# O'CIAO
# An Object Oriented Programming model using CIAO Prolog

**A. Pineda, M. Hermenegildo**

Computer Science Department

Technical University of Madrid (UPM), Spain

{apineda,herme}@clip.dia.fi.upm.es

**Abstract**

There have been several previous proposals for the integration of Object Oriented Programming features into Logic Programming, resulting in much support theory and several language proposals. However, none of these proposals seem to have made it into the mainstream. Perhaps one of the reasons for these is that the resulting languages depart too much from the standard logic programming languages to entice the average Prolog programmer. Another reason may be that most of what can be done with object-oriented programming can already be done in Prolog through the meta- and higher-order programming facilities that the language includes, albeit sometimes in a more cumbersome way. In light of this, in this paper we propose an alternative solution which is driven by two main objectives. The first one is to include only those characteristics of object-oriented programming which are cumbersome to implement in standard Prolog systems. The second one is to do this in such a way that there is minimum impact on the syntax and complexity of the language, i.e., to introduce the minimum number of new constructs, declarations, and concepts to be learned. Finally, we would like the implementation to be as straightforward as possible, ideally based on simple source to source expansions.

## 1 Introduction

Since the first steps on object oriented programming, many attempts has been made in order to join both logic and object oriented programming styles into a single language. Most of those proposals depart from the standard Prolog language, perhaps due to the starting point of view used to design them. In this paper, we will try another approximation taking an existing Prolog platform as starting point. First, we will try to locate those concepts present on the Prolog language which are very similar to those involved on object oriented languages. Then we will discuss how those concepts may be modified to reach object oriented behavior.

Traditionally, four main concepts are involved in Object Oriented Programming:

- State encapsulation.
- Instantiation.
- Inheritance relationships.
- Polymorphism.

State enacapsulation is already provided by Prolog in the form of *modules*. In this way, the concept of module is something very similar to the concept of *class*. Following this reasoning, we may say that the concept of *attribute* (also known as *property*) is something very similar to the concept of *dynamic predicate*. And the concept of *public method* is something very similar to the concept of *exported predicate*.

Polymorphism is also provided by a Prolog system. This is done in two ways:

- Since Prolog is not a typed language, polymorphic declarations inside the same encapsulation have no sense.

  For example, the following C++ polymorphic declarations :

  ```
  void myclass::method(int)
  void myclass::method(float)
  ```

  are written in Prolog as follows :

  ```
  method(Arg) :-
        integer(Arg),
        !,
        ...
  method(Arg) :-
        float(Arg),
        !,
        ...
  ```

- Support for polymorphic method calling is already present in the form of higher-order (or meta-programational) structures. For example, suppose that method/0 is both exported by modules m1 and m2:

  ```
  :- use_module(m1).
  :- use_module(m2).
  main :-
      aux(m1),
      aux(m2).
  aux(I) :-
      I:method.
  ```

  Notice that the *module qualificator* for method/0 predicate is just a variable.

As a result, we must center our efforts in providing **instantiation** and **inheritance** to our Prolog system. In the following sections, we will first describe our object-oriented programming model,then we will discuss some implementation issues.

# 2 The object-oriented programming model

Classes are declared the same way as modules, where original source code will suffer a source-to-source expansion which is called the *class expansion*. Since O'CIAO is based on the CIAO Prolog system, we will describe here the needed syntax for this Prolog system:

```
:- class(ClassName).
```

or (SICStus-like module declaration):

```
:- module(ClassName,[],[class]).
```

The following example declares a class which implements an "imperative-style" stack of elements:

```
:- class(stack).
:- dynamic storage/1.
:- export([push/1,pop/1,top/1]).
```

```
push(Item) :-
        nonvar(Item),
        asserta(storage(Item)).
pop(Item) :-
        var(Item),
        retract(storage(Item)).
top(Top) :-
        storage(Top),
        !.
```

Notice the similarity with a traditional Prolog module. In fact, this kind of class may be called an *instantiable module*, just because there is no usage of object oriented specific declarations (i.e. inheritance relationships). The concept of instantiable module is a previous step to the concept of class. There is only one point of difference between a module and an instantiable module: "copies" of the instatiable module may be (both dynamically and statically) generated:

```
:- use_class(stack).
:- stack1 instance_of stack.
:- stack2 instance_of stack.
main :-
        Stack3 new stack,
        stack1:push(a),
        stack2:push(b),
        Stack3:push(c).
```

In the example above, *stack1* and *stack2* are declared to be *instances* of the stack module. Those instances are used as they were loaded modules. Additionaly, instances may be also created at run-time (this is the case of *Stack3*) by the usage of the *new/2* operator.

From this point, the idea of instantiable modules is enriched in order to become an object oriented programming model based on the Prolog module system. Basically, this is reached by both adding a few new declarations and changing the meaning of some existing ones: dynamic/1 declarations change their semantics to *attribute declarations*. export/1 declarations also change their meaning to *public predicate declarations*.

### 2.1 Inheritance relationships

We have based most of the O'CIAO inheritance model on that provided by the Java programming language. This language makes use of two different kinds of inheritance relationships: code inheritance and interface inheritance.

Code inheritance is very similar to usage relationships already provided by Prolog[1], differences are relative to these points:

- Predicates are re-exported by default (known as public inheritance).
- As a consequence of previous point, there is transitivity on the inheritance relationship: whether "a" inherits from "b" and "b" from "c", there is an implicit inheritance relationship between "a" and "c".
- Ambiguous predicate callings are solved in a different way.
- Visibility of predicates can not be restricted: an inherited private predicate may be exported, but an inherited public predicate will not become a private one. This ensures an uniform public interface along the inheritance line.

[1] use_module/1 declaration.

Code inheritance works towards source code maintenance and reusability. Since it was the first kind of inheritance to be developed, it is usually known simply as "inheritance". We have implemented single code inheritance by adding a new declaration [2]:

```
:- inherit_class(Source).
```

Another declaration is needed in order to distinguish between private predicates and those that may be inherited by descendant classes [3]. Predicates are private by default, so inheritable predicates must be explicitly marked[4]:

```
:- inheritable(F/A).
```

The following example illustrates the usage of those mentioned declarations. The "item" class declares *set_value/1, get_value/1 and datum/1* to be inherited by descendant classes, in particular, they are inherited by the "tagged_item" class:

```
:- class(item).
:- export([set_value/1,get_value/1]).
:- inheritable(datum/1).
:- dynamic datum/1.
set_value(X) :-
        retractall(datum(_)),
        assert(datum(X)).
get_value(X) :-
        datum(X).

:- class(tagged_item).
:- inherit_class(item).
:- export([set_tag/1,get_tag/1]).
:- dynamic tag/1.
set_tag(X) :-
        atom(X),
        retractall(tag(_)),
        assert(tag(X)).
get_tag(X) :-
        tag(X).
```

Interface inheritance works towards external world (and concept) modeling and uniform usage of different pieces of code. Unfortunately, there is no similar concept present on Prolog systems. However, there is no much difficult on implementing it: the compiler must ensure that a particular source will export the same set of predicates as another one. Interface inheritance has been implemented by adding the following declaration:

```
:- implements(Source).
```

Where *Source* may be a class-expanded source or an *interface-expanded* source. *interface-expanded* sources are similar to the interface declarations provided by the Java language. Only *export/1* and *data/1* declarations are allowed. For example:

```
:- interface(must_be_implemented).
:- export([a/1,b/1]).
```

And this code illustrates how interfaces are used:

[2] Multiple code inheritance is not supported.
[3] Usually known as *protected* predicates.
[4] Except for exported predicates, which are inheritable by default.

```
:- class(itf_example).
:- implements(must_be_implemented).
:- implements(item).
a(_).
b(_).
set_value(_).
get_value(_).
```

The following example has the same effect as the previous one, and will introduce the meaning of interface inheritance:

```
:- class(itf_example).
:- export([a/1,b/1]).
:- export([set_value/1,get_value/1]).
a(_).
b(_).
set_value(_).
get_value(_).
```

### 2.2 Object Initialization

Classes may be improved by the usage of *constructors and destructors*. Constructors are implicitly declared by writing clauses of a predicate whose functor matches the class name (any and all arity is allowed). Destructors are also implicitly declared by writing clauses for a destructor/0 predicate. Those are mainly used to allocate/release system resources at the proper time. For example:

```
:- class(file_reader).
:- export(get_char/1).
:- dynamic handler/1.
file_reader(FileName) :-
        open(FileName,read,Handler),
        assert(handler(Handler)).
destructor :-
        handler(H),
        close(H).
get_char(C) :-
        handler(H),
        current_input(OldIN),
        set_input(H),
        get_code(I),
        set_input(OldIN),
        I =\= -1.
```

Constructors are usefull to perform some initialization just after object creation takes place. *State initialization* is a particular case which is also provided in the traditional Prolog way:

```
:- class(has_initial_state).
:- dynamic state/1.
state(active).
```

### 2.3 Virtual method calling

Virtual method calling allows descendant classes to provide different implementations for a predicate. This is a popular programming technique known as *code specialization*. The ancestor will allways call that version defined deeper in the inheritance line. O'CIAO provides the virtual/1 declaration in order to do so:

```
:- class(generic_item).
:- export([set_value/1,get_value/1]).
:- inheritable(datum/1).
:- dynamic datum/1.
set_value(X) :-
        validate_item(X),
        retractall(datum(_)),
        assert(datum(X)).
get_value(X) :-
        datum(X).
:- virtual validate_item/1.
validate_item(I) :-
        nonvar(I).

:- class(integer_item).
validate_item(I) :-
        integer(I).
```

In the previous example, any object derived from "integer_item" class will accept (only) an integer as argument to set/1, regardless of the validate_item/1 check implemented in the "generic_item" class.

## 2.4 Predicate overriding

Overriden predicates are also known as redefined predicates. A predicate is said to be overriden when it has been inherited from any ascendant class, but the same predicate has been defined at current source. In order to distinguish between both definitions there is an **inherited/1** predicate qualificator. By default, the local predicate is used, if the inherited predicate definition needs to be called, it must be qualified using the inherited/1 operator:

```
:- inherit_class(defines_predicate_foo).
foo(9).                             % predicate overriding
example :-
        foo(X),                     % local definition is called.
        inherited foo(X).           % inherited definition is called.
```

## 2.5 Retrieving self instance identifier

Sometimes, an object needs to know what its name is: the self instance identifier. There is a **self/1** predicate in order to do so. Usually this feature is used to tell another object who is making a call, this enables the called object to invoke the first one again. Example:

```
:- class(clonation).
:- dynamic some_data/1.
clone(Object) :-
     retractall(some_data(_)),
     Object:some_data(X),
     assert(some_data(X)),
     fail.
clone(_).

copy_to(Destination) :-
     self(Me),
     Destination:clone(Me).
```

## 2.6 Object creation and manipulation

First step in order to handle object oriented code is to load the *objects* CIAO package:

```
:- use_package(objects).
```

which is something very similar to an *include/1* declaration.

This procedure will load a run-time support library, declare some operators, and enable the **use_class/1** declaration: such declaration establishes an usage relationship between the current code and the given class. This behavior is analogous to those provided by the *use_module/1* declaration. For example :

```
:- use_package(objects).
:- use_class(library(file_reader)).
main([FileName]) :-
     File new file_reader(FileName),
     echo(File).
echo(File) :-
     File:get_char(I),
     put_code(I),
     fail.
echo(_).
```

Object creation will take effect by the ways of the **new/2** operator. This operator will take a free variable as first argument (which will be binded to an unique instance identifier), and a class constructor as second argument. For example:

```
?- X new xmessage('Hello there !!!').

X = '193847705' ?

yes
?-
```

Whether no class constructors were defined, a default constructor (with zero-arity) may be used. Instance creation will fail whenever the given constructor fails.

Once the object has been created, any exported (public) predicate may be called as a traditional module-qualified goal:

```
X new myclass , X:mymethod(Z) .
```

This package also provides the **instance_of/1** operator which checks whether an instance was derived from a class, or any of its descendants:

```
foo(Obj) :-
    Obj instance_of myclass,
    Obj:method.
```

# 3 Implementation issues

Intuitively, our main implementation challenge is derived from providing instantiation. There is a very simple solution which has been used before for some beginner Prolog programmers: the "extra" argument. This solution adds a new argument to every dynamic predicate which identifies the instance (object) owner. Such a solution *must* be avoided since Prolog indexing is broken[5]. It brings up important performance slowdown when the number of dynamic clauses grow up. This

[5]Notice that Prolog indexing is not broken at static predicates whether the "extra" argument is the last argument.

fact led us to a new implementation restriction: performance slowdown must be avoided as long as possible. As a result, any source-to-source expansion must also work in this way.

Another difficult on implementation is derived from the interdependency between inheritance and instantiation solutions. new/2 implementation and performance is fully dependant from that code auto-generated by the class expansor. Unfortunately, the class expansion robustness and efficiency is dependant from the underlining Prolog compiler capabilities. In general lines, the most information is known by the class expansion, the better performance is reached.

CIAO Prolog compiler allows code expansion in two stages. At the first stage, declarations are processed in order to generate any needed interface information. At the second stage, module dependences has been solved by the compiler. This seems that code may be generated *taking account* of other module (and class) declarations in relation with the current code being expanded. This interesting feature allows O'CIAO class expansor to suppress many run-time checks, generate many static code, and perform exhaustive semantic analysis on class code.

## 3.1 The class expansion

The *class expansor* performs several tasks:

- Semantic analysis.
- Enable instance distinction.
- Help on instance creation.

Semantic analysis on object oriented declarations will be easy to implement whenever module dependences are known by the code-expander. It will report any semantic error or warning due to incorrect usage of declarations. Currently, O'CIAO performs over 35 different checks over class-expanded code. To illustrate which kind of semantic errors could be found, we enumerate some of those checkings:

- Multiple inheritance declarations.
- Trying to establish an inheritance relationship with no class-expanded code.
- Trying to declare constructors/destructors as dynamic or multifile predicates.
- Exporting (publishing) predicates not related to object oriented programming.
- Trying to export (or mark as inheritable) a predicate not defined nor inherited at current class.

### 3.1.1 Enabling class code to distinguish between different instances

Each time a method predicate is called, the involved code must notice which is the current instance being executing. This is needed for the assert/retract predicates to operate over the proper state encapsulation.

This situation implies that the previously mentioned "extra" argument can not be avoided, but must be hidden from the user. In fact, this extra argument is also known as the *hidden argument* in other object oriented languages. These approach is also adopted by O'CIAO which will add a hidden argument to every **static** clause[6] present at the user code. The following approximation illustrates the idea[7]:

```
method(V) :-
        assert(attr(V)).
```

[6]And only to every static clause, in order to preserve Prolog indexing. Dynamic predicates must be treated in a different way.

[7]Instance identifiers are assumed to be simple atoms.

will be expanded to something like:

```
method(V,Instance) :-
        assert(Instance:attr(V)).
```

#### 3.1.2 Providing state encapsulation

As mentioned before, state encapsulation is also provided by Prolog in the form of modules. This seems that an instance (object) will be some kind of *run-time generated* Prolog module, which will be (basically) build of dynamic predicates. However, there are two implementation alternatives when inheritance is present:

**One run-time generated module in representation of each class:** For each class along the inheritance line, a module is generated in representation of that class state. Then, a simple relationship is internally allocated between those modules. Each time an inherited call is performed, the needed module must be first determined (at run-time).

Advantages:

- Attribute overriding is trivial. There is no possible confusion between two attributes with the same functor and arity defined at different classes.

Drawbacks:

- Attribute inheritance is **not** trivial. When a given attribute is not known to be defined at current class, we must determine which class (along the inheritance line) has the proper definition[8]. Then, the module in representation of such a class must be determined at run-time.
- Virtual method calling will be difficult to implement. Some kind of *virtual method table* will be needed in order to perform such a kind of calling. That table must be run-time generated for each instance.

Those drawbacks will have a direct impact on new/2 performance, and method calling slowdown.

**One run-time generated module in representation of all classes:** Only one run-time generated module will hold all attributes defined in all classes along the inheritance line.

Advantages:

- Attribute inheritance is trivial, since all attributes are allocated into the same encapsulation.
- Virtual method calling is trivial. This behavior will be discussed later.
- There is no method calling slowdown[9].

Drawbacks: attribute overriding is not trivial. Two identical attribute definitions on different classes can not be distinguished due to the use of a unique state encapsulation. Obviously, this problem can be solved by ensuring unique attribute names across the inheritance line. This can be easily reached by using some kind of name convention. For example:

```
:- class(myclass).
:- dynamic attr/1.
method(Value) :-
        assert(attr(Value)).
```

[8] Most times, this can be done at compile time.

[9] Except for that derived from the presence of an extra argument, which can not be avoided.

will be expanded to something like:

```
:- module(myclass).
method(Value,Instance) :-
      assert(Instance:'myclass::attr'(Value)).
```

Unfortunately, such a solution led us to another problem: attributes can not be trivially exported (published) because of that name translation. For example:

```
main :-
     X new someclass,
     assert(X:attr(V)).
```

needs to be expanded[10] to:

```
main :-
     X new someclass,
     assert(X:'someclass::attr'(V)).
```

This implementation will require some kind of **global analysis technique** in order to determine which class derived the instance denoted by the involved variable. As an example, consider the following code:

```
aux(X) :-
     assert(X:attr(V)).
```

Note that (using traditional compiling techniques) there is no chance to know whether *attr/1* is a valid attribute and how it must be expanded.

There are other chances to solve this problem:

- Not to allow attribute exportation. This is not a hard restriction since good programmers will usually provide a set of methods in order to control state access.
- To define *predicate aliases* for exported state. This feature must be supported by the underlining Prolog engine. Predicate aliases are symbol table entries which points to the same set of clauses as the aliased predicate[11].
- To allow attribute exportation only for goal calling porpouses. This solution is easy to implement using "chaining clauses" for attributes (this technique is discussed later in section 3.2).

O'CIAO is currently implemented using only one run-time generated module. Attributes are exported only for goal calling porpouses (assert/retract predicates are not allowed to operate over exported attributes).

#### 3.1.3 Helping on instance creation

As user-code is being expanded, some usefull information may be recorded in relation to instance creation. So, some new clauses (not present at the user code) should be generated using that information. The easiest way to store that information, which will be used at run-time, is by the usage of **multifile** predicates. That set of multifile clauses will build a *class template* in order to help instance creation.
That class template will hold the following information:

[10] Such an expansion should be performed by the *objects* package.
[11] Predicate aliases also solves some other problems present at traditional Prolog module system.

- The class name.
- The class ancestor (if present).
- A relation of public predicates: usefull whether run-time checks are needed.
- A relation of attributes and how to create them.
- A relation of methods and which classes defined them. The usage of this information will be discussed on section 3.2.
- State initialization information.

This is an example of those multifile clauses generated for the "generic_item" and "integer_item" classes:

```
class(generic_item).
public(generic_item,set,1).
public(generic_item,get,1).
attribute_template(generic_item,datum,1,'generic_item::datum').
method_template(generic_item,set(X),generic_item:set(X,_)).
method_template(generic_item,get(X),generic_item:get(X,_)).
method_template(generic_item,validate_item(X),generic_item:validate_item(X,_)).

class(integer_item).
super(integer_item,generic_item).
public(integer_item,set,1).
public(integer_item,get,1).
attribute_template(integer_item,datum,1,'generic_item::datum').
method_template(integer_item,set(X),generic_item:set(X,_)).
method_template(integer_item,get(X),generic_item:get(X,_)).
method_template(integer_item,validate_item(X),integer_item:validate_item(X,_)).
```

### 3.2 Providing instantiation

Object creation takes place by the usage of the new/2 operator. Most of its implementation is based on the underlining Prolog engine primitives, such as dynamic predicate creation. Those are the needed tasks for new/2 to perform:

- Generating an unique instance identifier.
- State encapsulation creation.
- Method interface creation (discussed below).
- Enabling Virtual method invokation.
- Calling the needed constructor.

State encapsulation is easily performed from the proper auto-generated template[12]:

```
state_creation(ID,Class) :-
      attribute_template(Class,Functor,Arity,RealFunctor),
      module_concat(ID,RealFunctor,NewFunctor),
      '$define_predicate'(NewFunctor/Arity,interpreted),
      fail.
```

[12]All those implementation examples has been simplified.

Public methods needs to be called in the same way as attributes: i.e. `Obj:method(Arg)`, but the implemented goal must be called as `proper_class:method(Arg,Obj)`. Method interface creation allows this situation using *chaining clauses*:

```
method_creation(ID,Class) :-
      method_template(Class,Goal,Class_or_ancestor:RealGoal),
      functor(RealGoal,_,LastArg),
      arg(LastArg,RealGoal,ID),
      assert( (ID:Goal :- Class_or_ancestor:RealGoal) ),
      fail.
```

Whether '123' were the instance identifier, and foo/3 a public method, the following clause should be asserted:

```
'123':foo(A,B,C) :- proper_class:foo(A,B,C,'123').
```

Knowing this, virtual methods are easily enabled. Whether a virtual method call is present, for example:

```
:- virtual validate_item(X).
set(X) :-
        validate_item(X),
        retractall(datum(_)),
        assert(datum(X)).
```

code must be expanded in this way:

```
set(X,Instance) :-
        Instance:validate_item(X),
        retractall(Instance:'generic_item::datum'(_)),
        assert(Instance:'generic::datum'(X)).
```

### 3.3 Enabling code to manipulate objects

As mentioned before, *usage relationships* are needed for some code to create objects or manipulate them. The *objects* package is in charge of such a task. Usage relationships may be also implemented by the usage of a single multifile predicate: `used_class`(*Class*,*Module*), which denotates "Class is used by Module"[13]. This information is used in conjuction with the `public/3` template in the following way:

The underlining Prolog engine will use some kind of dynamic predicate which holds information about module importation/exportation. That information is used to determine whether a module is allowed to call a given predicate from a given module. If not, an exception is usually raised. CIAO Prolog uses the predicate `imports`(*Module*,*From_module*,*F*,*A*), which stands for: "Module imports F/A from From_module". So, object's public interface accesibility may be controled using this trick[14]:

```
object_interface_creation(Object,Class) :-
        assert( (imports(Module,Object,F,A) :-
                        used_class(Class,Module),
                        public(Class,F,A) )),
        super(Class,Super),
        !,
        object_interface_creation(Obj,Super).
object_interface_creation(_,_).
```

[13]Where *Module* may be another class, a Prolog module, or a main program.

[14]Note: *Object and Class* variables will be binded to atoms at call time.

### 3.4 Object destruction

Object creation makes use of some amount of memory and system resources that must be released when no longer needed, this is called *instance destruction* or *object destruction*.

Two implementation alternatives are known to solve this problem: explicit and implicit destruction.

Explicit destruction makes use of some kind of operator in order to do so. This operator must be manually called by the programmer, who must ensure that no more references to the object are left in memory. Some popular languages uses explicit instance destruction, such as C++ and Object Pascal.

Implicit destruction works transparently to the programmer. This implementation first ensures that no references to a particular object are left in memory, then it destroys the given object. Implicit destruction needs a **garbage collector** to perform such a task. Java language has implemented this approach with successfull results. Prolog engine also works using a garbage collector, so there is a chance to implement implicit destruction.

At current stage of O'CIAO development, instance destruction is explicitly invoked by a *destroy/1* operator.

## 4 Performance tests

Three different benchmarks were used to compare O'CIAO performance against CIAO normal module system. At first sight, we may predict runtime slowdown due to instance creation (new/2 operator) and instance distinction issues. This last behavior has a direct impact on method execution performance. So, the main target of this test is to determine whether lost on performance is justifiable by the gaining on object oriented enhancement, or not.

### 4.1 Instance creation and destruction overhead

Instance creation as well as instance destruction may be considered as "wasted time" since it does not execute user code (except for constructors and destructors), but it can not be avoided whether object oriented programming is needed.
Execution time on new/2 for a given class depends on those factors:

- The whole number of attributes: all attributes defined in all classes along the inheritance line.
- The whole number of methods.
- Constructor code (if present).
- State initializations (if present).
- The actual size of the Prolog database, internal index tables, and so on. This is due to the usage of dynamic predicates and atoms as instance identifiers.

Constructor and state initialization execution time was not considered on benchmarking since it involves useful user actions which can not be avoided. Figure 1 shows execution time ratios on both creating and destroying an instance, the involved classes had the following characteristics:

- Different number of attribute declarations.
- No inheritance nor method declarations.
- No state initialization nor constructor/destructor definitions.

Figure 2 shows execution time ratios when involved classes declare methods instead of attributes.

Average execution time is 0.07 ms per attribute and 0.22 ms per method. So, first important result is that creating a method is three times slower than creating an attribute.

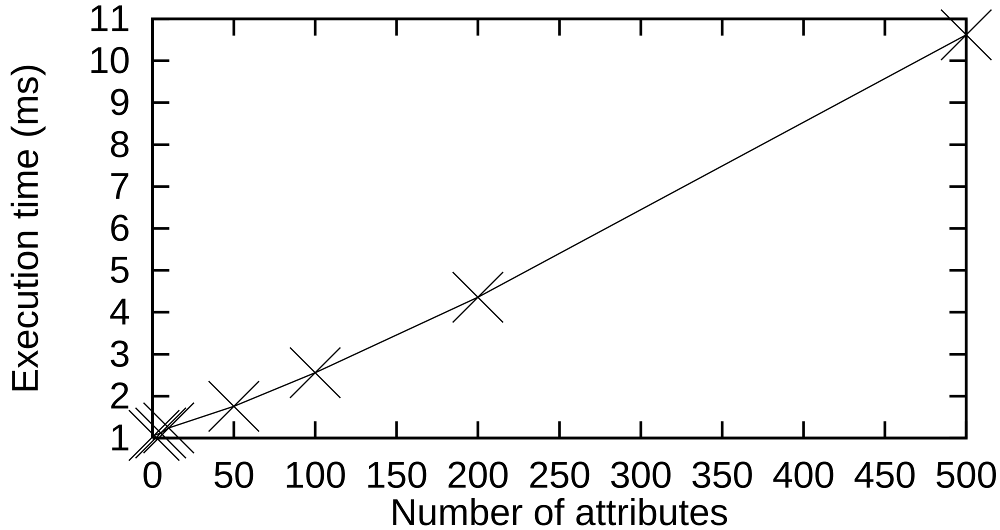


Figure 1: Involved execution time in creating and destroying an instance versus number of attributes.

### 4.2 Predicate calling overhead

The porpouse of this benchmark was to measure goal calling slowdown due to instance distinction issues. The test compared execution times for `Object:goal` versus `test_module:goal` once `Object new test_class` was executed. Code for "test_class" was just the same as "test_module" except for the necessary *class expansion*. Both dynamic predicate and static predicate calling was tested. Benchmarking result was:

- **9.5 %** slower on static predicate calling.
- **16.7 %** slower on dynamic predicate calling.

### 4.3 assert/retract overhead

This test was designed to evaluate assert/retract slowdown due to the usage of run-time generated state encapsulations. `assert(data(1)),retract(data(_))` was compared against `assert(Object:data(1)),retract(Object:data(_))` which is the same code but class-expanded.Such class-expanded code resulted to be **23.3 %** slower.

## 5 Previous related work and final conclusions

First steps on Logic and Object Oriented Programming are found in relation to Parallel and Concurrent Programming in the eighties. As signaled by [8], Object Oriented behavior may be used as a modeling tool for processes and their intercommunication. This tentative is based on the KL1 language, which is not a traditional Prolog system.

Then, many efforts has been focused on the formal semantics of objects and their logical meaning in First Order Logic ([5]) and High Order Logic ([4],[2]). Following this line, there have been also many advances in the area of Linear Logic ([1],[6]).

Another critical point of discussion is inheritance behavior ([3],[7]).

Most of those proposal has derived on a number of new languages which have not spread so much over the programming community, perhaps, due to the resulting complexity on language syntax.

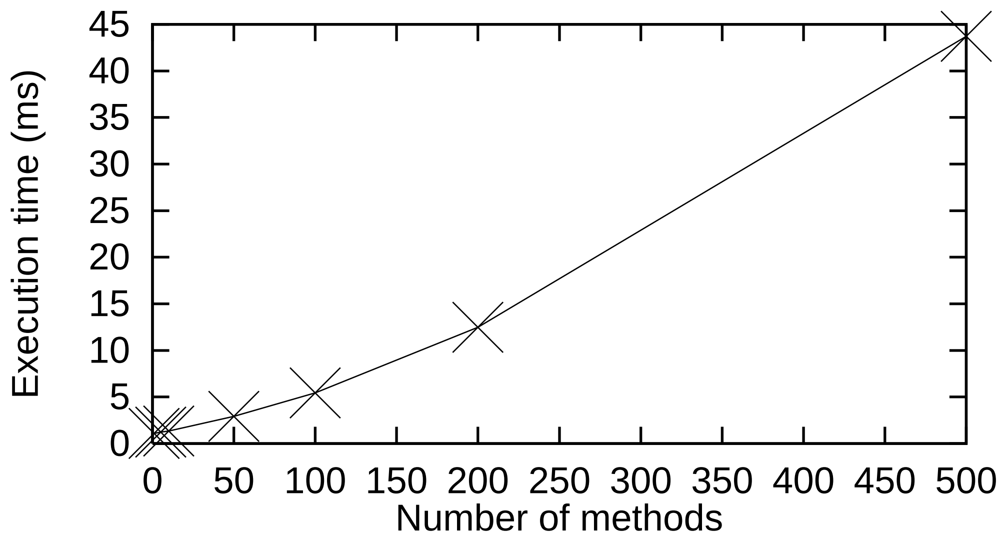


Figure 2: Involved execution time in creating and destroying an instance versus number of methods.

O'CIAO was designed keeping in mind that an experienced Prolog programmer will be reticent about learning a lot of new declarations and syntax extensions. On the other side, O'CIAO provides a good tool for the imperative-style programmer to learn about declarative languages. Most of the involved concepts (inheritance, state encapsulation, etc) are very similar to those related to imperative object oriented languages. Another point of difference between O'CIAO and previous proposals is that it has been implemented over a first order logic based language such as Prolog which, in addition, is becoming a very popular language day by day.

Other authors has focused their efforts on the declarative semantics of Object Oriented Programming. We prefer not to worry so much about this stuff basing our development on the well-known semantics of the Prolog module system, and focusing our efforts on implementation and performance of the system.

## References


[1] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[2] Hassan At-Kaci. An introduction to life – programming with logic, inheritance, functions and equations. In *Proceedings of the 1993 International Symposium on Logic Programming*, 1993.

[3] M. Bugliesi. A declarative view of inheritance in logic programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992.

[4] Weidoing Chen and David Scott Warren. Objects as intensions. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, August 1988.

[5] John S. Conery. Logical objects. In *Proceedings of the Fifth International Conference and Simposium on Logic Programming*, August 1988.

[6] Maurizio Martelli Giorgio Delzanno. Objects in forum. In *Proceedings of the 1995 International Symposium on Logic Programming*, December 1995.

[7] Laks V. S. Lakshmanan Hassan M. Jammil. A declarative semantics for behavioral inheritance and conflict resolution. In *Proceedings of the 1995 International Symposium on Logic Programming*, December 1995.

[8] K. Furukawa M. Ohki, A. Takeuchi. An object-oriented programming language based on the parallel logic programming language kl1. In *Proceedings of the Fourth International Conference on Logic Programming*, May 1987.