# Refining Classes in Statically Typed Object-Oriented Languages

Jean Privat, Roland Ducournau

# Class Refinement in Statically Typed Object Oriented Languages

Jean Privat and Roland Ducournau

LIRMM – Université Montpellier II
161 rue Ada
F-34392 Montpellier cedex 5

**Abstract.** Classes and specialisation bring simultaneously structuration and flexibility to object oriented programs. However, numerous model extension propositions (aspects, modules, etc.) prove these qualities are often considered insufficient. This article proposes two binded notions of class refinement and modules, the first one brings flexibility and the second one, the structuration. Modules contain a coherent set of class definitions and can modify classes defined in modules they depend. This proposition focuses on statically typed languages where modules can be separately compiled. It is based on a module metamodel similar to the class one and problems with multiple specialisation and refinement are managed like those in multiple inheritance.

## 1 Introduction

Object oriented programming languages offer to programmers a powerful development framework allowing at the same time to build stable and coherent entities (classes) while allowing a flexibility and evolutionarity thanks to specialisation and inheritance mechanism [1]. Nonetheless, this flexibility is not always enough and many approaches were developed, from reflection allowing program behaviour modification at compile-time, link-time or runtime, to aspects allowing *weaving* new behaviours to existing classes.

The class refinement model we propose belongs to these approaches but focuses on statically typed languages with multiple inheritance. It is characterised by a modular decomposition in which each module is autonomous, i.e. "separately compilable", but may *refine* – extend – one or more classes which exist in parent modules by adding new properties, modifying existing properties, adding new specialisation relation, unifying classes or generalising properties. Our proposition is based on a structural analogy between modules and classes where the module metamodel is isomorph to that of the class: mechanisms and difficulties that occur with multiple dependencies are analogous those that occur with multiple inheritance.

Section 2 exposes the refinement problematics and motivations. Section 3 revises the object metamodel we consider (without refinement), then focuses on multiple inheritance. Section 4 introduces the class refinement in the model and section 6 is about multiple inheritance and refinement. Section 5 depicts

implementation and compilation techniques. Section 7 gathers related works. Finally, section 8 presents conclusion and perspectives.

## 2 Problematic of refinement

### 2.1 Needs of flexibility and structure

Many works propose schemas and implementations of class refinement techniques that allow a class definition somewhere in a program and amendments of this class definition somewhere else. Flexibility brought by theses techniques in general goes hand in hand with program structuring, not only in the form of classes but also in the form of functionalities: functionalities strongly binded together may be gathered instead of dispersed through all the classes of a program. [2] evokes this need for orthogonality between objects and functionalities which is missing in the models with classes. [3] shows the need for modules.

Another interest is to be able *a posteriori* to make a whole set of classes (program or library) evolve without having to modify the existing classes (either the classes may be shared and must continue to exist in their original form, or the source code may be unavailable). Reflexive languages (CLOS [4], SMALLTALK [5]) or aspect oriented ones (ASPECTJ [6]) already allow this via a meta-object protocol (MOP) or a weaver. This paper considers a novel approach: the proposal of a technique applicable to statically typed languages separately compiled and not requiring the use of an external mechanism like a MOP or a weaver.

We focus on five class amendments: the method and attribute addition and redefinition (the attribute redefinition only makes sense with covariant typing context), superclass addition, classes unification (the expression that two classes defined in different places are the same, namely instances and properties of one class are those of the other one) and property generalisation (bring up the introduction of a property in a superclass).

We place ourselves within a multiple framework of heritage.

### 2.2 Intuitive idea of class refinement

Intuitively, one can present the refinement of a class $c_1$ by a class $c_2$ as an incremental definition of the classes in which the properties defined in $c_2$ are added or replace those of $c_1$. Such a process more or less frequently meets in the languages with dynamic typing. It is the case, for the methods at least, in the object oriented extensions of LISP where the methods are defined outside the classes. In a general way, it should be possible in all the languages equipped with a *meta-object protocol* [7] like CLOS, even if experiments show that these protocols are not always very adapted to the modification of the classes [8].

The intuitive character of refinement is lost a little with multiple refinement of the same class: the order of refinements and specialisation has consequences. If dynamic languages can base themselves on the chronological order, that is not possible for statically typed languages.

Thus, to give a more structured framework to our proposal, we add a concept of *module* to it, in its simpler form, rather traditional. A *module* is a set of classes, that dependent on a set of other modules. The order of refinements results then from the order on the modules. There is no concept of visibility (or export), nor namespace associated with the modules: it is not necessary for the moment.

## 3 Object metamodel

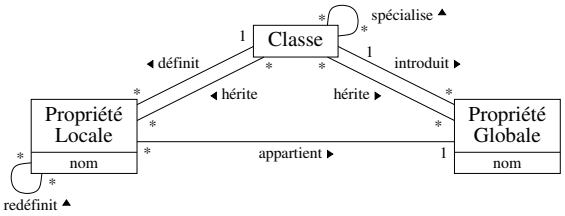



**Fig. 1.** Property metamodel

The starting metamodel is composed of three main kinds of entities: classes, local properties and global properties (Fig. 1). Although it is not explicit in any language, nor even in UML, we think that it has vocation with the universality: it is the implicit metamodel of JAVA and EIFFEL (in condition to make a limited use of renaming). This section is a resume of [9]

### 3.1 Entities and relations

When we wants to metamodel properties, late biding (message sending) forces to define two categories of entities. *Local properties* correspond to the attributes and the methods such as they are defined in one *class*, independently of possible redefinition. *Global properties*[1] correspond to the messages which the instances of a class can answer to: this answer corresponds to a local property of the class. Each local property belongs to a single global property and in each class there is a one-to-one mapping between its local properties (defined or inherited) and its global properties. The correspondence between these two entities is done by their names.

Inheritance mechanism through the relation of specialisation makes possible the subclasses to inherit the properties of their super-classes. This inheritance takes place on two levels. On the level of the global properties, called *name inheritance*, any class has the properties owned by its super-classes. On the level of the local properties, called *value inheritance*, if a class brings its own local

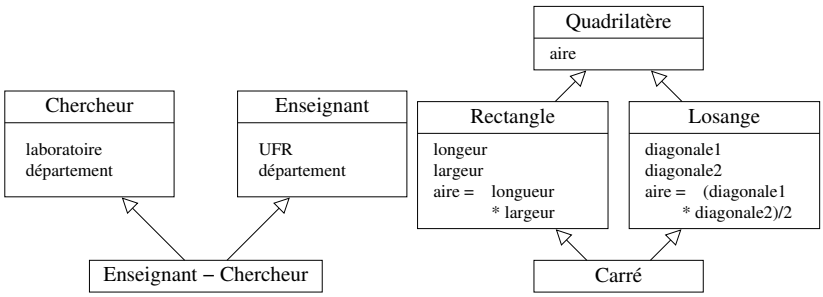
[1] Similar to *generic functions* of CLOS.

property for a given global property, it is a (re)definition; if not, the class inherits the most specific local property defined in its super-classes, i.e. the local property defined in the most specific super-class by the relation of specialisation. When a class defines a local property whose name does not correspond to any inherited global property, a new global property is implicitly introduced into the class and is associated to the local property.

Lastly, for a message send `x.foo(args)`, `foo` indicates the global property named `foo` of the static type of the receiver `x`. Let us note that dynamic typing makes this metamodel inoperative.

### 3.2 Multiple inheritance

With multiple inheritance, conflicts are the main difficulty. Obviously, the metamodel yields two kinds or conflicts.




2.1 Property name conflict 2.2 Property value conflict

**Fig. 2.** Multiple inheritance conflicts

**Name conflict.** A *property name conflict* occurs when a class specialises two classes having distinct but homonym global properties. Fig. 2.1 shows two classes (`Researcher` and `Teacher`) having both a global property named `department`. One indicates a department in a research laboratory, the other a teaching department in an university. It is then awaited that the common subclass inherits the global properties of its superclasses, but the name `department` is ambiguous in the context of the subclass.

Anyway, a name conflict is only a naming problem and a systematic renaming would guarantee the absence of name conflicts. This renaming may be automatic or specified, local or global.

**Forbidden** The specification of the language proscribes the conflicts of names. This situation forces the programmer to rename at least one of the two global properties in all programs that use it. This may imply the modification of

many classes (with the inherent potential errors) but it can be impossible if these classes are not modifiable (unavailable source code for example).

**Explicit designation** The litigious property name must be prefixed by a class where the name is not ambiguous, for example the class that introduces the global property. In the specification of the class `Teacher-Researcher` of the example, `Teacher:department` would design the global property known as `department` in the class `Teacher`. This solution is used in C++ [10] for the attributes[2]. It has two drawbacks: heaviness of writing and it does not respect the principle of anonymity which wants that the explicit references to the super-classes appear the least possible in order to improve the modularity.

**Local renaming** Local renaming makes it possible to change the designation of a global property in a class and its future subclasses. In the problematic class of the example, we can rename `department` inherited from `Teacher` into `dept-teach` and `department` inherited from `Researcher` in `dept-res`. Thus `department` in `Researcher` and `dept-res` in `Teacher-Researcher` indicate the same global property and as expected, in the class `Teacher-Researcher`, `dept-res` and `dept-teach` indicate two distinct global properties. This solution is used in EIFFEL [11] but has the major drawback to modify the vocabulary according to the classes.

**Unification** Dynamic languages as CLOS, JAVA via its interfaces and C++ for the functions consider that if two global properties are homonym[3] then they are not distinct. There is thus no possible conflict of name and ambiguities of multiple inheritance are deferred on value inheritance. The major drawback of this solution is that it expresses a bad class model : if two class introduce and share a common generic property, it means there is the lack of a common super-class that has to introduce this property.

**Value conflicts.** A *property value conflict* occurs when a class inherits two local properties from the same global property, none more specific than the other one. Fig. 2.2 shows two classes (`Rectangle` and `Rhombus`), both redefining the method `surface` whose global property was introduced into the class `Quadrilateral`. In the common subclass `Square`, which one is most specific? Contrary to the name conflict, there is no intrinsic solution with this problem (like a massive renaming in the name inheritance) and the programmer or the language must bring an additional semantics to resolve this problem.

**Forbidden** The specification of the language proscribes value conflicts. The programmer must redefine the global property by a new local property in the class where the conflict appears. In the redefinition, an explicit designation, as in C++, may be used to choose among the local properties in conflict.

[2] For methods, the operator :: corresponds to a static call: it is a local property which is denoted, not the global one.

[3] For C++ and JAVA, it is necessary to integrate into the name the number and the type of the parameters.

**Combining** For certain values or particular properties, the conflict resolution must be done by combination of the values: it is the case, for example, for the type in the event of covariant redefinition (the lower limit of types in conflict, if it exists) or for properties whose values cumulate. Combining is also found for EIFFEL contracts with disjunction of pre-conditions and conjunction of post-conditions.

**Choice** The programmer or the language arbitrarily choose the local property to inherit. In many dynamic languages like CLOS or DYLAN, the choice may be done by a linearisation [12]; in EIFFEL, the programmer can select the desired property using the inheritance clause `undefine`.

## 4 Classes and modules

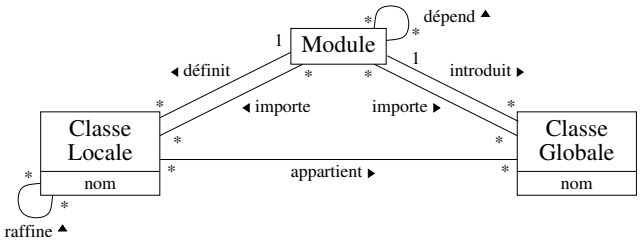



**Fig. 3.** Class and module metamodel

We now more formally present modules and refinement, by a strong analogy with the object metamodel presented above. Problems involved in inheritance conflicts multiple refinement will be dealt with in the following section.

### 4.1 Modules metamodel

A more rigorous approach is based on the fact that classes are with modules what properties are with classes, the relation of dependence between modules being similar to specialisation, and importation to inheritance. It is thus necessary to define two entities associated with the concept of class (figure **??**).

Local *classes* (similar to the local properties) are defined in modules. A local class is described by a name, names of super-classes and definitions of local properties. The global *classes* (similar to global properties) are orthogonal with the modules. Each module has global classes which correspond to the classes that it is statically able to handle. The global classes gather local classes. The correspondence between these two entities is done by their name. On the whole, the local classes are defined in a module exactly as in an ordinary object-oriented language, class names indicating as well local class as global classes.

### 4.2 Definitions and notations

A *module* is a set of local definitions of *classes*, which *import* (i.e. *depend of*) zero, one or more modules. We note $m$ a module and $<$ the module dependence relation: $m < n$ means that $m$ depends on $n$ ($m$ is a *submodule* of $n$, which is a *supermodule* of $m$). The dependence relation is a strict partial order. A *global class* is a set of local classes of the same name defined in different modules. In a way similar to the property name inheritance, a module imports all the global classes of its supermodules. When a local class is defined in a module $m$, two cases arise: If a *global class* of the same name is imported from supermodules, it is then a *refinement*. If not, a new global class of this name is *introduced* in the module $m$. In both cases, the new local class is added to the global class. We note $A_m$ the local class of $m$ owned by the global class $A$ and we abusively use the membership ensemblist notation between all these entities: $A_m \in m$, $A_m \in A$, as well as $A \in m$ if $A$ is imported or introduced into $m$. For any global class $A$ of a module $m$, the existence of the corresponding local class $A_m$ is supposed: it is either an explicit definition, or an implicit refinement, $A_m$ being then called *implicit class* since its description is empty. In corollary, to explicitly refine a class by a empty local class is equivalent to not to refine it explicitly. When two modules are in relation of dependence, local classes from their common global classes are in relation of refinement, also noted $<$:

$$A_m < A_n \Leftrightarrow_{\text{def}} m < N \text{ and } A \in N \tag{1}$$

In a module $m$, the definition of a local class $C_m$ can explicitly make it a subclass of a class $D$: we note this relation of *explicit specialisation* $C_m \prec_m D_m$. It is licit if the module $m$ introduce or import the global class $D$. The relation of *specialisation* between local classes is a strict partial order noted $\prec$ and is built by the importation of explicit specialisations of super-modules. It is defined by the transitive closure of the following relation $\prec'$:

$$C_m \prec' D_m \Leftrightarrow_{\text{def}} \exists N \text{ such as } m \leq n \text{ and } C_n \prec_n D_n \tag{2}$$

Remarks that a explicit relation of specialisation between local classes in relation of specialisation by importation or by transitivity is without effect.

Lastly, a *programme* is set of modules, closed by the relation of importation: it corresponds to a module (possibly virtual) that import every module of the program. We will call *local classes of a program* the union of the local classes of all its modules and *quasi-specialisation* the transitive closure of the union of the relation of refinement $<$ and relation of specialisation $\prec$ on the local classes of the program. In the figures, we adopt the following convention: the local classes appear as of small named box, inside large box possibly numbered, the modules. Only specialisation relations in a module ($\prec$) and of importation between modules ($<$) are drawn. Local classes of the same globlal class bear the same name: the relation of refinement between classes ($<$) thus remains implicit. Moreover, classes and specialisations that are implicit are illustrated in dotted lines.

Intuitively, the program corresponding to the module $m$ behaves like the hierarchy of its local classes, provided with quasi-specialization as relation of specialisation, and in the code of which any name of class $A$, used as type or for instantiation, would be interpreted as the local class $A_m$. All the other local classes of the imported modules behave like abstract classes. In an alternative way, the program can be seen like the hierarchy of the local classes of the module $m$, ordered by the relation $\prec$, the contents of each class resulting from its successive refinements from the imported modules.

### 4.3 Class unification

In a local class definition, the programmer may choose to explicitly unify this class with other classes. For example via a syntactic writing similar to the declaration of specialisation.

Class unification consists in binding the global classes: in the module and all the submodules, the two global classes will share a single local class. We note $\widehat{ab}_m = A_m = B_m$ the local class associated the union in the module $m$ of the global classes $A$ and $B$. As the local class $\widehat{ab}_m$ belongs at the same time to $A$ and to $B$, it refines classes $A_n$ and $B_n$ for any super-module $n$. Moreover as the relation of specialisation is antisymmetric, the equation (2), becomes:

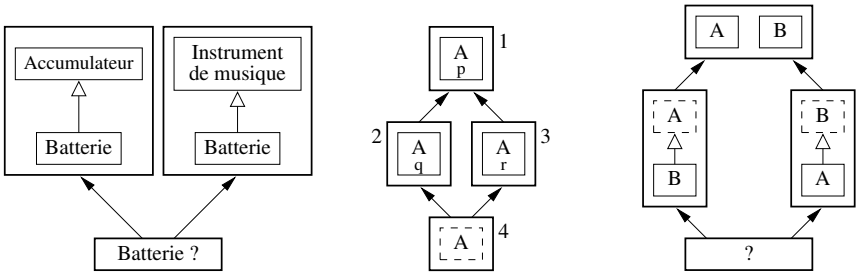
$$C_m \prec' D_m \Leftrightarrow_{\text{def}} C_m \neq D_m \text{ and } \exists N \text{ such as } m \leq N \text{ and } C_n \prec_n D_n \qquad (3)$$

The naming question of such local classes can be solved in two manners: either by considering that the names of the global classes are alias and can independently be used to indicate the local class, or by choosing a single name to identify it.

### 4.4 Constructors

In statically typed languages without refinement, instance constructors are not subjected to polymorphism: the dynamic type of instances that will be created is already determined by the static type used. Thus on the level of the languages, the particular methods which have a role of instance constructor are not inherited (Java or C++). In Eiffel, they are inherited but their instance constructor role is not.

With refinement, the things are a little different. Admittedly, the dynamic type of instances that will be created is statically foreseeable but it remains overall unknown, the local classes that the module statically handles being able to be refined in possible submodules. Thus, on a class refinement, on the one hand constructors must be fully inheritable and on the other hand, the refining classes must make sure that the constructors introduced into the refined classes remain coherent (by redefining them if necessary).




4.1 Conflit de nom de classe 4.2 Importation de valeur 4.3 Conflit de spcialisation

**Fig. 4.** Importation multiple et conflits.

## 5 Multiple importation and inheritance

It results from the definitions of the previous section that the refinement of a subclass (within the meaning of specialisation) automatically induce multiple inheritance (within the meaning of quasi-specialisation). It is thus impossible to make the economy of the problems arising from multiple inheritance. Moreover, the relation of dependence between modules can itself be multiple. The treatment of the conflicts will be more unusual, while remaining essentially in the way of section 3.

### 5.1 Multiple importation

A *class name conflict* occurs when a module imports two global classes of the same name from two different super-modules (Fig. 4.1). It is analogue with the property name conflict and can be solved by the same manner: the modules being generally spaces of names, explicit designation is the most natural solution here. However, a class renaming mechanism, as is EIFFEL and its configuration language LACE, could also solve the problem.

Fig. 4.2 illustrates the configuration similar to the conflict of value: module 4 imports $A$ introduced by module 1 and refines implicitly $A_2$ and $A_3$ by a local class $A_4$. In this case, the conflict of value will be solved by combination of the local classes in conflict, by inheritance of their properties through the specialisation and the refinement relations.

A new configuration conflict, the *conflict of specialisation*, appears when, in a module, two local classes are mutually a specialisation of the other (Fig. 4.3). It causes a circuit in the relation of specialisation which is not any more one partial order. This conflict may be solved by unifying the classes of the circuit.

### 5.2 Name property inheritance

Name property differs only a little with the introduction of refinement: Local classes have global properties of their super-classes within the meaning of quasi-specialisation. In Fig. 4.2, $A_2$ inherits the global property `p` introduced into $A_1$

and $A_4$ inherits the global properties `p`, `q` and `r` introduced into the classes which it refines.

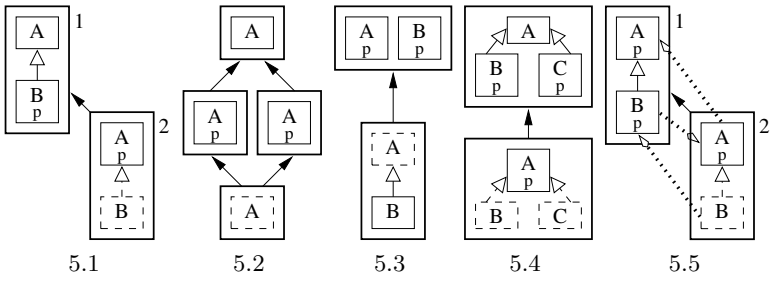



**Fig. 5.** Property and inheritance.

However, as the programmer knows the imported modules, some apparent name conflicts can be solved by the identity of the global properties. Addition of local properties to global properties does not have then any more reason to be limited to the redefinition in subclasses. The *generalisation of global property* makes it possible to extend the possession of global properties of the classes to the super-classes within the meaning of specialisation. The generalisation of a global property owned by a local class $C_m$ in a module $n$, submodule of $m$ consists in defining in $n$ a homonym local property of the global property in a class $D_n$ such as $C_n \prec D_n$. The example of the figure 5.1 shows the definition of a property `p` in a local class $B_1$ and its generalisation in the class $A_2$ (since $B_2 \prec A_2$).

Property name conflicts related on specialisation or refinement are still possible but must be individually solved by the techniques of the section 3.2.1. In addition to the name conflict related to multiple specialisation (Fig. 2.1), Fig. 5.2 shows a conflict related to multiple refinement and Fig. 5.3 and 5.4 show conflicts implying refinement and specialisation.

### 5.3 Value property inheritance

Either the global property of a local class is redefined, or the local property defined in the most specific super-class is inherited. However, it is necessary to combine the two local class orders which are specialisation and refinement to produce a single strict partial order of *specificity* used for determining the inherited local properties. This relation of specificity slightly differs from that of quasi-specialisation.

The relation of specificity for the module $s$ between the local classes $C_m$ and $D_n$ (with $s < m$ and $s < n$) is noted $C_m \ll_s D_n$ and is defined by:

$$C_m \ll_s D_n \Leftrightarrow_{\text{def}} C_s \prec D_s \text{ or } (C_s = D_s \text{ and } m < N) \tag{4}$$

the first part expresses that, in the module, the relation of specialisation between two local classes is reflected on any couple of classes of the same global classes. The second part of the definition expresses that a refined class is less specific than its refinement.

The example of Fig. 5.5, where the relation of specificity is represented by hatched arrows with white head, asks the question of the value inheritance of the global property `p` in the local class $B_2$. The equation (**??**) application says that $B_2 \ll_2 B_1 \ll_2 A_2 \ll_2 A_1$, so the local property `p` inherited has to be that one defined in $B_1$.

The relation of specificity contains the relation of quasi-specialisation. Indeed, the latter is insufficient : by considering the case of Fig. 5.5, the classes $B_1$ and $A_2$ would be incomparable. However the intuitive vision which considers refinement as an incremental modification of classes would give to $B_2$ the method $p$ defined in $B_1$ as it is the case with the relation of specificity $\ll$. Informally, that means that the relation of refinement is *stronger* that that of specialisation. However, during class unification, the relations of specialisation which could exist in possible super-modules do not affect the relation of specificity, only the relation of refinement keeps a utility in specificity.

With the redefinition or the inheritance of a local property $l$ in a class $C_m$, it is necessary to check that this one not only conforms to the local properties defined in the super-classes of $C_m$ (within the meaning of quasi-specialisation) but also that the local properties defined or inherited in the subclasses $C_m$ (within the meaning of specialisation) conforms to $l$. In the example of Fig. 5.1, it should be checked that the local property `p` inherited in $B_2$ (i.e. that defined in $B_1$) conforms with that defined in $A_2$. Conformity between local properties can take several forms according to the languages (arity, type of result, types of the parameters, declared exceptions, contracts, etc.) Let us note that, within a framework of sure typing, the rule of contravariance applies to refinement. Moreover, like the semantics of refinement is not a semantics of specialisation, a policy of refinement covariant [13] does not apply: on the contrary, contravariance could find its utility here.

## 6 Implementation

Implementing programs that use refinement is almost similar to the implementation of any object oriented program.

### 6.1 Overcost

On the first hand, global techniques *la* SmartEiffel [14] may be used. On the other hand, modules are good candidates for compilation units in a separate compilation scheme. However, refinement is not easily compatible with the dynamic loading, i.e. the refinement of classes during the execution of a program. Indeed, the difficulty of the implementation of the multiple inheritance in dynamic loading is exacerbated here: although it is possible to modify during

the execution the values of the pointers in the tables of methods (redefinition of methods), it is much more expensive to radically change these tables to reflect the addition of methods or specialisation relations. The extreme case can be the need for modifying the instances to reflect the addition of attributes, which imposes implementation techniques comparable with instance migration.

Because we do not have dynamic loading, the program (machine code and metamodel) is entirely known (at compile-time for global compilation and at link-time for separate compilation) so module importation and class refinement can be statically computed before execution. At run-time, various local classes of a global class have disappeared: only remain the local classes of the virtual module that is the program. The fact of having classes without modules and refinement makes it possible to assume that the implementation can be carried out without space or temporal overcost with the execution compared to the same language without refinement.

During the construction of a program, conflicts related to multiple importation of modules can appear. If necessary, those can be avoided by building a submodule dedicated to the resolution of conflicts. However, compiler has a total knowledge of the program. Therefore, only conflicts that having a real impact on the program, have to be signalled. For example, name conflicts (of classes or properties) which can appear in the virtual module do not have an influence on the behaviour of the program, unless the language allows introspection.

### 6.2 Separate compilation

The separately compiled schema we developed in [15] is used for our prototype. Here is a resume of its characteristics.

It benefits of two well known global techniques:

**coloration** [16–19] focuses specific data structure needed by object oriented specific behaviour (message sending, casting, attribute access). It is a global technique that allow implementation of object oriented languages to go beyond multiple inheritance and to bring back to single inheritance implementation.

**type analysis** allows to determine *dead code* that can be safely removed and optimises polymorphism related behaviours. It computes three sets : instantiated classes, dynamic types of expressions and local properties that may be executed. Theses sets are mutually dependant (executed properties depend on dynamic types of receivers, dynamic types depend on instantiated classes and instantiated classes depend on executed properties). This circularity explain the problem difficulty ([20]) and the diversity of solutions (many of them are described in [21]).

In first phase, modules has to be compiled to produce executable code and meta-data. As with usual separate compilation scheme [22], executable code is incomplete and some symbols are used instead of missing information. Some missing information are related to object oriented mechanism : there are symbols for method and attribute indexes. Meta-data contains the meta-model of the

module and some internal information as internal behaviour of each local property: instantiations, called methods, attribute access, etc. A module compilation does not depend on other modules except that the meta-model of super-modules is needed, so this phase may be recursively done.

In the second phase, compiled modules that compose the program are gathered, the global metamodel is built, type analysis is performed using internal metadata and dead part of the metamodel is forgotten, coloration is performed on the living part of the metamodel and incomplete information of the executable code is filled with the coloration results then linked to produce a complete executable.

## 7 Related works

MultiJava [23] proposes units of compilation, similar to the modules presented here, which are provided with a relation of dependence via the keyword `import`. It makes it possible to extend existing classes by adding functions by an ad hoc syntax. On the other hand the redefinition of methods, the addition of attributes or the declaration of implementation of interfaces Java are not allowed. Nevertheless, MultiJava is compatible with separate compilation and the dynamic loading. It also proposes an implementation of the multi-methods.

MixJuice [24], based on language Java, proposes modules in relation of dependence and amendments of classes authorising to it (re)definition of method, the addition of attributes and the additional declaration of implementation of interfaces Java. In the event of multiple dependencies between modules, the property name conflicts are solved by explicit designation; the value conflicts are solved by a linearization  la Clos. The approach is compatible with separate compilation but does not allow the dynamic loading.

*Classboxes* [25] make it possible to extend the classes in Smalltalk by additions or redefinitions of methods and attributes while controlling the visibility of the additions since these amendments have only local impacts, the answers to message sending are determined both by the receiver and the *classbox*. Thus, contrary to the contribution of this article, amendments of classes brought by a *classbox* are applied only to this *classbox* and to *classboxes* which import it, the message sending of messages coming from others *classboxes* will thus not be affected by the modification. *MixJuice*  does not reveal the analogy between classes and modules nor does analyse the various conflict configurations and their resolutions. It focus to be a functional implementation of the refinement mechanism (within the limits imposed by Java of invariance of the types and simple inheritance), which shows the feasibility of such approach.

*Hight order hierarchies* of [26] are quire similar with our modules. The motivations are identical and the isomorphism of the metamodels of classes (order 1) and modules (order 2) would allow an immediate any order generalisation. It is however not our objective and we do not push the class metaphor so far: finally, like says it [3], our modules are not classes, at least in the sense that they do not have instance. Module dimension is the program: it can have a single in-

stance, corresponding to the static data of an execution. But if the metaphor is continued, a order 3 class would be an operating system, and the proposal of the hierarchies of a higher nature becomes a little mysterious. Lastly, technically, the two approaches differ appreciably on the multiple heritage management, that is inevitable as we saw. [**?**] proposes a combination of completely ordered *mixins* whereas our proposal rests on the interpretation of the multiple heritage in the meta-model, which seems to us, since [**?**], the only good way of approaching the multiple inheritance.

## 8 Conclusion and perspectives

We proposed in this article an extension of the class model of statically typed languages in multiple inheritance which adds a concept of module and a mechanism of refinement between classes which makes it possible in a module to amend the imported classes of the modules on which it depends. The syntactic overload is weak since it requires only a rudimentary module language (to express that a module depends on another module and that a class (re)definition belongs to a module).

It allows a better structuring of the programs (functional or temporal) thanks to the modules and authorises more flexibility (facilitating amongst other things the evolution of the software) thanks to refinement, a very particular point being put on a coherent management of the conflicts aiming at respecting the principle of specialisation between classes. Moreover, it is fully compatible with separate compilation and without temporal or space overload for programs without dynamic loading of classes or modules.

Although the modules and the classes are of basically different utility [3], our proposal is based on an strict structure analogy between these two concepts since they are described by similar meta-models[4].

In order to try out the use of refinement on the development of software, a prototype of programming language and its compiler are being developed [15]. The objective is to test the refinement implementation in separate compilation.

However, the principal limitation comes from what it is not possible *a posteriori* to reconsider choices already carried out such as for example the suppression of properties and relation of specialisation or the unification of global properties. If the suppression is not clearly useful, the unification of properties would be a response to certain conflicts, with the proviso of managing to specify it properly.

## References

1. Meyer, B.: Object-Oriented Software Construction. Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor. Prentice Hall International, Hemel Hempstead, UK (1988)

[4] "Importation is inheritance, why we need both." to paraphrase [3].

2. Andersen, E.P., Reenskaug, T.: System design by composing structures of interacting objects. In Madsen, O.L., ed.: Proc. ECOOP'92. LNCS 615, Springer-Verlag (1992) 133–152
3. Szyperski, C.A.: Import is not inheritance — why we need both: Modules and classes. In Madsen, O.L., ed.: Proc. ECOOP'92. LNCS 615, Springer-Verlag (1992) 19–32
4. Steele, G.: Common Lisp: The Language, Second Edition. Digital Press, Bedford (MA), USA (1990)
5. Goldberg, A., Robson, D.: Smalltalk-80, the Language and its Implementation. Addison-Wesley, Reading (MA), USA (1983)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: Proc. ECOOP'2001. LNCS 2072, Springer-Verlag (2001) 327–355
7. Kiczales, G., des Rivieres, J., Bobrow, D.: The Art of the Meta-Object Protocol. MIT Press, Cambridge (MA), USA (1991)
8. Pavillet, G., Ducournau, R.: Implmentation des attributs boolens par un Meta Object Protocol. In Malenfant, J., Rousseau, R., eds.: Actes LMO'99, Herms (1999) 55–68
9. Ducournau, R., Habib, M., Huchard, M., Mugnier, M.L., Napoli, A.: Le point sur l'héritage multiple. Technique et Science Informatiques **14** (1995) 309–345
10. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, Reading (MA), USA (1986)
11. Meyer, B.: Eiffel: The Language. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK (1992)
12. Inconnu: Trouver des rfrences (0)
13. Ducournau, R.: Spcialisation et sous-typage : thme et variations. Revue des Sciences et Technologies de l'Information, TSI **21** (2002) 1305–1342
14. Colnet, D., Zendra, O.: Optimizations of eiffel programs: Smalleiffel, the gnu eiffel compiler. In: 29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99). Volume 10., IEEE Computer Society (1999) 341–350
15. Privat, J., Ducournau, R.: Intgration d'optimisations globales en compilation spare des langages objets. In Carré, B., Euzenat, J., eds.: Actes LMO'04 *in* L'Objet vol. 10, Herms (2004) 61–74
16. Dixon, R., McKee, T., Schweitzer, P., Vaughan, M.: A fast method dispatcher for compiled languages with multiple inheritance. In: Proc. OOPSLA'89, ACM Press (1989)
17. Pugh, W., Weddell, G.: Two-directional record layout for multiple inheritance. In: Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90). ACM SIGPLAN Notices, 25(6) (1990) 85–91
18. Cohen, N.: Type-extension type tests can be performed in constant time. Programming languages and systems **13** (1991) 626–629
19. Vitek, J., Horspool, R., Krall, A.: Efficient type inclusion tests. In: Proc. OOPSLA'97. SIGPLAN Notices, 32(10), ACM Press (1997) 142–157
20. Gil, J., Itai, A.: The complexity of type analysis of object oriented programs. In: Proc. ECOOP'98. LNCS 1445, Springer-Verlag (1998) 601–634
21. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. **23** (2001) 685–746
22. Levine, J.R.: Linkers and Loaders. Morgan-Kauffman (1999)
23. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: Modular open classes and symetric multiple dispatch for Java. In: Proc. OOPSLA'00. SIGPLAN Notices, 35(10), ACM Press (2000) 130–145

24. Ichisugi, Y., Tanaka, A.: Difference-based modules: A class-independant module mechanism. In: Proc. ECOOP'2002. LNCS 2374, Springer-Verlag (2002) 62–88
25. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A minimal module model supporting local rebinding. In: JMLC 2003 (Joint Modular Languages Conference). Volume 2789. (2003) 122–131
26. Ernst, E.: Higher-order hierarchies. In Cardelli, L., ed.: Proc. ECOOP'2003. LNCS 2743, Springer-Verlag (2003) 303–329