Object oriented programming and development in JxLogo

Michele Moro

University of Padova – Department of Information Engineering

Via Gradenigo 6/B – I 35131 PADOVA Italy

michele.moro@unipd.it

Abstract

JxLogo is an undergraduate student project that develops a complete Logo programming environment in Java. It includes an object oriented extension based on declarative structures similar to those used in common general purpose programming languages. The paper describes some details of the OO component of the language. It illustrates the effectiveness of the proposal describing the realization of an elementary geometric library. Some remarks on how following a straightforward progression to learn OO programming are also presented. In-line compiling of JxLogo classes and procedures into Java bytecode is a specific characteristic that is emphasized in the paper.

Keywords

JxLogo, Object oriented programming, Java, Multiprogramming

1. Introduction

JxLogo (Moro, 2001) is an undergraduate student project developed at our Department with the aim of experiencing modern technologies of Java in a well known and not so complex context as Logo. Through integration of methods, interfaces and protocols, students can explore modern programming techniques and, at the same time, build up a Logo programming environment with innovative characteristics, in order to make young students experience Logo language extensions, effective graphical interfaces, and new operative approaches.

Modern Logo environments privilege a visual and multimedia approach: this leads the user to exploring complex concepts such as objects, components, agents, events, actions and reactions. Notwithstanding, the level of the textual programming language maintains an important role. Provided that the basic Logo language is suitably extended with modern programming structures, even Logo can be a starting point to improve one's programming skills.

Historically Logo has a functional linguistic component so that programs are collections of variables and procedures with a general public visibility; the interpreted nature of the language permits a form of 'dynamic' programming ('code and try') that makes learning faster. If on the one hand Logo environments are used to effectively explore microworlds and the more or less sophisticated interactions between their components, on the other hand it could be used for older students as a link to face the comprehension of modern, professional general purpose programming languages. In these languages one of the main aspects is the object oriented (OO) approach.

According to this point of view, in the development of JxLogo we tried to define an OO extension of the Logo language with particular attention to the following key aspects:

- To maintain the substantial interpreted nature and dynamics of Logo but at the same time to accustom the user to complex declarative structures that are common in OO languages;
- To make the visibility control on program components evident and to take the advantage of specific aspects of the OO approach such as inheritance, overloading, overriding and polymorphism;
- To fully take advantage of the characteristics of the Java environment and of the JVM.

In section 2 the paper introduces the main elements of the proposed extension giving motivations and some illustrative examples. In section 3 a geometric library is briefly introduced: the presented code may give an idea of the characteristics of our proposal. Section 4 shows how JxLogo can support the learning of OO programming following an effective progression. The section also gives some remarks on how the implementation can directly map JxLogo classes onto Java classes. Some advanced experimentations are also presented. In the last section we provide some comparisons with other available Logo environments and some conclusions.

2. The object oriented component of JxLogo

2.1. The class

The fundamental construct in OO languages is the class. It realizes:

- an association between data elements and methods with the creation of a new data type and of a scope for the inner names;
- a visibility control of the inner elements;
- inheritance and polymorphism support.

Usually the class appears as a static declarative construct with the possibility of creating object instances: in pure OO languages only elementary data and references can be statically allocated, whereas objects are dynamically allocated.

In JxLogo it has been attempted to maintain this approach but without renouncing the possibility of defining new methods on-the-fly in favour of a compatibility with Logo procedures. Consequently the JxLogo class declaration includes only data elements, a constructor with possible parameters but not methods which are separately declared.

The constructor allows the specification of initializing values for a new instance of the class when it is allocated. Such values can be calculated on the basis of parameters, possibly with default values, thus the constructor acts as a special method. Because Logo is not a strong-typed language, the traditional method overloading based on prototypes is not possible: consequently in JxLogo the constructor is unique in each class and it is simply expressed as the code body of the class declarator.

The constructor parameters may be mandatory, with default values or specified with a list of arbitrary length. The three options can be present at the same time but exactly in this order (the mandatory first, then the default and finally the list);

Table 1. The class constructor and its parameters

`class P :x :y`
`; constructor`
`; body`
`End`
	`class P [:x 0] [:y 0]`
	`; constructor body`
	`end`
	`; (0,0) default params`
		`class P [:coordinates]`
		`; constructor body`
		`end`
		`; e.g. make "p1 P [3 5 7 9 11]`

For data elements it has been attempted to fix visibility rules more easily corresponding to the elementary operations of reading and writing, trying to make the first one almost always possible, as usual in Logo, and controlled the second one. Data elements are declared within one of the four following categories:

- `export` public visibility (both reading and writing)
- `own` public reading, protected writing (allowed within the class and its derivative)
- `hidden` protected reading, private writing (allowed within the class)
- `const` public reading only

The public visibility is extended to all class methods and all traditional Logo procedures. These rules give a good control on the visibility and add some useful simplification: as an example, the mere reading of `own` variables does not require any access method like in the case of protected variables in Java. When it is necessary to declare a data element with one of the visibilities above but common to all, the instances of the class (like static variable in Java), four specific keywords are provided (`commonexport`, `commonown`, `commonhidden`, `commonconst`), one for each type of visibility. A common element is allocated when the including class is loaded onto the system.

Class data elements may be initialized either contextually with their declaration, using expressions possibly referring to the constructor parameters, or later within the constructor body. In the example that follows, of the three elements given, i.e. x, y, distance, the last one is not initialized in the declaration (symbol ^) but in the code below. In order to distinguish the setting of data elements from the allocation and setting of Logo global variables, in JxLogo new values are assigned to data elements with the `set` command whereas the connector symbol `'` must be used to qualify an element for a class or for an object. When the context permits, the class name will be omitted.

```
class Point :x :y
    own [
      x :x
      y :y
      distance ^]
    set 'distance sqrt :x*:x+:y*:y
end
```

2.2. Instances

The name of the class, followed by a sequence, possible empty, of arguments within square brackets, represents the reference to a new instance of the class and may be assigned to a variable (data element) with a `make` (`set`) command. Thus the assignment between variables and/or data elements is only a reference copy and it does not cause an object copy.

Table 2. Class instances

`make "p Point[10 10]`	`class Square [:vrtx Point[]] [:edge 1]`
`make "q :p`	`hidden [center ^]`
`; q and p refer to`	`set 'center Point[:vrtx'x+:edge/2 :vrtx'y+:edge/2]`
`; the same object`	`end`
	`;:vrtx has the origin as default`

2.3. Methods

In order to maintain the maximum similarity with procedures, in JxLogo methods may be incrementally added to their class, in one comprehensive file or in separate files, giving the user the feeling of a dynamic definition (some implementation details, later described, will explain the limits of this assertion). All methods like procedure have public visibility. The absence of a greater control of visibility on the methods and of the distinction between instance methods and class methods (in Java with the use of the `static` attribute) has seemed acceptable in favour of a greater simplicity.

The declaration of a method has the following general structure:

```
to <class name>'<method name> :<param1> :<param2>. . . <paramh>
   [:<param(h+1)> <expr1>]. . . [:<param(h+k)> <exprk>]
   [:<list>]
end
```

Similarly to the interface of the constructor of a class, the parameters of a method can be mandatory (1..h), with default (h+1..h+k), a list of values with an arbitrary length.

Due to the compilation rules of JxLogo classes (see implementation details) variables (data elements) previously assigned to an instance of a class which is subsequently modified, e.g. adding or modifying a method, continue to refer to the old version of the class and may still be applied to the old methods during the current session.

The notation for a method invocation is straightforward: the name of a variable (data element) qualifies the method name, followed by the calling arguments within parenthesis. The couple of parenthesis is required even with an empty sequence of arguments in order to syntactically distinguish a method call from a procedure call:

```
make "c Point[10 20]       :c'draw()
```

Inside a method `m1` of a class `A` any reference to the same object on which `m1` acts is implicit and rendered simply omitting the name of the object (that is equivalent to the use of the keyword `this` in Java). For example:

```
to Point'setdraw :x :y
    set 'x :x
    set 'y :y
    'draw()
end
```

Remarkably the presence of parenthesis permits the concatenation of several method calls if each of them returns an object:

```
:<var> ' <method1>(...) ' <method2>(...) ' . . . ' <methodn>(...)
```

As already mentioned, method overloading is not immediately realizable since it is not possible to 'statically' differentiate methods with the same name on the basis of the type of their parameters because both Logo variables and JxLogo class data elements are not strong typed. Nevertheless the programmer can fuse in one procedure (one method) the overloadable meanings differentiating the code in the procedure (method) body on the basis of the -time contents of the parameters, using known primitives like `numberp`, `wordp` and the new primitive:

```
objectp <object> <classname>
```

that returns TRUE if the first parameter is an object of the class whose name is the second parameter.

2.4. Inheritance and polymorphism

A class B can extend a class A (keyword `isA`) inheriting the data elements: an instance of B includes both the elements of A and those of B. Generally speaking first the B constructor calls the A constructor with its specific arguments and then executes its body. Following the fixed visibility rules, the B methods may write `export` and `own` data in the A class. In the example:

```
class Pixel :x :y :c isA Point[:x :y]
    own [color :c]
end
```

`Pixel` extends `Point` adding an `own` element `color` to the `x` and `y own` members inherited from `Point` and visible in `Pixel`; of the three parameters of the `Pixel`'s constructor, the first two are passed to the base constructor.

Inherited methods can be redefined (overridden) in the derived class in order to realize more specific aims. The redefinition involves only the method name, therefore the interface of the method in the derived class can be different from that of the base method. Usually the application of a method `m` to an object of class `Z` causes the run.time searching and activation of the first version of `m` in the inverse derivation path (from `Z` toward its hierarchy root). If in a method of the B class you want to refer to a version of a method `m` in one superior class which is different from the version defined in B itself, you must use the keyword `old`. For example:

```
to Pixel'draw  setpc 'color      . . .       end
to Pixel'paint . . .
    old'draw(); it calls the method inherited from Point
    . . .
end
```

Therefore a call `:<var>'<method>` can activate various methods in various moments according to the object assigned to the variable (polymorphism): in this case, in order to avoid run-time errors, it is necessary that the various versions of 'polymorphic' methods have the same interface.

A similar masking behaviour regards class data elements: a `x` element in class B masks a `x` element defined in one of its superior class. `old'x` may be used in B to refer to the masked element.

2.5. Object clones and extensions, anonymous classes

Some special notation have been introduced to allow object cloning and extension. `(<var>[])` creates a new object of the same class and state of `<var>`.

```
make "p1 Point[10 20]
:p1'rotate(90); rotate of 90 degrees around the origin
make "p2 (:p1[]); it is a clone of :p1[- 20 10]
```

After the cloning the object clone has a life independent of the original object.

Cloning can be seen as a particular (reduced) case of the 'object extension' paradigm for which a new object is derived from the structure and the state of a preexistent object adding new data elements and methods (this feature is still under construction). For this case the proposed general syntax is:

```
(:<var>[<data element declaration>])
```

For example:

```
make "p2 (:p1[own[color "red]]); it extends the object of p1
```

On the new object it will be possible to define (or to redefine) specific methods only regarding that object. For example:

```
to :p2'draw
    show 'color
    old'draw()
end
```

Actually the extending object is the unique instance of an anonymous class deriving from the old object class; the new object also inherits the old object state. This requires the generation and compilation on-the-fly of a new ghost class when the new object is declared and its updating when new methods are added. If necessary, the user moves from the anonymous class to a named one by means of the command `classcode`. For example:

```
classcode "Pointc :p2
```

generates, and saves on the file `Pointc.lgo`, the source code of the class named `Pointc` which is the previous anonymous class of `:p2`, including the element `color` and the method `draw()`.

An anonymous class could also be defined extending a class: in this case the initial state of the inherited elements is specified, like in a normal instance, supplying the arguments to the constructor of the base class. For example:

```
make "p2 (Point[1 1][own[color "red ]])
```

2.6. System classes, multiprogramming

JxLogo includes some system classes to support the development of effective applications. The system classes are immutable (the Logo programmer cannot directly add elements and/or methods) but they can be extended by other classes, named or anonymous.

The `Turtle` class is obviously the fundamental class of the system: it includes all the elements that define the state of a turtle and its methods are implicitly represented by Logo commands who act on the turtle (forward, left, etc). When normal turtle commands are used, it is like referring to a default instance of the `Turtle` class represented by the active turtle.

When you want to directly refer to a specific instance, the syntax is the usual for objects, e.g. `t1'forward(80)`. The actual constructor parameters of the class are the two initial coordinates of the turtle. Derived classes may define more specific turtles taking advantage of the redefinition of methods and polymorphism. Some examples:

```
make "t1 Turtle[10 20]; base turtle in position [10,20]
make "t2 (:t1[ ]); a clone of t1 in its current position
make "t3 (:t1[own [color [255 0 0]])
    ; a turtle with the state of t1 and one more
    ; data element 'color' set up to red
to :t3'forward :s; it redefines the forward primitive forward for t3
    'setpc('c); it sets up current colour
    old'forward(:s); normal advance
end
make "t4 (Turtle[40 60][own [color [255 0 0]])
    ; the position is set up through the base constructor
class Cturtle :x :y [:c [0 0 0] ] isa Turtle[:x :y]
    own [color :c]
end
class Pturtle [:x 0] [:y 0] isa Turtle[:x :y]
    own [step 50      angle 90]
    commonown [num 0]; it counts instances
    set 'num 1+'num
end
to Pturtle'forward()
    old'forward('step)
    'rt('angle)
    ; with fd it jumps for a fixed step and it turns 90 degrees
end
```

JxLogo also supports multiprogramming taking advantage of the multithreading model of Java: the simplest form of concurrency is based on the `exec` primitive which spawns a new thread with its own environment of local variables. Threads can be suspended, resumed and killed from within other threads.

We decided to collect advanced multiprogramming features inside a new base class called `Agent` which extends `Turtle` and therefore includes at least one turtle. This class makes a message-based direct form of communication and synchronization between agent instances possible, with asynchronous methods (without waiting, `'sendnow()` and `'receivenow()`) and synchronous methods (waiting synchronization, `'sendwait()` and `'receive()`). The flexible syntax allows three variants, following respectively 1:1, 1:n and broadcast models (the last one refers to all the present agents in the system). For example (a1, a2, ... aN are agent instances):

```
:a1'sendwait("Hello :a2) ; synchronous send from :a1 to :a2
:a1'receivenow("msgbuf) ; immediate receive from any agent
```

```
:a1'sendnow("Hello [:a2 :a5 :a7]) ; immediate send to three agents
:a1'sendwait("Hello) ; broadcast with waiting for all agents
```

Initially there is an implicit instance of the `Agent` class including the default turtle and the command line executor: when a sequence of commands is given by the user, this instance activates a new thread to execute the sequence. In such a way the sequence can also contain the `forever` (unconditioned cycle) primitive since the activating thread concurrently waits for a new command.

Another form of concurrent activation is the request from one agent to one (the `ask` primitive) or more (the `askeach` primitive) other agents to generate a thread executing a sequence of commands. The syntax is the following:

Table 3. The ask and askeach commands

`ask <agent>` `[<istr1><istr2>. . <istrN> ]`	`askeach [<agent1><agent2 >..<agentK> ]` `[ < istr1><istr2 >. . < istrN > ]`

3. A geometry library

In order to verify the usability of the OO component of the JxLogo language, a library of classes representing geometric figures on the plain has been developed. A teacher could work at various levels: assuming all the library available, he/she could ask students to construct more complex derived figures or compositions; he/she could instead start from scratch and follow the model of development suggested by the implementation of the library. In this description we start from an intermediate level, assuming three classes available (here not detailed): `Coord` that represents a couple of coordinates, `Color` that represents a colour and `Dir` that makes a transformation between the angle orientation system common in Logo (0 corresponding to the vertical axis, clockwise positive angles) and the goniometric system (0 corresponding to the horizontal axis, counter-clockwise positive angles). The three classes in particular define the method `get()` that respectively returns a list with the two coordinates, a list with the RGB components, a goniometric direction (the `Dir'getInternal()` method returns instead the Logo-compatible one).

Now we construct the base class `Shape` that includes some characteristics common to several types of figures, represented by `own` elements: a reference point (`basepoint`), the colour (`color`), a direction (`direc`), a string identifying the type (`shapeType`). As exemplified, the constructor body may provide a control on the acceptability of the constructor parameters.

```
class Shape [:xy Coord[]][:c Color[]][:d Dir[]]
    own [basepoint :xy      color :c    direc :d
      shapeType "UNDEFINED_SHAPE ]
    if not objectp 'basepoint "Coord [
       localmake "defBP Coord[]
       (show "Bad "Base-Point "format: "default "set :defBP'get())
       set 'basepoint :defBP
    ]
    . . .
end
```

For the `Shape` class some methods of general use are defined such as the setting of a new controlled value, and transformation methods such as `move()` and `rotate()`:

```
to Shape'rotate :ang
    'direc'add(:ang); sum the direction with a given angle
end
```

A `Triangle` class can be therefore derived from `Shape` as follows:

```
class Triangle:e1:e2:a [:xy Coord[]][:c (Color[])][:d Dir[ ]]
  isa Shape[:xy:c:o ]
    own [edge1 :e1          edge2:e2          angle:a]
    set 'shapeType "TRIANGLE
    set 'edge1 ABS 'edge1
    set 'edge2 ABS 'edge2
    if not numberp'angle[ . . . ] ; set default
    'draw()
end
to Triangle'plot
    localmake "median (sqrt('edge1*'edge1) + ('edge2*'edge2) +
      (2*'edge1*'edge2*cos 'angle))/3
    localmake "medianAngle arcsin ('edge2*(sin 'angle)/(3*:mediana))
    pu                      setpos 'basepoint'get() seth 'direc'get()
    left :medianAngle       bk :median              right:medianAngle
    pd                      fd 'edge1               pu
    localmake "vert pos     bk 'edge1               left 'angle
    pd                      fd 'edge2               setpos:vert
    pu                      right 'angle      setpos 'basepoint'get()
end
to Triangle'draw
    setpc 'color'get()      penpaint                'plot()
end
to Triangle'erase
    penerase                'plot()
end
to Triangle'setDir :d
    'erase()                old'setDir(:d)          'draw()
end
```

The methods `plot()`, `draw()`, `erase()` and `setDir()` show the use of the data elements in the class and the reusing of inherited methods. Methods analogous to `setDir()` can be defined in order to update `basepoint`, and `color`, in order to move and to rotate the figure, in order to modify its dimensions. Auxiliary methods can be defined for the calculation of its perimeter and area. From a geometric point of view, modifications applied to

the parameters of the figure produce, by means of cancellation and redrawing, the new graphical layout of the figure. For instance, the command:

```
make "t1 Triangle[70 70 30 Coord[100 120] Color[[255 0 0]] Dir[0]
```

draws in red an isosceles triangle with the barycentre (encounter of the medians) placed in (100,120), with the first of the two equal edges parallel to x axis (`direct` equal to 0) and the second edge at 30 degrees (in counter-clockwise sense).

With the same approach we have derived from Shape classes like Ellipsis, Parallelogram, RegularPoligon, Trapezium; from Ellipsis the class Circle; from Parallelogram the classes Rectangle, etc.. As an example for the Square class it suffices to add to Rectangle only a specific constructor (two equal edges); instead Rectangle masks the method `setAngle()` for himself and for any derived class (this method loses its meaning because of the geometric constraint on the angles of the figure).

4. Some relevant characteristics of JxLogo

The OO approach in JxLogo supports a natural progression of learning assuming an initial knowledge of standard Logo based on global variables and procedures. One first makes some simple extensions, even on single objects, modifying the characteristics of the turtle. For example it is possible to define an alternative to the forward primitive with the turtle drawing a dash line:

```
to :t1'forward :x
    localmake "rep int :x/10
    repeat :rep [
          old'forward(5) ('penup()) old'forward(5) ('pendown()) ]
    old'forward (:x-:rep*10)
end
```

At this level it is already possible to emphasize concepts like object instance, the visibility control of the elements, the association between a model and its procedures. Once the anonymous class is realized, it is possible to fix, through the automatic generation of the code (`classcode`), the complete definition of one new class which can be manually annotated with comments for documentation purposes. Then the visibility rules regarding data elements can be presented in a more complete way and the role of the parameterized constructor clarified. The class can subsequently be improved with other adjustments and/or new data elements and methods editing its source code. The new class becomes patrimony of the system, enriching the resources already available, and it can be later extended with one or more classes to realize specialization or particular behaviours. In these further classes students can be invited to try out more advanced aspects of the language (multiprogramming, polymorphism, masking, etc.).

JxLogo is still a project under development. The current version includes the usual 2D GUI for turtle moving end, on a separate window, a syntax-highlighted editor and graphical debugger able to effectively support the development of the applications. The online compiler, when requested, translates the JxLogo source of procedures, classes and methods directly in bytecode Java. Also the sequence of instructions given to the command line executor can be pre-compiled instead of being interpreted. When a method is added to a class or it is modified, the entire JxLogo class is compiled and it produces a new Java bytecode and therefore the class be reloaded into the JVM. Java instance objects are definitely associated with their class: when a JxLogo class is modified, an entirely new Java class is compiled and

loaded. This is why objects allocated before this modification continue to refer to the old class, still in memory, and they are not automatically adapted to the new definition. A similar behaviour is true also for anonymous classes.

We have been also carried out two interesting experimentations that reveal the flexibility of the language and of the JxLogo environment together with advantages coming from the power of Java.

In the first experimentation we defined a set of primitives, alternative to the usual movements of the turtle, aimed to command the RCX component of Lego Mindstorms™. We used an adapting software bridge towards the mobile component and we provided a robotics configuration that could realize on the plain the movements of a turtle ('roboturtle'). The JxLogo environment could interact with RCX during the execution of the program through the infrared communication. Elementary commands are sent and executed in real time (usually the robotics program is entirely downloaded onto the RCX and then autonomously executed). In order to maintain as permanent as possible the communication between JxLogo and the robot, we have used three infrared RX/TX turrets, placed at the vertices of an equilateral triangle around the active area of the robot.

In the second experimentation we tried to introduce in JxLogo the concept of a mobile agent: in our scenario an Agent instance should be able to migrate with (part of) its state towards another JxLogo instance, possibly on a different computer. We are therefore faced with the problem to transferring objects from one machine to another using standard mechanisms supported by Java. The preference went to SOAP (Simple Object Access Protocol) (Englander 2002), a specialized dialect of XML (Extensible Markup Language) to define a portable RPC (Remote Procedure Call) mechanism. SOAP is significantly simpler than complex and proprietary architectures like CORBA, RMI and DCOM, and it can use HTTP as normal transport protocol. Dealing with the transfer of Java object, the techniques used in the experimentation was Apache-SOAP[1] and XSOAP[2], with preference to the second due to its relative greater semplicity and affinity with the RMI mechanism. For the transfer of an `Agent` instance from M1 to M2, first M1 serializes the object, executes a lookup of the acceptance service in M2 and then executes one remote call to M2 to transfer the state of the object; in its turn M2 executes calls to a class loading service in M1 in order to obtain the code of the methods requested by the agent in the executive environment of M2.

5. Comparisons with some other Logo environments and conclusions

In the development of primitives in JxLogo a substantial compatibility with MSWLogo, due to its vast diffusion, has been adopted. Environments like TurtleTracks, NetLogo, StarLogo, and XLogo, though portable on various OS because written in Java, do not explicitly introduce extensions OO to the Logo language and they do not appear to adopt the approach of JxLogo with the in-line compilation of procedure. With respect to environments that integrate robotic commands (Microworlds, Terrapin Logo, WinLogo, YellowBrick Logo), in JxLogo a user with Java programming experience can realize personal extensions by easily adding Java classes to the system in form of new (or alternative) primitives, provided they implement pre-defined interfaces. As far as the OO component of the language is concerned, the comparison must primarily be done with Imagine Logo (Tomcsanyi, 2003) (also Object Logo and Elica include an OO extension). The marked difference is that in JxLogo we made a compromise between the syntactic/semantic accordance between the basic Logo and the OO extensions, that is one of the main characteristics of Imagine Logo (see for example the `newClass`, `<instance>'to` and `<class>'to` constructs), and the use of constructs more similar to those adopted by widely diffused OO languages. In particular the class

declaration recalls its generative nature, including the possibility of specifying instance parameters and a constructor. In our opinion this approach is more suitable for developing applications that must be well documented and debugged. Another relevant aspect in JxLogo is the role of the class Agent specifically devoted to concurrency, communication and synchronization, and also to the implementation of mobility, aspects that in Imagine Logo are more or less integrated in the Turtle class: we think that the Agent class manifests these specific aspects more explicitly when the user is acquiring a deeper knowledge in multiprogramming.

JxLogo will continue to be developed as a student project and we expect to investigate some further aspects like the handling at the JxLogo language level of events and exceptions, the enrichment of synchronization constructs and the improvement of the graphical interface.

6. References

Moro, M (2001), *JxLogo: A new Integrated Java-based Programming Environment for Logo*, EUROLOGO ’01: Proceedings of the 8th European Logo Conference, Linz, 21-25 August, pagg. 209-217

Englander, R (2002), *Java and SOAP*, O’Really (ed.)

Tomcsanyi, P (2003), *Implementing object dependencies in Image Logo*, EUROLOGO ’03: Proceedings of the 9th European Logo Conference, Porto, 27-30 August, pagg. 127-140

Web Sites

[1] http://ws.apache.org/soap/ Apache-SOAP official site

[2] http://www.extreme.indiana.edu/xgws/xsoap/ XSOAP official site