An Iconic Framework for Learning the Art of Programming

Guido Averna, Biagio Lenzitti and Domenico Tegolo

Abstract: *The integration of programming teachings, in all levels of education, highlights the need to acquire the art of programming for each individual student through versatile tools based on specific cognitive methods. Diversified linguistic metaphors have to be adopted by the developing frame, in order to highlight the qualities of each student. Therefore, a framework, oriented to learning the art of programming, must foster polychrome constructs representations, a number of data structures and an intuitive interfaces in order to make easier to understand the evolution of the algorithm that have to be developed. The following contribution will present a theoretical formalization of a framework for teaching and learning the art of programming and therefore its development will propose as a graph-based execution and iconic modular interfaces on the methodologies to be developed.*

Key words: *Iconic Language, Visual Programming Language, Unified Modeling Language, Graphical Language, Coding, Programming.*

INTRODUCTION

In the last decade, due to the spread of computers and communication networks in the field both academic and scientific, it has made the need to teach the programming languages in all levels of education. There are numerous starting and specialized courses to train students in the art of programming, and a number of worries can involve students, that must address and resolve. Skills, for a correct solution, have be acquired and they can be grouped into two categories: problem solving and details on the semantics and the syntax of programming languages.

These two categories play one of the main aspects in the activity of learning of the art of programming. Languages, in general, can be defined as a formalization of conceptual schemes that the human being intend to express. Patterns are interpreted and converted into visual representations with images [1], and finally presented with visual or iconic languages.

The iconic representation, significant element in the area of visual languages [2,3], requires the understanding of specific cognitive processes which facilitate the construction of mental images useful for a more rapid and accurate acquisition of skills.

The numerous fields of application support the diversification of the use of graphical symbols and constructs, thus in the iconic programming, images or icons visually represent the entities, the alphabet or symbols of language, and each of them is associated to a particular meaning or to a predefined action.

The languages based on data, constructs and iconic interfaces were presenting in the literature as Visual Programming Language (VLP) and some consolidated examples were Labview, the UML, as well as the numerous examples of prototype were proposed in which they provided some descriptive information without representative entity. The block diagrams or flow charts, universally known and widely used are primordial examples of iconic representation. This representation has a close analogy with the concept maps [4] and represent learning and teaching tools. Moreover, they allow to establish, organize and represent, in a visual way, the various concepts and their relationships.

Relations between the icons are defined according to the context of language, in this sense it is possible to highlight the natural relationship between the conceptual and the visual language maps. CmapTools, Scratch and Labview are just some of the examples of iconic languages, the first is a general purpose framework used for creating a conceptual maps [5], Scratch language, brought into vogue by code.org, is surely one of the most interesting examples of visual language-oriented teaching program for primary school pupils [6], while the last one has been developed for the definition and evaluation of electronic circuits.

The representation of algorithms in term of execution graphs, relationships and properties are described using symbols [7]. Visual representations provide a more intuitive reasoning, and from this point of view it improves the based knowledge [8]. This makes it easier to understand the logical implications of the various icons [9], and in addition, this improves the understanding about the possible modification of these properties and their effects on physical objects [10, 11].

This representation has always been used for teaching programming, but also to digital tools to help students develop programming skills as "Raptor" [12], "ProGuide" [13], ect.

This work will present a framework designed for learning the art of programming. Iconic data structures, visual constructs, visual working space are some of the main elements to define, in a simple and meticulous way, a set of execution graphs which represents algorithms. In order to validate the goodness of the code and test it on conventional processors, the framework offers the possibility of transforming the execution graph in one of the text-based programming languages. This option allows both to test the iconic code, but also to evaluate the dual implementation version (visual and textual) of the graph execution.

In order to make this framework understandable to the neophytes and also usable both by inexperienced students and professionals, it will be offered some details of the framework, and explained the various aspects of graphics features found on the icons. Similarly, we will analyze the statements of the data structures that have been implemented on the iconic framework. Finally, it will be analyzed the key aspects which are useful to understand which icons are desirable, their association and the interconnection among constructs and data.

Iconic Language Definition

Data structures, constructs and libraries are presented in any programming both textual and visual language, these in own representation provide tools for modeling algorithms and methodologies to be run on standard computer systems. There are considerable differences in representation between textual and iconic languages or visual, defining a clear boundary between their micro and macro structures, including data and constructs, although sometimes the constructs can be treated as data. It is well known that in programming languages elementary structures can be treated as monolithic objects and represented as textual or iconic elements. Therefore in the follow, the language will be structured by more general macro facility developed on the micro structures, the icons. Moreover, the constructs can be designed both as an iconic formalization of relations between micro structures and as elementary icon with which it is possible to express methods and algorithms in accordance to other elementary structures,.

There are many examples of meta-languages and programming languages that have been presented and developed in last century, all sharing the user to offering general purpose tools and not only for solving special matters on standard computer system.

In the coming sessions we will discuss on the formalization of an iconic language and its implementation as a tool for learning the art of programming. Therefore, it will give great importance to the definition of the icons is in term of shape and color and all of them in order to make more simple and intuitive their use.

Formalization of the Iconic Programming Language.

The formalization of the proposed iconic programming language has been addressed through the definition of its grammar and the related symbols. The symbols are represented by elementary icons and a contex-free grammar has been defined as a Visual Language Grammar GVL (N, T, S, P), where N is the set of not terminal icons,
$T = T_1 U T_{WS}$, S = Start symbol, P = set of production rules.

$$T_1 = \left\{ I(1) : I = \left(I_l(WS,t), I_p(WS,t), Type(t), RelAction(WS,t) \right) cont. UI \right\}$$

$$T_{ws} = \{ set\ of\ operations\ on\ WS\ to\ modify\ I \}$$

$$I = \left(I_l(WS,t), I_p(WS,t), Type(I), Relaction(WS,t) \right)$$

$$I_l = The\ logical\ part\ of\ I(Graphical\ representation)$$

$$I_p = Phisical\ part\ of\ I$$

$$Type = The\ set\ of\ Data\ Structure\ of\ I$$

$$RelAction = The\ relation\ between\ I\ and\ WS$$

$$UI = Universal\ Space\ of\ icon.$$

In the following it will be given the logical implementation and the development of an iconic language. The basic concepts are:

- The iconic configuration of the main constructs;
- The logical flow of instructions to the language;
- The visual representation, in a working space (WS), of the programming code developed during the writing of the iconic code.

Visual representation of the main constructs.

Each construct has a uniquely determined by its features icon:

- Border color;
- Filled color of the icon;
- Shape of the Icon;
- Owned text of the icon;
- Creating a custom image for the functions created by the use.

The boundary of the icons provides the first cognitive approach for the representation and the commit of the construct. The icon of the initial Start symbol, for example, has a red color boundary and it represents the key element to analyzing the graph of execution. Variables and constants have been defined with the same boundary, but a big difference has been adopted according to the different type: the numeric data type or string, discrete or floating point, union or based.

A variable or constant discrete was defined with a blue color, while the floating point adopts orange thus the string data type was associated fuchsia color. This allows us to perceive without any mnemonic effort the data type that marks the variable or constant.

The icons representing the *cycles construct* are indicated by a green boundary thus within them the user can define the own textual conditions, the iconic representation of the *conditional construct* was adopted yellow for its boundary. Finally, the boundary of icon assigned to the *sequential block* was black.

These are just some examples of boundary color definition for the iconic data structures and constructs and other diversified representations were fixed both for shape and color of other objects (break, continue, exit, home, return, assignment). Particular attention can be devoted to sequence constructs (begin ... end, {...}, ...) which use a dynamic color subjected to the level of nesting, that is, the color will be linked to the color of the flow, and this in order to facilitate the perception of the nesting level.

The icons representing functions will take two different colors for the boundary (purple, blue), and this in order to make easily perceptible those functions which return output values from the other one. There are numerous options available in the icons in relation to the activities it carried out or on, making this icon dynamic in time: thickening of the boundary shows an active icon, icon with its handle positive changes in the color, and finally blue color when it is selected.

In addition to the color that highlights the state and/or the type of structure or context, each icon is also characterized by a special shape. From a cognitive point of view, the alliance of color and shape, which in a naïve analysis may seem more complex, facilitates the understanding of the syntactic and semantic and in general the meaning of the icon and therefore their rapid use.

Various shapes were defined for different icons of the Visual Language, the icon for the initial Start symbol, for non-terminal symbols (variables, constants, elementary constructs, ...) and for the icon of the terminal symbols. Some of them capture the attention using intense colors, others fascinate the user with their shape, thus all of these strategies were approached following the Gestalt theory [14].

In order to give some details on design definition of the icons related to constructs and data structures, in the following will be proposed some examples and their iconic representation will be shown in Table 1.

Conditional Control Structures: A shape of rhombus was adopted as icon for this structure. This choice was tuned with the need to visually highlight all the possible branches of execution, and also to satisfy a general assumption proposed by the literature for some metalanguages. The hosted conditional expressions can be obtained as a composition of expressions both iconic and textual.

Loop Control Structures: they provide a breakpoint for the execution flow, they refer to variable length code portions, thus its iconic representation is obtained with a quadratic shape with smoothed corners, its size is larger than the others in order to highlight both the dynamism on the size of the instructions to be executed and that the block is subject to execution conditions.

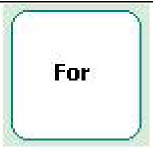
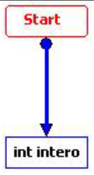
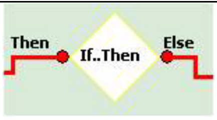
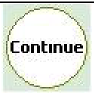
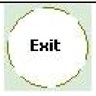
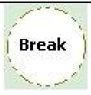
		
		

Table 1: Example of Icons.

Sequence Control: The icon for a code block uses a square shape, similar to that of the loop control. However, unlike these, its corners are not smoothed. It, from a cognitive point of view, evokes the encapsulation of code.

The Break, Continue and Exit: their constructs are represented by a circle, and in this case the type of the edge differentiates their meaning. The edge with a continuous circle evokes the continuity construct, the Break Interrupt without leaving the function is obtained with a weakly dashed border, while the Exit has a heavily dashed border as cognitively evokes exit the function.

The root of the tree, the Start icon: this icon is hosted inside the Working Space and is defined in the grammar is not necessarily in a exclusively way. In each WS, more of them can coexist and thus multiple concurrent processes can be executed. In accordance to the production rules, the arches represent the icons which connect structured icons to follow a stream. This ensures the continuity between the constructs and allows you to have a control over the variables, constants, functions and procedures, following the *RelAction* production rules defined in the grammar.

The iconic view of the code provides a good way to understand the execution of the flow behavior of the code and of the interactions between the local parts. The connection between the icons with the arcs metaphor improves the visibility conceptual, guarantees the logical continuity of the flow and provides a pictorial display of the theoretical formalization of an algorithm. Furthermore, the arcs may assume a different color selected from ten different size and shape, depending on the nesting level.

Some details of the Implementation of the Framework.

A framework, in order to improve both the logical understanding and the drafting of a generic program, must enjoy an aptitude of understanding of the objects present in it and also a naïve use during the development of complex methodologies. Therefore, the framework must enjoy the following properties:

- To hold objects to be manipulated with intuitive feedback which identify significant sections of the algorithms that have to be developed.
- Based on a contex-free grammars.
- Open to extend the own iconic code.
- Provide an easy readability of the code and an adequate perception of the logical behavior of the developed methods.
- Exportable in other text-based programming languages.

The Object-oriented programming has allowed to use the concepts of encapsulation, inheritance, polymorphism and abstraction. Therefore for the iconic representation of the proposed framework, some objects from predefined classes have been created in which shapes, colors, production methods or rules were introduced. A parser based on regular expressions was adopted in order to define and validate the syntax of the production rules.

Finally, graphs execution present in the Working Space can be divided into logical and graphics streams. The logic flow is implemented by arrays of connections implemented in TForme class. While, the graphical one is made by the arcs incoming or outgoing from the icons. The class that implements those graphics is called TLinker.

To save the project of execution graph, its files and functions, three classes were implemented derived from TXMLDocument class: TXML_Progetto, TXML_File, TXML_Funzione.

At startup, the application dynamically builds objects useful for creating graphs of execution (see figure 1). Whenever an icon object is inserted in the Working Space, it is assigned its iconic features and a set of methods for their control.

In addition, in order to analyze the code in terms of both iconic and textual view it has been placed in a working window in which its equivalent text format is displayed written in XHTML. It notes that this window is read-only, thus its code cannot be changed.

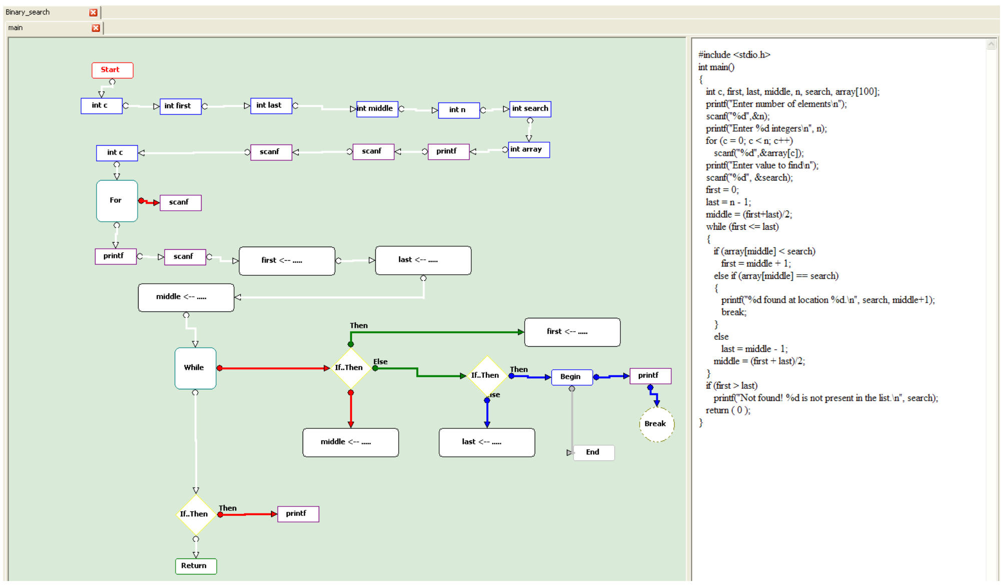
Each icon provides a particular textual code associated with it, such as the icon of the loop ***For*** provides a textual code like this ***"For (k = 1, k <10, k ++)"***. Furthermore, when the user selects an icon, automatically it is also selected the source code generated in the test format and this for an higher cognitive understanding of the process of the algorithm.

Completing the framework the classic functions for saving the project and file, as well as their initialization, the configuration of the project and the IDE. Moreover a special section is dedicate to save the compiler functions and the starting of the developed program.

CONCLUSIONS AND FUTURE WORK

This paper was presented an iconic framework designed and built for learning the art of programming in computer systems.

This framework, structured on a context-free grammar, ensures the proper use of icons and provides a parser to locate syntax errors also in the working process of the code. The iconic representation of the constructs and data structures provides a clear and immediate display of the operating logic which ensure a high level of abstraction useful for both local and global perception of methods or algorithms that the neophyte implements. The framework also creates a robust and correct source code written in text-based language, and provides a visual representation of logical operations hosted in the implemented procedures.



```
#include <stdio.h>
int main()
{
  int c, first, last, middle, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d",&n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d",&array[c]);
  printf("Enter value to find\n");
  scanf("%d", &search);
  first = 0;
  last = n - 1;
  middle = (first+last)/2;
  while (first <= last)
  {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search)
    {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;
    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d is not present in the list.\n", search);
  return ( 0 );
}
```

Figure 1: An Example of Framework in progress

A validation phase in the classroom, planned as the next step in this paper, will focus on seeking feedback from those students will use this programming tool. Therefore, in order to understand the degree of learning of programming with the iconic language, it will be defined as a set of background processes that evaluate the types of errors made during the drafting of standard algorithms and the timing to solve them.

In addition, the characteristics of the framework enable to realize in a naïve way an online web version, in order to expand the audience of potential purchasers and for potentiating its use in an e-learning platform.

REFERENCES

[1] Arnheim, R., Visual Thinking, University of California Press, Berkeley, 1969.

[2] Blackwell A. F., Metacognitive theories of visual programming: What do we think we are doing? Proc. IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1996, pp. 240-246.

[3] Petre M., Blackwell A. F., Green T. R. G., Cognitive questions in software visualization, in J. Stasko, J. Domingue, B. Price, M. Brown, Software Visualization: Programming as a Multi-Media Experience, MIT Press.

[4] Saarinen, S., Hietala, P., Mikkilä-Erdmann, M., Mäkiaho, P., Poranen, T., Turunen, M., “Improving E-Textbooks: Effects of Concept Maps”, International Conference on e-Learning (2014), ISSN: 2367-6698, pag. 77-82.

[5] Novak, J.D. and Canas, A.L. (2006). The Theory Underlying Concept Maps and How to Construct and Use Them. Technical Report IHMC CmapTools 2006-01.

[6] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. Commun. ACM 52, 11 (November 2009), 60-67. DOI=http://dx.doi.org/10.1145/1592761.1592779.

[7] Larkin J. H., Simon H. A., Why a diagram is (sometimes) worth ten thousand words, Cognitive Science, Carnegie-Mellon University, 1987, pp. 65-99.

[8] Narayanan N. H., Suwa M., Motoda H., A study of diagrammatic reasoning from verbal and gestural protocols, Proc. 16th Annual Conference of the Cognitive Science Society, Lawrence Erlbaum Associates, 1994, pp. 652-657.

[9] Larkin J., Display based problem solving, in D. Klahr, K. Kotovsky, Complex Information Processing, Lawrence Erlbaum Publishers, Hillsdale, New York, 1989.

[10] Narayanan N. H., Suwa M., Motoda, H., Diagram-based problem solving: The case of an impossible problem, Proc. 17th Annual Conference of the Cognitive Science Society, Lawrence Erlbaum Associates, 1995, pp. 206-211.

[11] Narayanan N. H., Suwa, M., Motoda, H., Behavior hypothesis from schematic diagrams, in Diagrammatic Reasoning: Cognitive and Computational Perspectives, J. Glasgow, N. H. Narayanan, B. Chandrasekaran, (Eds.) AAAI Press and MIT Press, 1995, pp. 501 -534.

[12] Dr. Nikunja Swain, Wanda Moses, James Allen Anderson "RAPTOR - A Vehicle to Enhance Logical Thinking" 2013 ASEE.

[13] C. Areias and A. Mendes, "A tool to help students to develop programming skills", Proc. 2007 Int. Conf. Computer systems and technologies, ACM, New York, NY, USA, 2007, Article 89, 7 pages.

[14] Datta, R., Joshi, D., Li, J., Wang, J.Z.,, ``Studying Aesthetics in Photographic Images Using a Computational Approach,'' Lecture Notes in Computer Science, vol. 3953, Proceedings of the European Conference on Computer Vision, Part III, pp. 288-301, Graz, Austria, May 2006.

ABOUT THE AUTHOR

Dr. Guido Averna, M.Sc. Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Italy. averna_guido@libero.it

Assist. Prof. Biagio Lenzitti, Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Italy. biagio.lenzitti@unipa.it

Assoc. Prof. Domenico Tegolo, Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Italy. domenico.tegolo@unipa.it

The paper has been reviewed.