**An extensible environment for views in Eiffel**

Robert Chignoli, Pierre Crescenzo & Philippe Lahire

chignoli@unice.fr, crescenz@tigre.unice.fr, lahire@unice.fr

Laboratoire I3S – URA CNRS 1376 & Université de Nice-Sophia Antipolis (UNSA)
250 Av. Albert Einstein - Sophia Antipolis - 06560 Valbonne - FRANCE

Tel : 33 92 94 26 85 – Fax : 33 92 94 28 96

## Key words :

Object technology, view, persistence, meta-programming, Eiffel

## Topics :

Class management and class evolution - Reuse, components and frameworks - Applications and experiences - OO databases and object persistence

# Contents

**An extensible environment for views in Eiffel**

Robert Chignoli, Pierre Crescenzo & Philippe Lahire

Laboratoire I3S – URA CNRS 1376 & Université de Nice-Sophia Antipolis (UNSA)
250 Av. Albert Einstein - Sophia Antipolis - 06560 Valbonne - FRANCE

## Abstract :

We are explaining here the motivations and the main points of a project which aims to modelise and implement mechanisms of view around the Eiffel language. The concept of view we are developing here, aims at enabling to describe in the same time the needs coming from the DBMS, where a view is basically a way for handling and organizing into a hierarchy, collections of persistent entities, and the needs for object type evolution. The approach we have carried out is based on four notions :

- the notion of "basic language" ( *Ofl*$_{zéro}$ language) provided with a set of primitive mechanisms derived from Eiffel, but without inheritance mechanisms.
- the notion of "link" enabling to add additionnal mechanisms (a type of inheritance, etc)
- the notion of "meta view" which enables to build "new languages" by combining the primitive language together with the links (Eiffel 3, Eiffel provided with a mechanism "is a version of", etc)
- and finally the notion of "view" which will intuitively correspond to extensions of the classic concept of class and will be directly manipulated by the programmer.

Close to initiatives like CLOS/MOP, our project is rather different, first of all because of its Software Engineering aspect, or in other words because the basis we are taking is the industrial development languages ; and secondly, we will be giving much importance to the persistent problems.

## Key words :

Object technology, view, persistence, meta-programming, Eiffel

# 1 Introduction

The emergence of the object Technology in the Software Engineering field [MNC+91] has had, is continuing to have, and will probably still have a great impact on all the fields dealing with the problems of software design and more particularly with the problems of components reuse and reverse engineering.

Actually, this technology allows to consider a larger and a less expensive automatisation of activities dealing with the evolution of applications and thus, it could lead to a better control of the computerization process.

In particular, the emergence of sophisticated technics of "persistent object"[1] will allow to widen the application field of this new technology which should in the end become widespread in the software industry.

Object, re-use and persistence are thus in the heart of the actual problematics and one of the main stakes consists in reconciling these three directions.

But some non trivial problems are resulting of this association. For example, since its birth, the inheritance concept[2] is a matter of heavy discussions, which we could sum up here with the following questions :

- inheritance vs evolution ([BK87])
- inheritance vs aggregation ([Mey90])
- inheritance vs points of view ([Car89], [Bou94], [CK94])

[1] Via extensions to existing languages or through OODBMS.

[2] To be more precise, the inheritance concepts (simple, multiple, Eiffel like, Smalltalk like, etc)

- inheritance vs persistence [Lah92] [ADF+94]

One promising direction seems to be to develop a concept of view/point of view integrated in programming languages which are dedicated to Software Engineering (Eiffel, C++, etc). This article is a contribution to this theme ; it is introducing the main characteristics of the $O^{fl}$ computerization environment which is under going definition by our team.

$O^{fl}$ is an extensible computerization environment with views derived from Eiffel [Mey94]. According to its architecture, $O^{fl}$ is made of a *minimum* basic language (Eiffel like without its inheritance mechanisms but with a minimal persistence service), and of a *system combining links* which enable to add additionnal service levels ("Eiffel like inheritance" link, "Points of view" link, "ODMG-93 relation like" link, etc) This providing thus the programmer with a library of compatible languages[3].

The $O^{fl}$ project is the continuation of the FLOO project which has leaded to the development of a persistent computerization environment for Eiffel[CFLR93b] [CFLR93a] .

After this introduction, section 2 is proposing a short overview on the notions of *views* and *points of view* through the litterature and it is explaining the choices which have been done in $O^{fl}$ . Section 3 is introducing in a more detailed way, the model we have chosen. Section 4 is giving some possible use of our environment and finally there is a detailed conclusion on the main points dealing with the implementation of such an environment.

# 2 Points of views on views

In this section, we are successively introducing the potential uses of the notions of *views* and *points of views*, the main types of possible approaches, and finally, a recent example focussed on Eiffel (the VBOOL project [MKC95]). The last paragraph is giving the conclusions of this study and setting out the main choices done for $O^{fl}$ .

## 2.1 The potential uses of the views

The notions of *view* and *point of view* are put forward in many situations in order to solve different types of problems. The main proposals are based on the "object model" and consist in installing *new* mechanisms above and thanks to the inheritance relationship. They are supposing the existence of *schemes of classes* which define hierarchies of types and allow to *create* occurences of typed objects. Most of them are concerned by the handling of the occurences via the concepts of *collection* and *query*. The potential uses are :

- hiding informations
- restructuring informations
- simplifying queries
- dynamic definitions of collections
- making models evolve
- making objects evolve
- checking the access to objects according to their content

[3] The notion of compatibility is covering different aspects : a technical aspect (inter-operability of the entities generated by the programs) and a logical aspect (execution of the work which the programmer is expecting for). This theme is mentionned in section 3.1 (page 9).

- integrating heterogeneous systems
- authorizing the independence of data
- ensuring the compatibility with the relationnal model

We are briefly talking about each of these uses hereafter. In this first approach, two *classes* (figure 1, page 3), with the usual meaning of object languages, are taken as illustration basis. An intuitive notion of *view* is going to be introduced thru a *had hoc* syntax.

```
CLASS Car                      CLASS Motor
   mark :  String                 capacity :  Integer
   model :  Model                 type :  String
   motor :  Motor                 power :  Integer
END CLASS Car                  END CLASS Motor
```

Figure 1: *Car* and *Motor* classes

### Views for hiding information

The lay out of the information has to be changed by masking source class characteristics.

```
VIEW Simple_Car
   FROM Car IMPORT model
END VIEW  Simple_Car
```

### Views for restructuring information

Here, the view enables to change the lay out of the information by bringing together characteristics coming from different source classes. We can consider this case as a generalisation of the previous one.

```
VIEW Model_MotorType
   FROM Car IMPORT model
   FROM Motor IMPORT type
END VIEW  Model_MotorType
```

This is corresponding to the following effective view :

```
VIEW Model_MotorType
   model :  Model
   type   :  String
END VIEW  Model_MotorType
```

`Model_MotorType` is thus offering a partial view on the `Car` and `Motor` classes and is specifying the *couples* associating a name of car type together with its motorisation capabilities.

### Views for simplifying the queries

A query allows to select a part of a collection of objects (usually persistent). In the object model, an essential point consists in locating a query result at the right level in the graph of classes. In this context, the notion of view may provide an answer on how to solve this problem, a query result is associated with a *view* which is something different from a class.

#### Views for defining collections dynamically

We need here to partition a collection of objects in collection of smaller size. The adjunction of a *criteria* in the view clauses allows such a use.

```
VIEW Car_Peugeot
    FROM Car IMPORT model
    WHERE Car.mark = "Peugeot"
END VIEW  Car_Peugeot
```

The reader will notice that the criteria chosen in this example is particularly simple. In a more general manner, there is going to be in this context an expressiveness close like "SQL" languages.

#### Views for making the models evolve

A concept of *view* is integrated in order to allow to test the changes to be done to the model. (*Scheme of classes*). In a soft and dynamic way, the scheme of views wants to give here a new look to the real scheme and wants to enable to a priori study the effects of the changes done to the built-in scheme of classes.

#### Views for making objects evolve

We are looking here at the situation in which an objects basis is containing different versions of objects. In this situation, the views may provide an unified mechanism for accessing these objects.

#### Views for checking the access to the objects

Any mechanism of protection is offering to the user a point of view on real entities. Introduced like this, this use is becoming a special case of collection views.

#### Views for integrating heterogeneous systems

This use is dealing with inter operability problems (different data handlers, different operating systems, etc). The views can provide a transparent mean to access the information. This situation could be associated with the more general case : information *restructuration*.

#### Views for authorising the independence of the data

The object engine of the computerizing environment enables to define the *hard* types of the objects (when creating them) and the views are the only way to access existing objects. Views are here, for example, implemented by a "client" application manipulating objects which are handled by a "servor". This case could be considered as a generalisation of *test of evolution* mechanisms.

#### Views to ensure the compatibility with the relationnal model

One concern in the computer science world is the durability of the investment. The relationnal model has been for a long time the solution for storing data and now, the object oriented database managment systems are coming to maturity. The transition from a model to another one should be done in a soft way. A system of views could be then a solution for accessing from the object world to relationnal database in a quite satisfactory manner. The views can be used as a gateway towards relationnal data bases.

### 2.2 The different approaches

The uses we have introduced above are missing the main motivations of the different proposals of *systems of views*. In fact, each proposal is putting ahead a specific field (data base, software

maintenance, etc) and one or several uses. For example, among the proposals which are developed hereafter :

- in [BK93], views are first of all a mean to access in a different way to already existing data
- in [AB91], views are a mean to compensate for the lack of object models flexibility
- in [SS89], a view is a simplifying abstraction of a complex structure. The use of views is considered in an environment of software development and maintenance
- in [Ber92], views are representing a multi-uses unified mechanism which is covering at the same time the handling of collections of occurences, the problems of scheme evolution and the problems of *points of view* on objects.
- in [SLT91], the OODBMS are supposed to be offering at least the functionnalities of the relationnal DBMS. A solution has thus to be found for separating the global- and conceptual-scheme of the data base from the external scheme - lower scheme - which is specific to one task. The views are here a mechanism for the independence of the data.

Each proposal is thus establishing a relationship between the potential users, the priviledged fields and the specific motivations. [CK94] is making a synthesis of it by proposing a partitionning of the approaches. And it is this partitionning which is taken here : the *methodological* approach set a framework for simulating the concept of view with the language existing concepts (library of classes provided with ad hoc methods). The approach via the *queries* is mainly dealing with the partitionning of collections of objects and with the *dynamic* typing of the collection join/project results. Morever the approach based on the *scheme* is bringing together proposals of a more general incidence which are defining and introducing in the heart of the model the additionnal concept of view by binding it with the semantics together with the other available concepts.

### Methodological approach

An example of this type of approach is described in [BK93]. Its advantage is that it does not require the definition of new concepts for the object paradigme. Views are described thanks to the classes and the interaction between basic objects is done thanks to methods. The authors are giving examples of operations allowing to define views with selection, projection, and join.

From the authors point of view, when combining these three possibilities, it is possible to represent *anything* but people must be aware that these views should only be used for accessing data in a different way and that they are filled with existing objects. So it is not possible to create new objects through the view. In this approach, the programmer is fully in charge of views. In fact, the authors are providing a method for implementing the views manually. They do not define a concept of view.

### Approach based on the queries

[HS90] [SLT91] are describing a proposal based on this approach. The field which is concerned about is mainly the data base one and it is quite inspired by the relationnal model. In this approach, a view is defined, first of all, through a query. The existence of a powerfull mechanism of queries containing plenty of operators is mandatory. The developer starts describing his view thanks to a query. An automatic mechanism is responsible of generating classes thanks to the descriptions of views. These classes are then inserted at the right place in the scheme of the application. More complex is the view, more complex will be the insertion in the scheme of the classes which have been generated. In order to make the insertion possible, it is in some cases mandatory to automatically reorganise the scheme[4].

[4] With the considerable drawback that it becomes difficult to understand when the scheme is modified in an automatic way.

**Approach based on the scheme**

The expression *based on the scheme* means that the approach is offering to the programmer capabilities for modeling an application[5], in order to directly manipulate an additionnal concept of *view* which is clearly identified in the language. For example :

- [AB91] is proposing a concept of view which enables to mask a hierarchy of classes and enables operations for generating, deleting and modifying underlying objects.
- [Ber92] is giving a greater number of uses with a unique mechanism able to provide :
  - short cuts for query description and the storage of query results as views,
  - the dynamic definition of sets and partition of classes,
  - an authorization mechanism,
  - a mean for making the scheme evolve,
  - a version handling,
  - several lightening on objects.
- [MKC95] [MEC95] is proposing to add to the Eiffel language two concepts, a concept of *views* and a concept of *visibility* which lead to the definition of the VBOOL language which expands the syntax and semantics of Eiffel. In the mind of the VBOOL designers, classes and views are cohabiting within the language :
  - a view of a class C is another class declared in a `seen_as` clause of C. A class which have a `seen_as` clause is a *flexible class*.
  - a point of view on a class is a relationship associating this class with a subset of its views. An attribute `X` may then have a type corresponding to view names of the flexible class, expressed by `feature X : FLEXIBLE_CLASS_NAME ( VIEW1, VIEW2, ...)`.

  Two types of classes can be found in this proposal and they are designed for different views : the implementing classes, in other words the classes with the usual Eiffel meaning, and the flexible classes which handle the mechanisms of views. People has to notice that the model is proposing solutions to different problems (mutual exclusion of views, dynamic evolution of points of views).

## 2.3 Discussions and proposals

The above study has given a short overview of the variety of proposals dealing with views[6], which makes us now able to define in a more precise way the context of our study. It is worthwhile to remember our starting motivations which were found in the FLOO project in which we have proposed a model and an implementation of a persistent environment for Eiffel:

- *(a)* providing a framework which allows the automatic and controlled evolution of persistent objects and their models,
- *(b)* Widening the application area of the FLOO query mechanisms by allowing the creation of dynamic types.

In other words, this is corresponding to the 7 first uses of the list given section 2.1[7].

[5] In other words, to define the scheme of entities.

[6] We did'nt talk about fields dealing with knowledge representation where the reader will find additional points of view on views.

[7] restructuring, hiding, simplifying, defining, testing, handling, checking

**Approach *versus* use**

A *methodological* type approach is not sufficient for covering the set of uses we wish to deal with. An approach with *queries* could be the answer to our purpose, but such approach being essentially focussed on the algorithmic aspects of occurences collections, it would not be sufficient for our purpose *(a)*. On the other hand, as the VBOOL project is showing it, a *scheme* type approach which brings an additionnal modelisation concept, may be considered.

*The programmer has to be allowed to access directly the concept of view.*

**Inheritance, persistence and evolution**

The inheritance is a mechanism installed in the heart of object language because, at the same time, it is its major characteristic and secondly because it gives this way the warranty of a certain efficiency. The inheritance concept may be considered, in a general manner, as a mechanism for evolution, but it is more specifically adapted to the evolution of the models (re-use of classes in order to create occurences on *new* objects) and it is used not really *accurate* for the handling of the evolution of a couple (scheme-collections of objects).

*The inheritance concept is not sufficient for a persistent language.*

**Inheritance, views and points of view**

The cohabitation in the same language of both inheritance and concepts of views is making important problems (cf VBOOL for example). It is particularly true if we consider that although both exist in the same language, they don't have the same status, the inheritance keeping its supremacy. Then it may be interesting to examine this new contribution in a different situation : for a given occurence, the inheritance is nothing but a particular handling of a link which is associating it to its model. In other words, the inheritance may be handled at the same level as other links like those reducing the view on the characteristics of an occurence or those reducing the view on a collection of occurences, or even those *dynamically* combining two models.

*An inheritance mechanism may be interpreted as a special case of link.*

# 3 General introduction of the model

The last paragraph has put forward the difficulty to integrate mechanisms of view/points of view in languages which provide built-in inheritance mechanisms. Also, it has shown the concepts leading our work. This let us going further, in this paragraph, according to the concepts kept for $O^{fl}$ and its architecture.

## 3.1 The basic concepts

The programmer has to be able with $O^{fl}$ to handle generic views. The basic operations which will be available for the programmer are the following ones :

- the capability to *create* a view thanks to other views. For example, to type an attribute A1 of a view V2 and then to *go through* attribute A1 for accessing its characteristics,
- the possibility to *generate* (eventually persistent) occurences of these views.

To do it, we have specified a language provided with a *minimum* semantics, $O^{fl}_{z\acute{e}ro}$ . This language allows to define generic views, to instanciate *private* objects (expanded object) or *shared* objects (reference).

$O^{fl}$*zéro* is also provided with a panel of *redefinable* operators which allow to define the semantics which is relevant according to needs. This deal with the most common operators (assignment, message passing) but also with more specific operators which are provided by $O^{fl}$*zéro* (the different ways for handling the collections of persistent objects, etc). For instance, $O^{fl}$*zéro* implement a two step message passing mechanism : message searching / message execution inspired by the *lookup* $\circ$ *apply* model [MDC92] and it will be possible in the $O^{fl}$ environment to program both steps.

Because our approach is refering to a "Software Engineering" context, we have chosen the Eiffel language as the basis for the language used by the programmer. It is true for both the style (syntax) and also the expressiveness. These choices are summarized in paragraph 3.1.3. But it is important to mention right now that $O^{fl}$*zéro* is differing from Eiffel on a fundamental point : it does not provide any inheritance mechanism at all.

With $O^{fl}$, such a mechanism is going to be available for the programmer thanks to a *mechanism of links* which is outside the basic language. So, the additionnal concepts, which a programmer may need for modeling or implementing an application, are going to be introduced thanks to this mechanism of links : for example, *a certain way of inheriting* or *a certain mechanism "is a version of"*.

More precisely, these links are acting at a *meta* level : there is no way for the programmer to define a new link (for example, the simple inheritance) for a given application. On the other hand, a meta programmer will be able to add new links for defining a new language (provided for example with the simple inheritance, and the programmer will then be able to use this new language[8].

For the meta programmer, *to define a link* is one of the task leading to the definition of a new language. The set of tasks is the following one : *to choose* a kernel language ( $O^{fl}$*zéro* or another higher one), then *to define* the new links and to integrate these new links by composition with the existing links.

The next part of this section is giving precisions successively on our concept of view, on the implementation of links and gives some elements on the basic language.

### 3.1.1 The notion of view

A view is a describing model for its occurences. It is containing objects[9]. It is defined by :

**a name :** representing the name of the view for the programmer.

**characteristics :** corresponding to attributes (constants or variables) and to routines of an *Eiffel 3* class in the $O^{fl}$ environment. These characteristics may for example come from declarations of characteristics *redefinition or renaming* coming from other views (depending on the links implemented in the application).

**exporting indications :** which enables the programmer to handle the visibility which is provided to the clients views, thanks to a built-in exportation mechanism copied from the *Eiffel 3* one.

**a collection :** it is defined at the moment of the declaration of the view with a criteria or an extension. Its composition will, one more time, depend on the links being implemented at the moment of the declaration of the view.

[8] The meta programmer will then set the semantics of generic operators within the link.

[9] As soon as it is dealing with a non generic view or with a creation of a generic view occurence.

### 3.1.2 The notion of the link

We are explaining here how to *define a link* and what does the expression *add a link* mean.

To define roughly a link is the same as describing all the aspects of a relationship between two views, or even the same as pre-determining, at run-time, the behaviour of objects of any view built by using this link.

Any definition of a link is associating with it a *name* and it is containing 3 clauses (*execution*, *modelisation* and *extension*) giving the semantics of the link by establishing the relationship (which will exist in the language using this link) between any *mother* view (origin of the relation) and any *daughter* view (destination of the relation). These definitions are based on the availability in *Ofl zéro* of a panel of redefinable operators which the meta programmer may redefine[10]. Each of these clauses is related to different operators.

**the execution clause** is describing the behaviour generic operators at run time for this link (as message passing, assignment, creation, copies and comparison). It is defining the behaviour of the daughter view on the basis of the mother view.

**the modelisation clause** is determining the semantics given to the generic operators for modelisation (redefinition, renaming, selection and deletion); in other words, for instance, if it is possible and what it would mean for this link to rename features of a daughter view in a mother view.

**the extension clause** is setting the composition of collection of occurences of any view built with this link (combination or not with the extension of mother views, etc)[11].

In this context, *to add a link* is the same as *to do a composition* of the above different clauses to determine the semantics of the new language. In fact, this composition will consist in the definition (for each clause) of new algorithms based on the implementation of the link redefinable operators available in the chosen language and the new added link.

It must be noticed that, according to the clauses, these algorithms will address different aspects of the language : regarding execution and extension clauses, the algorithm is mainly dealing with the handling of occurences ; regarding the modelisation clause, it is rather distinguish the handling of views, or in other words, the types of occurences.

Finally, we have to be aware that the basic purpose of our study is to look for solutions for improving the capabilities of object evolution; so that, we are not *a priori* planning *a priori* any limitations in the combinations of language towards objects (especially persistent ones). Thus, a persistent object which is generated by an application written in *Ofl* language will be directly accessible with an application written in a language of any level (which is intuitively completely corresponding to a point of view). It is up to the programmer to be consistent according to his problem or up to the meta programmer to define the new language adapted to the problem.

### 3.1.3 A short overview of the basic language

A detailed study on the Eiffel language has enable us to define a subset likely to behave as a minimum language of our environment. We have kept the following Eiffel concepts, after having adapted them : control structures, exportation, assertions, exceptions, genericity and indexation.

The inheritance relationship is not a concept present in *Ofl* . This choice lead us to make the assumption that there is a library that may be used by *Ofl zéro* programmers. Right now,

[10] The expression redefinition has to be understood here with its the largest meaning.

[11] *Ofl zero* is provided with a set of generic operators for handling persistent objects and collections of objects (persistent or non persistent).

the library contains views *derived*[12] from the following *Eiffel 3* classes : `ANY`, `ARRAY`, `BOOLEAN`, `CHARACTER`, `DOUBLE`, `EXCEPTION`, `FILE`, `INTEGER`, `REAL`.

In order to get a better picture, one may find hereafter in the annexes a short example of a program written in *Ofl*$_{zéro}$ . The reader will notice that, for the basic points, we have taken the Eiffel syntax. It is however obvious that the syntaxical aspect of the language is not its main aspect and that other formalisms are going to be implemented in the *Ofl* environment (in particular, an abstract representation of programs).

### 3.2 An introduction to the model

The figure 2 page 11 is describing the model which is underlied by our approach. It is worthwhile to immediately precise that our model is making no difference between the traditionnal steps of programming and execution : we actually want to remain completely independent from the statical, dynamic aspect. The picture is showing the two levels we have chosen :

**Meta programming level :** It allows to generate a language. We are then be able to emulate a real language or to generate a new language in order to test its capabilities. The languages we are generating are modelised thanks to a meta-view containing the different types of links existing in this language. A link is a relationship existing between several views. For example, an inheritance link is defining the relationship between the inherited views and the heiresses views.

**Programming level :** This is the level where application programmer should play a role by declaring views. These views are occurences of meta views and they are especially referencing the occurences of the links that meta-views are using. For example, if the `A` view has inherited from the `B` view, the instance of the link "`A inherit B`" is generated and it becomes component of the `A` and `B` views.

Finally, it is important to understand that entities like the *link*, *meta view*, and *view* are reified, which will make them directly accessible at run time.

## 4 Using the model : an example

In order to make the things easier to understand, we are showing (fig. 3 page 12) with an example how we are going to be able, step by step, to use this system. To do that, let us imagine that we need for a program a language provided with a "`Eiffel3-inheritance-like`" relationship. We are describing here step by step an example of such program description.

### 4.1 Meta programming step

1. First of all, we have to generate "`Eiffel3-inheritance-like`" link. It is describing the way the views are going to react when they will be linked by an inheritance realtionship

2. We then have to generate an *Eiffel 3* meta view (*Eiffel* `star`) which is built with this link and which represents the language we need. Here is the end of the meta programming step. Nevertheless, we have to mention that if we had needed several types of links, we would then have defined in *Eiffel* `star` the method of composition of these links.

### 4.2 Programming step

1. During the programming step, we are generating the views we need by generating *Eiffel* `star` occurences. When we specify links between views, occurences of the concerned links are

[12] In the sense of services provided to the programmer.

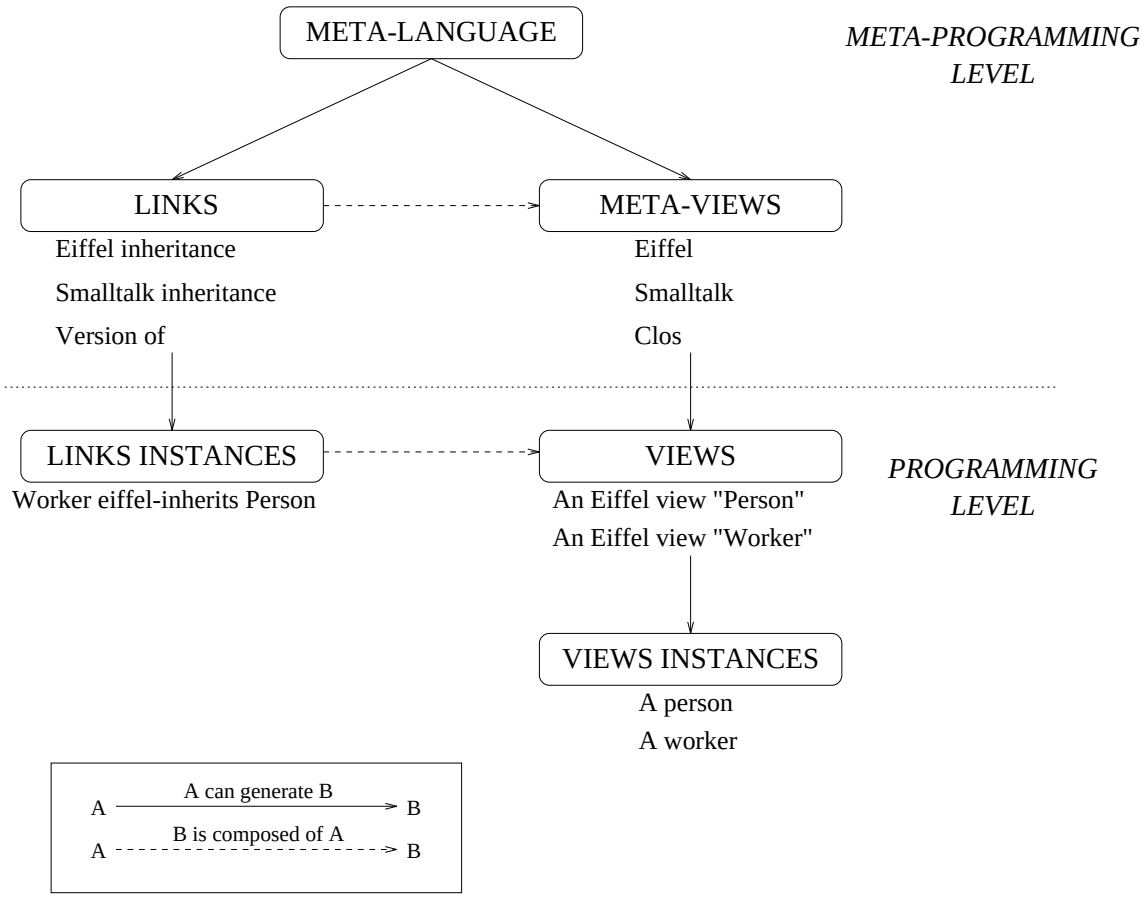


Figure 2: Components of OFL

automatically generated. We can see that on our example. We are creating the `A`, `B` and `C` views and we want the inheritance graph to be the one of picture 3 : we are thus going to specify that `C` is inheriting from `A` and `B`. This is generating the occurences of the "`Eiffel3-inheritance-like`" link : "`C-inherit-A`" and "`C-inherit-B`". "`C-inherit-A`" becomes then a component of `A` and `C` while "`C-inherit-B`" becomes a component of `B` and `C`.

2. Finally, we can access, at run time, all occurences of `A`, `B` and `C` views which are making our program.

# 5 Conclusion

In this paper, we have been introducing the main points of a model which aims to concile the concerns of "Software Engineering" together with the most advanced features dealing with object language evolution.

In the first part, we tried to remind some of the limits of the "industrial" objects environments, especially regarding the management of persistence, the introduction of view mechanisms and the status of the inheritance concept. The results of these studies show that it should be interesting to introduce a higher reflexivity and extensibility to these environments.

To do that, the $O^{fl}$ model is proposing, briefly :

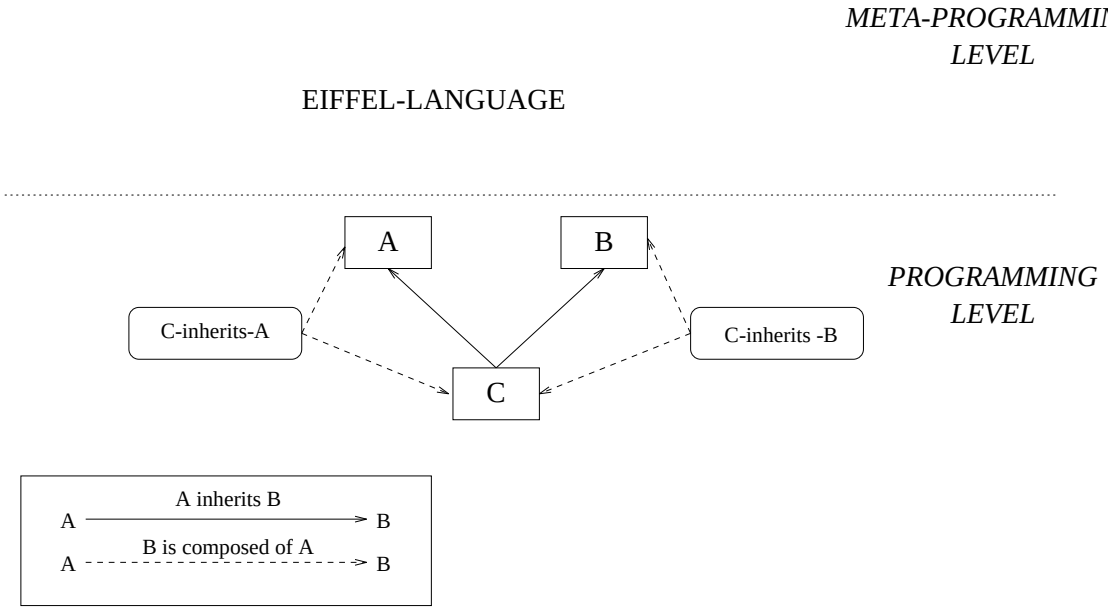


Figure 3: Using OFL : an example

- *an unique concept of view* as a mean of typing all the occurences handled by the applications
- *a programming language* derived from Eiffel, but not provided with inheritance built-in mechanisms. This language is provided with a minimal semantics (expansed objects or references, exceptions, etc) and extensible semantics, which means that the basic operators (assignment, renaming, etc) could be redefined.
- *two programming levels* :
  - a meta programming level allowing to extend the basic language with the definition of the meta-views which are generating views and carrying with it the semantics (for example, a style of inheritance and/or a "is a version of" relationship).
  - a programming level allowing to declare and use (within applications) views which are generated by one or several meta-views.
- *an extension mechanism* of the basic language, located at the meta programming level which allows to define a meta view as a composition of links : a link is the entity which enables to give a new semantics to the operators of the basic language. The composition enables to give the semantics of each redefinable operator for a given meta-view (in other words, actually, to forecast the behaviour of the views which have been generated from the meta views composed of several links).

In fact, our concerns in this project are multiple and should lead us to concile several research themes and reuse their results :

- in the persistence field, *O*$^{fl}$*zéro* should allow, in the most transparent way, both storage/incremental loading of objects and the handling of persistent collections of views [ADF+94] [CFLR93b].
- the engine of the environment should allow to generate meta-views, links and their occurences. In some way, it is dealing with the idea of being able to adapt the programming language to the needs of the programmer. This idea is quite present in the Clos/MOP environment [Kee89], which could be used as the implementation kernel of our model. More

generally, we are quite close to the works dealing with the evolution of object oriented languages (LMO'95 [Nap95])[13].

- In the $O^{fl}$ environment, the coexistence of an internal engine based on MOP and a classical application programming language ( $O^{fl}$*zéro* ) will lead to study and solve problems of languages translation/interpretation.

# Annexe

```
view PERSON
-------------------------------------------------------------------------------------
-- A view PERSON written with the basic language OFLzero:
-- * Use of  "kernel" views STRING, INTEGER and CHARACTER to set types to the attibutes,
-- * Use of another  "application" view TOWN,
-- * Explicit use of the "kernel" view ANY (for I/O, cf. DisplayNames),
-- * Free access to atributes (cf. SetSpouse et AddChild).
-------------------------------------------------------------------------------------
feature
   p_name : STRING; f_name : STRING; age : INTEGER; sex : CHARACTER;town : TOWN
   spouse : PERSON; children : linked_list [PERSON]; kernel : ANY;

   Create (n : STRING; p : STRING; a : INTEGER; s : CHARACTER; v : TOWN) is
      do
         p_name := n.duplicate; f_name := p.duplicate; age := a; sex := s; town := v;
      end; -- create

   SetSpouse (p : PERSON) is
      do
         spouse := p; p.spouse := current;
      end; -- SetSpouse

   AddChild (p : PERSON) is
      local
         l_children : LINKED_LIST [PERSON];
      do
         if children.void then children.create end ; children.start; children.put_right (p);
         if not spouse.void then   spouse.children := children; end;
      end; -- AddChild

   DisplayNames is
      do
         kernel.io.putstring (f_name); kernel.io.putstring (" "); kernel.io.putstring (p_name);
      end; -- DisplayNames

   DisplayTown is
      do
          io.putstring (town.town_name); io.putstring (" "); o.putint (town.nb_inhabitants);
      end;    -- DisplayTown
end -- feature
```

# References


[AB91] Abiteboul S. and Bonner A. "Objects and Views". In *Proceedings of the ACM SIGMOD on International Conference on Management of Data*, pages 238–247. ACM Press, 1991.

[ADF+94] Atwood T., Duhl J., Ferran G., Loomis M. and Wade D. *The Object Database Standard : ODMG-93.* Catell R. (ed.) . Morgan Kaufmann Publishers, San Mateo, California, 1-55860-302-6, 1994.

[Ber92] Bertino E. "A View Mechanism for Object-Oriented Databases". In *Proceedings of the International Conference on Extending Databases Technology*, vol. 580, pages 136–151. Lecture Notes in Computer Science, Springer-Verlag, 1992.


[13] LMO (Langages et Modèles à Objets) is the annual congress of the research groups "Evolution des langages à objets" of GDR programmation and "Classification et objets" of PRC-IA (France).

[BK87] Banerjee J. and Kim W. "Semantics and implementation of schema evolution in object-oriented databases". In *International conference on management of data*, pages311 à 322. ACM SIGMOD, San Francisco, Californie, 1987.

[BK93] Barclay P. J. and Kennedy J. B. "Viewing Objects". In *Proceedings of Advances in Databases*, pages93 à 110. $11^{th}$ British National Conference In Databases, Keele, Royaume Uni, 1993.

[Bou94] Bourdin C. *Points de vue et représentation multiple et évolutive dans les langages à objets.* Rapport de stage de D.E.A. d'Informatique, Université de Montpellier II, 1994.

[Car89] Carré B. *Méthodologie orientée objet pour la représentation des connaissances : concepts de points de vue, de représentation multiple et évolutive d'objet.* Thèse de doctorat en Informatique, Université des sciences et techniques de Lille Flandres Artois, Laboratoire LIFL, Lille, 1989.

[CFLR93a] Chignoli R., Farré J., Lahire P. and Rousseau R. "FLOO : Principes et Illustration des mécanismes automatiques de persistance pour Eiffel". In *6th International Conference on Software Engineering and its Applications*, pages181 à 190. Kluwer Academic Publishers, Paris-La Défense, 1993.

[CFLR93b] Chignoli R., Farré J., Lahire P. and Rousseau R. "FLOO : Une implémentation de la persistance pour Eiffel". In *RPO'93 - Représentation par Objets*, pages131 à 142. EC2 (Paris), La Grande Motte, 1993.

[CK94] Chan D. K. C. and Kerr D. K. "Improving One's Views of Object-Oriented Databases". In *Actes du Colloque "Orientation objet en bases de données et génie logiciel"*, pages123 à 137. Soixante-deuxième Congrès de l'Association Canadienne Française pour l'Avancement des Sciences, Montréal, Canada, 1994.

[HS90] H.Scholl M. and Schek H.-J. "A Relational Object Model". In *Proceedings of the International Conference on Database Theory*, vol. 470, pages 89–105. Lecture Notes in Computer Science, Springer-Verlag, 1990.

[Kee89] Keene S. E. *Object-Oriented Programming in COMMON LISP : A Programmer's Guide to CLOS.* Addison-Wesley, Reading, Massachussetts ; Menlo Park, California, 1989.

[Lah92] Lahire P. *Conception et réalisation d'un modèle de persistance pour le langage Eiffel.* Thèse de doctorat en Informatique, Université de Nice-Sophia-Antipolis, Laboratoire I.3S., Sophia-Antipolis, 1992.

[MDC92] Malenfant J., Dony C. and Cointe P. "Behavioral Reflection in a Prototype-Based Language". In *Proceedings of the International Workshop on New Models for Software Architecture'92, Reflection and Meta-Level Architecture, Nov. 1992, Rise (Japan).* ACM Sigplan, JSSST, IPSJ, 1992.

[MEC95] Marcaillou-Ebersold S. and Coulette B. "Extension du modèle objet par introduction de la visibilité". ENSEEIHT, Toulouse, 1995.

[Mey90] Meyer B. *Conception et programmation par objets pour du logiciel de qualité.* InterÉditions, Paris, 1990.

[Mey94] Meyer B. *Eiffel, le langage.* InterÉditions, Paris, 1994.

[MKC95] Marcaillou S., Kriouile A. and Coulette B. "Vbool, une extension d'Eiffel intégrant le concept de point de vue". ARAMIIHS, UMR 115 du CNRS, Toulouse, 1995.

[MNC+91] Masini G., Napoli A., Colnet D., Léonard D. and Tombre K. *Les langages à objets : langages de classes, langages de frames, langages d'acteurs.* InterÉditions, 1991.

[Nap95] Napoli . (ed.) . *Actes de LMO'95 - Langages et Modeles a objets, Nancy, 12-13 Oct. 1995.* INRIA, Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Cheynay (France), 1995.

[SLT91] Scholl M. H., Laasch C. and Tresch M. "Updatable Views in Object-Oriented Databases". In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, vol. 566, pages 189–207. Lecture Notes in Computer Science, Springer-Verlag, 1991.

[SS89] Shilling J. J. and Sweeney P. F. "Three Steps to Views : Extending the Object-Oriented Paradigm". In *Proceeding of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 353–361. ACM Press, 1989.