each word is a sequence of letters. And each letter is but a little image simply representing an only symbol of the whole text.

Even though these are reasonable arguments, they are based on intuition and can only be verified by means of empirical studies.

Different studies have been carried out in order to measure the benefits of the use of visual programming for programs development [Pandey93]. These studies have not determined that, in a general sense, visual languages are better than textual languages for programming activities. However, they at least suggest that, for many tasks, a suitable visual programming language is potentially better than any textual language.

It is clear that visual programming has a great potential. However, there are some problems with the existing visual languages.

Visual languages tend to use a relatively disperse notation. This may mean that they use more space of textual languages screen. This can be solved by using the procedural abstraction concept. That is, by representing a sub-diagram with an only simple symbol, and by treating it as a "black box".

Many visual languages are interpreted, and, in most of the cases, the interpreter directly operates on diagram components representations. This obviously results in a very low efficiency.

In addition to this, many of them are latently typed, which means that the verification that an operator may receive adequate type values can only be done when that operator is applied, that is, during running time.

## Development environment

Visual DaVinci development environment is formed by five windows: the main window, the visual diagrams editor, the textual code editor, the city, and the system variables inspector.

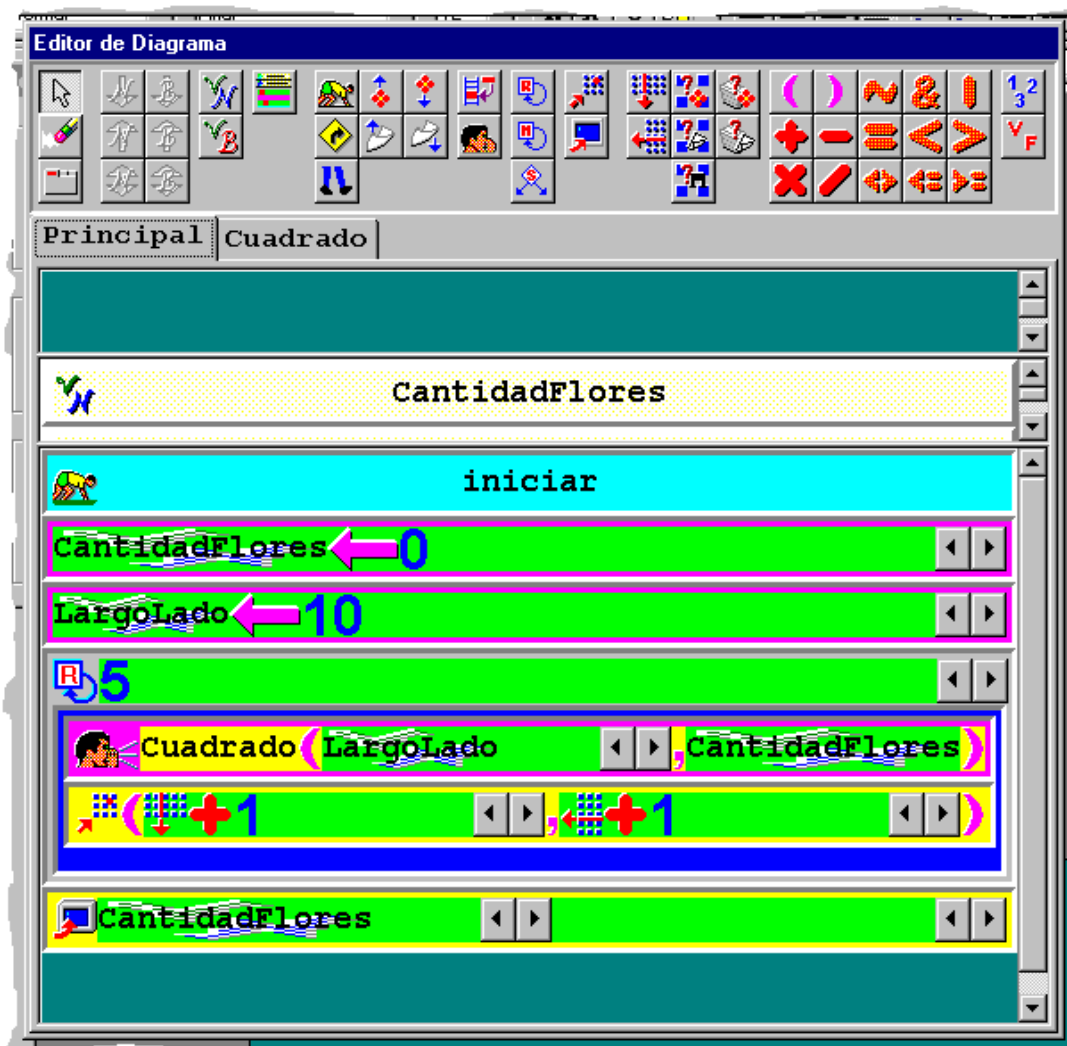


Figure 1.a. : Diagrams Editor

The main window is in charge of assisting the user with the global handling of program files, from edition, saving and restoring to syntactic verification, execution and depuration.

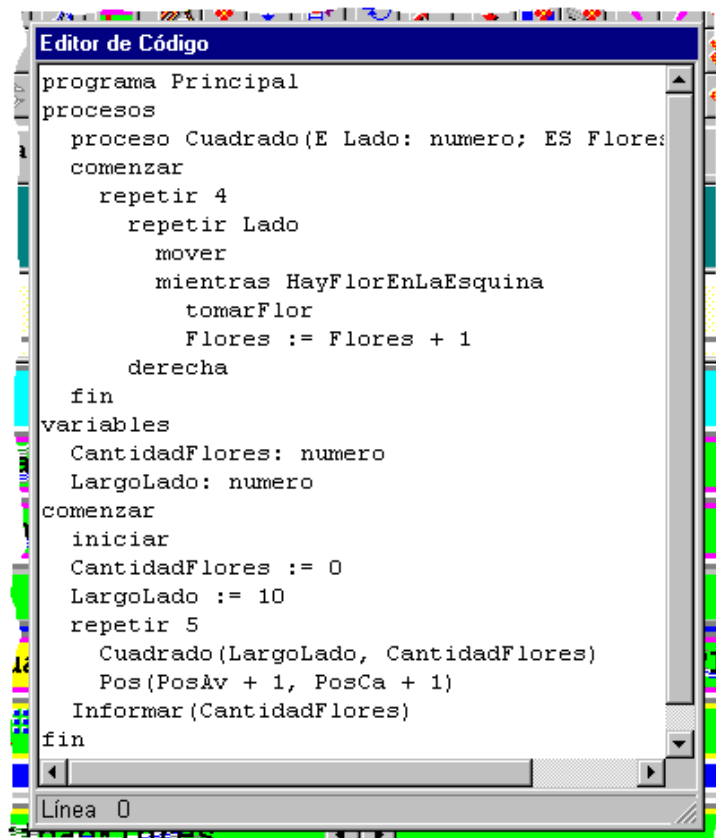
Programs visual specification is done by means of the confection of a Nassi-Schneiderman-style diagram [Nassi73] [Yourdon93] in the diagrams editor (Figure 1.a.).

It may contain several pages. The first of them corresponds to the main program, and the following ones to the different sub-programs.

Each page is divided into three parts: one for formal parameters definition, a second one for local variables declaration, and a third one for the body. Naturally, the first page formal parameters definition part is non usable, since the main program does not receive nor return parameters.

Diagram confection is done by means of the insertion of different elements of a program, selected from the editor buttons bar, and using the technique known as "drag&drop" for the cases in which a change in the position of an element is required, or when a variable, an output parameter, or an input/output parameter has to be placed in an expression or at the left of an assignation.

As it can be seen, no disperse visual notation is used. Still, it uses more screen space than the code. Procedural abstraction solves this inconvenient; however, in this particular case it would be desirable to keep such inconvenient in order to encourage modular decomposition.



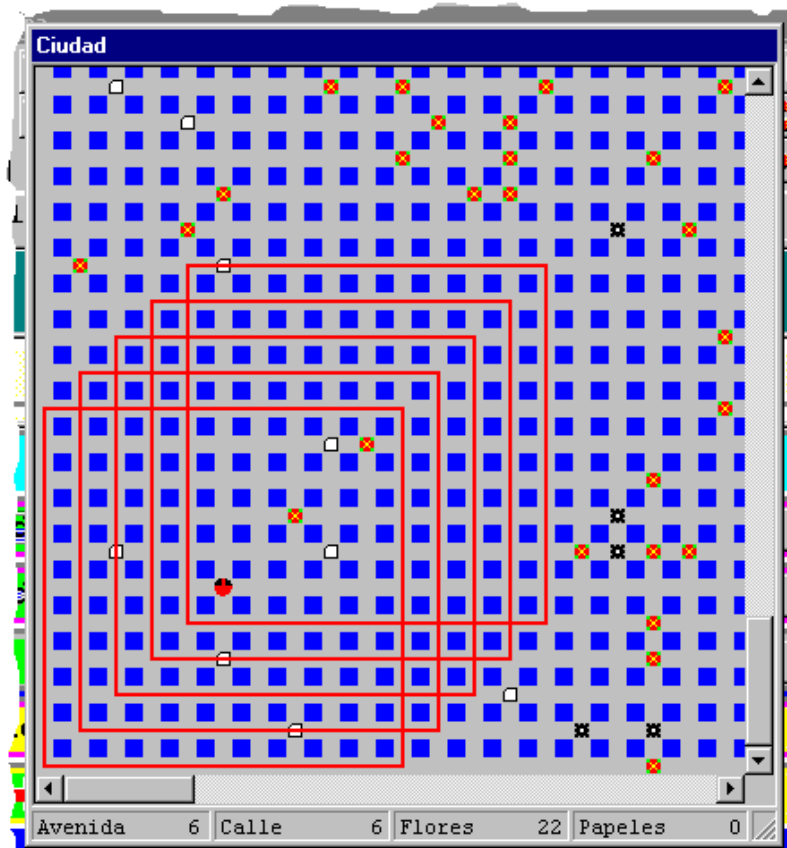
```
programa Principal
procesos
  proceso Cuadrado(E Lado: numero; ES Flores: numero)
  comenzar
    repetir 4
      repetir Lado
        mover
          mientras HayFlorEnLaEsquina
            tomarFlor
              Flores := Flores + 1
            derecha
          fin
    fin
variables
  CantidadFlores: numero
  LargoLado: numero
comenzar
  iniciar
    CantidadFlores := 0
    LargoLado := 10
  repetir 5
    Cuadrado(LargoLado, CantidadFlores)
    Pos(PosAv + 1, PosCa + 1)
  Informar (CantidadFlores)
fin
```

**Figure 1.b. : Code Editor**

Every modification made to the diagram is automatically translated to the textual code. This translation is instantly reflected at the corresponding position of the code editor (Figure 1.b.).

This is a simple text editor whose only assistance to the user is related to a semiautomatic indentation of code lines, and to execution line highlighting during the depuration process.

Robot behavior during the execution of a program can be seen on the city window (Figure 1.c.).



*Figure 1.c. : City Window*

## Implementation aspects

In general terms, Visual DaVinci is a development environment. A program can be specified in a visual way, in a textual way or in a combination of both. The syntax of the developed program must be always verified, which generates an intermediate code which is in turn interpreted for its execution.

From the interpretation of Visual DaVinci, the most important aspects are those related to diagrams visual edition, automatic translation from diagrams to code, code verification, programs execution, and effects visualization.

The system variables inspector allows to assign an initial value to them before the visualization of their contents during the execution of a program.

Syntax verification, as well as execution, is directly done based on the textual code. Therefore, the efficiency of the developed programs does not depend on the language visual condition, this problem inherent to visual languages being thus avoided. This is the basis of the importance of the automatic derivation from diagrams to textual code.

Compilers and interpreters generally have an analyzer that generates data structures such as parsing trees, symbol tables, etc., and a synthesizer that generates machine code or intermediate code [Ghezzi87] [Hunter85].

The reasons for choosing an interpreter are its development relative simplicity as compared with a compiler, and the absence of important requirements on execution efficiency.

However, for the implementation of Visual DaVinci, the scheme chosen was quite different from that of conventional interpreters and compilers.

Instead of going through the code to ensure its syntactic correction, building a parse tree, which will be then run through to generate some kind of intermediate code, a program object is created, instance of the `TPrograma` class, which "knows" how to analyze the corresponding code.

In turn the program creates a sub-program object for each of the sub-programs defined in the program, a variable object for each declared variable, and a body object.

Similarly, a subprogram object will create a series of parameter objects (instances of the `TParametro` class), a list of subprogram objects, a list of variable objects, and a body.

Thus, different classes of objects will successively be part of this process, such as `TSecuencia`, `TSentencia`, `TPrimitiva`, `TAsignación`, `TInvocacion`, `TEstructura`, `TSi`, `TMientras`, `TRepetir`, `TExpresion`, etc., which form the hierarchy in Figure 2.a.

Each of these objects has a method called `Analizar`, by means of which syntax verification of its own code is possible. If the syntax is incorrect, the object verifying it produces an exception which aborts verification and shows the user the error detected in compilation time by means of a message.

With this scheme, the result of the verification (which could be called compilation though pretentious it may sound [Hunter85]), is the intermediate code in itself. On the other hand, all aspects related to tables and trees implementation used by conventional interpreters can be completely unattended.

In addition to the `Analizar` method, each of the objects of the mentioned hierarchy has a method called `Ejecutar` giving it the necessary knowledge to execute itself.

Thus, the result of the syntactic verification is not only the intermediate code, but also its own interpreter. Therefore, in order to execute a program, only an instance of the `TPrograma` class needs be executed.

The same as before, the program object will tell each sentence of the body to execute, following the sequence in which they were specified.



As a way of complementing the previous one, there is another hierarchy of objects, shown in figure 2.b., which is in charge of visualization and translation into code of visual diagram elements. To do so, a class has been defined for each visible element of the program. Each of these classes knows how to visualize itself within the program and how to translate itself into code.

Each time a change in the diagram is produced, be it an insertion, a deletion, or a modification, the `Codificar` method of the involved objects is called, to reflect those events in the code editor text.

Even if both hierarchies - execution and verification objects on the one hand and translation and visualization objects on the other - could have been implemented as an only hierarchy, it is preferable to respect the separation of visible from non-visible objects, as most object-oriented languages do.

## Improvements and Extensions

Visual DaVinci design and implementation have been developed bearing in mind possibilities for improvements and extension of their application to other areas.

One of the improvements proposed is to include a diagram/code synchronization, so that every modification of the diagram be automatically reflected in the code (current practice), and that every modification of the code be automatically reflected on the diagram.

To do this, the code editor could be changed by other of the so called "assisted" ones, in which each input textual code line is analyzed to determine the type of sentence and then create the corresponding visible object and insert it at the adequate spot of the visual diagram.

Another improvement proposed is the possibility of easily re-using the developed code by encapsulating subprograms in entities similar to Delphi components.

As regards extensions, there are two of them proposed.

The first one intends to place an arbitrary amount of robots in the city, which will have different synchronization methods and will communicate with each other, in order to teach concurrent programming.

The other is the incorporation of a real robot on a maquette of the city to execute the developed programs.

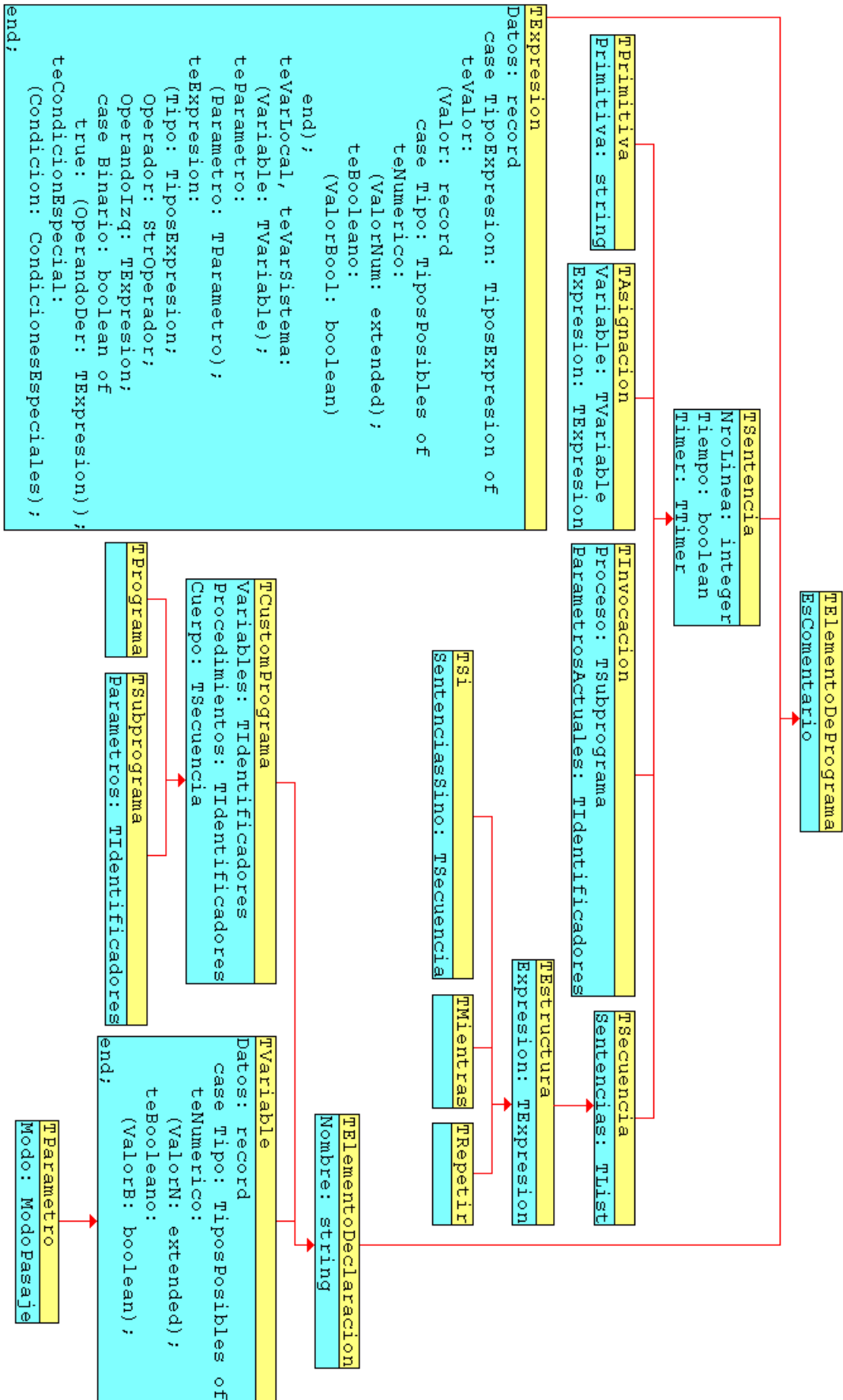


Figure 2.a. : Execution and verification objects hierarchy

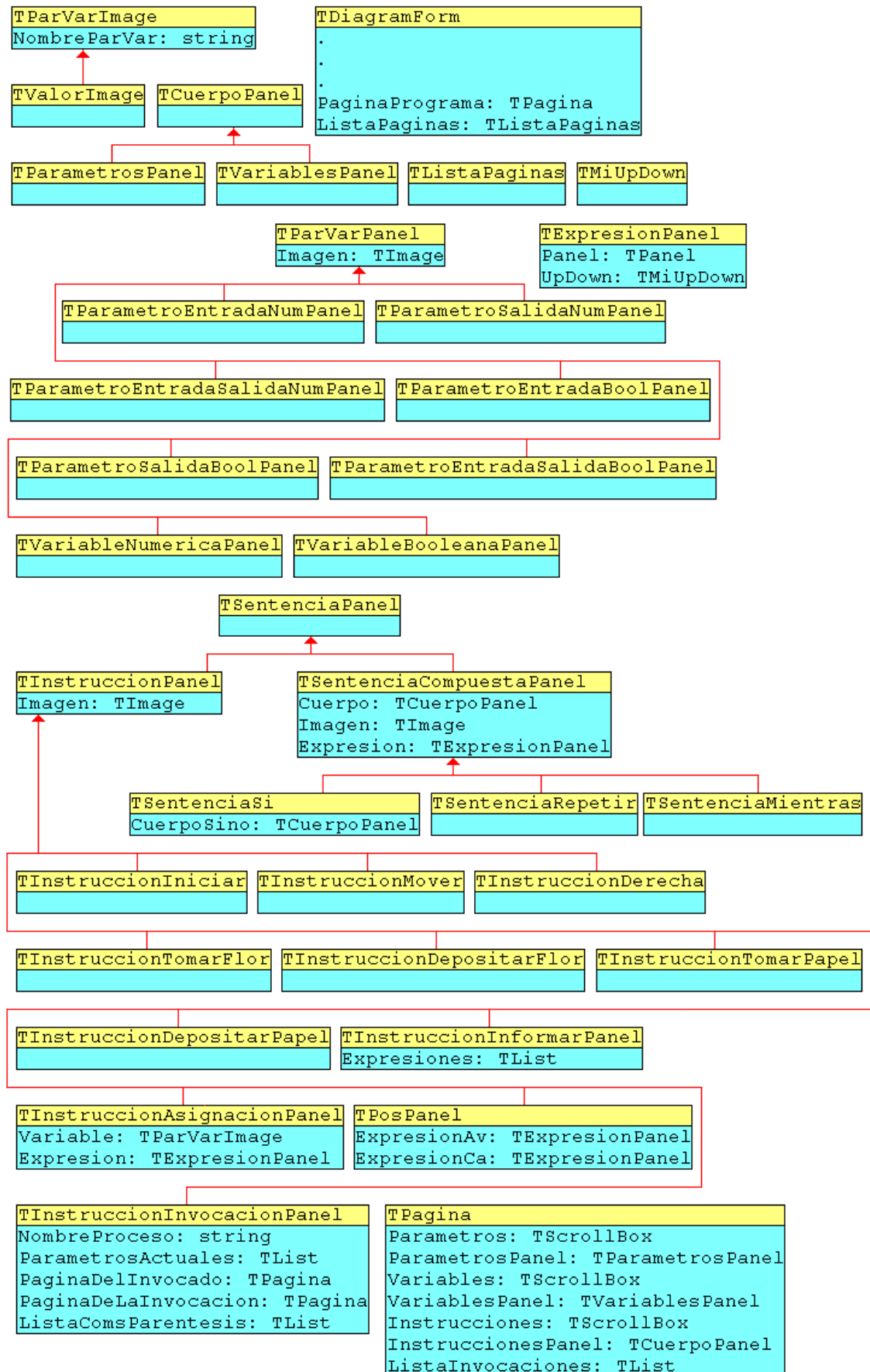


Figure 2.b. : Translation and diagrams objects hierarchy

## Conclusions

A visual language based on the control flow paradigm, which uses a diagram similar to the Nassi-Schneiderman one for programs specification which is automatically and simultaneously translated into code, has been presented.

The automatic translation from diagrams to code presents certain advantages, such as elimination of useless programs definition, independence of the syntax to which it is derived, code organization, style, etc.

Execution efficiency of the developed programs is independent of the fact that we are dealing with a visual language, since syntactic verification and execution are based on the textual code and not on the diagram.

Maintenance complexity and auxiliary structures run time usually used by compilers and interpreters, are replaced by a scheme of simple objects hierarchies.

Diagrams confection and automatic translation into code is simplified by means of a similar mechanism.

Even though the program does not use a disperse visual notation, the spatial problem persists, and it is solved by using procedural abstraction. This is only a partial solution, used as a way of promoting decomposition into modules.

## References

- [Burnett94] "Visual Object-Oriented Programming". M. M. Burnett, A. Goldberg, T. G. Lewis. Prentice Hall and Manning. 1994
- [Chang86] "Visual Languages". S.-K. Chang, T. Ichikawa, P.A. Ligomenides. Plenum Press. 1986
- [Chang90] "Visual Languages and Visual Programming". S.-K. Chang. Plenum Press. 1990
- [DeGiusti88] "LUBO-1: Una Máquina Abstracta para un Primer Curso de Programación". A. De Giusti, C. Madoz, P. Pesado. Anales de XIV Conferencia Latinoamericana de Informática. 1988
- [DeGiusti89] "Abstract Machines in a first Course of Computer Science". A. De Giusti, L. Lanzarini, C. Madoz. Proceedings 11th International Symposium "Computer at the University", Zagreb, Yugoslavia
- [DoD] "The Programming Language ADA. Reference Manual". Proposed Standard Document. United States Department of Defense. Springer-Verlag. 1981
- [Ghezzi87] "Programming Languages Concepts". Ghezzi, M. Jazayeri. John Wiley and Sons. 1987
- [Glinert90a] "Visual Programming Environment: Paradigms and Systems". E. P. Glinert. IEEE Computer Society Press. 1990

- [Glinert90b] "Visual Programming Environment: Applications and Issues". E. P. Glinert. IEEE Computer Society Press. 1990
- [Golin90a] "A Method for the Specification and Parsing of Visual Languages". E. J. Golin. Brown University. 1990
- [Golin90b] "The Specification of Visual Language Syntax". E. J. Golin, S. P. Reiss. J. Visual Languages and Computing. Volumen 1. Número 2. 1990
- [Hunter85] "Compilers. Their Design and Construction Using Pascal". R. Hunter. John Wiley & Sons. 1985
- [McIntyre92] "Visual Tools for Generating Iconic Programming Environments". D. W. McIntyre, E. P. Glinert. 1992 IEEE Workshop Visual Languages. 1992
- [Myers90] "Taxonomies of Visual Programming and Program Visualization". B. A. Myers. J. Visual Languages and Computing. Volumen 1. Número 1. 1990
- [Najork94] "Programming in Three Dimensions". M. Najork. Ph. D. Thesis. University of Illinois. 1994
- [Nassi73] "Flowchart Techniques for Structured Programming". I. Nassi, B. Schneiderman. ACM SIGPLAN Notices. Volumen 8. Número 8. 1973
- [Olsen83] "Ada for Programmers". E. Olsen, S. Whitehill. Reston Publishing, Prentice-Hall. 1983
- [Pandey93] "Is it Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study". R. Pandey, M. Burnett. IEEE Symposium on Visual Languages. 1993
- [Raeder85] "A Survey of Current Graphical Programming Techniques". G. Raeder. IEEE Computer. Volumen 18. Número 8. 1985
- [Shu88] "Visual Programming". N. C. Shu. Van Nostrand Reinhold Company. 1988
- [Yourdon93] "Análisis Estructurado Moderno". Yourdon. Prentice Hall. 1993