

Natural Object-Oriented Programming: The Evolution Metaphor In Elica

Pavel Boytchev

Dept. of Information Technologies
Faculty of Mathematics & Informatics
University of Sofia
5, James Bourchier blvd
1164 Sofia
Bulgaria
elica@fmi.uni-sofia.bg

Abstract. OOP is a collection of concepts, tools and styles that are supposed to help programmers build more complex programs and support them easily. However some OOP benefits become disadvantages. The Natural OOP (NOOP) in Elica, does not use any specialized syntax to utilize OOP. The evolutionary metaphor applied to NOOP changes the focus of OOP by treating classes and objects as species and individual creatures. The paper confronts activities done in a NOOP-based program with evolutionary processes like inheritance, birth, mutation, life, and death.

OOP vs. NOOP

Object-Oriented Programming (OOP) is a collection of concepts, tools and styles that are supposed to help programmers build more complex programs which are easier to support. Although there are many benefits in using OOP, there are cases when those benefits vanish [6, 9]. A typical example for the failure of OOP is its implementation in languages that are simple by design and are intended to be used by non-professional programmers – like students, teachers, and researchers. Unlike professional programmers software amateurs are frightened when they have to learn and apply OOP.

A possible reason for this might be the fact that OOP is often introduced at the cost of changing the syntax of a language, or at least, adding numerous new commands and functions that must be used if one wants to use OOP [10]. The situation is even worse, because beginners are flooded with a list of new words like classes, objects, instances, methods, fields, inheritance, etc [8].

In attempt to address these issues, Elica (Educational Logo Interface for Creative Activities, [5]) uses a non-traditional OOP approach, called Natural Object-Oriented Programming (NOOP). Elica is a modern dialect of the programming language Logo.

In communities of low-level programmers, the word NOOP is usually has the meaning of “No-Operation”. In the context of Elica ideology, NOOP has almost the same meaning – users do OOP without OOP. The implementation of NOOP does not require the study of special new commands [2]. What is needed is some knowledge of the well-known Logo commands like MAKE, LOCAL, RUN and TO...END.

There are few concepts behind NOOP. Historically, they were not articulated in advance, but evolved together with the development of Elica [3]:

- There are no specialized data structures for OOP elements.
- There are no special commands and functions to manage OOP elements.
- Every variable is local to another variable, except for the root variable.
- Activities are always done inside a variable, called current variable.

The variable in Elica is the core element, which properties make it possible to have NOOP. Namely, Elica's variables can be defined at runtime and can contain local variables. A function is also represented as a variable. During its execution all its locals are defined as variables local to the function's variable.

Consider executing the commands:

```
MAKE "TOM.AGE 10
MAKE "TOM.HEIGHT 150
MAKE "TOM.HEIGHT.METRIC "CM
```

They create a variable TOM with local variables AGE and HEIGHT. Nested variables are accessed by the dot-selector, which is used in Delphi, C, C++, Java, etc.

Elica objects

A key feature of NOOP is that almost every aspect of OOP can be represented in a natural way. In Elica an object is a group of variables inside another variable. A description of an object is called class. Variables inside an object are its fields. Local functions are called methods. When the description of one object uses the description of another object this is inheritance.

Metaphor 1: NOOP is like an evolution. If classes are species, then objects are the individual creatures. Their properties like height and weight are fields. Their behavior is expressed in methods. The inheritance corresponds to the genealogical relations between species.

To demonstrate how NOOP works, we will focus on several common activities: object creation, destruction and identification.

Creating objects

Every object is a variable with zero or more nested variables. Following the evolutionary diversity, Elica can create an object in four different ways. It is just a matter of programming taste which one to use.

Metaphor 2: Creating an object corresponds to giving birth to a creature.

Creating by assignment. The most straightforward way to create an object is to copy its structure directly from another already existing object. Assume we want to create PETER as a copy (clone) of TOM. The command to do this is:

```
MAKE "PETER :TOM
```

which is exactly how variables are assigned. After the execution PETER will have the same age and height as TOM. When Elica assigns one variable to another it duplicates not only its value, but also all local variables (if any).

Metaphor 3: The most primitive creatures on Earth reproduce themselves by cloning. Technically, this means creating a creature as an exact copy of another one. Naturally both creatures are from the same specie and are indistinguishable unless later on one is changed for some reason.

Creating field by field. A little bit more flexible method of creating objects is to explicitly say what fields and methods are in it – i.e. the object is created field by field. We have already described this method when creating TOM.

To define object's behaviour we need to provide local functions. Because of Elica's flexibility, there are more than one ways to do this, and each way is natural. We can use the standard for Logo syntax mixed with the dot notation. The example below will add method HELLO to PETER:

```
TO PETER.HELLO
  PRINT [Hello World, I am Peter!]
END
```

Alternatively we can create the method treating it as a variable:

```
MAKE "PETER.HELLO [ PRINT [Hello World!] ]
```

Metaphor 4: Explicit creating of an object is like playing God. The user has a total control over her Elica-based microworld and can create creatures by saying their properties and behavior.

Creating by modification. Sometime we need to create an object that is almost like another object. The easiest way is just to clone the first object and change some of the variables. For example, if we want to make object ANA that is the same as TOM, but has an additional field EYES=BLUE, then here is how to do it:

```
MAKE "ANA :TOM
MAKE "ANA.EYES "BLUE
```

Metaphor 5: Creating by modification corresponds to mutation. Sometimes mutations are so small, that the effect is invisible. However, mutations could change the appearance and the behavior of a creature to the extend that it looks as if from another specie. Sometimes mutations are fatal or even lethal. Be careful when playing God.

Modifying objects made by other users might be dangerous especially if you change important fields. However, object modification is powerful because it allows to completely and gradually change an object.

Metaphor 6: Object's fields are creature's organs. Destroying some of them may make the object invalid. Some modifications, however, are good and lead to improvement. All depends on the internal structure and the environment where the object lives.

With the command MAKE we can add and modify a field, with the command DELETE we can delete a field. If we delete AGE and HEIGHT from ANA it will become totally different from TOM, although it was cloned from TOM.

Creating from classes. The method of creating objects from classes is the traditional method in OOP. And usually it is the only one. For now let's assume we have a class HUMAN with two initializing parameters for the age and the height. Let's create two other objects using the HUMAN class:

```
MAKE "PAUL HUMAN 15 160
MAKE "MARIA HUMAN 21 175
```

Metaphor 7: Using classes to create an object is like giving a birth to a new creature from specific specie that determines its features. In a more religious context, creating from classes is more close to incarnation. The result of executing the class is the spirit of the creature that is later incarnated into a variable-receptionist.

The most important thing here is that defining an object from a class is just like calling a function. There is no any specific syntax niceties. Even the definition of the class looks like a normal subroutine – this is the unique way of NOOP to handle classes without changes in the core language syntax.

Let's now define the class HUMAN:

```
TO HUMAN :AGE :HEIGHT
  MAKE "HEIGHT.METRIC "CM
END
```

It looks like a procedure – there is no any result which is returned to the caller. How could this be a constructor of an object? Let's see what happens when HUMAN is called as a function. It first creates an empty run-time object and local variables AGE and HEIGHT in it, which will be initialized by values passed as parameters, then adds METRIC. And that's all. We have two local variables in the variable of the executing the routine.

The twist here is that HUMAN is called as a function, but there is no any OUTPUT command. In this case, Elica returns back the run-time object which now has AGE and HEIGHT. The returned object is assigned to PAUL or MARIA, which become HUMANS.

Metaphor 8: There is nothing closer to mammal's pregnancy than this. Embryos are being developed (a kind of constructed) inside the parent, and the delivery is done by moving the new body outside the mother.

The preferred way to define classes in Elica is just to define procedures that are used as functions. Most elements of such procedures can be related to the OOP terminology. The procedure itself is the constructor, the inputs are initialization variables, the local variables become fields of the created objects, and the local routines become methods.

But that's not all. Elica NOOP can represent other OOP-related stuff, like polymorphism; single and multiple inheritance; virtual, overloaded, inherited and overwritten methods; virtual and static fields; and so on.

Actually there is one thing that was decided that NOOP cannot do – i.e. to define access privileges to fields and methods. In Elica the user can inspect and change every local variable, because this is in the educational spirit of the Logo programming language.

Destroying objects

Whatever objects the user creates, it comes the time to destroy them. Destroying is needed mainly to free memory resources. Generally there are two ways to do destroy a variable – automatic and manual.

Automatic destruction. When a variable is destroyed (whatever the reason is), all its local variables are automatically deleted. Because every execution of a procedure uses something like an object (i.e. the stack for local variables) when the procedure's execution is finished, Elica destroys this object and all its local variables.

Metaphor 9: Objects live in the execution context of the routine where they are defined as local. This execution context can be treated as the life line of the creature. When the execution terminates, the creature dies by natural cause.

Manual destruction. In some cases it is needed to destroy an object at a specific moment, rather than to wait the end of the execution. One of the most frequent reasons to delete an object is when the object is graphical and its image must be removed from the screen.

The manual destruction is done by the DELETE command demonstrated above. Note that DELETE is not OO-specific. It just removes a variable from the memory.

Metaphor 10: Manual destruction is de facto a murder. Period.

Class identification

After an object is created it is impossible to find out its class. This is so because objects can be modified at any time, and because no information is stored in the object as to what is its class. This can be better explained by one conceptual decision about Elica objects. When they are created from classes, all definitions from the class are copied to the object and it becomes effectively independent of the class.

Metaphor 11: This is how things work in Nature. When a creature is born, its parents' DNA is copied, so that the child is independent. In the OOP style, no DNA is copied, just a link is created to the parent. Indeed, the link is to the class, which plays the role of a common, virtual and spiritual DNA source. Such species are found only in SF books – they are insect-like brainless bugs with collective society-based memory.

There is one simple explanation why definitions are copied, rather than just pointed. Variables in Elica memory are isomorphic. And there is no way to say whether an object is created by cloning or from a class. However, there is a neat solution to this problem. The object may keep in a local variable the name of its class.

Even if such a variable is not created, it is possible to find the class (if any) by inspecting what local variables exist in it.

Metaphor 12: The two methods for distinguishing objects match real life extremely well. A local variable containing the name of the class corresponds to creatures that know what they are, like we, humans, do, and possibly dolphins and monkeys too. The other method, recognition by properties, is related to more primitive creatures that recognize themselves by the smell, for example.

Other properties of objects

It is not possible to describe in details all features of Elica's NOOP. However, this section will enumerate briefly some of them.

The polymorphism of Elica NOOP is not a special feature, but a core functionality inherited from its Logo roots. Logo is a weak-typed language, and this applies to objects too. When an object is constructed, the initialization parameters have undetermined type. The object constructor decides at run-time how to use them¹.

The class inheritance is implemented by “inserting” the definition of the parent class in the body of the child class [7]. Assume we want to define classes MAN and WOMAN based on the class HUMAN, but having one more field: SEX. The natural way to do this in Elica is:

```
TO MAN :AGE :HEIGHT          TO WOMAN :AGE :HEIGHT
  RUN :HUMAN                  RUN :HUMAN
  LOCAL "SEX                  LOCAL "SEX
  MAKE "SEX "MALE            MAKE "SEX "FEMALE
END                            END
```

RUN is a standard Logo command for executing a list of command. The :HUMAN expression above extract the value of HUMAN as a list of commands.

This is not the only type of inheritance in Elica. A unique feature is the conditional inheritance where at run-time it is decided what object to be a parent. Parent-child relationships are not fixed and can be established and removed during program execution. This allows to supply several parents to a class, so the so called multiple inheritance is also supported.

¹ Example for polymorphism is the graphical library Geomland. The object LINE can be created by two POINTs, by a POINT and an ANGLE.

Metaphor 13: Single inheritance corresponds to monosex species, inheritance from two parents – that's the current top of evolution. It is difficult to imagine any interpretation of multiple inheritance and the weird conditional inheritance. May be they correspond to adoption?

In traditional OOP there are various types of methods and fields. Without adding new syntax structure and reserved words, Elica can support in a natural way overwritten, abstract and virtual methods, static fields, event handlers (for handling mouse, keyboard, and graphical events).

However Elica goes beyond this. NOOP objects are used to implement many other entities, like multidimensional, fractal-dimensional and associative arrays, sets (unordered collections of data), OLAP and relational databases, and libraries. The implementation of each of these entities deserves a separate paper [2]. Let's take, for example, the libraries. In Elica they are realized by creating an object and putting all library functions in this object. Then this object is declared as open object, which means that it exposes its methods and fields for everyone to access directly (i.e. without mentioning library's name). This technique is used for all Elica's libraries.

And one final note about objects in Elica – they are truly dynamic and may evolve as the program runs [1]. So, there are specific means, generally called indirect access or access by name, for inspecting objects and finding what fields and methods are defined. Indirect access allows to access fields and call methods when their names are not known in advance, but are computed at run time.

NOOP Applications

The theoretical foundation of the Natural Object-Oriented Programming is a nice area of research, but the ultimate goal is to apply this into practical programming. The Elica system was designed and implemented to validate the concepts and the vitality of NOOP. Although it supports a dialect of Logo, the core of the system processes only 10 reserved Logo words. All the rest of the Logo language, i.e. hundreds of other primitives, is provided as a user-defined library called LOGO. The core of Elica is so miniature that without the LOGO library it cannot even add two numbers.

Several other libraries are built upon Elica's core and LOGO. They cover areas of practical programming – traditional and nontraditional Turtle Graphics, 3D graphics and real-time animation, graphical user interfaces with multimedia support...

Geomland is a geometry-oriented Logo which helps students and teachers to explore various problems by programming geometrical constructions. Geomland is implemented as a library in Elica.

The strength of Elica is mainly in the area of modeling and simulation. Geometrical objects are programmed in Elica. The minimalistic core of Elica makes it possible to enrich it in virtually any direction. Systems with rich core are usually closed for further improvement by users.

NOOP model is suitable for building 3D models of bodies and animate them in real time. The first two images in Fig. 4 are of human and robotic bodies. The main benefit of using NOOP is that it is easy to define any form of bodies, like the spider-like tripod. Also, it is quite straightforward to modify already defined models by reattaching different body-parts.

Defining postures of models can be done with a simple GUI posture editor, or programmatically. Animation can be achieved by interpolating between postures, or by programmatically control over each joint in the body definition.

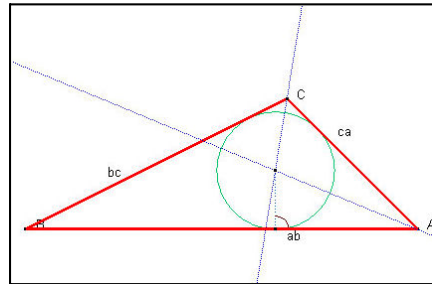


Fig. 1. Geomland Microworld in Elica. Users may drag any vertex or side of the red triangle. The rest of the geometrical construction will adjust automatically to the new configuration. Geomland microworld is not a part of the main Elica setup file. It can be downloaded and installed separately

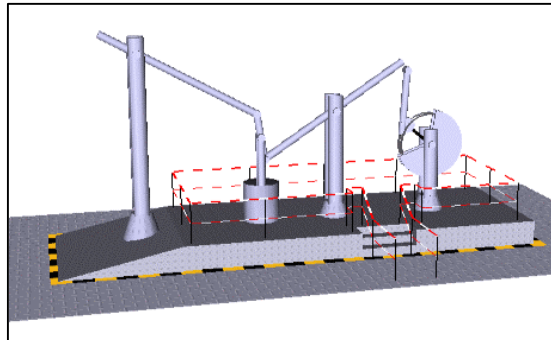


Fig. 2. An animated virtual reconstruction of a steam machine. The original steam machine converts circular movement into linear movement. The initial structure of the virtual model is plain 2D but as it works it gradually becomes 3D solid model [11]. The model is available as a sample inside the main Elica setup file

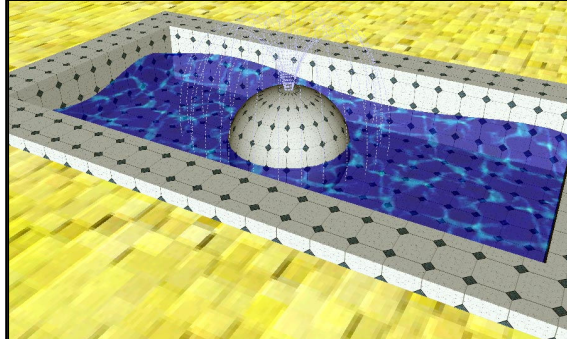


Fig. 3. A virtual pool. Waves on water surface are animated as well as the fountain. The animation is accompanied by sounds of water waves

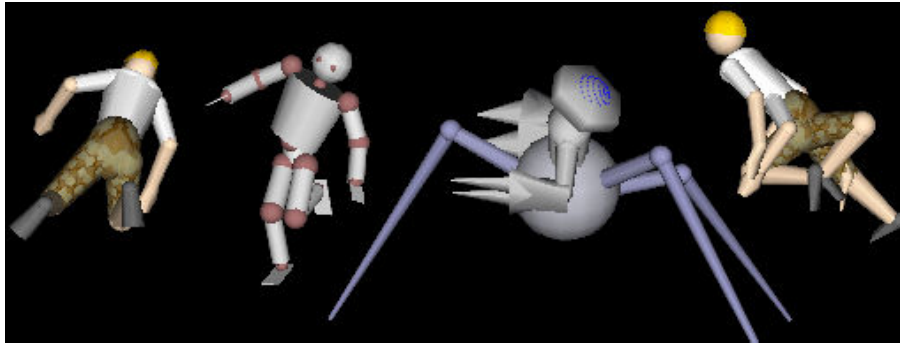


Fig. 4. A composition of snapshots of different Androids creatures. From left to right: the default human body, a humanoid robot, a 3-legged spider warrior, and a 3-legged, 4-handed mutant. The Androids library is not included in the main Elica setup file. It can be downloaded from the Elica site and installed additionally

Final words

Objects-oriented programming in Elica is designed in the way of how users think about the entities in their programs. It is somewhat impolite if the semantics of the program is hardcoded in the syntax. That's what most other OOP-enabled languages do.

If user writes a program having in mind elements like arrays and libraries, just because she feels more comfortable with them, than this program will give hard time to another user, who prefers to think in OOP style. As a result the second user has to wreck her mind or in some cases to rewrite the program in her favorite OOP.

The Natural OOP in Elica addresses this problem by separating the form from the meaning. The form is coded in the program, and it is as simple as dealing with variables nested in other variables. The meaning is something that is a result of user's in-

terpretation of the program. That's why the same program can fit well in the mental patterns of users with various programming backgrounds and preferences.

It is the same with many other things in Nature. For example, a linear system of equations is perceived differently by different people. An algebra teacher will think about sets of equations, a geometry scientist will think about analytical definition of a geometrical object defined as intersection of hyper planes, an engineer will think about forces at joints of a truss. Whatever all these people think about, its nature is the same.

The expressive power to apply different projections to the same reality makes Elica unique. To think about something as a library, then to use it as an array, and eventually to modify it as an object, right after using it as a variable, is something that frightens people who prefer to follow fixed stereotypes. There are dozen of programming languages made for exactly this type of programmers. Elica, however, is made for the rest of the programmers – those who want to experiment with ideas, to explore various relations and interconnections, to vary models while they are being built, and eventually to create things that make other people think.

References

1. Boytchev, P.: Using Logo to Model and Animate. Proceedings of 10th European Logo Conference EuroLogo'05, Warsaw, Poland (2005)
2. Boytchev, P.: Student-oriented Software for Application of Geometry in 3D Modeling and Animation. Abstracts of International Congress of Mathematical Society of South Eastern Europe MASSEE'2003, Bulgaria (2003)
3. Boytchev, P.: Turtle Metamorphoses (From "FD 1" To 3D Animated Characters). Proceedings of 9th European Logo Conference EuroLogo'03, Porto, Portugal (2003)
4. Boytchev, P.: The Very Logo Way. Logo Update, Logo Foundation, <http://el.www.media.mit.edu/logo-foundation/pubs/papers/vlw.pdf> (2002)
5. Boytchev, P.: Elica Home Page. <http://www.elica.net> (2006)
6. Mansfield, R.: OOP Is Much Better in Theory Than in Practice. <http://www.devx.com/opinion/Article/26776> (2005)
7. Hatton, L.: Does OO sync with how we think? IEEE Software , Volume: 15 Issue: 3 , May-June 1998, pp. 46 -54
8. Swartz, Fr.: C++ Notes: OOP Terminology. <http://www.fredosaurus.com/notes-cpp/oop-classes/oopterms.html> (2006)
9. Findy Services, Jacobs, B.: Object Oriented Programming Oversold! <http://www.geocities.com/tablizer/oopbad.htm> (2006)
10. Lee, X.: What are OOP's Jargons and Complexities. http://xahlee.org/Periodic_dosage_dir/t2/oop.html (2006)
11. Woodfield, Sc.: The Impedance Mismatch Between Conceptual Models and Implementation Environments, Proceedings of the ER'97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling, UCLA, Los Angeles, California (1997)