EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN/ECP-96-10

31/07/96

COMO: An approach to Object Oriented Analysis and Design for High Energy Physics applications of algorithmic nature

Francis Bruyant
CERN/ECP

Abstract

The emergence of the Object Oriented technology is having major consequences on the evolution of the software for High Energy Physics. Given the lack of maturity of the commercial products currently available, it is too early to make a statement on which OO languages and methodologies will survive up to the start of the Large Hadron Collider. The opinion of the author is that many traditional design features successfully tested during the last two decades are rather independent from any specific language characteristics. They still appear as valuable and could be seamlessly transposed into the Object world. They constitute the skeleton of the COMO approach presented in this note.

1 Introduction

This report reviews

- the motivation for the COMO project and its initial aims,
- the arguments behind its main concepts,
- the proposed approach.

It summarizes research work undertaken part-time over the last 5 years into the adequacy of Object-Oriented (OO) software technologies for High Energy Physics (HEP) applications.

COMO, an acronym for Constituent Model, originates from ideas developed in the context of Event reconstruction for the L3 detector at LEP [1]. The concepts have emerged from observations based on past experience. During the period of incubation they were exposed partially and discussed on several occasions, not always with much success! It is only recently that, with the encouragements and suggestions of more experienced people[1], they have reached coherence and maturity and can be presented hopefully in a more convincing way, with personal comments to shed some light on their *raison d'être*. This, however, does not mean that the proposed approach makes use of all potentialities of the Object paradigm. COMO does not have such an ambition. The OO purists shall see it as a revisited SA/SD programming technique. I do not share their view, to the extent where the "object structures" mentioned in this report should not be seen as "data structures", despite of analogies in the semantics of the relations used to define them.

It will take years of hard work before a statement can be made on the possibility to master the entire OO technology and to use it with a real benefit. Any effort in this direction has to be encouraged, with hope that it will succeed.

2 Initial aims

In the early 90's it was fashionable to share the opinion that OO languages would bring the solution to our software problems. Undoubtedly, some of the OO features (e.g. encapsulation, inheritance) looked like good ideas, useful for implementing truly modular and re-usable pieces of code. There was enthusiasm for commercially available OO methodologies proposing attractive recipes to make programmer's life easier and to increase their productivity, guaranteeing at the same time the quality of the products. However, books dedicated to the OO technology and to its applications usually warned the reader about the trauma of entering the OO world, about the necessity of a complete mental revolution.

During the same period, I was investigating the *pros* and *cons* of approaches followed by several HEP Collaborations, trying to identify which developments could be of general interest and have some intrinsic value, at least for the specific domain of applications we were concerned with. I had come to the conclusion that it was possible to propose general guidelines [3] to build, maintain and execute in an efficient way the large programs one had to write for HEP event simulation and reconstruction. I thought this could avoid

[1] I am indebted in particular to John Deacon (Consultant, Software Engineering) for a constructive criticism of my earlier notes on COMO. His comparative study of methodologies [2] has helped me to clarify many points.

the rather sterile replication of effort for software organization that takes place in every new Collaboration. Then, dreaming of "grandiose" ideas about software design (of course based on FORTRAN, the traditional language I was familiar with) and being convinced that they could solve our problems in an elegant manner, I have been shocked by the faith and confidence, not to say the arrogance, of the OO fans.

How to accept that concepts which have to do with pure logic could be so strongly dependent on a technology? The willingness to clarify that point has been, since then, the only motivation behind this work. Therefore, the initial aims of COMO were nothing else than studying the differences and the analogies between an OO approach and traditional software approaches rather successful in the past, in order to understand in depth the intrinsic benefits of the new technology and, if possible, to demystify the aura which surrounds it.

What was at stake appeared to me to be of fundamental importance for the HEP community, not only from a technical point of view, but also from a sociological point of view: how to revolutionize programming techniques without taking the risk of losing the invaluable experience accumulated over the years by physicists and programmers who could suddenly feel lost.

Today, I have no doubts that the main features of the OO languages are valuable and that it is necessary, and most likely unavoidable anyway, to switch to the new technology. I also firmly believe that going the OO way blindly, without regard for the specific features of our environment, might result in a major disaster: this is explained in section 3.1 and the suggestions to avoid falling into a fatal trap are described in section 4.1.

Regarding the OO methodologies and their notation systems, I have come to the sad conclusion that none of the proposed commercial products is acceptable to the physicists. Given the notoriety and the talent of their authors this is rather surprising. It is partly due to the fact that the work done in this new area of research has not yet reached maturity. It might also be due to the tribute paid by all methodologies to their common ancestor, the Entity-Relationship (ER) data model [4]. This is explained in section 3.2 and what COMO proposes instead is reviewed in section 4.2

It is assumed in the following sections that the reader understands, at least superficially, the OO jargon.

3 Arguments behind the COMO concepts

3.1 The OO grey areas

It is not rare that, after the initial phase of excitement, the OO neophyte enters a period of doubt about the infallibility of the dogmas: the feeling grows that things might not be as simple as some books claim. Of course, the intensity of the quarrels around the OO languages and their respective churches contributes to that situation, but the doubt may also come from the confrontation of what the books say with the teachings of personal experience.

There are many areas where the proponents of OO seem to have not initially given enough attention to the fact that concepts which look simple may cause problems when confronted with the complexity of reality, e.g.

- messaging,
- inheritance and run-time reclassification,
- object persistency.

However, with the research carried out, a consensus seems to be emerging in these areas.

Object persistency, of utmost importance for HEP, is the most striking example. Let us discuss it first. Quite rightly, an object's **class** is regarded as the **essence** of the objects that it generates. It describes their common characteristics, through the definition of attributes, and their common behaviour, through the specification of methods, namely the implementation of the services that the objects can provide. Primarily related to the object creation process, known as class instantiation, the concept of class-method had to be introduced: sending a message to an object, to invoke a service, requires that the object exists, therefore the creation of an object cannot result from a message sent to it. Class-methods offer also the possibility to define services that operate on all objects at once. In theory the latter facility is technically acceptable. It leads to what is called the ensemblist approach. In practice it is often a source of problems as it usually results in developing classes which become monstruous, often modified and therefore unstable.

Let us go through the first part of the argument. Most examples given in the books to illustrate the OO concepts concern objects usually created through a process which one may qualify as creation **by assignment**: their ingredients are either preset according to decisions taken by a privileged "user" or derived through trivial deterministic computations. In that context, the **existence** of the objects results from a deliberate intention to assign to each of them, at creation time, specific identification properties: e.g. the objects of a **bank_account** class created, for instance, each one with a given account number and for a given person, with given opening date and initial deposit.

For high level scientific applications of algorithmic nature the situation is different. There, objects that one often needs to consider are created through a **stochastic**[2] process: they result from algorithms that decide which ingredients are assigned to every of them. In an earlier note [1] such ingredients were given the name of (hierarchical) **constituents**. The constituents are themselves objects belonging to other classes, defined a priori as **constituent classes** of the new objects' class, and the algorithm selects amongst the objects of the constituent classes the ones assigned to each object. In such a context, one cannot predict how many objects will be created and which constituents will be assigned to them; furthermore, one cannot exclude that some constituents will be shared by different objects: e.g. in a track pattern recognition context, the creation of **track_candidate** objects, starting from a bunch of **space-hit** objects, their constituents. What seems to be specific of a stochastic creation process is the sensitivity of the results to even small changes in the implementation or in the tuning of the algorithms, as well as the difficulty to compare the quality of the results other than on a statistical basis.

Let us now go through the second part of the argument, related to two essential requirements which one has to consider for HEP applications (and most likely also in other scientific domains) and on our ability to fulfill these requirements.

The first requirement concerns I/O **modularity**: during event simulation or event reconstruction one may often need to store intermediate data, keeping the history of

[2]Since S. Van der Meer's Nobel award it has been well known that the word **stochastic** does mean exactly what one wants it to mean!

what has happened, then to restart from any given intermediate state, possibly with different assumptions.

The second requirement concerns **reproducibility**: namely, the possibility to record the conditions under which any results have been obtained, to reproduce these results at any time if needed.

Commercial approaches are developed to store objects, keeping track of their class characteristics. In OO applications where only the creation by assignment is considered, recoding a method does not affect the identity of the objects and has usually no side effects, therefore keeping an exact record of what a class consisted of at the time when its objects were created and stored is often irrelevant. The situation is different for applications where the stochastic creation takes place. There, modifying an algorithm within a method may change dramatically the results. It is well known that, due to the imagination and to the perfectionism of the physicists, one has to expect such modifications rather frequently. Then, keeping track of the conditions under which the objects are obtained, namely of their class characteristics, though technically feasible, becomes critical, even unrealistic when dealing with huge amounts of data processed, and re-processed, over long periods of time. Alternatively, assuming that objects formerly stored could be "re-activated" while their class has been slightly modified would correspond to a situation where the **existence** of the objects precedes their **essence**, an existentialist view hardly compatible with the requirement of reproducibility.

If the above arguments illustrate one of the potential problems of the OO technology, they also show where the trouble comes from and suggest the way to cure it, as discussed in section 4.1.

Another problem with OO is related to the different interpretations of the concept of inheritance at the level of the languages. In object modelling context, inheritance is an abstraction process which consists of describing new concepts by making reference to similar ones whose definition is already known. What *similar* means is however rather subjective. Again, judging from the often contradictory opinions expressed by the authors of OO languages and methodologies, things are not as simple as they look at first sight. It is an aspect of the technology which is undoubtedly fascinating and it is also where one finds the most frustrating, irritating and sometimes frightening developments. Quarrels on subtyping/supertyping versus subclassing/superclassing[3] or on aspects of multiple inheritance seem to indicate that the rules of the game are not well established. Section 4.3, describes the inheritance features needed, at least for our applications, and proposes rules to figure them out.

A last remark, concerning messaging: this is also an area where there is some confusion. One could have expected that the way the objects interact, a fundamental OO issue, would have been defined unambiguously and exploited accordingly by the methodologies. For some conventional OO languages, like C++, messages are nothing else than procedures[4] and concurrency features have to be implemented independently, at least for real time or

[3] As an example, I shall mention a paper (where other points of interest are also discussed) by Gabriel Eckert, Department of Computer Science, EPFL, Lausanne, CH: *Types, Classes and Collections in Object-Oriented Analysis*, Proceedings ICRE'94, Colorado Springs, Colorado, USA.

[4] "Procedures issue requests for external input and then wait for it; when input arrives, control resumes within the procedure that made the call", J.Rumbaugh [5]. Nevertheless, in the OO context, messages are crucially different from subroutine calls, a message being a communication which allows the receiver to choose the code and to take the appropriate response.

multi-process applications. General modes of communication, possibly influenced by the observation of how evolution proceeds (a highly controversial issue anyway), have been defined, for instance the broadcasting mode, where an object (the sender) is assumed to issue a message to the whole world, expecting that another object (the receiver) will interpret it and take action. I do not believe that they should be considered for our domain of applications. Nature displays an extreme prodigality, leading to a lot of unavoidable wasting, and human beings, with their capacity of thinking, should avoid to imitate it. Unfortunately, they often do[5]. A programmer is assumed to know what he/she wants to do and can therefore establish *a priori* all connections between the relevant classes in order to ensure that the most logical path, usually also the most economical one, will be followed. This is discussed again in section 4.2.

3.2 Weaknesses of the OO methodologies

The OO methodologies need be built upon models which, because of the nature of the objects, cannot be simply data models. They have to take into account dynamic features in addition to the static ones. But the static model (also called structure model) is the essential part of an OO model.

The OO methodologies rely usually on the well known ER data model as the starting point for a static description of the objects. This model has been for decades a reference, successfully used for data base related applications [6] and also within general proprietary data systems such as ADAMO [7].

Before the emergence of the OO technology the world wide diffusion of the ER model, or of many variations built around it, had given rise to a powerful commercial lobby. For the authors of OO methodologies there was clearly an obligation, not necessarily only financial interest, to make reference to the ER model and to integrate its basic concepts and terminology.

When trying to model data in the HEP context of event simulation or event reconstruction, I have never felt comfortable with the ER concepts, for many reasons. The first one concerns the graphism and notation system which is confusing, not compact, and becomes unreadable as soon as the number of entities to be considered simultaneously grows, not to speak of the interpretation of the cardinalities in the case of n-ary relationships with $n > 2$. This could be an unimportant subjective opinion if it did not reflect deeper problems. There comes the second reason: the over-simplification of the semantic of the relationships between entities, usually regarded as bi-directional relations[6]. Such relations do not take into account the hierarchy which may naturally exist between entities, a frequent situation for the kind of applications one is concerned with. In other words, the ER semantic forces the user to degrade the representation of what he/she wants to model, which is not quite what one expects from a data model. The third reason, as important as the second one, is that the ER model, not surprisingly I should say, is nothing more than what it means to be, namely a **conceptual** data model and only a conceptual one. Due to the absence of a powerful semantic, it does not carry any **logical**

[5] Who has not been irritated by the avalanch of uninteresting or irrelevant prospectus distributed every day in all private mail boxes?

[6] Indeed, additional information can be provided by giving suggestive names to the relationships or by assigning rôles to the entities, but this makes an immediate interpretation of the ER diagrams even more difficult and the relationship names are usually meaningful in only one of the two directions.

information on the real nature of the relation between data and therefore does not give any hint on how to structure a program to exploit the information that it describes. One can stay for hours in front of a complex ER diagram and, assuming that one understands what the labyrinth of entities and relationships means (a performance on its own!), not have any idea on what to do with it. Furthermore, the detection of errors of logic or of inconsistencies is by no means evident [7]. Physicists expect more than what the ER model can give them, or they shall not accept to use any model at all.

Up to recently, when refering to the ER terminology to describe their structure models, the books on OO methodologies were often introducing the concept of object as an extension of the notion of entity. The notions of relationship and of cardinality were used to describe additional features of the static model (e.g. association, aggregation) and, ultimately, to illustrate some aspects of the dynamic model relevant to the messaging between objects. This has led to a great confusion. Of course, a trivial object can primarily be seen as an entity but most objects are non-trivial: they can be represented by systems of simpler objects, assembled according to **composition** rules, based on simple uni-directional relations. The composition relations include the concept of functional **delegation**, which happens to be also in many situations a flexible way to avoid making a poor usage of inheritance, as discussed in section 4.3.

Independently, other relations (uni-directional too) are needed to enable the interaction of the objects with the outside world.

All these points are discussed in detail in the following chapter. In short, all relations in the OO world are of uni-directional nature and this does not match very well the ER notion of relationship. For some OO methodologies, in particular the Class Relation approach [8], the adoption of the ER concepts derives from a deliberate initial choice, regardless of the consequences. For others, the integration of the ER technology, sometimes made at the cost of tortuous arguments [8] , is less fundamental and appears so artificial that one may wonder why their authors do usually praise the ER model in the same books. I called one of the most reputed world experts in OO methodologies and expressed to him my concern. Speaking of the ancestor, the ER model, his answer was: "It is painful, but sometimes necessary, to kill the grandfather". I cannot agree more!

There are other weaknesses of the OO methodologies which the ER model is also partly responsible for. I shall just mention briefly two of them.

Most methodologies do not consider the **existence dependence** between objects as a useful modelling concept. As a result, the notions of **aggregation** and of **association** to which they refer are often nebulous and not of much practical interest for most applications. The COMO semantics is based on a precise definition of these notions.

The OO methodologies do not pay much attention to promoting the stability of the classes, so necessary for defining re-usable class libraries. This increases the complexity of the problems mentioned in the previous discussion on persistent objects. Though it was not initially perceived as an important point, it turns out that many features of COMO

[7] For instance, the figure 2.33, page 47 of Batini's book [6], with the cardinalities proposed for the relationship "taught by", does not prevent that a professor could give courses in different rooms at the same time, a weakness corrected in the text, where the relation "taught by" is described as a one-to-one relationship. Nevertheless, this book is remarkable and the distinction made between the conceptual, the logical and the physical phases is most interesting.

[8] For instance, the non-demonstration, pages 72-73 of Rumbaugh's OMT book [5], for the ternary association Student-Professor-University!

have an impact on the stability of the classes.

A last remark: the unification of the various aspects of an OO model has, to my knowledge, never been achieved by any OO methodology, despite some of them advertising that they have. COMO proposes some steps towards this objective, as discussed in section 4.6.

4 The COMO approach

4.1 The transient and the persistent objects

The OO philosophy relies on the basic concept that an object is defined only by the services that it can provide, the internal data representation inside the object being of no concern to the users. Respecting too strictly this concept seems to be a narrow-minded attitude. It reminds me of a well known sketch by Coluche[9] on advertising for the new OMO, "the washing powder cleans the dirt even inside knots made around the spots", giving the same results as the old OMO without knots. The housekeeper makes the knots on Monday, washes on Tuesday and has the rest of the week for untying ..., great progress! Without any violation of the fundamental encapsulation principle, it is perfectly acceptable to have, within the objects of a given class, structures built from simpler objects, known to the user and accessible as such through appropriate services of that class: these structures may look like knots, but one does not need to untie them! Of course, any service may also define and use freely internal data whose representation remains irrelevant to the user.

As mentioned before, a coherent and efficient storage of persistent objects may turn out to be a tricky matter, at least for applications with classes whose services consist of sophisticated algorithms subject to frequent changes. Furthermore, the way classes are usually built encourages the introduction of services which have to do more with the environment of the objects than with their intrinsic properties and this is a source of class instability.

More generally, though this is apparently conflicting with the strict OO orthodoxy, one is led to the evidence that an object has **essential** properties, regardless of the context in which it is considered – these are specified in its own class, and additional **auxiliary** properties, which depend on the context and should be specified in other classes.

These remarks lead to the distinction made in COMO between various categories of user-defined classes:

- the **user_data** classes, whose objects may have to be stored and retrieved frequently. These objects need not to be too intelligent. They are usually single-state, with explicit attributes (something like a C structure or like the data part of a ZEBRA bank [9]). The user_data classes are equiped with the set of basic (object-)services needed.

- the **environment** classes, of "singleton" kind (namely with a unique instance), whose responsibility is to prepare the creation of their related user-data objects. Because they know why and under which conditions these objects are created it is natural that they keep control of them. The environment classes are often bulky

[9] A french humorist (1944-1986), founder of the *Restaurants du coeur* to feed people without resources.

and complex, each with a multi-state composite instance, with all kinds of services, possibly including those where sensitive algorithms of stochastic creation processes are implemented. All this has however no major inconvenience to the extent where their instances are transient, namely do not usually need to be stored persistently. Keeping a record of the successive modifications of these classes does not imply anything else than traditional source code versioning techniques.

- the (intermediate) **tactical** classes, to which the environment classes allocate part of their responsibilities and whose objects, created under their control, have the specific mission to feed them with the information required to build the user-data objects. The objects of the tactical classes are always transient. They have an autonomous life, namely they control themselves their evolution, their changes of state. They play a fundamental role and the success of an OO application depends greatly on a skillful design of these classes.

The user_data classes are nevertheless classes, therefore *a priori* compatible with any OO data base management system (OODBMS) commercially available. The benefit comes from their expected stability, and this is what makes the difference. Several user_data classes can be related to the same environment class. The construction of a user_data class has to follow simple rules in order to achieve efficient and modular I/O, for instance the use of some attributes as keys for object retrieval.

The construction of an environment class may also follow some rules: in COMO it is suggested that all environment classes inherit from the same abstract class **Processor_**, whose properties, in relation to the dynamic aspects of the model (section 4.6) and to the construction of large programs (section 4.7) are of interest [10].

During the execution of a program, an environment class' instance can be regarded as the repository of the user_data objects that it has to keep under control. All relevant user_data classes are then accompanied by their respective environment classes, which can either be tailored for the appropriate application or be more general classes of whose services only some will be used. Mechanisms to "reload" the repositories from the external medium where the user_data objects are stored have to be made available as services of the environment classes, as well as mechanisms to "empty" the repositories to store the required user_data objects. These tasks could be steered by specific I/O processors coupled to a commercial OODBMS.

4.2 The object modelling features

Object modelling consists of describing the static and dynamic properties of the objects. It relies on concepts expressed through an unambiguous semantics and on a graphical representation expected to match faithfully the underlying semantics.

The static aspect of object modelling concerns primarily the description of the stable states of an object. The dynamic aspect concerns the creation of the objects and what causes their state transitions.

The user-defined classes, described above with the main objective of ensuring a greater stability for the classes whose objects have to be stored persistently, illustrate the need to

[10] The processor characteristics could be viewed as a design pattern, in the spirit of the recent remarkable book on Design Patterns [10].

handle composite objects and therefore to define precise **composition** rules. These rules should give the users all flexibility required for a modular construction of classes/objects based on simpler and stable ones. Composite objects need not be nebulous: whenever it is possible to describe them in a logical and transparent way, by identifying carefully their **components**, there is only a benefit to do so. The components of an object **characterize** its state.

Among the **essential** properties of an object, some can be regarded as belonging exclusively to the object and others can be regarded as sharable. This leads to the fundamental distinction made in COMO between **aggregation** (exclusive) and **association** (sharable).

In the aggregation case, the characterising properties are either all settled at the time of creation of the object (internal components) or specified **asynchronously** through the definition of external components referred to as **parts** of the object. An object with external parts is defined in COMO as an **aggregate**. The parts do not know anything about the aggregate. The parts *belong* to the aggregate, which means that their existence depends on the existence of the aggregate. However, they can be regarded as "dismountable" and, when relevant, be transfered elsewhere prior to the disappearance of the aggregate. This is in particular the case for the repositories of user_data objects, parts of their respective environment class instances. The asynchronism mentioned above does not necessarily imply that the aggregate exists prior to the parts, as shown in sections 4.3 and 4.6.

In the association case, some properties characterising the state of a given object are "borrowed" from other objects which have their own independent existence. One cannot guarantee that these external components will not be also associated to other objects. They are *a priori* sharable and, in COMO, are referred to as **constituents** of the object. The object knows how to access its constituents but the constituents need not know anything about the objects to which they have been associated. Coming back to the ER model, one may notice that all relationships (of any n-arity) between entities could be represented by objects having these entities as constituents.

The fact that the external components of an object (both the parts and the constituents), themselves objects of other classes, contribute to its essential properties implies that the object knows which component classes are expected and how to use their objects.

The **auxiliary** properties of an object are always specified asynchronously and the object, which in this case has to pre-exist, is unaware of their existence (relational aggregation). They are usually induced by the conceptual association of an object (e.g. A) to another object (e.g. B): B is assigned a part C which has A as constituent and where the auxiliary properties of A in the context of its association to B are described.

The characterising relations within an object are then, in COMO, classified as follows:

- Internal component, a language feature

- Aggregation, a relation defined through the following properties:
  an **aggregate** has access to its **parts** through uni-directional relations named **APASI** (inverted IS A PArt of),
  the parts are created asynchronously,
  they can be accessed only through the aggregate,
  they do not know anything about the aggregate,

their existence depends on the existence of the aggregate (as long as they have not been dismounted).

- Association, a relation defined through the following properties:
  an object has access to its **constituents** through uni-directional relations named **OCASI** (inverted IS A COnstituent of),
  the constituent objects do not know anything about the objects to which they are associated and, most important,
  the object is **not** allowed to change the state of its constituents.

- Relational aggregation **APOCASI**, an APASI relation for the description of auxiliary properties of a constituent in the context of its association to a given object.

The dynamic aspect of object modelling is related to the capacity of an object (the sender) to send a message to an object of another class (the receiver). This implies that the sender *knows* the receiver, namely has a (uni-directional) relation to the receiver. When the receiver is an external component of the sender, the condition is naturally satisfied through any one of the specific composition relations described above. When the receiver is an object of the "outside world", namely not a component of the sender, the relation to the receiver, a property of the sender, is given the name **USES**. The USES relation is a **class**-relation within the class of the sender, namely it is the same for all its objects, and the receiver is always the unique instance of a singleton[11]. Conceptually, a message sent to the "outside world" invokes either a service of type **action**, which does not modify the state of the sender, or a service of type **access**, which induces a change of state of the sender. In practice, the syntax of the procedures available in most OO languages allows to combine both operations at once and, often, the sender exposes within the message, either by value or by reference, those of its components that are expected to be modified. This is convenient, though not quite in line with the concept of messaging.

Any USES relations needed by the tactical objects activated during the execution of a program are derived from the USES connections pre-defined in the set of environment classes which contribute to the strategy for the given application.

4.3 The inheritance features

As used by the OO technology, inheritance appears as a pure *inter-class* relation concept: a "subclass" is derived from a "superclass" by declaring explicitly the inheritance relation within the definition of the subclass. In theory, an instance of a subclass is expected to fulfill all commitments of the superclass and can therefore be viewed also as an instance of the superclass, but this is not what one means by *inter-instance* relation. I mentioned *in theory* because, the above condition is not always satisfied. Often, one should rather speak of **partial** inheritance, which has more to do with code borrowing.

So far two questions can be raised. The first is related to the fact that inheritance is defined only as an inter-class relation: is it the unique way, and the best way, to see the things?

As mentioned before, an instance of a given subclass is also an instance of the superclass. Is it useful to have the possibility to turn it into an instance of another subclass, a

[11]The latter statememt should be considered for the moment as a postulate. I do however believe that its validity could be deduced from purely logical considerations.

situation that would somehow simulate a change of nature of the object? The answer is yes and this is the reason why non-trivial developments on run-time re-classification and migration are currently taking place. Are these developments going in the right direction? The answer might be yes or no, depending on the context. Inheritance can often be simulated by an appropriate usage of composition relations: this is the **inheritance by delegation**, expressed in COMO by the relation **ISALSO**, both an inter-class and an inter-instance relation. It gives the user the possibility to play simultaneously with the two related instances and to declare their relation to be either of aggregative or of associative nature (following the COMO terminology). The inheritance ISALSO is an elegant way to solve the problems of run time re-classification and of migration, simply by eliminating them. It also provides a way to handle the delicate situation of multiple inheritance and gives a possibility to treat alternative inheritance properties (the nightmare of modelling **amphibian** vehicles, which are **vehicles**, like **cars** and **boats**, and which inherit from both **cars** and **boats**!). It finally enables the definition of "roles" for objects of a given class without modifying the definition of the class: e.g. in the appropriate context, a class **husband** can be defined on the top of a general class **male**, through an associative ISALSO relation.

The second question, less fundamental, concerns the semantics: should a clear distinction be made between the situation where a superclass is concrete, namely describes the characteristics of objects which may have an existence, and the situation where the superclass is abstract, namely cannot support the existence of such objects?

COMO suggests the name **ISA** when the superclass is abstract, and the name **ISAKI** (for IS A KInd of) when the superclass is concrete. The ISA inter-class relation corresponds to a **specialisation** process. The superclass, despite its abstractness, has the knowledge of all properties of the objects of its subclasses: e.g. in the relation "Box ISA Solid_", the class Solid_ knows which services can be invoked from the objects of the class Box. The fact that an object of the class Solid_ happens to be a Box appears as a detail. It could as well be a Tube, a Cone, .., it reacts primarily as a Solid_, a property known as polymorphism.

The ISAKI inter-class relation between a concrete subclass and its concrete superclass has to do with the extension of the properties of the superclass and may also be used to modify some of its properties through method overloading. It corresponds to a **generalisation** process. The objects of a subclass have the obligation to fulfill all commitments of the superclass. It is recommended to avoid dealing at a given time with groups of objects belonging to several 'ISAKI related' superclass and subclasses.

The ISALSO relation can be used to cover all other inheritance situations. New classes can be defined to simulate the addition of **essential** properties to classes which already exist and are simultaneously used in the same context. The ISALSO inter-class relation is superimposed to an inter-instance relation of either aggregative (APASI → **APALSO**) or associative (OCASI → **OCALSO**) nature; the aggregative case, an alternative to the ISAKI inheritance, illustrates the possibility mentioned in section 4.2 of a **part** existing prior to the aggregate.

Last, anticipating on the introduction of utility processors in section 4.6, the ISALSO relation gives great freedom to the users to build specific classes upon general utility classes, then achieving an effective re-usability of code.

4.4 The graphism and notation system

All relations defined in COMO for the description of an object are uni-directional relations. The class diagrams can therefore be sketched for each (composite) class separately. As a consequence, the graphism adopted to represent a class can also rely on topological conventions: one iconic box per class, with full or dotted, straight or kinked lines leaving the box either from the top side or from the bottom side to identify the various relations, as shown in Fig. 1, 2 and 3.

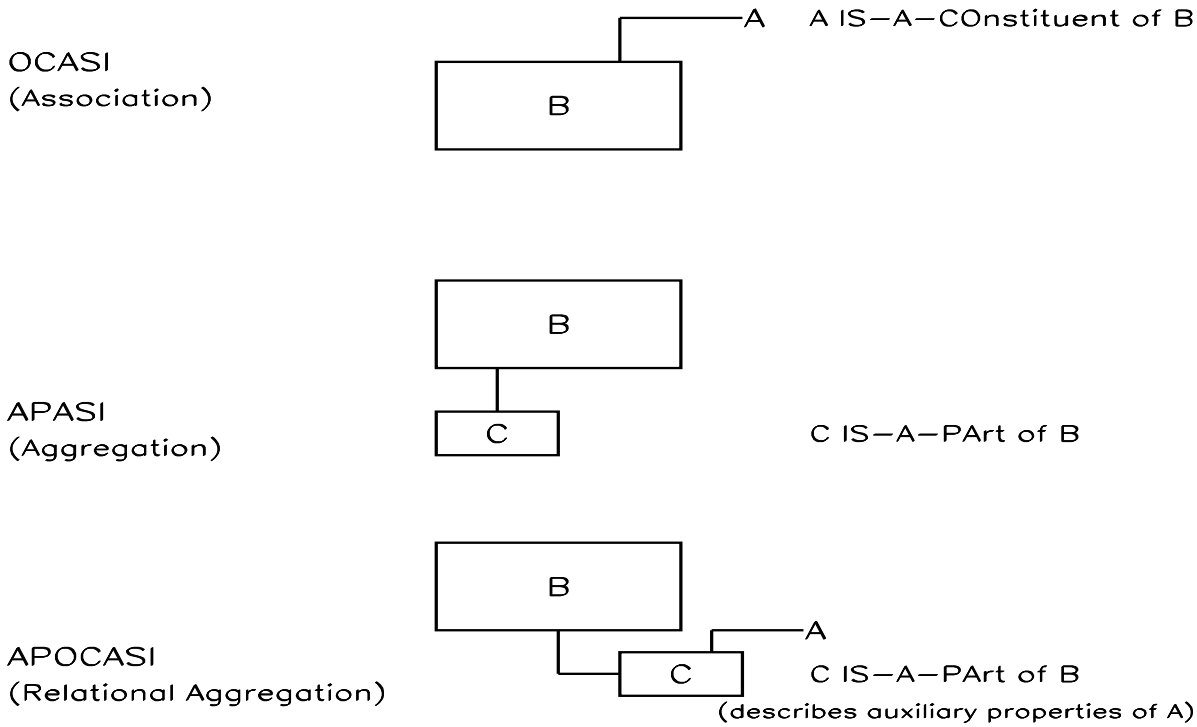

Figure 1: Composition relations

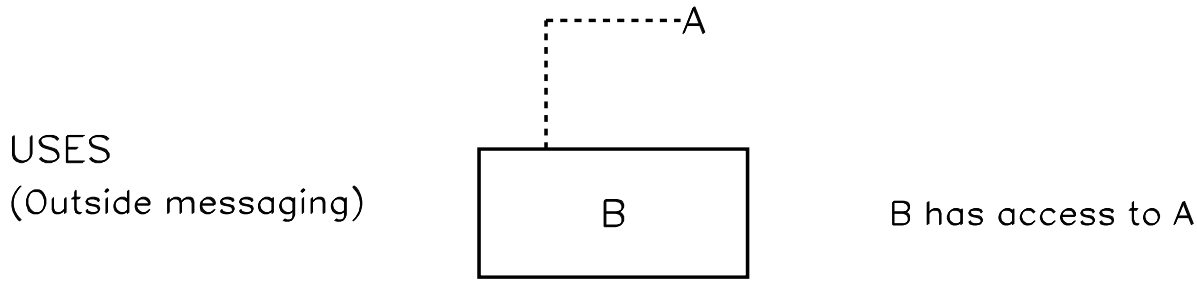

Figure 2: Outside messaging relation

The object diagrams proposed by most methodologies seem to be not much useful. A scenario can as well be described, much more economically, by a sequence of messages listed in clear.

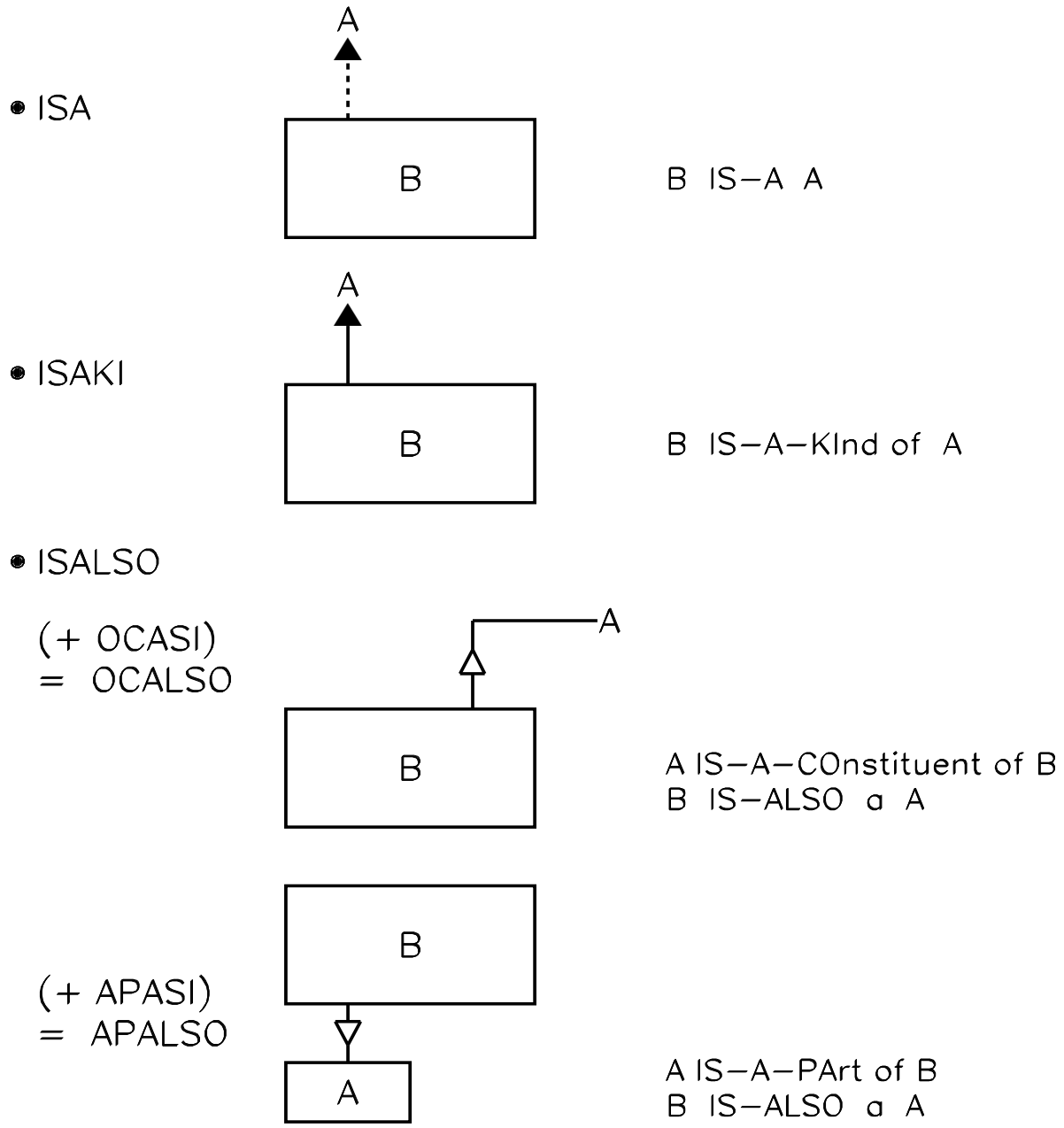

Figure 3: Inheritance relations

Participation symbols, to express cardinalities unambiguously, can also be proposed, as shown in Fig. 4. It should be noticed that the symbol is attached to the class to which the relation belongs. This is the inverse of the usual anglo-saxon convention.

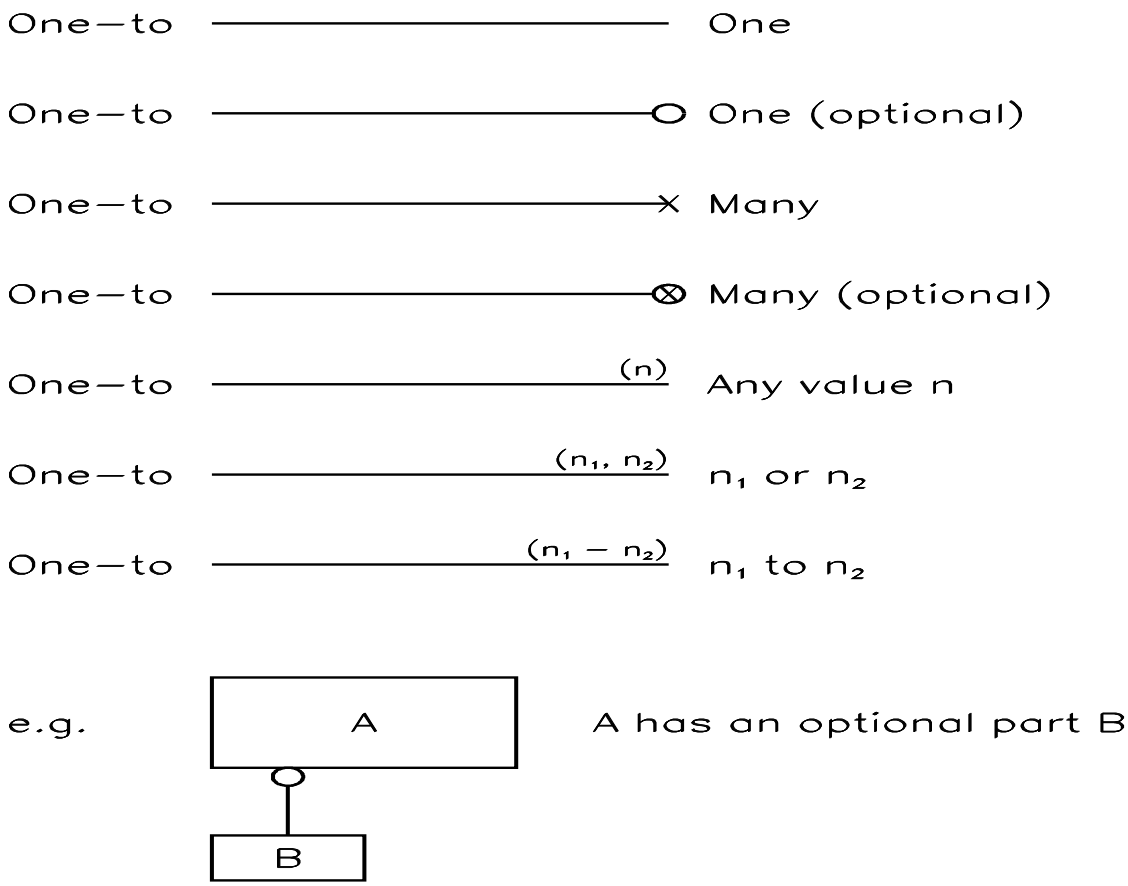

Figure 4: Participation symbols (cardinality)

4.5 The basic generic classes

The following generic (template) classes, which define specific characteristics of groups of objects, are proposed:

Family object, a container which has as **parts** any number of objects, all from the same class (or from the "ISA related" subclasses of the same class).

Collection object, a kind of container for a stable family of objects. The **parts** of the family are in a given order, not modifiable except by addition of new objects.

Fan-out object, an artificial aggregate whose **parts** can be objects of any classes.

List object, which has as **constituents** any number of objects, all from the same class.

Because of the aggregative nature of the parts, killing a Family, a Collection or a Fan-out has the effect of also killing their parts. Because of the associative nature of the constituents, killing a List does not affect the constituents.

Letter shapes are tentatively suggested for the iconic representation of these template classes (Fig. 5).

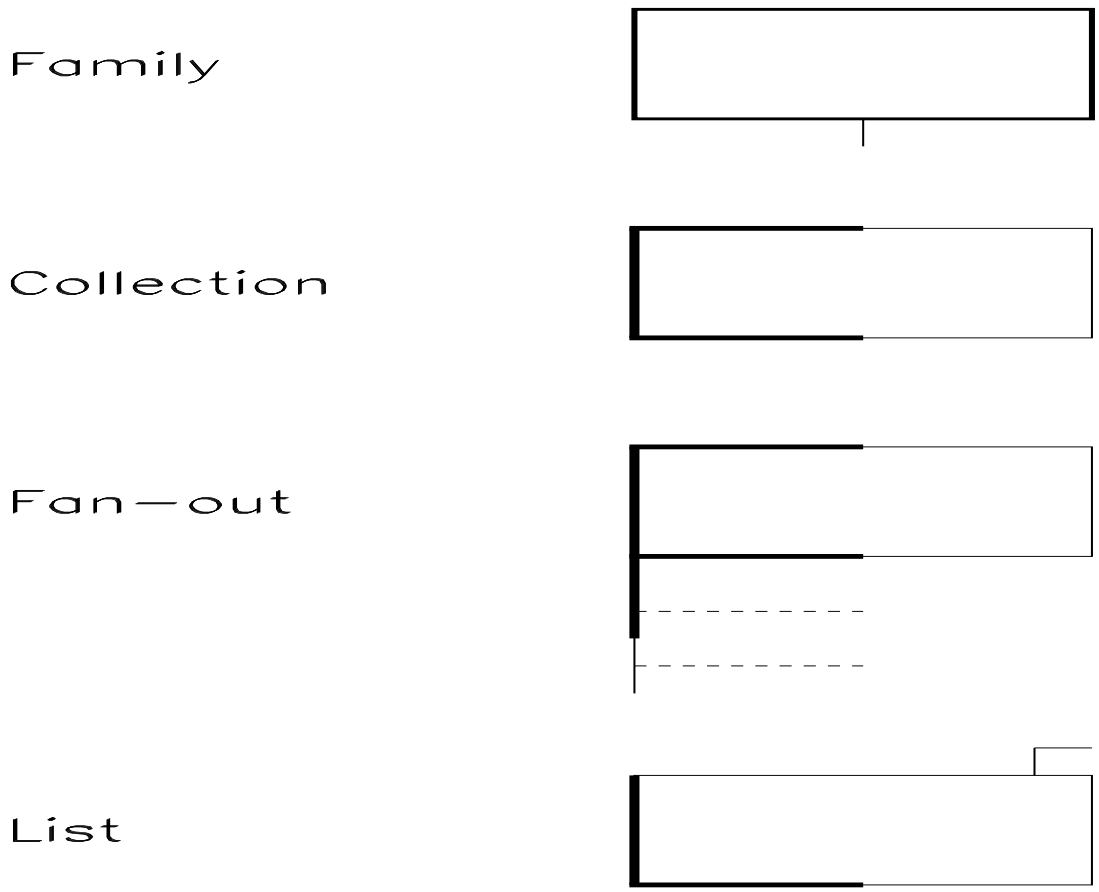

Figure 5: Basic generic classes

4.6 The processor modelling features

As mentioned earlier, all environment classes, also called **user_processors**, are subclasses of the abstract class **Processor_**. They can be thought of as "sub-steering" kinds of classes which, in addition to their internal (private) services, have public services performing given major tasks. The public services provided by the user_processors are of two kinds, either **Access** services or **Action** services. They can be designed so as to make possible the evolution of the objects, when executing in a multi-process environment.

An **Access** message is sent by an object, the sender, to a user_processor object, the receiver, to collect information required from the receiver. The corresponding Access service modifies the state of the sender but does not necessarily change the state of the receiver. An Access message is expected to return a status, to tell the sender whether the request has been honored or cannot be honored. The first case corresponds to the situation where the receiver either was ready to honor the request and has honored it, or has successfully triggered an Action service to honor it. The receiver has usually an internal status saying whether the relevant information is available or not. The second case

means that an exception has occured upstream, which inhibits delivery of the information required by the sender.

An **Action** message is sent to a user_processor object to execute some specific task, and is not expected to return any results. The corresponding Action service is expected to change the state of the receiver. An Action message returns control to the sender either as soon as the task is initiated, when the service is under control of a different process, or after completion of the task, when executed as part of the same process. In order to achieve the invariance of the code where the messaging sequences occur, an Access service may execute a "wait for completion of Action" when the receiver is under control of another process than the one of the sender.

When relevant, the user_processor objects can control the correctness of the sequencing of Actions, through the use of appropriate internal status variables. Of course, the user-processor classes may also have internal services other than Access and Action ones.

The bulk of the code that a programmer has to write is clearly located in the Action services, which often themselves need to invoke Action services from other user_processors. It seems therefore desirable, in order to minimize the amount of code initially loaded when executing a program interactively, to give to the user_processors a modular internal structure. This could be achieved in the following way: every substantial Action service is regarded as a service of a **user_action** class encapsulated in the user_processor class. When receiving an Action message, a user_processor object checks that the relevant user_action object exists (namely is loaded and ready to execute) and, if not the case, it takes steps to create it, then delegates the message to the user-action object.

The user_action objects are aggregated to the user_processor objects through APASI relations.

It is rather frequent that different user_processor classes do have Action services which, ultimately, need to perform similar sub-tasks of purely deterministic nature through general, sometimes complex, mathematical or logical algorithms. To avoid duplication of code there is a benefit to define general classes whose services can take care of these sub-tasks. They are called **utility_processors** and play the role of utility routines in traditional software.

The utility_processor objects may also be repositories of **utility_data** objects that they create. The utility_data objects can be treated as pre-existing **parts** of user_data objects created later by Action services of the relevant user_processors. When a user_data object has the same basic characteristics as a utility_data object its class can be declared as related to the utility_data class through an ISALSO relation. This gives the great flexibility to play with utility_data objects which, on their own or in comparison with others of the same class, might not fulfill criteria of acceptance required by the client and would have then to be deleted, the creation of the user_data objects taking place only when they have successfully passed the tests.

Libraries of utility classes can be developed, partly *a priori*, by anticipating on the expected usefulness of their functionality, and completed *au fur et à mesure* when the benefit of a generalization *a posteriori* becomes evident.

4.7 Program construction and execution control

With traditional software, a program consists of a piece of code, usually named "main", and of all subroutines called either directly by the main program or indirectly by the

subroutines themselves.

The OO integrists often show some commiseration for the poor traditional programmers who, like Sisyph or Penelop, have painfully to redo the entire work every time they design a new program. For them, one does not write programs, one writes classes! Their opinion, though not formally wrong, seems to be a damned exaggeration, most likely induced more by a lack of experience with well structured large traditional software than by the deliberate intention to bluff.

Quite a number of OO practitioners, not to say victims, know where they have been led by giving too much credibility to such an idyllic vision: in many cases, re-usability has so far turned out to be an illusion or, at best, a source of inefficiency. Conversely, there are large traditional software programs whose effective re-usability has been pushed very far: at CERN, in particular, many utility packages, including the Detector Simulation program GEANT3 [11], are remarkable examples of re-usable code, despite other weaknesses due mainly to the absence of encapsulation.

This being said, I nevertheless believe that the OO argument of (efficient) re-usability still can apply to our specific domain of applications provided the way the classes are built follows adequate design concepts. This point seems not to have been given enough attention so far. The proposed COMO approach is **one way to give some coherence to the construction of large programs and to guarantee that they can be developed without any risk and without loss of efficiency**.

From the COMO point of view, the execution of a given OO program starts with the creation of an object, a unique instance of the corresponding **user_program** class (a sort of user_processor class), and with the triggering of a pseudo-service (a sort of processor action) which itself invokes services of user_processors.

All user_program classes inherit from the same abstract class **Program_**, one of whose attributes is a **JobID** object identifying uniquely the job which executes the program. The user_program object is the repository of the information (the input parameters) required for a given execution of the program and of any information reported during the execution and considered as useful to save.

When running a **production** job, a dedicated **production control** server allocates a JobID object to the given job. After completion of the program task, and before the execution stops, it is recommended, at least for any official production, to store some parts of the user_program object in an appropriate **production database**. It is suggested that any object created during the execution of a given program and stored persistently has access, directly or indirectly, to the corresponding JobID object. The JobID value can also contribute to the identification of any file (of event data or of calibration data) produced by the execution of the program and recorded on an external medium.

The messaging and modelling features of the user_processor concept provide clear guidelines to the programmers for structuring programs to be executed in batch mode. Ultimately, the sequence of services invoked during the batch execution of such an OO program should not differ significantly from the subroutine flow chart of the equivalent traditional program except in the areas, usually well localized, where it is possible to simplify it by using the inheritance relations.

5 Conclusions

When deciding, years ago, to study the adequacy of the OO technology for the HEP software, I could not imagine how long, and sometimes painful, the climbing path would be, and how strong the feeling of loneliness. From the intermediate point reached today the panorama is nice and rewarding: one can contemplate the hidden beauty of the OO world and even see the top! However, going on climbing is too risky, given the rarefaction of oxygen and the absence of sherpas! So I shall just write a few comments, dedicated to whoever will participate in the next expedition.

It is a priority for the HEP community, at this point in time, to think about our specific needs, our user requirements, and to brainstorm about possible ways to fulfill them. In parallel, one has also to study the evolution of solutions proposed outside and made commercially available. Then, later, it will be possible to choose, and maybe to adapt, any products that would seem appropriate.

In the recent period I have often reviewed in my mind a number of software problems encountered in the last three decades, from the Bubble Chamber era to the LEP era, through the European Hybrid Spectrometer and GEANT3. This has been most helpful to assess my own conviction on the validity of the COMO approach, regardless of the surrounding technology.

Independently, I am seduced by the OO concepts and hope very much that the weaknesses of the OO languages will be cured in a near future. OO Analysis and Design, however, is far from having reached maturity. It is not yet a science, just bricolage with recipes which sometimes, like for cooking, become an art.

Of course, the eventuality to consider COMO as appropriate to our domain of applications does not provide us with a complete methodology, namely with friendly tools needed to develop, use and maintain efficiently a large distributed software, for data visualization, coherent documentation, code generation or quality assurance. Existing methodologies could possibly be adapted, and their tools interfaced, to the specific COMO semantics.

For what concerns the implementation of COMO itself, it would be a very small investment, because COMO consists mainly of an *état d'esprit*, of guidelines for building re-usable classes in a coherent way. It would undoubtedly be more economical than the recurrent payment of license fees for commercial methodologies.

Time is not critical as it seems unfortunately that we have more time than initially planed, but this is not an argument to waste it. Indeed, there is a reason why, I believe, one should move ahead as soon as possible and this is related to the fact that, right now, many physicists are busy working on the software needed for the optimization of the Atlas and CMS detectors: the sooner they could integrate, if not formally at least in spirit, features which could help for a later conversion to the OO technology, the better. The physicists would then be given the possibility not only to understand and to accept the change over, but also to actively contribute to it, a necessary condition for the success of the future experiments.

Acknowledgments

†*Merci, Olivier, pour ta présence et ton soutien toujours aussi stimulants*

I would like to thank P.G.Innocenti (ECP Division Leader, 1990-1994) for his continuous support and also D.O.Williams (CN Division Leader) for his friendly encouragement.

I am grateful to M.Storr (ECP Divisional Training Officer) and to J.Deacon (Consultant for training in Software Engineering matters) for many useful discussions. Their interest stimulated me to complete this work.

Finally, I would like to thank S.Banerjee (TIFR, Bombay) for his continuous assistance.

References

[1] S.Banerjee, F.Bruyant. CERN-L3 Note 748 (May 1990).
F.Bruyant, N.Colino. CERN-L3 Note 761 (July 1990).

[2] J.Deacon. *Object–Oriented Modelling: the Requirements.* October 1994.

[3] F. Bruyant. *Skeleton Logic for Analysis Chains.* MC91 Workshop on Detector and Event Simulation in High Energy Physics, Amsterdam, April 1991.
F. Bruyant. *HEP Software Design: How to reconcile the immediate needs and the trends for the future.* 2nd International Workshop on Software Engineering, Artificial Intelligence and Expert Systems in High Energy and Nuclear Physics, La Londe-les-Maures (France), January 1992.

[4] P.P. Chen. *The Entity-Relationship Model: Toward a Unified View of Data.* ACM transactions on Database Systems, no.1, p 9 March 1976.

[5] J.Rumbaugh et al. *Object Oriented Modeling and Design.* Prentice-Hall, 1990.

[6] C. Batini et al. *Conceptual Database Design. An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[7] S.M. Fisher, P. Palazzi. *The ADAMO Data System.* CERN, 1992.
This paper gives a list of all contributors since 1987.

[8] Ph. Desfray. *Ingénierie des Objets - Approche CLASS-RELATION - Application à C++.* Editions MASSON, 1992.
Ph. Desfray. *Object Engineering - The fourth dimension.* Addison Wesley, October 1994.

[9] R. Brun, J. Zoll. *ZEBRA User Guide.* CERN Program Library Q100.

[10] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, September 1995.

[11] R. Brun et al. *GEANT3 Users Guide.* CERN Program Library W5013.