

Turtles with Roles: an application of an Object Oriented Pattern in JxLogo

Maria Vittoria Bedin, *bedinmar@gmail.com*

Bentec S.p.A. (Benetton Group), Via Villa Minelli 1, 31050 Ponzano Veneto, Treviso, Italy.

Michele Moro, *mike@dei.unipd.it*

Dept of Information Engineering, University of Padova, Via Gradenigo 6/B, 35131 Padova, Italy

Abstract

JxLogo is an undergraduate student project developed at Padua University and entirely realized in Java. It adds an object oriented extension to the Logo language and it integrates in the environment an effective graphical interface and useful tools. The introduction of this extension makes it arise some general questions about its educational motivations, as pointed out in the introduction. Then the paper discusses how the object oriented component can effectively support Logo users of different ages in learning complex concepts. The power of the proposed approach is showed through the introduction of the *Role* abstraction as a form of dynamic inheritance, in contrast with the traditional static inheritance based on derivation between classes. Such an abstraction is introduced through the known Role Object Pattern: two implementations of this pattern are presented, one that uses the *JxLogo* language constructs, and another based on the *TurtleRole* class that is provided at low level. Both levels of implementation permit to add transitory behaviours and attributes to the *Turtle* basic class. Other relevant elements of the environment are also presented, specifically how events are currently supported. Finally some didactic motivations try to answer to the questions aforementioned and to justify the introduction of advanced programming issues as accessible learning options.



Figure 1. The JxLogo online help

Keywords

Logo, JxLogo, Object Oriented, Role Object Pattern, Dynamic Inheritance

Introduction

Logo is a well known learning tool (Papert, 1980) (Papert, 1999) where relatively simple graphical actions (the turtle geometry) coexist with more complex concepts like recursion, interactivity, multithreading. The procedural approach of its language has been proved sufficiently expressive to support pupils during their exploration of new concepts. With respect to the past, recently developed Logo environments pay more attention to the graphical interface adding features such as icons, animated elements, control components etc., but in the Logo history the general architecture and the operative modes have remained more or less the same as the original ones. Actually only in a few cases significant extensions to the language have been introduced to empower its expressiveness: the objective is to allow skilled users to develop projects with new structured, complex abstractions. One of these extensions is the object oriented (OO) component (Boychev, 1999) (Tomcsanyi, 2003) (Kanemune and Kuno, 2005) (Lehotska, 2005) which is the validated paradigm that characterizes most of the widespread general purpose languages such as Smalltalk, C++, C# and Java. This paradigm introduces a relevant number of benefits in terms of software robustness and reliability, software reusing, and a more precise resembling of the real world.

Now some general questions arise: is the OO approach too complex for pupils involved in Logo projects? Does it obscure the clearness and the straightforwardness of the Logo procedural structure? What kind of didactic motivations should the introduction of the OO component in Logo suggest? Finally, at what levels should OO programming be exploited?

As mentioned above, the OO approach can more precisely describe entities expressing characteristics of the real world where 'objects' act according to their properties, with some exposed and some hidden attributes, with some described and some not known behaviours, objects which can be activated through well defined interfaces. In a Logo system the turtle, which is its main entity, may be seen as an object and its model (class) may be used to create different instances in a multiturtle scenario. In this sense a proof consisting in a complete hierarchy of classes describing elementary geometric shapes and their dependencies was proposed in (Moro, 2005). Moreover, the integration of the OO component must be realised preserving the native procedural approach of the Logo language, in order to provide a smooth progression conducting the user to fully exploit the OO features like inheritance and polymorphism.

JxLogo was already presented in previous Eurologo conferences (Moro, 2001): it is an undergraduate student project developed at Padua University and entirely realized in the Java language. Its goal is twofold: it is intended on the one hand, to encourage experimentations and usage of Java technologies by university students, on the other, to implement a complete and innovative environment to support advanced Logo programming. Starting from a strict compliance to MSWLogo for the name and meaning of the basic primitives, JxLogo includes the OO component as an actual extension of the traditional Logo language. The hoped progression in learning complex structures derives from the attention paid during the design of the language and partly from the interpreted nature of the language.

This paper, after a brief description of the main actual aspects of the JxLogo environment, presents some recent improvements that can enforce the justification of introducing a OO programming style in a Logo environment. Then the concept of *Role* is explained as a form of dynamic inheritance which can be used in all those cases when temporary capabilities have to be added to already created objects. JxLogo has sufficient expressiveness to easily implement this concept but some reasons to introduce the same mechanism at a lower level for the important turtle object are also presented. A section deals with events which are other recent empowering elements giving the turtle more reactivity to external stimuli. The last section contains some final remarks and ideas for future work.

The JxLogo environment

In order to give classes a more incremental development, in JxLogo the class construction is divided into two separate steps. In the first one we declare the attributes and the single constructor: the initialization code is quite similar to a usual Logo procedure body. In the second step methods are separately defined and compiled so that they can be added to the class one-by-one in a way similar to the adding of new procedures during a Logo session. Some syntactic extensions have been defined to represent hierarchy aspects, visibility rules, assignments and references to the class attributes and methods (for more details and explanations regarding the syntax that JxLogo uses to manage objects refer to (Moro, 2001) and (Moro, 2005).

The following explanatory example defines a specialised turtle with an overridden `forward` method and the specific method called `square` (the added comments help for an intuitive understanding):

```
class newTurtle [:x 100][:y 100] isa Turtle[:x :y]
  ; [100 100] is the default position of the turtle
  ; :x and :y are passed to the superclass constructor
  ; own attributes: globally visible only for reading
own [initialX :x
initialY :y
distance ^]
  ; set: the keyword to assign values to attributes
set 'distance sqrt sum :x*:x :y*:y
end

to newTurtle'forward :m    ; forward :m*20 pixels drawing a dash line
  repeat :m [
    old'forward(10) ; it refers to the inherited version of forward
    'penup()
    old'forward(10)
    'pendown()]
end

to newTurtle'square :m    ; old-style square
  repeat 4 [old'forward(:m) 'rt(90)]
end
```

To create an instance of the class the programmer has to call its constructor through the name of the class giving the required arguments enclosed between square brackets (some of them can be optional and therefore defaulted); the reference of the created instance can be assigned to a variable with the usual `make` command:

```
make "t1 newTurtle[150 150]
```

The class construct leads to a static description of derivable data types, whereas the dynamic approach of the Logo language is maintained with the on-the-fly declaration of methods. JxLogo introduces another kind of 'dynamic' extension of a class in form of the extension of an already created object. In the following example:

```
make "t2 (:t1[hidden [step 1]])
  ; t2 is derived from t1 with a new hidden attribute (step)
```

```
to :t2'setStep :s      ; a method defined on t2
  set 'step :s
end

to :t2'forward :m     ; another method defined on t2
  old'forward(:m * 'step)
end
```

an anonymous class is derived from `newTurtle`, the class of the `t1` instance, with the new hidden attribute `step` (visible only within the new class) and a single instance of this new class is created and represented by the `t2` variable. The state of the inherited attributes is copied from the original `t1` object to the new `t2` object. It is also possible to use a similar construct directly referring to the superclass:

```
make "t2 (newTurtle[200][hidden [step 1]])
```

In this case the declaration can include some or all the constructing arguments.

To transform JxLogo into a suitable Logo environment, able to support and help the user during the entire work session, two kinds of editing tools have been defined together with a online help system. In fact JxLogo recognizes two different types of users:

- children and very young students using Logo as a learning tool, who need an easy, handy environment suitable to their limited abilities and focused interests;
- advanced older users ready to face the object oriented programming challenge, in order to experiment relative complex projects and to improve their programming skills.

For the first type of users an elementary editor to code simple Logo procedures is almost enough, together with a command line where it is possible to invoke a list of instructions. The second type of users are invited to take advantage of the full potentiality of the system that includes an integrated editor/debugger to code and test procedures, classes and methods.

Currently JxLogo provides a total of more than 250 primitives and several built-in classes whose interface and semantics must be known by the user for an effective programming. The need of a complete online documentation is therefore mandatory. We decided to implement an ad-hoc tool, using the Sun Microsystem JavaHelp: an XML structure for the help files has been defined in order to separate the content management from the contents themselves (Figure 1).

Dealing with Roles

In object oriented programming abstractions like specialization and adaptation may be easily implemented using the class derivation paradigm. This is a static declarative construct by which hierarchically organized models may be defined and instantiated. In this scenario each object is univocally associated with one generating class during all its life.

Basically Logo tends to encourage users to incrementally define new elements (variables and procedures) that can be coded and tested on-the-fly exploiting the interpreted nature of the language. In JxLogo this characteristic was improved when the object oriented component was defined. In fact, even if the class header includes data elements and initializing code, we saw that methods can be incrementally added to the class. When a new method is defined, a new class version is compiled and stored in memory: thus objects created before this modification are associated with the old version of the class whereas objects created hereafter refer to the new one. Even when a new anonymous class is created and compiled, the object 'extending' the old one is an instance of the new class. These three approaches to extend a class (usual class derivation, method adding, anonymous class) are more or less static constructs which lead to permanent models which do not allow removing of previously added components and maintain the strict relation between an object and its class.

Notwithstanding, there are situations when a greater dynamism is required: new components and behaviours should be temporarily added to an object in order to satisfy specific runtime requirements. Well-known and immediately understandable examples are all those situations in which an object describes the attributes of a person (or an equivalent entity) who enters an environment where new capabilities are assigned and/or required. These capabilities represent the concept of *Role* and specific architectures are proposed in literature to support effectively this concept (Fowler, 1997) (Baumer, 1997) (Kendall, 2000). A role summarizes capabilities used temporarily and under conditions, and it must express a kind of dynamic inheritance, i.e. a part or all the behaviours of the original object can be assumed by the object even when it plays one or more specific roles. New or modified behaviours may be related to its current active roles and, generally speaking, they could depend on the context in which they operate.

Roles implemented in the JxLogo language

In order to show that JxLogo is sufficiently expressive to support one of the most flexible patterns for roles, the so called *Role Object Pattern*, proposed by Fowler, we present a demonstrative example referring to a turtle object which has well-known basic behaviours in Logo. In this pattern the original object (and its class) acts as a *core object* which includes some general functions to manage roles defined as separate classes. A core object may be dynamically linked to a set of separate role objects: each one, as explained above, describes specific capabilities. To implement the required form of inheritance, all the functions, which are available in the core object and are chosen as functions to be inherited by its possible roles, must be applicable to each role object. This is obtained applying a simple delegation, i.e. a role class must implement each 'inherited' method calling the homonymous method in the core object, apart from a possibly added specific code.

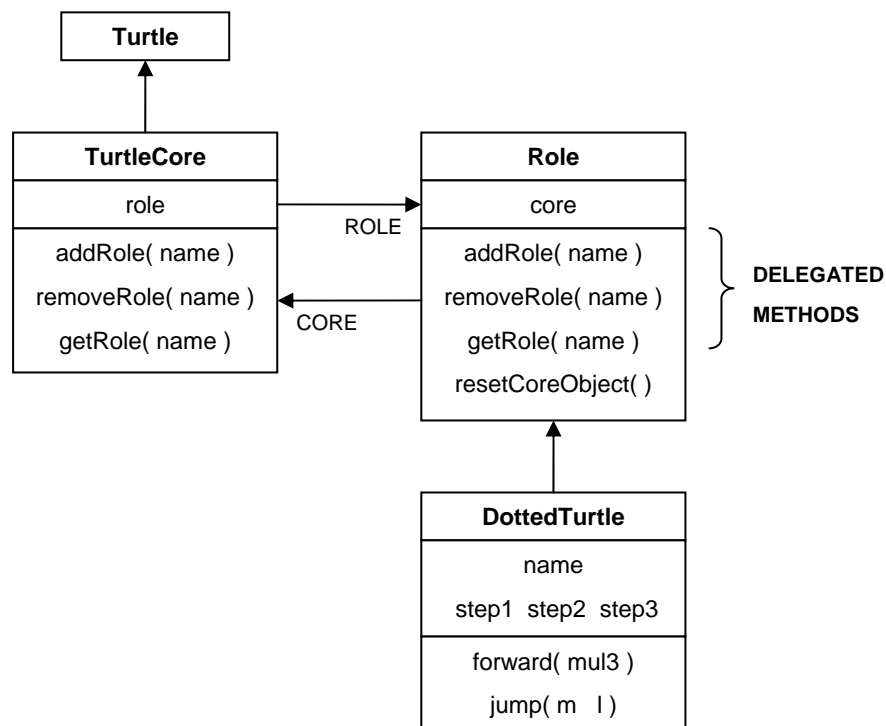


Figure 2. The role example structure

Considering that in a Logo environment the turtle is the main graphical component with proper pre-defined functions, it is reasonable to think to transforming a turtle object into a core object and to implement some special movement behaviours as instances of different classes. More in detail, in the following example the class `TurtleCore` extends the base built-in `Turtle` class,

and defines itself as a *core class* including three general managing methods for roles (`addRole`, `getRole` and `removeRole`) of straightforward meaning (Figure 2). Without loss of generality, instead of a list of roles, the class includes a reference to a single role object. Moreover, for the sake of simplicity, the `addRole` method manages directly all the allowable roles instead of relying on a suitable data structure where to register them in a configuration phase (notice that classname symbols, like `DottedTurtle`, may not be passed as parameters because they must be translated at compile time, thus the role name is provided to the method as a string parameter). Delegated methods in each *role class* should be the same three managing methods (e.g. adding on a new role to a previous role means adding it to the associated core object) plus other 'inherited' methods, in this example, the `forward` method. Considering that the role managing methods are completely general, they are included in a base `Role` class which must be extended by all the specific role classes. The `DottedTurtle` class defines a different `forward` method drawing a dotted (dash) line instead of the usual straight line.

```
class TurtleCore :x :y isa Turtle[:x :y] ; compulsory coordinates
  own [role "null] ; the 'list' of roles (actually at most one role)
  (show "The "current "role "is 'role)
end
to TurtleCore'addRole :name ; the role name
  if (equalp "DottedTurtle :name) ; code to recognize every assumable role
    [set 'role DottedTurtle['this]]
    . . . ; similar for other roles
end
to TurtleCore'removeRole
  'role'resetCoreObject() ; if necessary
  set 'role "null ; remove association with the role
end
to TurtleCore'getRole
  output 'role
end
```

To support the delegation to the associated core object, the `Role` class contains a reference to the core object, which is cleared when the role is removed.

```
class Role :core ; the associated core object
  hidden [corRef :core]
end

; delegated methods
to Role'addRole :name
  'corRef'addRole(:name)
end
to Role'removeRole
  'corRef'removeRole()
```

```

end
to Role'getRole
  'corRef'getRole()
end
to Role'getCore
  output 'corRef
end

      ; specific methods
to Role'resetCoreObject
  set 'corRef "null
end

class DottedTurtle :core isa Role[:core]      ; a specific role
  own [name "DottedTurtle
    step1 10
    step2 5
    step3 2]
  (show "Added "the "role 'name)
end

      ; delegated methods
to DottedTurtle'forward :mul3 ; triplicate the step?
  if (not equalp :mul3 0)[set 'step3 3*'step3]
  repeat 10 [ ; ten separated segments of step3 units
    'getCore()'forward('step3)
    'getCore()'penup()
    'getCore()'forward('step3)
    'getCore()'pendown()]
end

      ; specific methods
to DottedTurtle'jump :m :l ; new position
  'getCore ()'penup()
  'getCore ()'setx(:m)
  'getCore ()'sety(:l)
  'getCore ()'pendown()
end

```

The following list of instructions:

```
make "t1 TurtleCore[10 20] ; a new turtle in [10 20]
```

```
. . .  
:t1'addRole("DottedTurtle) ; add a 'dotted' behaviour to t1  
make "r1 :t1'getRole()  
:r1'forward(0) ; dotted line with step = 2  
:r1'forward(1) ; dotted line with step = 6  
:r1'jump(40 -100) ; jump to a new position  
:r1'removeRole() ; the role is unlink from its core  
:r1'forward(0) ; run-time error: the delegated method is no longer applicable
```

creates a TurtleCore instance and, in different moments, add a role, use that role for delegated and specific activities, and eventually remove it from the core object.

The example shows that the role becomes a sort of second skin that empowers the original object during the period in which the role is active: once the object gives up the role, it gives up everything new or different the role represents. The method delegation is the price to pay to 'simulate' the inheritance of the 'superclass' (core) methods.

Roles implemented within JxLogo

Other Logo environments allow a form of dynamic association of specific behaviours with the fundamental turtle object (Tomcsanyi, 2003). The role pattern reveals a greater power because several roles can be added to an object and singularly referenced by means of a unique identification (e.g. the role class name). Therefore, even if the JxLogo language provides the sufficient expressive power to implement the Role Object Pattern on generic objects, there has been the convenience of implementing a built-in structure to effectively define roles for a turtle. Such a structure is based on the availability in the system of the built-in TurtleRole class which includes all the role managing methods, present also in the built-in Turtle class, and the basic turtle methods (forward, back, penup, etc.). The user can easily derive its own role classes from TurtleRole specifying new turtle behaviours and/or modifying its default behaviours. This structure seems to positively support both multiturtle interactions in complex systems and the typical enrichment of a turtle 'agent' when the mobility from one machine to another will be provided.

Named again DottedTurtle the role class that defines the new version of the forward method for drawing dotted lines, the following instructions, using the built-in class, create a turtle, add this kind of role, use it and then remove it (Figure 3). Notice that, when a first instance of the class derived from TurtleRole is created, the system registers a 'prototype' that can be successively used to add this role to core objects by means of its symbolic name string.

```
class DottedTurtle isa TurtleRole[]  
. . . ; same definitions as in the previous example  
. . .  
make "t1 Turtle[100 100]  
:t1'forward(90)  
:t1'right(90)  
make "r1 DottedTurtle[] ; a DottedTurtle prototype is registered  
:t1'addRole("DottedTurtle) ; the role is added  
make "r1 :t1'getRole("DottedTurtle) ; set a reference to the role  
:r1'forward(90) ; 'dotted' forward  
:r1'left(90) ; equivalent to :t1'getRole("DottedTurtle)'lt(90)  
:r1'forward(90)
```

```
:r1 'removeRole( "DottedTurtle)
:t1 'left(90)
:t1 'forward(90) ; normal forward
```

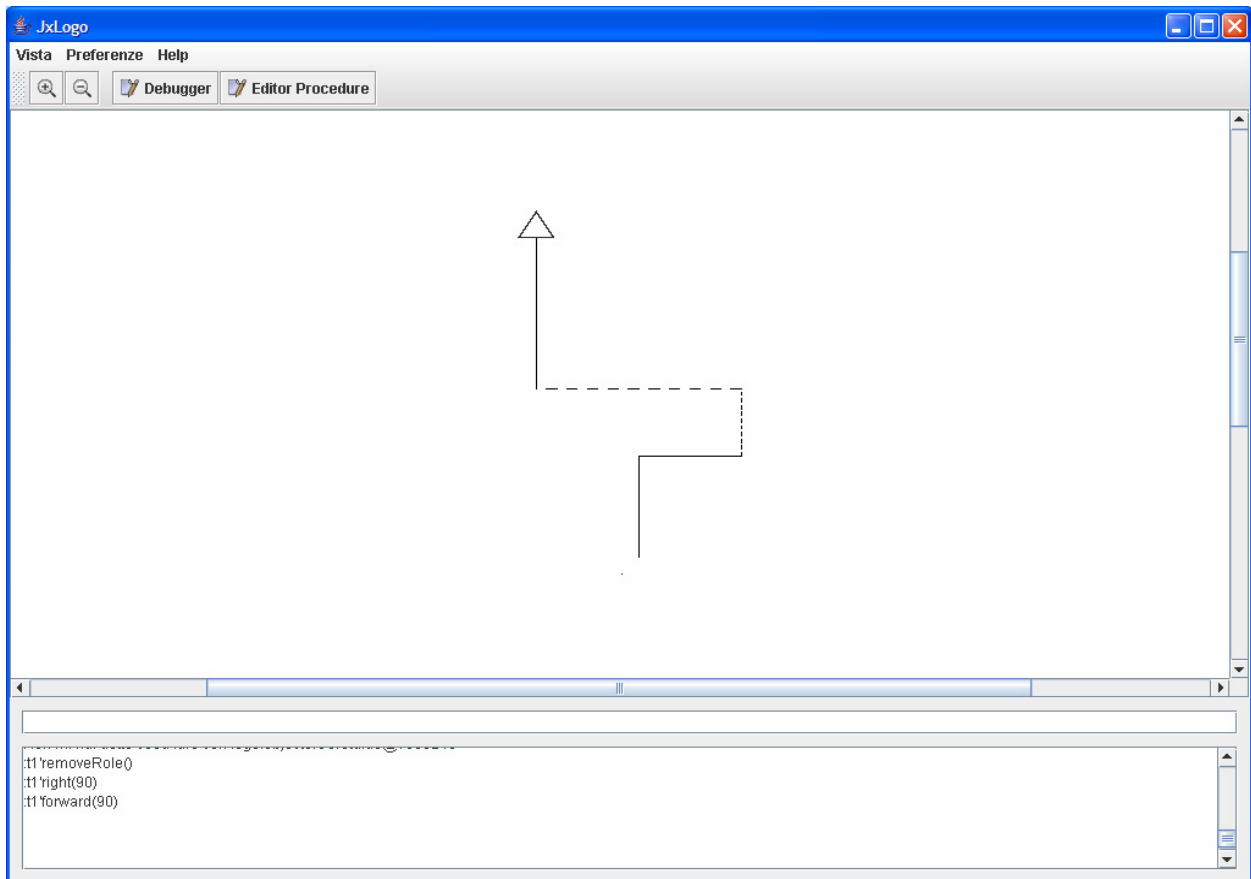


Figure 3. Use of a role: during the section in which the turtle is associated with the 'dotted' role, its behaviour is specific.

Events

In JxLogo the graphical interface is a relevant key aspect of the environment: the major part of the user's attention is devoted to graphical objects and interactions. The turtle is the most important active element with a lot of commands related to the drawing facilities of the system. The essence of the turtle as an active object is guaranteed by the programmability of its preordered actions, whereas the responsiveness to asynchronous events (the *reactive* turtle) may be delegated to the definition of *callback* actions to be performed when the user interacts with the graphical environment during the execution of the program. A third vision of the turtle as a *proactive* entity is connected with the notion of agent and it will be subject for future studies. For example, in the realization of a game application, the user could either point and click with the mouse on one of the defined turtles or press an arrow key on the keyboard to force special behaviours while those turtles are moving and drawing within their operational area. These considerations led to the introduction of *event management* that, together with the role support, provides a powerful tool for behavioural control.

The management of events is limited to the turtle because this is the only recognizable object for which the meaning of events is immediately understandable. It is realized using the low-level Java events: the turtle becomes the source of the event, the callback function is implemented as a method of a dedicated object (listener). Such a method has a predefined interface: when an event takes place, only if the specific callback has already been registered within the graphical

object that represents the turtle, the foreseen reaction is executed; otherwise nothing happens. The declaration of a callback function is actually done in the form of a list of instructions that is to be executed when the specified event is received by the associated turtle. For example, if `t1` is a turtle instance, after the following command is executed:

```
:t1'setEvent("MouseClicked  
    [t1'penup() :t1'fd(40) t1'pendown()])
```

any time the user clicks with the mouse on the graphical representation of that turtle it jumps 40 units ahead.

The list of the event names currently registered within a certain turtle can be obtained with the built-in Turtle method `events()`. Another Turtle method, called `sendEvent("eventName")`, may be used on the command line to test the action registered for the *eventName* event.

The association of the callback function is not permanent and it may be successively removed or redefined:

```
:t1'setEvent("MouseClicked [t1'rt(45)])  
. . .  
:t1'removeEvent("MouseClicked")
```

Reasonably the model enables the user to operate with predefined events (some of the events supported by Java) but a similar approach may be easily extended to user-defined events seen as a form of very simple asynchronous interactions between turtles. In other words, usually a method applied to a turtle is executed by the only thread which has the responsibility to control that turtle on the basis of the defined program. Instead the `sendEvent()` method may be executed by a different thread that wants to force the controlling thread to execute the associated callback action. Accordingly *sendEvent* may be used in the command line to test also actions for user-defined events.

Discussion and final remarks

The objectives reached by the current version of the JxLogo environment and the experimentations carried out until now reveal some agreeable answers to the questions posed in the introduction of the paper. The complexity is maintained under control proposing simplified declarative structures but at the same time allowing users to exploit all the advanced features of the OO programming. The teacher can choose at what level making the experimentations (basic Logo, anonymous extensions, traditional class inheritance). The basic Logo level is in any case available with some facilities regarding the compatibility with other systems and a national language support. Therefore it is the starting point to explore more advanced concepts under both programming and application point of view. The OO programming style may improve the correspondence between the application and the real world (or the real personal experience) through a progressive learning path that guides young users to a more professional approach to programming. Consequently its level depends on the user's age, on the kind of projects, and on the teacher's didactic aims.

Roles underline the usefulness of the OO component in a Logo environment even in the case of a complex goal like defining data types in a dynamic hierarchical structure. At the same time the built-in support of roles for turtles allows a user to manage 'dynamic classes' without been bored with technicalities regarding the role pattern. Moreover roles represent a substrate to support an agent-based view of the turtle where interaction-related issues are developed within roles whereas algorithmic-related actions are included in the core objects (Cabri et al., 2005).

Future developments should therefore regard the transformation of turtles from simple drawing atoms to movable, proactive entities able to interact in a controlled way both with other turtles in the same machine and with other turtles or other Logo objects in different machines where the original turtle has been moved. In this sense some experimentations have already been carried

out in terms of multiturtle interactions and of using standard protocols to support the turtle migration, which may be improved in the near future. The JxLogo experience has also been offered as a experimental platform for educational robotics in the scope of the recently started TERECoP (Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods) project, a Socrates Programme, Comenius 2.1 Action (Training of School Education Staff).

References

- Papert, S. (1980) *Mindstorm, Children, Computers and Powerful Ideas*. Basic Books, New York.
- Bäumer, D., Riehle, D., Sibersky, W., Wulf, M.: *Role Object*. In: Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois (1997).
- Fowler, M. (1997) *Dealing with Roles*. In Collected papers from the PLoP '97 and EuroPLoP '97 Conference, Technical Report wucs-97-34, Washington University Department of Computer Science (<http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/>).
- Baumer, D. and Riehle, D. and Siberski, W. and Wolf, M. (1997) *Role Object*. In Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois.
- Boychev, P. (1999) *Elica Logo and Objects*. In Proceedings of the 7th European Logo conference, Sofia, pp. 160 - 168.
- Papert, S. (1999) *Introduction: What is Logo? And Who Needs It?* In Logo Philosophy and Implementation, LCSl, Montreal, pp. V-XVI.
- Kendall, E. A. (2000) *Role Modelling for Agent Systems Analysis, Design and Implementation*. IEEE Concurrency, 8(2), pp. 34 - 41
- Moro, M. (2001) *JxLogo: A new Integrated Java-based Programming Environment for Logo*. In: Proceedings of the 8th European Logo conference. Edited by Futschek G., Linz, pp. 209 - 218.
- Tomcsanyi, P. (2003) *Implementing object dependencies in Imagine Logo*. In Proceedings of the 9th European Logo conference. Edited by Cnotinfor, Porto, pp. 127 - 140.
- Lehotska, D. (2005) *Advanced programming classes in Imagine*. In Proceedings of the 10th European Logo conference. Edited by Gregorczyk G. and oth., Warsaw, pp. 154 – 163.
- Kanemune, S. and Kuno, Y. (2005) *Dolittle: an object-oriented language for K12 education*. In Proceedings of the 10th European Logo conference. Edited by Gregorczyk G. and oth., Warsaw, pp. 144 - 153.
- Cabri, G., Ferrari, L., Leonardi, L. (2005) *Injecting roles in Java agents through runtime bytecode manipulation*. IBM System Journal, Vol. 44, no. 1, pp. 185 - 208.
- Moro, M. (2005) *Object Oriented programming and development in JxLogo*. In Proceedings of the 10th European Logo conference. Edited by Gregorczyk G. and oth., Warsaw, pp. 132 - 143.