Deductive Object Programming
trying to make object-oriented programming less complex

François Colonna
Laboratoire de Chimie Théorique
Université Paris 6
email:colonna@lct.jussieu.fr
home page http://galileo.lct.jussieu.fr
this document url:
http://galileo.lct.jussieu.fr/~frames/ARTICLES/DOP/Provider.tex

February 1, 2008

Preamble
This document is a working document, it may never be published in a scientific journal. It is aimed at starting a discussion on the interest of the kind of programming method explained below.
Any comments or corrections are welcomed can be written in color in this document and sent back to me.

Abstract
We propose some slight additions to O-O languages to implement the necessary features for using Deductive Object Programming (DOP). This way of programming based upon the manipulation of the Production Tree of the Objects of Interest, result in making Persistent these Objects and in sensibly lowering the code complexity.

1 Motivation
It is a real frustration, when writing some code, not to be able to use the values of a functionality of an object by simply referencing it. If you do so, you will obtain an error as soon as you will use the functionality of a just created object.
In this paper we show that, in fact, this can be achieved quite easily. The only information necessary to obtain values as soon as an object is referenced, is its production tree. Having remarked that any object has necessarily been produced by a tree (see below, paragraph 2), the trick is therefore to make this tree accessible to the programmer.
On the other hand, in the usual way of programming, the produced-by relation is never made explicit but hidden in calls to object creation methods,
scattered in the code and difficult to follow. It is therefore a common experience, when trying to modify somebody else code, to be stuck during hours or days at the same point because we have no clear idea of what we have to do before being able to use a given object. We shall show that, by making explicit the produced-by relation between objects, one can avoid to get trapped in this kind of problem: any reference to an object can be removed or added to a class without any further remodeling of the code.

We shall show how deductive programming is implied by this way of managing objects which may become persistent (see [Wik]) and can easily be distributed.

We shall give as an example how we have implemented this programming mechanism in Eiffel shall conclude by the proposition to add some new features to the language to hide the management tools inside the compiler and make the new mechanism transparent to the programmer.

2 The Production Tree

2.1 definition

By production tree we mean the tree resulting from the relation produced-by between two objects. Contrarily to the client-of relation, the graph generated by the produced-by relation cannot have cycles, otherwise an object could never be produced.

This tree defines in a unique way any ground-state (see paragraph 2.3.3) of a given object.

As the edges of the tree are totally defined by the code, the only degrees of freedom left to define an object state are the values of the leaves (the parameters values). The production tree and a set of coherent parameters are nothing else than the persistence closure of the object, this point will be developed below (see paragraph 3). It is therefore sufficient to know the production tree of an object and to give its calculation conditions (parameters) to define an object state before any calculation has been done.

Once this simple proposition has been stated, there is no difficulty to make it effective, that is:

1. To make possible the automatic production of any object in any state.

2. To define a key from the production tree to access to the corresponding object state (stored in a data-base, for example) before any calculation has been done. This possibility allows to reuse the object values not only its functionalities, thus extending the capabilities of programming with objects.

3. To make unimportant (but easily providable) which path the author has decided to follow to build some target object \( t \) from pre-existing objects \( a, b, \ldots, x \), making the code modification by an alien programmer easier.
2.2 new kinds of attributes

In the whole paper we shall speak of attribute to designate the couple (memory-function, memory-attribute).

Let an _attribute_ be a memory-function returning some value of type A stored at an _attribute_memory_ address, we can write:

```plaintext
feature {ANY}
  an_attribute : A is
  do
    Result := an_attribute_memory
  end -- an_attribute

feature {NONE}

an_attribute_memory : A
```

The couple (an _attribute_, an _attribute_memory_) represents what we shall abusively call the attribute an _attribute_ of type A.

Classical OO design concentrates (see OOSC2 [Mey97]) not on what attributes a class has but on what methods a class can offer to manipulate them. Of course we agree with this view, nevertheless it leads to hide the important question how can I build an instance of this class to use its methods?

As the attributes are supposed not to appear in the interface, their role as a builder or as an internal sub-state provider is never mentionned. Focusing an method for reusability purpose is fine, but methods do not determine the state of an object, while attributes do. And, using an object in a given state is what a programmer is at first concerned with, then he is concerned with applying methods on it.

We do not violate the OO principle to hide attributes. Looking at the example upper you can see that only an _attribute memory-function_ will appear in the interface and not an _attribute_memory_ which is hidden, as it should be.

We will now define two kind of attributes: internal attributes and builder attributes or external attributes.

2.2.1 internal attribute

An internal attribute is an attribute calculable from the values of other attributes of the same object. For example, the perimeter of a TRIANGLE (see Annexe B in paragraph 8.2) is calculable if we know the positions of the 3 vertices.

2.2.2 builder attribute or external attribute

An external or builder attribute is an attribute calculated outside the current object. For example, the 3 vertices of a TRIANGLE, (see paragraph 8.2).
These attributes are a source of complexity as they usually need a lot of information to be built. Our programming mechanism consists in providing these builder attributes in the correct state (attribute of the attribute) with the only knowledge of the name of the attribute and the name of the state we want it to be in. For example, in a TRIANGLE we can directly ask for the vertices the state “position”, using this syntax:

```plaintext
needs (vertices('position'))
vector_1 := vertices.item (1).position
```

Which ensures that after the call to the procedure `needs`, the code can use the value of the position of a vertex as shown. The `vector_1` variable will be assigned to the correct value.

### 2.2.3 parameter attribute: calculation conditions

We call parameter a kind of builder attribute not built in the production tree, it has to be provided by the User of the code (i.e. read from the input). A parameter is a leaf of the production tree. Any builder attribute of basic type (BOOLEAN, INTEGER, REAL, STRING) is necessarily a parameter: it cannot be built.

We call calculation conditions the whole set of parameters for a given production tree. They define the persistence closure (see [Mey97] page ) of the production tree and determine completely all the states of any object inside the tree. They are pure basic types or collections of basic types.

In the case of a TRIANGLE the parameters are the coordinates of the 3 vertices, i.e. 9 real numbers in a 3-dimensional space.

### 2.3 Object state, sub-state and ground-state

#### 2.3.1 object state

We say that an object is in a sub-state s if the exported attribute s has been computed.

#### 2.3.2 object sub-state

We shall consider two types of sub-states corresponding to the two types of attributes already mentioned in paragraph 2.2, the internals and the externals (or builder sub-states).

The State of an object is characterized by the list of its sub-states.

The production-tree handles only builder sub-state.

#### 2.3.3 object ground-state

We shall say that an object is its ground-state if it has all its builders built. If an object is in its ground-state any of its internals state can be built.
2.3.4 well-built object

We propose to say that an object is well built if all its internals sub-states need the same builders. In other words, all internal trees share the same leaves: the builders.

For example, a TRIANGLE with 3 vertices as the only builder is well built. If a builder color is added it is not, as a color is not needed to build the perimeter for instance.

2.4 cyclic connections

If an object is produced by a tree, its relations with other objects form a graph which can even be cyclic. We show here, that this case can also be managed. Consider the following example:

Object a has an attribute b_in_a of type B
Object b has an attribute a_in_b of type A

If - in the code - object a is created first, then object b is a builder of A, because it is referenced in A but not created in A.

Therefore, a is created, then b is created, a_in_b is computed, then b_in_a.

If the situation is the opposite you will have the inverse order of computations.

So, cyclic connections can also be handled, with the production tree mechanism.

2.5 The solution we propose: The Object Manager

How to implement this mechanism in an Object-Oriented language? The solution that we have implemented consists in associating an Object Manager to each object of interest (see paragraph 3).

2.5.1 the Object Manager specifications

An Object Manager is an object, biunivocally associated to a “real” object, and able to answer the programmer’s request provide me with this object in this sub-state.

In some sense it is more than an object and less than an “agent”. An Object Manager obey to the following requirements: when asked for providing an external object object an_object in state s it

I. tries to retrieve object an_object in state s from a data-base

1. if an_object in state s is stored returns an_object
2. if an_object in state s is not stored
   i. creates the instance an_object
   ii. launches the memory-function of an_object corresponding to the attribute s.
iii. stores an object in state s in data-base.

II. returns an object in state s

2.5.2 How the Object Manager is used now?

We give below an example in Eiffel of the part of class TRIANGLE using the Object Manager triangle Om for a TRIANGLE (line 1 of code below):

1: triangle om : TRIANGLE_MANAGER
2: vertices memory : ARRAY [POINT]
3: vertices (sub-state : STRING) : ARRAY [POINT] is
4: do
6: if vertices memory = Void then
7: vertices_from_key (sub-state)
8: end
9: Result := vertices_memory
10: ensure
11: Result = vertices memory
12: end -- vertices

13: vertices_from_key (sub-state : STRING) is
14: local
15: vertices om : ARRAY POINT_MANAGER
16: do
17: vertices om := triangle om.child om extract (‘‘vertices:ARRAY[POINT]’’)
18: vertices_memory := vertices om.provided (sub-state)
19: ensure
   vertices_memory_is_built: vertices_memory /= Void
20: end -- vertices_from_key

21: centroid : POINT
22: do
23: if centroid_memory = Void then
24: centroid_memory_build
25: end
26: Result := centroid_memory
27: ensure
28: Result = centroid_memory
29: end -- centroid
30: centroid_memory : POINT

31: centroid_build is
32:  local
33:   vertices_local : like ARRAY[POINT]
34:   do
35:     vertices_local := vertices ("position")
36:     create centroid_memory.make
37:     centroid := (vertices.item (1) + vertices.item (2) + vertices.item (3))/3
38:   ensure
39:     centroid_is_built: centroid /= Void
40: end -- centroid_build

We show upper how the centroid attributes uses the vertices attributes as if it were already calculated. The procedure centroid_build (line 31) refers to vertices in sub-state “position”. This assignment (line 35) launches the execution of the memory-function vertices (line 3). The first time the code is executed, vertices_memory is Void, the procedure vertices_from_key is therefore called (line 7 and 13). It asks the Current’s Object Manager (line 17) triangle_om to extract from itself the Object Manager of its son class ARRAY[POINT] (line 15). This Object Manager vertices_om provides vertices_memory in the correct sub-state (line18), i.e. provides of the vertices positions.

2.6 Proposed Extensions to the Eiffel language

Most of the code in paragraph 8.1 upper can become transparent to the programmer if taken into account by the compiler. For this, four new keywords have to be added to the Eiffel language. Two new requirements and two new declaration keywords.

2.6.1 the needs requirement

To be provided with a needed builder attribute in a given sub-state:
needs (object_1 (sub-state) ,..., object_n (sub-state))

To be provided with a needed internal attribute:
needs (object_1 ,..., object_n)

2.6.2 the uses requirement

defines the list of building procedure depending on the context
uses (context_1 : object_1_build, ..., context_n : object_n_build)

We propose 3 kinds of contexts : build, read and set as shown in table
Table 1: 3 kinds of contexts, and the procedure suffixes associated

| context | procedure suffix |
|---------|------------------|
| build   | a_build          |
| read    | a_read           |
| set     | a_set            |

2.6.3  the internal keyword

The syntax looks like:

attribute : SOME-TYPE internal (building-procedure)

To the type SOME-TYPE the keyword internal is added and the name of the building-procedure is given between parenthesis.

2.6.4  the builder keyword

The syntax looks like:

attribute : SOME-TYPE builder (sub-state-name)

To the type SOME-TYPE the keyword builder is added and the name of the sub-state to be provided is given between parenthesis.

2.6.5  How the Object Manager could be implemented?

3  Persistent Objects : a new object sub-category

Because the ground-state (paragraph of an object is equivalent to its persistence closure (see [Mey97], page 252) and because we have a mechanism allowing to define the state of an object before it is created it easy to make it persistent and consequently to check if it has not been stored somewhere (a data-base) in that state.

This is of course not as easy with usual programation not making the production tree explicit. Most of the time, if the object has been computed during a previous task, you have no mean to know it and the object has to be recomputed.

Using the Object Manager mechanism, it is possible to build a key to characterize uniquely any ground-state of an object of interest and to store them in a data-base, we call persistent objects the objects managed in this way. Sub-keys can also be defined to handle sub-sates.

Here, the builder (not creation) procedures are closed: no further information is needed to invoke them (this information is already known by the production tree).

In Chapter 8 of his book OOSC2 [Mey97] at the top of page 236, Bertrand Meyer says (although in an other context):
what you need is, rather than a creation instruction, an assignment operation that attaches a reference to an already existing object.

It is exactly what Persistent Objects are aimed to: create an object in one of its possible states as soon as it is assigned. While what Eiffel propose is to create an object in an empty state or in a unique built state, not made clearly explicit, defined by its creation routine and its class invariant, this is not enough.

### 4 What is lacking in the Class Interface

An object Class is designed to provide a set of functionalities to manipulate their instances: the interface.

Looking at this interface, the question *what can I do with it?* can be answered, and how to reuse a piece of code already written by somebody else.

However, before using a functionality of a class you have to create an instance, and to create it in such a state as to be sure that this functionality will be usable (will have values solving your problem not default values). That is precisely the information lacking in the class interface: *how to reach the state in which you wish to use one instance, not bothering with all information necessary to reach it?*.

It is most of the time a complex task to build an instance needed in order to use it. Why? Because in usual OO programming the object builder procedures are opened that is to say when you invoke them, you need to feed them from outside with some necessary information in an argument list, they do not provide a closed object state.

In Eiffel you write

```
create this_attribute.make (some parameters)
```

and not

```
create this_attribute.make ("some_state")
```

For example, if you are writing a new class (for example, TRIANGLE_PYRAMID of paragraph 5.3.1) needing an attribute base of type TRIANGLE to use its surface, what you need is not only to know that surface is of type REAL and requires that the sides (of type ARRAY[SEGMENT]) should be defined. What you need is a mechanism replacing the require statement on the necessary existence of the sides by the effective provision of the surface of base, i.e. a REAL number which represents its computation, whenever you make a reference to it and whatever the method to built it could be.

By looking at the class interface of TRIANGLE (see Annexe 8.1 - the class of a triangle - you have no access to this essential information, but you need it, here is one of the reason why codes are still complex, even if written according to the best O-O style. Because authors are not forced to make explicit the route they have decided to follow, leading from one object to its son in the production tree.
4.0.6 Notes

If you code surface_build (s1, s2, s3: SEGMENT) you suppose that s1, s2, s3 are already calculated, **outside surface_build** which is not.

If you code

```
surface_build is
  needs (s1, s2, s3)
do
end -- surface_build
```

you tell the code: if s1, s2, s3 are not yet calculated, calculate them.

The calculation of s1, s2, s3 is done **inside surface_build**

5 Deductive Object Programming

It was a way of procedural programming which used to be popular in the seventies (see references [FM78],[Les78]). It is a top-bottom approach:

Start from the final result you want to reach.
Write the procedure to build it. Then write the procedure building the immediate needed objects.
Iterate until the data.

This way of programming has been re-actualized for Object Oriented Programming as follows:

5.1 definition

- start from the final result (the Target of the Task, an Object in some state: the centroid of a TRIANGLE)
- design objects immediately needed to build this Target (the sons of its production tree) as function-attributes (lines 2 and 3 of paragraph 2.5.2)
- in the building procedure of the Target, write a reference reference to the son-objects this make them available in the desired state. (lines 35 of paragraph 2.5.2)
- iterate over the objects needed and the objects needed by the objects needed until parameters (not buildable but readable objects) are reached, read them.

In other word, instead of building an intermediate object needed to reach your goal, write your code as if these needed objects were already built, and defer their building or retrieving from a storage to a specific object, here the *Object Manager*. 
Programming with Persistent Objects or Deductive Object Programming are a same thing: the programmer never cares of providing values to a function-attribute-persistent-object, he declares it as a builder function-attribute and uses it.

It must be remarked that the sequence of calls to the building routines are included in each other

\[ a \text{ build calls } b \text{ build which calls } c \text{ build } \ldots \text{ until the leaves} \]

In the usual (bottom-up) way of programming one starts building what is needed first (the data-leaves) then climbs up the tree, the sequence of calls in not inclusive:

- build the leaves when done build x, ... when done build c when done build b when done build a
- and the programmer has to known the whole tree.

5.2 connection with the Production Tree

The connection between DOP and the Production Tree is clear: both needs to make explicit the sons of the relation built-by for any class to be programmed. The whole Production Tree will be built automatically from this basic information.

As we have already mentioned upper, the production tree is the main information necessary to let the compiler be able to produce an object in a given sub-state by simple reference (point two of deductive programming pre-requisite).

5.3 improvement of code Quality

5.3.1 lower complexity

The complexity is lowered because the only information a programmer has to known to program a new class is the interface of the son-classes. There is no need for him to know the whole production tree as it is implicit in usual programming.

The complexity is also lowered because the state of the objects are now made explicit and one can access to the values of any object without taking care the complex way to get them.

For example, suppose we want to program a class TRIANGULAR_PYRAMID to use the surface of its triangular base, we shall write:

class TRIANGULAR_PYRAMID

apex : POINT builder ("position")

base : TRIANGLE builder ("surface")

base_surface : REAL is

\[ \text{needs base("surface")} \]

\[ \text{do} \]
Result := base.surface
ensure
   Result = base.surface
end -- base_surface

This is sufficient to obtain the surface of attribute base computed according to the context of the Current object TRIANGULAR_PYRAMID. No other knowledge is necessary.

The compiler will be able to tell the programmer that the three vertices coordinates of this particular base-triangle have to be provided as data of the Task, as well as the apex coordinates, to defined completely the new class instances.

The procedures `triangle_from_key` and `apex_from_key` (similar the that of line 13 of paragraph 2.5.2) are taken into account by the compiler.

5.3.2 objects reusability

Persistent Objects have the property of **object reusability** i.e. : as objects the code of their Class is reusable, as persistent objects their values are reusable.

5.3.3 more evolutionary capabilities

Persistent objects are more independent than usual objects.

If you need to modify a piece of code, that is to say, take out some attribute and put it elsewhere, the code will re-adapt automatically, because the *production tree is not hard coded* as in an usual program but **dynamically built by the compiler** from the father-son couples.

This property is important for the maintainability of always growing systems as those used in scientific simulations.

5.3.4 better class design

A good design implies a few builders and that the **internal attributes** all share the same *calculation conditions*. That is to say all **internal trees** share the same leaves. If it is possible to cluster the leaves, because some **internal attributes** use only a sub-set of the leaves, it can be a sign of a bad design. The class has to be splitted in one or several heirs, each of them being well-built (see paragraph 2.3.4).

5.4 distribution of the building of Objects Persistent

To manage the distribution of objects building first of all one needs to set up their *production tree*. This can be a huge task to be done with a usual code.

Using **DOP**, it is easy to distribute the building of the nodes of the *production tree* on some defined processors of a cluster or the nodes of a grid : this new functionality can be implemented in the **Object Manager**.
5.5 extending the Concept of a Calculation

Instead of computing values we can use the same mechanism to compute whatever property of an object. For example we can compute the cpu time, the memory, the disk space to be used, the choice of a given processor or grid node.

The management of the production tree at compilation time, allows any kind of work flow simulation on the code.

5.6 iterative processes

Iterative processes (optimization, Monte Carlo simulation, molecular dynamics) are very common in scientific calculations.

An iterative process is a process which computes iteratively the same Target object and modifies at each iterations the calculations conditions of this Target.

As the DOP mechanism aims at computing an object in a well defined state, modifying the calculation conditions will modify de facto all the objects whose state is no more consistent with the new calculations conditions. And only these objects will be recalculated automatically.

We may point out that using DOP allows a code to optimize any object attribute against any parameters of the code (this facility is extensively used in the QMCMOL code [QMC]).

These processes needs to use:

- a boolean function like \texttt{is\_not\_ready} instead of the condition \texttt{attribute = Void} (see lines 6 and 23 of paragraph 2.5.2. This function is true whenever some node of the production tree of \texttt{attribute} has been modified.
- an iteration counter

6 Conclusion

We have shown that combining deductive programming with making explicit the production tree of the objects of interest (persistent objects) increases the re-usability and lowers the complexity of codes.

By adding a few functionalities to any OO language may totally hide the agent-like mechanism necessary to manage the persistent objects in a today language like Eiffel. Doing this, the compiler can take an active part in the automatic building of any persistent object in any of its states. This helps considerably the possibility to modify somebody else’s code in full security without any deep knowledge of the code and allows a simulation of the calculation flow.

Moreover, the objects managed in this way are persistent by construction and can also have their calculation distributed.

Anyway, this programmation improvement is not yet sufficient to write codes fully understandable by an alien programmer expert of the domain, which is the ultimate goal of programmation. A mechanism to make the author’s intentions immediately understandable, is still lacking.
7 Acknowledgements

We are grateful to Nicole Levy, Parinaz Davari and Francisca Losavio (PRISM, Versailles) for their contributions and to Dominique Colnet and Frederic Merizen (Loria, Nancy) for their decisive help in the SmartEiffel implementation. We thank Gilles Blain and Zahia Guessoum (Lip6, Paris) for reading the manuscript.

8 Annexes

Comparison between the usual and new implementation of class TRIANGLE, what has changed.

8.1 A : an usual implementation of class TRIANGLE

Below, we give an example of what the interface of class TRIANGLE like now :

class TRIANGLE

feature {ANY}

sides : ARRAY[SEGMENT]
  require
    vertices_are_defined: vertices /= Void
  ensure
    sides_are_defined: Result /= Void
end -- sides

centroid : POINT
  require
    vertices_are_defined: vertices /= Void
  ensure
    centroid_is_built: Result /= Void
end -- centroid

perimeter : REAL
  require
    sides_are_defined: sides /= Void
  ensure
    perimeter_is_built: Result > 0.0
end -- perimeter

surface : REAL
  require
    sides_are_defined: sides /= Void
  ensure
surface_is_built: Result > 0.0
end -- surface

vertices : ARRAY[POINT]

make (points : ARRAY[POINT])
require
  points_defined: points /= Void
ensure
  vertices = points
end -- make

invariant
  vertices_are_built: vertices /= Void
end -- class TRIANGLE

As far as the objects produced by the class (surface, perimeter, sides, centroid) things are pretty fine and their relations are clearly described by the assertions:
  to compute the surface you need the sides and the perimeter, to define the sides you need the vertices, to obtain the perimeter you need the sides.

So, supposing you obtain the vertices, you have no difficulty to understand how to compute any other property of a TRIANGLE, just by looking at the interface.

The problem starts with the vertices, where do they come from ? Knowing that they are an array of POINT, which has to be provided not Void, does not help to answer the question how to obtain the vertices ?.

The new implementation below show the answer: the vertices are a builder, the Calculation Manager will take care of providing them in the sub-state “position” i.e. with their positions valued as needed for the current calculation.

This TRIANGLE case may seem trivial, when the same mechanism is applied to the calculation of a density matrix from a precise kind of wave-function in quantum physics, the programming effort stays as low as it is here, which is less trivial to do with usual programming.

8.2 B : a new implementation of class TRIANGLE

Below, we give an example of what the interface of class TRIANGLE could look like using the Eiffel extensions :

class TRIANGLE

feature {ANY}
sides : ARRAY[SEGMENT] internal (sides_build)
  needs
  vertices ("position")
  ensure
  sides_are_built: Result /= Void
  end -- sides

centroid : POINT internal (centroid_build)
  needs
  vertices ("position")
  ensure
  centroid_is_built: Result /= Void
  end -- centroid

perimeter : REAL internal (perimeter_build)
  needs
    sides
  ensure
    perimeter_is_built: Result > 0.0
  end -- perimeter

surface : REAL internal (surface_build)
  needs
    perimeter,
    sides
  ensure
    surface_is_built: Result > 0.0
  end -- surface

vertices : ARRAY[POINT] builder ("position")
  ensure
    vertices_are_defined: Result /= Void
  end -- vertices

invariant:
  Current /= Void

end -- class TRIANGLE

• *internal* keyword means that the attribute is internally built by the procedure in parenthesis.

• *builder* keyword means the that the attribute has to be externally built in the sub-state where the attribute *position* of each POINT is defined.
We want to emphasize on the fact that the sub-state “position” of class POINT, is also apparent in this interface of TRIANGLE. Which tells the programmer in which sub-state a POINT will be used in a TRIANGLE. Now, some context appears in the interface.

8.3 C : the external tree of a TRIANGLE
8.4 C: the internal trees of a TRIANGLE

8.4.1 C-a: the internal tree of surface

triangle
  └── surface
    ├── sides 1
    │   └── vertex 1
    ├── sides 2
    │   └── vertex 2
    └── sides 3
          └── vertex 3

8.4.2 C-b: the internal tree of centroid

triangle
  └── centroid
    └── vertex 1
          └── vertex 2
                     └── vertex 3
References

[FM78] J.P. Finance and J.F. Mari. Méthode de programmation déductive et structures de données. Technical Report 78-E-042 CRIN - 139, CRIN, 1978.

[Les78] P. Lescanne. Sémantique d’un langage adapté à la construction déductive des programmes. Technical Report 78-R-0152 CRIN - 139, CRIN, 1978.

[Mey97] B. Meyer. Object Oriented Software Construction. Prentice Hall, 1997.

[QMC] QMCMOL. http://galileo.let.jussieu.fr/~qmcmol/mediawiki/index.php/main_page.

[Wik] WikiPedia. http://en.wikipedia.org/wiki/persistence.
Contents

1. Motivation ........................................... 1

2. The Production Tree .................................. 2
   2.1 definition ......................................... 2
   2.2 new kinds of attributes ............................. 3
      2.2.1 internal attribute ............................... 3
      2.2.2 builder attribute or external attribute .......... 3
      2.2.3 parameter attribute : calculation conditions ...... 4
   2.3 Object state, sub-state and ground-state ............ 4
      2.3.1 object state .................................... 4
      2.3.2 object sub-state ................................. 4
      2.3.3 object ground-state .............................. 4
      2.3.4 well-built object ................................ 5
   2.4 cyclic connections .................................. 5
   2.5 The solution we propose : The Object Manager ....... 5
      2.5.1 the Object Manager specifications ................ 5
      2.5.2 How the Object Manager is used now ............. 6
   2.6 Proposed Extensions to the Eiffel language ......... 7
      2.6.1 the needs requirement ............................ 7
      2.6.2 the uses requirement .............................. 7
      2.6.3 the internal keyword .............................. 8
      2.6.4 the builder keyword .............................. 8
      2.6.5 How the Object Manager could be implemented ? . 8

3. Persistent Objects : a new object sub-category ........ 8

4. What is lacking in the Class Interface .................. 9
   4.0.6 Notes ........................................... 10

5. Deductive Object Programming .......................... 10
   5.1 definition .......................................... 10
   5.2 connection with the Production Tree .................. 11
   5.3 improvement of code Quality ........................ 11
      5.3.1 lower complexity ................................ 11
      5.3.2 objects reusability ............................... 12
      5.3.3 more evolutionary capabilities ................... 12
      5.3.4 better class design .............................. 12
   5.4 distribution of the building of Objects Persistent .... 12
   5.5 extending the Concept of a Calculation .............. 13
   5.6 iterative processes ................................. 13

6. Conclusion ............................................ 13

7. Acknowledgements ...................................... 14
8. Annexes

8.1 A : an usual implementation of class TRIANGLE ............... 14
8.2 B : a new implementation of class TRIANGLE ............... 15
8.3 C : the external tree of a TRIANGLE ....................... 17
8.4 C : the internal trees of a TRIANGLE ....................... 18
8.4.1 C-a : the internal tree of surface ....................... 18
8.4.2 C-b : the internal tree of centroid ....................... 18