# Beyond the Object

Cristina COCULESCU
Romanian-American University, Bucharest

*Object orientation (OO) is regained not only in all components of integrate development media but it remarks in the rest of software world from operating system to last application domain – of course, with different intensity and success. A clear prove of OO application in all situations is the development of a wide range of industrial applications.*
*OO technology allows drawing of relation between the geometry, topology and dimensions of data on a class hierarchy; thus, the observation of the amount of data gained by research in many scientific domains is facilitated through class libraries both for graphic primitives and for events examination. In conformity to all waiting, OO asserts in every distributive system, there are very important the applications for making open systems customer-server and distributed applications in Java. Finally OO application in robot's programming and modeling needn't be omitted. However, far to be panacea, OO has also shades which will be researched so on.*
***Keywords:*** *object orientation, adaptability, re-usability component wares, mega-programming, generative programming.*

## 1 Limitations

However OO has gained the right of "city", some aspects of OO show a certain inconsequence of thinking – that prove the youth of paradigm. One of the land problems is given by a famous polemic older more than ten years: "is there square a class of rectangles?" The answer, apparently clear, is yes. Indeed, in what concerns the classification, like an expression – and likely – as taxonomy, the problem is clearly solved on all the chain of scientific tradition, from Aristotle to Linn : specific difference adds proprieties to nearest type: on other side, mathematically, inclusion into a (sub)manifold is defined through a property set (additional, seen like restrictions). The content decides the sphere: every class is defined through conditions or rules; an object is a member of a class if it satisfies all its conditions.
This intentional approach, logically normal and historically established, is fit in theoretic studies where there is searched the proving of some collective properties (of the class itself). Contrariwise, the establishment of the fact that an object is part of a class is difficult not only in informatics but also in mathematics (a famous example : the property of a number to be prime).
Real world classes – both of problem and informatics are explicitly definite through enumeration of their members, that is, the definition of the sphere is made by extension. At each level of classification, classes gave attributes and behavior in conformity with shown rules but now they don't define the extension of the class and they affirm properties of members. Thus, squares are not defined through their enumeration though they are a sub-manifold of rectangles, in a graphic editing program, in the class Figure_Square are included objects that the user draw then explicitly like squares, not "mathematical squares" but "graphic objects whose form is constrained to be a square". Now the surprises appear: square has less attributes then rectangles and even with other names: edge instead length and breadth; formal attributes are no more independent; separate update is dangerous (Modif_X is a good operation for rectangle but not for square too).
However both aspects are clear, in OO prevails extensional approach that intentional is generally avoided – probably also under psychological pressure of implementation through languages (especially that, actually, prevails the simplicity "type = class" where the appertaining to a class is explicitly established either at the declaration of a variable

or at the making of an object. There is true that all the characteristics of the class (attributes and behavior) are defined intentionally, by rules but the class is not established through the checking of these rules but first reversely : the rules must be defined so that to be compatible to the classes.
The problem is that objects hardly change the class (In Smalltalk an explicit operation is necessary but in C++ is impossible). For implementation, it would be terrible non-efficient to have an intentional definition, relied on rules, especially in the case of some wide classes because these rules would be checked after every operation which affect an object.
Additionally, an operation in super-class can change ”involuntary” the subclass of a member. Therefore, generally, a class must be closed relative to its operation and those of its ancestors (if Rectangle has an operation Modif X, Square needn't be a subclass but a twin of the rectangle, derived, for example, from Parallelogram). Otherwise, classes must be defined through augmentation (addition of new data and behavior structures) but not restriction. Hence, there is the trouble (as not to say it impasse).
After this description that deserves to point to one of the barriers came from deep principles of paradigm, we follow to have an overview to some practical inconveniences:
a) The mind complexity of big programs increases (heredity and polymorphism go to dependencies hard to follow; thus immediate taking and viewing of changes, characteristic of exploring programming, asks navigation through difficult graphs); the trouble goes bigger in the case of multiple heritage.
b) The acceptability is still poor (a whole generation of programmers has remained tributary to procedure paradigm).
c) Complexity of software development, especially in distributive work out (partially grace of the paradigm itself or of the first to inconveniences);
d) In simple operations, the code is not efficient (the burden of redundant complexity);
e) OO fell in re-usability (to this key problem there us given the paragraphs as follow).

## 2. Component wares adaptability and re-usability

For more then 30 years there is heard in different kinds about programming crisis (*software crisis*): the pessimists give it apocalyptic colors, the optimists consider it as defeat from the paradigm in vogue. However, all these, feel it and – in relation to other industrial domains – is full of paradoxes. Here are only three examples:
**1)** It is intense though it is truncated only to development phase (opposite of other industries there needn't the problem of product multiplication).
**2)** However, paradigms change very often (structural programming is only a quarter of a century old) but all programs almost the same (lack of work division is a clear symptom of industrial immaturity);
**3)** Geometric increase of available resources, far to cancel it gives it shape in at least two aspects: additional megahertz don't decrease clearly moon-men and additional gigabytes are filled immediately. Event the conclusion is driven by the paradox; the more the crisis becomes stronger, the better it becomes acuter!
For clarification, there is utile a rapport to other industries: there are large-usage products (TV-set, text-editors) where wide production allow the development of a small number of variants with functional flexibility and also products (almost) unique (bridge, driving system for a complex technological system), where the variance is hardly imagined because the cost of non-used functionality is too expensive.
Between these extremes there are many domains (for example car industry or software for InInd) where, in slang OO, there would speak about a class “auto” with subclasses like “limousine” , “kombi” , “cabrio” . Contrariwise, OO modeling of engine or heating is difficult because either the tree of class hierarchy would be too wide or the numerous relations between classes would go to a complex and hard project activity. Here there becomes true that OO is neither a sufficient base for adaptability (for example the changing of an engine with benzene without lead,

respectively of a program-product for localization) nor for re-usability (if there is consolidated a new necessity of use), usually it relies on the kind of development and on the experience achieved at existing similar products: in time it would have to appear elements that can be used to many products). In fact, the problem is very old (from bricks and slabs to containers and even to architectural styles, the efforts for modularization traverse all civilization's history) and has penetrated many years ago the technology of information both through electronic components of equipments and through modular programming rules. To conjectures deeps it: the spread of OO systems generally distributed and globalization of enterprises, which asks more and more distributed applications on communication networks. More, InInd systems become complex and complex clearly mirroring the complexity of real world but the world remain intelligible, being composed of components that can be found (for example operators, products, equipments); of course, conceptual description of complex informatics systems as a projection of modeled world, is often modular but this vision seldom overcome the concept level, the system remaining monolithic as it is - with very strong links between „models" – because it isn't clearly joined to real world but by means of "someone" (man or machine) who use its information to control it. Therefore modularity must be born to implementation – through the technology of distributed objects.
But re-analyzing the problem with available instruments, the conclusions can be formulated in terms more strong; though the eves of the time of half-made software can be delayed, there is plenty of inconveniences, most of OOP languages – including C++ haven't means of packing and efficient distribution of objects as binary; the customs and practice needed for development of components, is clearly unlike those necessary to use they, in fact, exaggerating a little, OO breaks the principles of modular programming (a program developed like a hierarchy of classes, has a maximum coupler).
Because standard elements are few and rigid, for solving the problem, there are suggested three ways to fight, shown in respect to the distance to OO paradigm (already) "conventional", the development of flexible models (shown in component wares, described as follow), development of half-finished materials (as abstract classes used in frame technology) and – last solution? – renewing the paradigm (passing to generative programming).
Availability of computers is precarious and there is (still) hope – if they will be enough - to be more efficient search, checking and understanding of a component then it conception. In fact, what is a component? A class or a group of classes with a tight coupler but in concepts an atomic piece good to be used again, that is, their components are sold individually and must impose trust, its subcomponents (for example other objects) always co-operate for the achievement of a function.
An important role in the insurance of the lustiness of systems with distributed objects relayed on components has tolerant communication between objects: a system is considered as tolerant if it accomplishes its functions even when a component is changed or deleted. A wide treatment through InInd systems, of passing from monolithic to component-based systems – including showing of demands, system projection, facility of "mediation" of the action between components and the description of the class of system components – is presented in [Coutts and Edwards, 1998].

## 3. Megaprogramming frame technology

An approach right more flexible of this problem which offer partial solutions too, but of bigger efficiency, is based on frame technology, where re-usability points not only the code but also the projection. An OO frame (framework) defines a feature of classes and a model of interaction for the objects which co-operates, being made a generic architecture; the places where there must be added additional functionality are predefined. Therefore, unlike usual applications, the mainframe and also a big part of needed functionality pre-exists, every time there

must be added only proper functions of actual application. The frame calls additional functionality, developed by user. Of course, as a frame is used to more and more applications, as the number of components which must be developed decreases, so that complex applications "can be put out the shelf". Entities are defined functionally, relations between they, abstract, grace of insertion of many abstraction levels in system specification, the complexity of each level can be managed, thus it increases not only the quality of system arrangement but also re-usability. Through aggregation of frame components, the craft of traditional programming supports a movement to mega-programming – direction so more marked in distributed systems. In consequence, the frames can cover a wide range both as granularity of domain (support frames, domain, application ones) and like power of re-usability (*white-box, glass-box, black-box* , of course, the more transparent is the frame, the better it is re-usable.

However, how do we operate, for example with an "abstract resource"? A resource is something which can be got and, after that, released (file handle, memory area, counter). Extending the idea; every object is a resource that is got by constructor and is released by destructor (resources are packed in objects). An abstract resource is something that isn't an object itself but the status of another object. In this point of view, abstraction force of used language has an important function.

Difficult problems appear – almost always – in practice. Difficulties and also their passing with a methodic development of frames are synthesized related to: big development costs, limited experience in the domain of object technology, the long time for habit with pre-existing frames, lack of proper instruments of frame description and use. Proposed solution: a main model, on three subdomains a) development b) documentation c) application.

In first stage, domain theory (here, knowledge regarding the kind for solving the problems) are implemented as generic entity, re-usable (main problem: the splitting of functions in general valuable and specific for applications).

Both development and documentation are realized by iterations (while there are added new properties, it goes from established zones to flexible ones of the frame). A meta-model of the frame is the same time base for documentation and application process. The "networks" (general, without implementation details) for development of applications, are part of documentation (and they are filled by the users of the frame with specific information about classes, methods, parameters etc.). Through software quality instruments (with incremental code generator, for example) the application is divided in two phases: the creation of an implicit application and its gradual specialization.

## 4. Generative programming

Unfortunately, not only conventional paradigms of software development but also, right now even OO technology help insufficiently the adaptability and re-usability. Hence there are new searches. The motivation, main concepts and principles of one of most promising "trans-object orientations", generative programming introduces itself in abstract as follow with [Eisenecker and Cazanecki, 1997].

Between the problems not solved with components and frames there are : lack of some procedures of development, abstract classes implements arbitrary some features, reducing the adaptability, abstract classes with different functionalities are hardly used together in the same application because most of applications don't use all (intentioned) facilities of abstract classes, much of their functionality remain useless and redundant in code: flexibility is often got with the price of increasing of the time of operation, need of detailed knowledge regarding internal structure (hiding of information become barrier). Additionally, there are also big maintenance problems that appear. Sincerely, if there is previewed a variable feature , its "factorization" in an abstract class is easy; but, if several features are kept, the number of 'artificial" classes, without a link to basic concepts of

application domain increases – and the same time with it, the coupler given to heritages and associations and also the complexity of projection.

There is true that adaptability and re-usability of the amount of classes increase but, especially, isolated classes can't be used. In consequence, it is preferred that the variability be got by configurability, that is, instead to derive new classes, a class is configured with classes (with minimum dependence) having role of "configuration parameters". In this regard, generic units of real-time programming languages give an important potential, without purpose even a systematic identity of definition and configuration dimensions of component. Here comes the new approach.A main concept of generative programming aspects is projection space (design space). Whose dimensions concern characteristics recognized as relevant for building a component. Examples of dimensions: interfaces, implementation, synchronization, structure, error detection. A dimension is relied on its aspects like an attribute to its values (for example the characteristic "type of data" can receive as types: integer, real with mobile coma, complex). Expressions give basic nature and functions of one component. Among the numerous problem put by projection space here are just some: Are the dimensions complex, that is, decomposable in other dimensions or elementary? Are the expressions dichotomist, discrete, continue, (in)finite, ordered and so on? Is the space extensible and/or changeable?

In this context, the objects of generative programming are formulated like this: increasing of adaptability and re-usability; improving of the control of complexity; availability of a big number variants; increasing the efficiency (as memory and time).

Therefore, the five **principles of generative programming** are:

- founding and splitting of relevant domains of projection space (*separation of concern, how many they are and how they are established?)*;
- opening of implementation: every component allows the access to its implementation strategies as expressions of dimensions;
- the spreading of expressions: information concerning expression of a dimension of a component can be sent further to its subcomponents, neighbor or over-ordered (for avoid redundancy and global optimization relative to the domain);
- illation of complexity through configuration rules: a component has external and internal expression (these can be deduced generally through configuration rules that establishes the associations of non-valid expressions);
- avoiding of useless costs (*zero-overhead rule*) final product haven't redundant components or functions (for example, it resigns dynamic binding if that static is enough).

Of course, such approaches must be accompanied also to a cluster of (formal) methods which help them; for example, a level of inter-connection of components is purposed in [Hirschfeld and Schonefeld, 1997]. Finally, to this paradigm (ones already grant it this title) there are associated a lot of techniques that are still in syncretism (*Intentional Programming*, *Aspect-Oriented Programming*, *Subject-Oriented Programming*). It is to view which of them will pass the stage of promising.

**References**

[1]. BĂRBAT, B., FILIP, F. G., *Informatică industrială. Ingineria programării în timp real,* Vol. I, Editura Tehnică, Bucureşti, 1997.

[2]. HELLENDOORN, H., DRIANKOV, D., *Fuzzy Model Identification*, Springer-Verlag, Berlin, 1997.

[3]. HERMANS, B., *Intelligent Software Agent son the Internet: an inventory of currently offered functionality in the information society& a prediction of (near-) future developments*, Tilburg University, Tilburg, 1996.

[4]. HUNT, J., *Smalltalk and Object-Orientation*, Springer-Verlag, Berlin, 1997.

[5]. MC ELLIGOTT, M., SORENSEN, H., *A Connectionist Approach to Personal Information Filtering,* University College, Cork, 1995.

[6]. METAKIDES, G., NERODE, A., *Principii de logică şi programare logică*, Editura Tehnică, Bucureşti, 1998.
[7]. NIELSEN, J., *Usability Engineering*, Academic Press, New York, 1992.
[8]. SHIN, Y. C., VISHNUPAD, P., *Neuro-fuzzy control of complex manufacturing processes*, International Journal of Production and Research, v. 34 no. 12, S.U.A., 1996
[9]. SMITH, L., *An Introduction to Neural Network*, 1998
[10]. STONE, P., VELOSO, M., *Multiagent Systems: A Survey from a Machine Learning Perspective*, Carnegie Mellon University, Pittsburgh, 1997.
[11]. TURNER, R., *Logics for artificial intelligence,* Ellis Horwood, Chichester, 1984.
[12]. WOOLDRIDGE, M., JENNINGS, N. R., *Intelligent Agents: Theory and Practice,* Knowledge Engineering Review, 1995.