Class Refinement and Modules Hierarchies: A MetaModeling Approach for Statically Typed Language in Multiple Inheritance

Jean Privat, Roland Ducournau

LIRMM

Laboratoire d'Informatique, de Robotique, de Microlectronique de Montpellier

Unité Mixte de Recherche 5506
CNRS – Université de Montpellier II

Class Refinement and Modules Hierarchies: A MetaModeling Approach for Statically Typed Language in Multiple Inheritance

Jean Privat and Roland Ducournau

Rapport/Report 06002

January 6, 2006

Abstract

Many extensions of the object-oriented model were proposed to provide a better separation of crosscutting concerns of object-oriented programs. However these approaches are mainly developed for dynamically typed languages or for languages in simple inheritance.

This paper addresses the same need of separation of crosscutting concerns. We propose a simple extension of the object-oriented model of statically typed language in multiple inheritance. Our proposition involves two coupled notions of *class refinement* and *module hierarchy*. And is based on a formal analogy between the metamodel of modules and classes and the metamodel of classes and properties: (i) modules contain a coherent set of class definitions and can refine classes defined in required modules, (ii) conflicts raised with multiple module dependence, multiple class refinement, and multiple class inheritance are treated in the same sound way.

1 Introduction

Object-oriented programming languages give to programmers a powerful development framework to build both stable and coherent entities (classes) while allowing reusability and maintainability thanks to specialization and inheritance mechanism [20].

However, several studies show that classes pose some reuse problems [28] especially because a single concern (ie. a feature of a program) can be dispatched over several classes [17]. Thus, the observed orthogonality between classes and concerns [1] yields a need for explicit structures at the concern level. Methodologies like *Feature Object Programming* [2] and *Multi-Dimensional Separation of Concerns* [24] consider that improving reusability can be achieved by the separation of crosscutting concerns. Such a separation has the following merits:

- programs can be easily extended as new concerns can be added to existing programs,
- units of reuse are no longer classes, but are orthogonal to classes—thus new concerns can crosscut classes,
- programmers can combine existing concerns and build their own application with very few (or without) glue code.

Many approaches have been developed to achieve this separation. In *Aspect Oriented Programming* [17] systems, like HYPER/J [24] or ASPECTJ [16], the programmer can define new concerns (called *aspects*) and *weave* them to existing classes. *Mixins* [5] are defined as parts of classes. They are combined to define the classes of a program. *Mixins layers* [26] are sets of related mixins. They are an improvement over simple mixins since they make the composition of related concerns easier. The last approach is based on two coupled notions of *class refinement* (a class can be extended while retaining its name) and *module hierarchy* (a module can refine classes imported from its 'required' modules). One can see this approach either as an incremental class definition, or as inheritance between class hierarchies. This idea can be found in several works : *Virtual Classes* [18], *Open Classes* [7], *Difference-Based Modules* [14], *Classboxes* [4] and *Higher-order hierarchies* [13]. However, these works have some limitations since they are based on single inheritance class hierarchies. Therefore, multiple refinement and combination of refinement and inheritance are not always fully and correctly handled.

Our proposition belongs to the class refinement family but integrate the specificity of multiple inheritance and statically typed languages. Hence, the main contributions of the present paper is a class refinement and module hierarchy approach that permits an incremental class definitions for statically typed languages in multiple inheritance. Our proposition is formalized with a module and class metamodel, isomorphic to the class and property metamodel. It extends results on multiple inheritance for two purposes: multiple dependence between modules is analogous to multiple class inheritance, and multiple class refinement is handled like multiple class specialization.

The present paper is organized as follows. Section 2 exposes the class refinement and module hierarchy principle. Section 3 outlines a simple class and property metamodel for statically typed

object-oriented languages which gives a sound basis to multiple inheritance. Section 4 is about the module and class metamodel. Section 5 discuss about related class refinement approaches. Finally, we conclude in Sect. 6 and discuss our perspective.

2 Class Refinement Approach

We first present the two dual notions of class refinement and modules, then we illustrate them with an example.

2.1 Class Refinement and Modules

Intuitively, one can present the refinement of a class c_1 by a class c_2 as an incremental class definition in which the properties (attributes and methods) defined in c_2 are added or take the place of those of c_1. In contrast with specialization, once c_1 has been refined by c_2, the latter takes the place of the former in all its occurrences in the whole program. Such a mechanism is rather common in languages with dynamic typing. In many object-oriented extensions of Lisp, like Flavors and Clos, this is reserved to methods as they are defined outside classes. In Yafool [9], incremental class definition was the standard use. Without loss of generality, it should be possible with all languages supporting a *metaobject protocol* [15] like Clos, even if experiments show that these protocols are not always really adapted to dynamic class modifications.

Class refinement is coupled with a dual notion of *modules*, in its simplest and quite traditional form.

- A *module* is a class hierarchy, i.e. a set of classes ordered by specialization.
- A module is an unit of reuse [28], which can be separately compiled then linked to produce a final executable. On the contrary, classes are no longer units of reuse and there is no class nesting.
- A module *depends on* a set of other modules (called *required modules* or *supermodules*): the dependence relation is acyclic, like class specialization.
- A module *imports* classes from its supermodules and can *refine* them. The class refinement order is then deduced from the dependence order between the modules.
- Finally, we do not associate with modules a notion of visibility (or export), or even namespaces, because we do not need it for the present paper.

Regarding class refinement, we identify five atomic mechanisms:

- *adding a property*, i.e. a method or an attribute (aka instance variable);
- *redefining a property*—this is more common for methods than for attributes;
- *adding a superclass* (multiple inheritance);
- *generalizing a property*, i.e. defining a property in the superclasses of the class which introduced it in the supermodules;
- *unifying classes*, i.e. asserting that two classes defined in different modules are the same, by merging their definitions.

However, this is not obvious to extend the intuitive class refinement mechanism to multiple refinement of one class or to combination of refinement and specialization, since their ordering does matter. While dynamic languages can follow the run-time chronological ordering, this is not possible when the effects of class refinement are statically computed at compile-time.

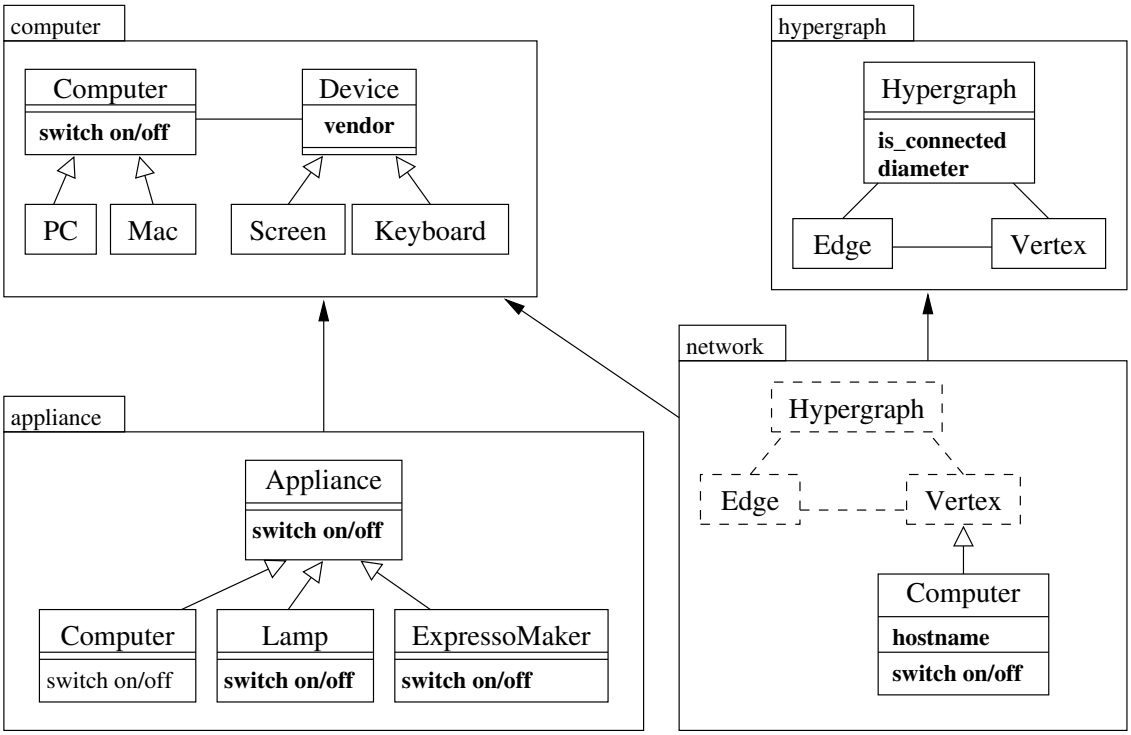


Figure 1: Concrete Example with Module and Class Refinement

```
class Computer
   def switch_on_off
       do ... end
   ...
end
```

```
class PC
   inherit Computer
   ...
end

...
```

Figure 2: Computer Module Implementation

2.2 Example

This section presents an example of four modules (see Fig. 1) to illustrate a concrete application of the class refinement mechanism.

This example is written in PRM, a small statically typed object-oriented language currently under development[1] [25]. The syntax has a simple straightforward style: class definitions start with the `class` keyword and property definitions (attributes and methods) start with the `def` keyword. Class names start with an upper case letter, method names start with a lower case letter and attribute name start with the `@` character ('at'-tribute). Declarations follow the Pascal style (`identifier: Type`). As modules are reuse units, each PRM source file is a module.

2.2.1 Computer Module.

The first considered module models how a computer works (Fig. 2). It contains a `Computer` class, subclasses for different computer types and other classes for related computer things. The `switch on/off` method of the `Computer` class is used to power on and off computers.

[1]The PRM homepage: `http://www.lirmm.fr/$\tilde{~}$privat/prm`.

```
require computer

class Appliance
   def switch_on_off
       do ... end
   ...
end


class Computer
   inherit Appliance
end
```

```
class Lamp
   inherit Appliance
   def switch_on_off
       do ... end
   ...
end

class ExpressoMaker
   inherit Appliance
   def switch_on_off
       do ... end
   ...
end
...
```

Figure 3: Appliance Module Implementation

```
class Hypergraph
   def @nodes: Array[Node]
   def @edges: Array[Edge]
   def is_connected: Boolean
       do ... end
   def diameter: Integer
       do ... end
   ...
end
```

```
class Node
   def @edges: Array[Edge]
   ...
end

class Edge
   def @nodes: Array[Node]
   ...
end
```

Figure 4: Hypergraph Module Implementation

2.2.2 Appliance Module.

The programmer wants to generalize computer behavior with other appliances like lamps or expresso makers. For some reason, the computer module cannot be changed. A module that requires it has to be built.

A new abstract class, `Appliance`, is created as an ancestor of `Computer` class (*adding a superclass*), and is specialized by two classes: `Lamp` and `ExpressoMaker`. Appliances use electricity, and can be powered on and off: the `switch on/off` method defined into the `Computer` class must be generalized into the `Appliance` class (*property generalization*), and then specialized into both `Lamp` and `ExpressoMaker` subclasses.

The `require` keyword at the first line of the implementation (Fig. 3) declares that the `computer` module is a supermodule. Hence, classes from the `computer` module are imported and the `Computer` class is refined.

2.2.3 Hypergraph Module.

This module models finite hypergraphs i.e. a generalization of graphs in which edges contain any number of vertices (Fig. 4). Methods of the `Hypergraph` class implement two algorithms: `is_connected` returning `true` iff there exists a path between any couple of vertices and `diameter` computing the maximal distance between two vertices of the graph.

2.2.4 Network Module.

Both computer and hypergraph modules are independent and might be developed for different programs by different programmers. Class refinement allows programmers to easily create a derived

```
require hypergraph
require computer
```

```
class Computer
   inherit Vertex
   def @hostname: String
   def switch_on_off
       do ... end
   ...
end
```

Figure 5: Network Module Implementation

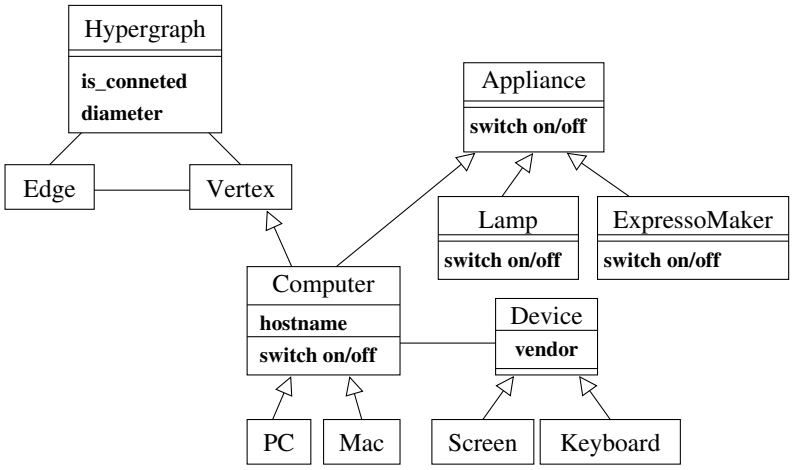


Figure 6: Final Class Model of the Program

module and to model networks (Fig. 5).

On a network, every computer has an hostname. Hence, a new attribute, `hostname`, is added to the `Computer` class (*property addition*). Moreover, complex networks, for instance the Internet, can be represented by hypergraphs with computers as vertices. For this reason, a specialization relation is added between `Vertex` class and `Computer` class (*adding a superclass*). When a computer is powered on, it becomes reachable on the network and unreachable when it is powered off. Hence, in order to characterize the computer state on the network, the `switch on/off` method is redefined (*method redefinition*).

All of these modifications made on the `Computer` class are naturally inherited by PCs and Macs. Hardware failures implemented in the `PC` class in the computer module, induce computer shutdown by a `switch on/off` message sending. Therefore, they make the PC unreachable on the network.

2.2.5 Final Program.

For the programs which merge these four modules, each class is imported (Fig. 6) and the final `Computer` class is made of the combination of the `Computer` class declarations from the above modules as follows: `Computer` is a superclass of `PC` and `Mac` (from the `computer` module), it is also a subclass of `Node` (from the `network` module) and of `Appliance` (from the `appliance` module). It has a `switch on/off` method (the one declared in the `network` module overrides the one from the `class` module) and a `hostname` attribute (from the `network` module).

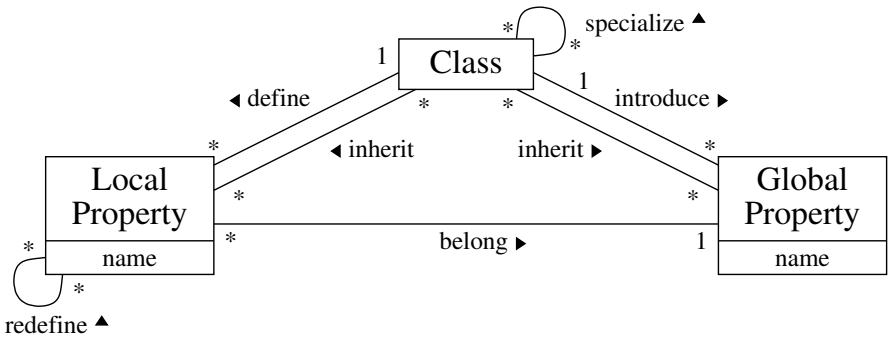


Figure 7: Metamodel of Classes and Properties

3 Class and Property Metamodel

In this section, we present a metamodel made of three main kinds of entities: *classes*, *local properties* and *global properties* (Fig. 7). It is intended to be both intuitive and universal. It is likely to be very close to the intuition of most programmers, when they think of object-oriented concepts. It is universal in the sense that it is not dedicated to a specific language and it is very close to the specifications of most statically typed object-oriented languages, at least when they are used in a simple way. For instance, it is an implicit metamodel of JAVA and EIFFEL[2], but it has never been explicitly described in any programming language nor even in UML [23].

In the following, we present successively an informal idea of the metamodel, followed by a more formal, set-theoretical description, then an analysis of multiple inheritance conflicts, and a comparison with existing languages.

3.1 Entities, Relations and Inheritance

Metamodeling some part of an object-oriented programming language consists in defining an object model—i.e. entities like classes, associations, attributes, methods, etc.—for modeling a subset of the considered concepts of the language. We add to this basic specification the following requirement: any occurrence in the modeled program of an identifier denoting a modeled entity must unambiguously maps to an unique instance of the metamodel.

Therefore, when one wants to metamodel properties, late binding (aka message sending) implies to define two categories of entities. *Local properties* correspond to the attributes and methods as they are defined in a *class*, independently of possible other definitions of the 'same property' in superclasses or subclasses. *Global properties* are intended to model this idea of the 'same property' in different classes. They correspond to messages to which the instances of a class can answer—in the case of methods, the answer is the invocation of the local property of the dynamic type of the receiver. Each local property *belongs* to a single global property and is *defined* in a single class. Global and local properties should be in turn specialized into attributes (aka instance variable) and methods. However, there is no need to detail this here. Note that, in the following, 'property' stands for both, though the complication comes mainly from methods—indeed, attributes are usually quite simpler.

A class definition is a triplet constituted with the name of the class, the name of its superclasses, presumably already defined, and a set of local property definitions. The specialization relation supports an inheritance mechanism—i.e. classes inherit the properties of their superclasses. The two kinds of properties yield two-level inheritance. First of all, the new class *inherits* all global properties of its superclasses—this is *global property inheritance*. Then, each local property defini-

[2]In the EIFFEL case, with a limited use of renaming. Indeed, extreme renaming makes the specification ambiguous.

```
class A {
   public void foo() { ... }
}
```

```
class B extends A {
   public void foo() { ... }
   public void bar() { ... }
}
```

Figure 8: A simple JAVA Example

tion is processed. If the name[3] of the local property is the same as the name of an inherited global property, the new local property is attached to the global property. If there is no such inherited global property, a new global property with the same name is *introduced* in the class.

Local property inheritance takes place at run time—though most implementations statically precompile it. A call site `x.foo(args)` represents the invocation of the *global property* named `foo` of the *static type*[4] of the receiver `x`. At run time, this call site is interpreted as the invocation of the *local property* corresponding to both the *global property* named `foo`, and the *dynamic type* (i.e. the class) of the value of `x`. Therefore, when no local property named `foo` is defined in the considered class, a local property of the same global property must be inherited from superclasses.

The JAVA listing example in Fig. 8 defines seven entities of our metamodel—two classes, `A` and `B`; three local properties, the method `foo` defined in `A` and the methods `foo` and `bar` defined in `B`; two global properties, respectively introduced as `foo` in `A` and as `bar` in `B`.

3.2 Notations and definitions

Notations. Let E and F be sets. Given a function foo : $E \to F$, the definition of foo is extended to the parts of E in the usual way—i.e. $\forall F \subseteq E$, $\text{foo}(F) = \{\text{foo}(x) \mid x \in F\}$. $|E|$ is the cardinal of E, and $E \uplus F$ is the union of the disjoint sets E and F.

Definition 1 *A model of a hierarchy, i.e. an instance of the metamodel, is a tuple* $\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, \text{name}_{\mathcal{H}}, \text{glob}_{\mathcal{H}}, \text{def}_{\mathcal{H}} \rangle$ *where:*

- $X^{\mathcal{H}}$ *is the set of* classes*.*
- $\prec^{\mathcal{H}}$ *is the* class specialization *relationship, which is reflexive, antisymmetric and antireflexive, i.e.* $(X^{\mathcal{H}}, \prec^{\mathcal{H}})$ *is a strict partial order.* $\preceq^{\mathcal{H}}$ *(resp.* $\prec_d^{\mathcal{H}}$*) denotes the reflexive closure (resp. the transitive reduction) of* $\prec^{\mathcal{H}}$*.*
- $G^{\mathcal{H}}$ *and* $L^{\mathcal{H}}$ *are the sets of* global *and* local properties*.*
- $N^{\mathcal{H}}$ *is the set of* property identifiers *(names).*
- $\text{name}_{\mathcal{H}} : G^{\mathcal{H}} \uplus L^{\mathcal{H}} \to N^{\mathcal{H}}$ *is the naming function of properties.*
- $\text{glob}_{\mathcal{H}} : L^{\mathcal{H}} \to G^{\mathcal{H}}$ *associates with each local property a global property.*
- $\text{def}_{\mathcal{H}} : L^{\mathcal{H}} \to X^{\mathcal{H}}$ *associates with a local property the class where it is defined.*

Moreover, the model $\mathcal{H}$ *is constraint by the following equations (1–9).*

The model is generic, as all its components are parameterized by $\mathcal{H}$. However, for the sake of readability, in the next two sections, the parameter $\mathcal{H}$ will remain implicit. The reader must keep in mind that all the components of the model $\mathcal{H}$ are relative to it and that the parameter must be explicit as soon as one deals with more than one model, as in Sect. **??**.

[3]The 'name' of properties is a shorthand for more complex identifiers. For instance, static overloading of C++, JAVA, C#, etc. forces, in these languages, to mangle names with parameter types in order to unambiguously denote properties.

[4]The static typing requirement appears here. In a dynamic typing framework, for instance in SMALLTALK, there is no way to distinguish different global properties with the same name. Of course, static typing allows to ensure that the global propery exists.

3.2.1 Global Properties.

Given a class $c \in X$, G_c denotes the set of global properties of c. Global properties are either *inherited* from a superclass of c, or *introduced* by c. Let $G_{\uparrow c}$ and G_{+c} be the two corresponding subsets:

$$G_c = G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'} \ , \tag{1}$$

$$G_{\uparrow c} = \bigcup_{c \prec_{\mathrm{d}} c'} G_{c'} = \biguplus_{c \prec c'} G_{+c'} \quad \text{and} \quad G = \bigcup_{c \in C} G_c = \biguplus_{c \in C} G_{+c} \ . \tag{2}$$

Note that all G_{+c} are disjoint—hence a global property is introduced by a single class. This implies static typing.

Given a class $c \in X$ and two global properties $g_1, g_2 \in G$, a *global property conflict* occurs between g_1 and g_2 when:

$$\begin{aligned} &c \prec_{\mathrm{d}} c_1 \quad \wedge \quad c \prec_{\mathrm{d}} c_2 \quad \wedge \quad c_1 \neq c_2 \quad \wedge \quad g_1 \in G_{c_1} \quad \wedge \\ &g_2 \in G_{c_2} \quad \wedge \quad g_1 \neq g_2 \quad \wedge \quad \mathrm{name}(g_1) = \mathrm{name}(g_2) \ . \end{aligned} \tag{3}$$

In the following, we assume that there are no global property conflicts:

$$\forall c \in X, \forall g, g' \in G_c, \mathrm{name}(g) = \mathrm{name}(g') \Rightarrow g = g' \ . \tag{4}$$

Therefore, the restriction of the function $\mathrm{name} : G \to N$ to G_c is injective and, in the context of a class c, the identifier of a global property is unambiguous.

3.2.2 Local Properties.

Given a class c, L_c denotes the set of local properties defined in c and, conversely, the function $\mathrm{def} : L \to C$ associates with each local property the class where it is defined:

$$L = \biguplus_{c \in C} L_c \qquad \text{with} \qquad L_c = \mathrm{def}^{-1}(c) \ . \tag{5}$$

The function $\mathrm{glob} : L \to G$ associates with each local property a global property, in such a way that:

$$\forall l \in L, \quad \mathrm{name}(\mathrm{glob}(l)) = \mathrm{name}(l) \ , \tag{6}$$

$$\forall c \in X, \quad \mathrm{glob}(L_c) \subseteq G_c \ . \tag{7}$$

Conversely, the restrictions of glob and name to L_c are injective. Thus determining the global property associated with a local one is unambiguous:

$$\forall l \in L_c, \forall g \in G_c, \mathrm{name}(l) = \mathrm{name}(g) \Rightarrow \mathrm{glob}(l) = g \ . \tag{8}$$

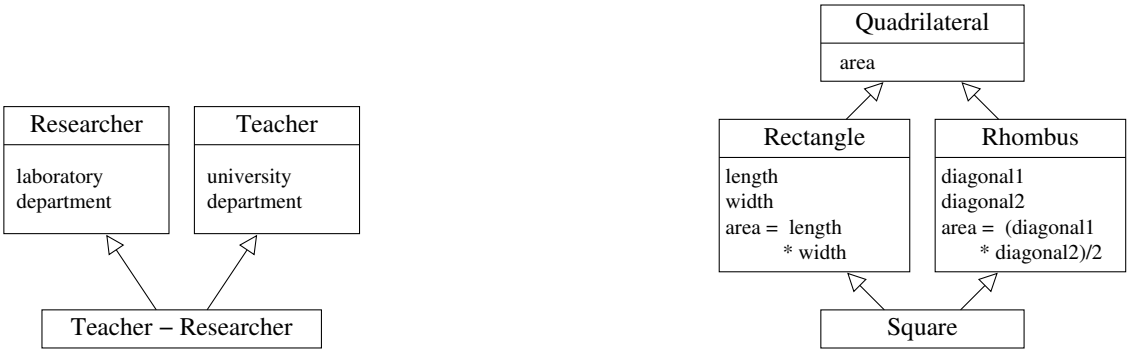
A local property is either a redefinition of an inherited property, or the introduction of a new global property. Let $L_{\uparrow c}$ and L_{+c} be the corresponding sets.

$$L_c = L_{\uparrow c} \uplus L_{+c} \quad \text{with} \begin{cases} L_{\uparrow c} & = L_c \cap \mathrm{glob}^{-1}(G_{\uparrow c}) \ , \\ L_{+c} & = L_c \cap \mathrm{glob}^{-1}(G_{+c}) \ . \end{cases} \tag{9}$$

Moreover, G_{+c} and L_{+c} are in one-to-one correspondence and $G_{+c} = \mathrm{glob}(L_{+c})$.

Redefinition (aka *overriding*) is defined as the relationship $\ll$ between a local property in $L_{\uparrow c}$ and the corresponding local properties in the superclasses of c:

$$l \ll l' \overset{\text{def}}{\Longleftrightarrow} \mathrm{glob}(l) = \mathrm{glob}(l') \wedge \mathrm{def}(l) \prec \mathrm{def}(l') \ . \tag{10}$$



(a) Global Property Conflict

(b) Local Property Conflict

Figure 9: Multiple Inheritance Conflicts

Finally, given a class c and a global property $g \in G_c$, *late binding* involves the selection of a local property of g defined in the superclasses of c (including c), hence in the set $\text{loc}(g, c) = \{l \in \text{glob}^{-1}(g) \mid c \preceq \text{def}(l)\}$. Hence, the model is completed by a function $\text{sel} : G \times X \to L$, such that $\text{sel}(g, c) \in \text{loc}(g, c)$. Usually, this selection is restricted to the subset of the most specific properties, i.e. $\text{spec}(g, c) = \min_{\ll}(\text{loc}(g, c))$. A *local property conflict* occurs when $|\text{spec}(g, c)| > 1$. Discussion of this selection process is not the point here. We just assume that, either there are no local property conflict—i.e. $\text{spec}(g, c) = \{\text{sel}(g, c)\}$—or the selection function sel chooses an element of $\text{spec}(g, c)$, as linearizations do [11].

3.2.3 Class Definition and Model Construction.

The model is built by successive class definitions.

Definition 2 *A* class definition *is a triplet* $\langle$`classname`, `supernames`, `localdef`$\rangle$, *where* `classname` *is the name of the newly defined class,* `supernames` *is the set of names of its direct superclasses—they are presumed to be already defined—and* `localdef` *is a set of local property definitions. A local property definition involves a property name and other data which are not needed here.*

A class c with name `classname` is added to X. For each superclass name in `supernames`, a new pair is added to $\prec_d$. $G_{\uparrow c}$ is computed (2) and global property conflicts are checked (3). For each definition in `localdef`, a new local property is added to L, with its corresponding name—this yields L_c. $L_{\uparrow c}$ is determined by (9). Then, G_{+c} is constituted as the set of new global properties corresponding to each local property in $L_{+c} = L_c \backslash L_{\uparrow c}$. L_c and G_{+c} are then respectively added to L and G. Finally, local property conflicts are checked for all inherited and not redefined properties, i.e. $G_{\uparrow c} \backslash \text{glob}(L_{\uparrow c})$.

3.3 Multiple Inheritance

In multiple inheritance, conflicts are the main difficulty. The metamodel yields two kinds of conflicts which require different answers. The following analysis is the same as [12], enhanced with the metamodel.

3.3.1 Global Property Conflict.

A *global property conflict* (in [12], it was called "name conflict") occurs when a class specializes two classes having distinct but homonymous global properties (3). Figure 9(a) shows two classes (`Researcher` and `Teacher`) both having a global property named `department`. The first one specifies a department in a research laboratory. The other one specifies a teaching department in a

university. It is then natural to expect that the common subclass inherits the global properties of its superclasses. However the name `department` is ambiguous in the subclass context.

Anyway, a global property conflict is nothing but a naming problem. It must be solved and a systematic renaming would guarantee the absence of such a conflict. This renaming can be constraint or free, local or global, according to whether the considered language provides:

Nothing: the language does not specify any answer to global property conflicts. This forces the programmer to rename at least one of the two properties in every program using them. Renaming does not only imply the modification of classes (with inherent potential errors) but it can be impossible (unavailable source code for instance).

Explicit Designation: i.e. an alternative fully qualified syntax, which juxtaposes to the property name the name of a class where the property name is not ambiguous, for instance the class that introduces the global property. In the specification of the class `Teacher-Researcher` of the example, `Teacher::department` would denote the global property known as `department` in the class `Teacher`. This solution is used in C++ [27] for attributes[5].

Local Renaming: Local renaming changes the designation of a property, as well global as local, in a class and its future subclasses. In the problematic `Teacher-Researcher` class of the example, one can rename `department` inherited from `Teacher` as `dept-teach` and `department` inherited from `Researcher` as `res-dept`. Thus, `department` in `Researcher` and `res-dept` in `Teacher-Researcher` denote the same global property and as expected, in the class `Teacher-Researcher`, `res-dept` and `teach-dept` denote two distinct global properties. This solution is used in Eiffel [21].

Unification: Dynamic languages like Clos, Java for interfaces and C++ for functions, consider that if two global properties are homonymous then they are not distinct. Hence, there is no global property conflict and the multiple inheritance ambiguities are deferred on local property inheritance. This solution has a major drawback: it does not allow to express the programmer intention of distinct global properties—in Fig. 9(a), the two departments represent distinct concepts. If the programmer intention was a single concept, then the programmer should have defined a common superclass introducing a single global property for this concept.

3.3.2 Local Property Conflict.

A *local property conflict* (in [12], it was called "value conflict") occurs when a class inherits two local properties from the same global property, none of them more specific than the other. Figure 9(b) illustrates two classes (`Rectangle` and `Rhombus`), both redefining the method `area` whose global property was introduced into the class `Quadrilateral`. In the common subclass `Square`, which one is most specific? Unlike the global property conflict, there is no intrinsic solution to this problem. Consequently, either the programmer or the language must bring additional semantics to solve local property conflict:

Nothing: The considered language does not specify any answer to local property conflicts. This forces the programmer to define a local property in the class where conflict appears. In this redefinition, an explicit static call, as in C++, can be used to choose among the local conflicting properties.

Combining: For some values or particular properties, the conflict resolution must be done by combining the conflicting values: for instance, it is the case for the type with covariant redefinition (the lower bound of conflicting types, if it exists). Combining is also needed by Eiffel contracts with disjunction of preconditions and conjunction of postconditions.

[5]For methods, the operator :: corresponds to a static call—therefore, it denotes a local property.

Selection: The programmer or the language arbitrarily select the local property to inherit. In many dynamic languages, such as CLOS, DYLAN or PYTHON, the choice is made by a linearization [11, 3]; in EIFFEL, the programmer can select the desired property by using the `undefine` inheritance clause.

3.4 Comparison with Existing Languages

The main contribution of this metamodel is to make unambiguous the specifications of languages when naming problems occur, i.e. when there are several properties with the same identifier in the context of a single class. Such naming problems mostly occur with multiple inheritance or with *static overloading* [22]. In the absence of these naming problems, there is no need, but conceptual, to distinguish global properties and property identifiers and all languages agree with the metamodel. On the contrary, when naming problems arise, different languages present different behavior, and our claim is that the present metamodel is a good basis for better specifications of the relationship between object-oriented entities and their names. We review hereafter most commonly used languages.

In the SMALLTALK terminology, *methods* and *method selectors* denote respectively local and global properties. Nevertheless, selectors are simply reified as symbols and there is no equivalent for attributes (*instance variables* in SMALLTALK terminology). In CLOS, *methods* and *generic functions* stand for local and global properties—they are reified, but multiple dispatch changes the model as they do not belong to classes, hence are not inherited in the usual meaning. As for attributes (*slots* in CLOS terminology), they are reified into two kinds of *slot descriptions*, which can be *direct* or *effective*: but both can be understood as local properties, effective slots being only the result of inheriting slot descriptions from superclasses [15]. Moreover, as previously noticed, in both SMALLTALK and CLOS, dynamic typing does not allow to distinguish two entities with the same name, whether it is a selector, a generic function or a slot. Despite this last point, those languages are the only one whose terminology is suitable for distinguishing the two key notions that we have called local and global properties. In the following languages, one word (method, feature, etc.) stands for the two notions.

JAVA is almost fully compatible with the metamodel. However, multiple inheritance is possible with JAVA interfaces and, when global property conflicts occurs, it is not possible to distinguish two methods with the same signature. Moreover, there is no reification of global properties, neither in the introspection facilities (package `java.lang.reflect`), nor in reflective extensions of the language, as OPENJAVA or JAVASSIST [6].

EIFFEL also is fully compatible in common usage. In the terminology, "feature" stands for property, without distinguishing the two kinds, even though a notion of *feature seed* could be understood as the introduction of global properties [21]. Feature renaming allows to deal with global property conflicts in the desired way. However, a full usage of the `rename` clause is not compatible with the metamodel: (i) a feature with the new name can coexist in subclasses with the old-named feature, as two distinct features, (ii) in a class, if two features from different seeds are inherited under the same name, then they are locally merged.

Among the most commonly used languages, C++ is obviously the less compatible with the metamodel, as soon as one uses multiple inheritance. Firstly, the keyword `virtual` is mandatory in inheritance, for avoiding the duplication of the attributes introduced in a superclass inherited through multiple paths (e.g., `Quadrilateral` in Fig. 9(b)). But, this is not enough for distinguishing two methods with the same signature, introduced in different unrelated superclasses.

Eventually, one can't speak of metamodeling without considering UML, where all entities are metadefined. As a matter of fact, concerning properties, the UML metamodel is left unfinished: the *Features diagram* of [23, page 27] shows only one kind of entity called "feature". Page 38, the specification says "One classifier may specialize another by adding or redefining features". Does it mean that their "features" correspond to our local properties?

Concerning the present metamodel, the specifications of most languages could adopt it without changing but marginally either programming habits or program behavior. It would afford

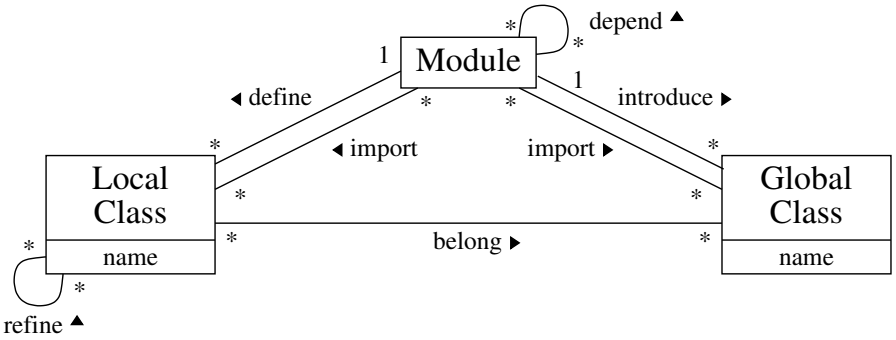


Figure 10: Metamodel of Modules and Classes

some terminological and conceptual basis to object-oriented programming, either for program documentation or teaching.

4 Modules and Classes

Let us now more formally present modules and class refinement, with a strong analogy with the class and property metamodel previously presented. In the following, we present successively the metamodel, its formalism, an analysis of multiple import and multiple inheritance conflicts, and a discussion about static typing.

4.1 Module and Class Metamodel

A more rigorous approach comes from the observation that classes are to modules what properties are to classes, the dependence relation between modules matches specialization, and import matches inheritance. It is thus necessary to define two entities associated with the concept of class, and one entity associated with the concept of module (Fig. 10).

Modules are hierarchies of classes. A module *depends* on zero, one or more others modules—if a module m depends on a module n, n is a supermodule of m, and m is a *submodule*[6] of n. Alike the class specialization relation, the module dependence relation is a strict partial order.

Local classes (alike local properties) are defined in modules. A local class is described by an ordinary *class definition* (Def. 2). *Global classes* (alike global properties) gather local classes and are orthogonal to the modules. Each module has global classes which correspond to the classes that it knows, i.e. those that can be used in the module.

A module definition is a triplet constituted with the name of the module, the name of its supermodules, presumably already defined, and a set of local class definitions[7]. The dependence relation supports an import mechanism—i.e. modules import the classes of their supermodules. The two kinds of classes yield two-level importation. First of all, a module *imports* all global classes of its supermodules—this is *global class importation*. Then, each local class definition is processed. If the name of the local class is the same as the name of an imported global class, the new local class is attached to the global class. If there is no such imported global class, a new global class with the same name is *introduced* in the module.

For any global class of a module, the existence of the corresponding local class is supposed: it is either an explicit definition, or an implicit refinement. In the latter case, local classes are said to be *implicit classes* since they do not have explicit definitions.

The *class refinement relation* (alike property redefinition relation) is deduced from the module dependence relation. However, since modules are class hierarchies, there is a *specialization relation*

[6] A submodule is not a module inside a module (*nested module*). It is a module that depends on another module.

[7] The local classes are defined in a module like classes in an ordinary object-oriented language

between local classes of a module. This relation is deduced from the explicit declarations of superclasses: (i) in a local class of a module, a declaration of a superclass named n is licit if the module introduces or imports a global class with the same name n. (ii) The specialization relation is transitive (it is a partial order). (iii) A module imports specialization relation of its supermodules. Let us notice that an explicit specialization relation between local classes already in specialization relation by import or by transitivity has no effect.

In figures, we have chosen the following convention: local classes appear as small named boxes, inside larger numbered boxes, namely the modules. Only specialization relations in a module and dependence between modules are drawn. Local classes in the same global class have the same name—the refinement relation between classes thus remains implicit. Moreover, implicit classes and implicit specializations are illustrated in dotted lines.

Then, a *program* is a set of modules, closed by the dependence relation. It corresponds to a main module (possibly implicit) that requires every other module of the program. There are two intuitive views of the behavior of the program which correspond to a main module m:

- The program can be seen as a class hierarchy $(C, \prec)$ where C is the set of all the local classes of all modules, and $\prec$ is the transitive closure of the union of the refinement relation and the specialization relation. In the whole program, each instantiation of a class is interpreted as the instantiation of the local class of the module m that has the same name—local classes of the supermodule of m are interpreted as abstract classes.
- The program can be seen as the hierarchy of local class of the module m where the contents of each local class results from its successive refinements from the required modules.

4.2 Notations and definitions

A *module* is a hierarchy of classes and a *program* is a hierarchy of modules.

Definition 3 *A model of a program is a tuple* $\mathcal{P} = \langle X^{\mathcal{P}}, \prec^{\mathcal{P}}, G^{\mathcal{P}}, L^{\mathcal{P}}, N^{\mathcal{P}}, \mathrm{name}_{\mathcal{P}}, \mathrm{glob}_{\mathcal{P}}, \mathrm{def}_{\mathcal{P}} \rangle$ *where:*

- $X^{\mathcal{P}}$ *is the set of* modules *and* $\prec^{\mathcal{P}}$ *is the* module dependence *relationship, which satisfies the same equations as class specialization, together with the same notation.*
- $G^{\mathcal{P}}$ *and* $L^{\mathcal{P}}$ *are the set of* global *and* local classes.
- $N^{\mathcal{P}}$ *is the set of* class identifiers *(names).*
- $\mathrm{name}_{\mathcal{P}} : G^{\mathcal{P}} \uplus L^{\mathcal{P}} \to N^{\mathcal{P}}$ *is the naming function of classes.*
- $\mathrm{glob}_{\mathcal{P}} : L^{\mathcal{P}} \to G^{\mathcal{P}}$ *associates with each local class a global class.*
- $\mathrm{def}_{\mathcal{P}} : L^{\mathcal{P}} \to X^{\mathcal{P}}$ *associates with a local class the module where it is defined.*

All notations and equations available for $\mathcal{H}$ are also available for $\mathcal{P}$, apart substituting $\mathcal{P}$ (resp. 'module', 'class') to $\mathcal{H}$ (resp. 'class', 'property').

Module dependence entails *class refinement*, which is simply the analogue of property redefinition, i.e. the relation $\ll^{\mathcal{P}}$ (10). Each module $\mathcal{M} \in X^{\mathcal{P}}$ is itself a class hierarchy $\mathcal{M} = \langle X^{\mathcal{M}}, \prec^{\mathcal{M}}, G^{\mathcal{M}}, L^{\mathcal{M}}, N^{\mathcal{M}}, \mathrm{name}_{\mathcal{M}}, \mathrm{glob}_{\mathcal{M}}, \mathrm{def}_{\mathcal{M}} \rangle$ where $X^{\mathcal{M}} = L^{\mathcal{P}}_{\mathcal{M}}$ is the set of local classes of $\mathcal{M}$ in $\mathcal{P}$ and all $N^{\mathcal{M}}$ are the same nameset. Once again, all notations and equations available for $\mathcal{H}$ and $\mathcal{P}$ are also available for $\mathcal{M}$, apart from (1) which is slightly modified into (13).

However, modules are not exactly ordinary class hierarchies. Indeed, class refinement in turn induces inheritance of global properties. When $c \ll^{\mathcal{P}} c'$, then $G^{\mathcal{M}'}_{c'} \subseteq G^{\mathcal{M}}_{c}$—where $\mathcal{M} = \mathrm{def}_{\mathcal{P}}(c)$, $\mathcal{M}' = \mathrm{def}_{\mathcal{P}}(c')$ and $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$. As a small difficulty arises regarding $X^{\mathcal{M}}$, for simplification's sake, one assumes that $X^{\mathcal{M}}$ includes a local class definition, possibly empty, for each class in the supermodules of $\mathcal{M}$, which entails that:

$$\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}' \Rightarrow \mathrm{glob}_{\mathcal{P}}(X^{\mathcal{M}'}) \subseteq \mathrm{glob}_{\mathcal{P}}(X^{\mathcal{M}}) \ . \tag{11}$$

Moreover, one must also assume that the specialization relationship $\prec^{\mathcal{M}}$ include its analogue in supermodules. Given $c_{\mathcal{M}}, c'_{\mathcal{M}} \in X^{\mathcal{M}}$:

$$c_{\mathcal{M}} \ll^{\mathcal{P}} c_{\mathcal{M}'} \quad \wedge \quad c'_{\mathcal{M}} \ll^{\mathcal{P}} c'_{\mathcal{M}'} \quad \wedge \quad c_{\mathcal{M}'} \prec^{\mathcal{M}'} c'_{\mathcal{M}'} \Rightarrow c_{\mathcal{M}} \prec^{\mathcal{M}} c'_{\mathcal{M}} \ . \tag{12}$$

Finally, the following equation replaces (1):

$$G_c^{\mathcal{M}} = (G_{\uparrow c}^{\mathcal{M}} \cup G_{\Uparrow c}^{\mathcal{M}}) \uplus G_{+c}^{\mathcal{M}} \quad \text{with} \quad G_{\Uparrow c}^{\mathcal{M}} = \bigcup_{\substack{c \ll^{\mathcal{P}} c' \\ \mathcal{M}' = \mathrm{def}_{\mathcal{P}}(c')}} G_{c'}^{\mathcal{M}'} \ , \tag{13}$$

and class refinement entails also a redefinition relationship, noted $\lhd$, on local properties. Given $lp \in L^{\mathcal{M}}$ and $lp' \in L^{\mathcal{M}'}$, with $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$:

$$lp \lhd lp' \overset{\text{def}}{\Longleftrightarrow} \mathrm{glob}_{\mathcal{M}}(lp) = \mathrm{glob}_{\mathcal{M}'}(lp') \quad \wedge \quad \mathrm{def}_{\mathcal{M}}(lp) \ll^{\mathcal{P}} \mathrm{def}_{\mathcal{M}'}(lp') \ . \tag{14}$$

The program $\mathcal{P}$ defined as a module hierarchy is equivalent to a class hierarchy $\mathcal{Q} = \langle X^{\mathcal{Q}}, \prec^{\mathcal{Q}}, G^{\mathcal{Q}}, L^{\mathcal{Q}}, N^{\mathcal{Q}}, \mathrm{name}_{\mathcal{Q}}, \mathrm{glob}_{\mathcal{Q}}, \mathrm{def}_{\mathcal{Q}} \rangle$, where $X^{\mathcal{Q}} = G^{\mathcal{P}}$ is the set of global classes of $\mathcal{P}$ and $N^{\mathcal{Q}}$ is the common nameset for all $\mathcal{M} \in X^{\mathcal{P}}$. In the program $\mathcal{P}$, any reference to a classname actually references the single corresponding class in $X^{\mathcal{Q}}$. $\mathcal{Q}$ gives the semantics of the program $\mathcal{P}$ according to the following definitions.

The resulting class specialization $\prec^{\mathcal{Q}}$, is the union of the image by $\mathrm{glob}_{\mathcal{P}}$ of all local class specializations, i.e. the smallest set such that:

$$\forall \mathcal{M} \in X^{\mathcal{P}}, \forall c, c' \in X^{\mathcal{M}} : c \prec^{\mathcal{M}} c' \Rightarrow \mathrm{glob}_{\mathcal{P}}(c) \prec^{\mathcal{Q}} \mathrm{glob}_{\mathcal{P}}(c') \ . \tag{15}$$

The set of global properties, is merely the union of all sets of global properties:

$$G^{\mathcal{Q}} = \bigcup_{\mathcal{M} \in X^{\mathcal{P}}} G^{\mathcal{M}} = \bigcup_{gc \in X^{\mathcal{Q}}} G_{gc}^{\mathcal{Q}} \ , \tag{16}$$

and $G_{gc}^{\mathcal{Q}}$ is defined as the union of the sets of global properties of all local classes of gc:

$$G_{gc}^{\mathcal{Q}} = \bigcup_{\substack{lc \in \mathrm{glob}_{\mathcal{P}}^{-1}(gc) \\ \mathcal{M} = \mathrm{def}_{\mathcal{P}}(lc)}} G_{lc}^{\mathcal{M}} \ . \tag{17}$$

Finally the local properties of a class are defined as the most specific, according to $\lhd$, local properties of all local classes of the considered global class:

$$L_{gc}^{\mathcal{Q}} = \min_{\lhd} \left(\biguplus_{\substack{lc \in \mathrm{glob}_{\mathcal{P}}^{-1}(gc) \\ \mathcal{M} = \mathrm{def}_{\mathcal{P}}(lc)}} L_{lc}^{\mathcal{M}} \right) \ . \tag{18}$$

Of course, this is a sound local property set only if the function $\mathrm{name} : L^{\mathcal{Q}} \to N^{\mathcal{Q}}$—defined as the disjoint union of all $\mathrm{name}_{\mathcal{M}}$—is injective on $L_{gc}^{\mathcal{Q}}$ (8). Otherwise, a conflict occurs, which must be solved. The whole program is constructed starting from this last sets $L_{gc}^{\mathcal{Q}}$ (Section 3.2.3).

4.3 Multiple Import and Inheritance

As a result of the previous definitions, refinement of a subclass automatically induces multiple inheritance, through class specialization ($\prec^{\mathcal{M}}$) and class refinement ($\ll^{\mathcal{P}}$). Hence, it is impossible to ignore the problems caused by multiple inheritance. Moreover, the dependence relationship between modules ($\prec^{\mathcal{P}}$) can itself be multiple. In this section, we analyze these new conflicts and investigate their treatment at the light of the metamodel.

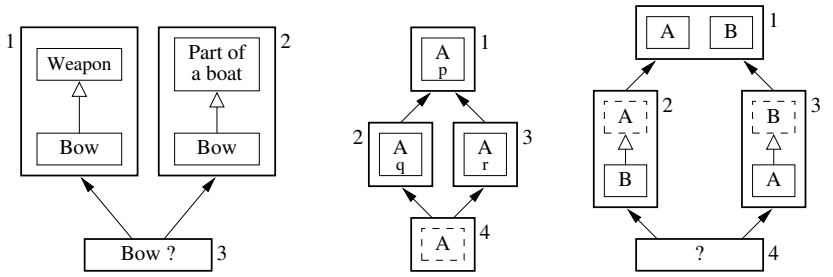


Figure 11: Class Import and Conflict

4.3.1 Multiple Import.

A *global class conflict* occurs when a module imports two homonymous global classes from two different supermodules (Fig. 11(a)). It is analogous to, and has the same definition as the global property conflict (3) and it can be solved in the same way. As the modules are generally namespaces, explicit designation is the most natural solution here. However, a class renaming mechanism, as in EIFFEL and its configuration language LACE [21], would be an alternative.

In a similar way, a *local class conflict* occurs when a local class refines several most specific classes in supermodules, i.e. when $|\text{spec}_{\mathcal{P}}(c, m)| > 1$ for a module m and a local class c. Figure 11(b) illustrates this configuration: module 4 imports A introduced by module 1 and implicitly refines A_2 and A_3 by a local class A_4. Furthermore, the situation must be quite frequent since all the classes in supermodules are implicitly refined by empty local classes (11).

Nevertheless, apart from the structural analogy, *local class conflicts* are quite different from *local property conflicts*. Whereas a local property is required to be unique and atomic in a class, local classes can be understood as a set of local properties. Therefore, the natural solution to *local class conflicts* is a combination of the conflicting local classes—this is done through refinement based inheritance of their properties (13). Of course, some property conflicts may follow, that we will examine hereafter.

4.3.2 Cyclic Specialization and Class Unification.

A new conflict configuration, named *specialization conflict*, appears in a module when two local classes specialize each other, after inheritance of the specialization from supermodules (12). Figure 11(c) illustrates this case. It creates a cycle in the specialization relation which is no longer a partial order, but only a preorder. A natural solution would be to forbid such conflicts, which means that for each module $\mathcal{M} \in X^{\mathcal{P}}$, the part of $\prec^{\mathcal{M}}$ declared by the programmer should not entail such cycles in conjunction with the part inherited from supermodules by (12). This is however not enough since the cycle may follow only from the specialization inherited from two unrelated supermodules.

An alternative would be to unify all the classes on the cycle, since they are presumed to all have the same instances and the same properties. Moreover, class unification could be an explicit feature of class refinement language. We shall no more formalize this idea, which seems quite feasible. Of course, the occurrence of the same property conflicts would be likely.

4.3.3 Global Property Inheritance.

Like usual classes, a local class resulting from specialization, refinement and maybe unification consists in sets of global and local properties.

A local class has the global properties of its superclasses, i.e. the classes that it specializes and refines (13). This should entails the usual global property conflicts (3) together with their

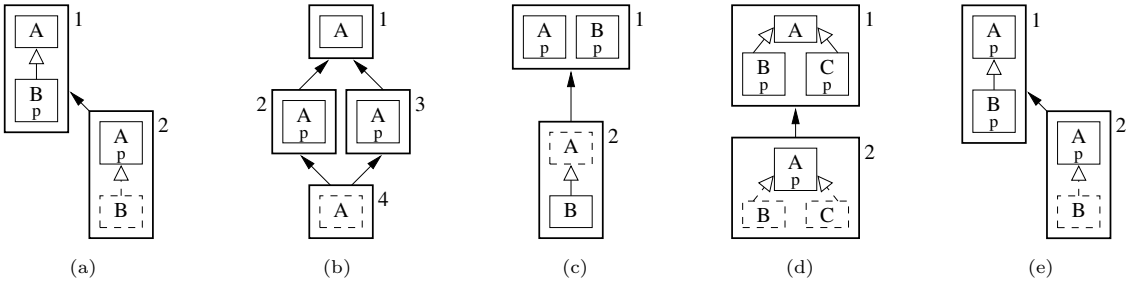


Figure 12: Property Inheritance

generalization to refinement, by replacing $\prec_d^{\mathcal{M}}$ by $\ll_d^{\mathcal{P}}$. In Fig. 11(b), A_2 inherits the global property `p` introduced into A_1 while A_4 inherits the global properties `p`, `q` and `r` introduced into the classes it refines.

However, as the programmer of a module m knows the modules required by m, some apparent global property conflicts can be solved by the identification of global properties. Thus, addition of local properties to global properties no longer needs to be restricted to redefinition in subclasses. *Global property generalization* allows the extension of global properties from some classes to their superclasses. The generalization of a global property owned by a local class C_m in a local class D_n (with $n < m$) consists in defining a local property in D_n with the same name as the global property, since $C_n \prec D_n$. The example in Fig. 12(a) illustrates the definition of a property `p` in a local class B_1 and its generalization in the class A_2 (since $B_2 \prec A_2$).

Global property conflicts are still possible for specialization or refinement. They must be individually solved by the techniques of the Sect. 3.3.1. In addition to the global property conflict related to multiple specialization (Fig. 9(a)), Fig. 12(b) illustrates a conflict related to multiple refinement, Fig. 12(c) illustrates a conflict implying refinement and specialization, and Fig. 12(d) illustrates a conflict involved by property generalization.

4.3.4 Local Property Inheritance.

Local property inheritance can present three kinds of apparent conflicts, when there are several most specific inherited local properties, for a given class and a given global property: (i) usual local property conflict, when specificity is related to $\prec^{\mathcal{M}}$ and $\ll^{\mathcal{M}}$, (ii) the analogue when specificity is related to $\ll^{\mathcal{P}}$ and $\lhd$, (iii) a mixed situation, when the two conflicting properties are inherited respectively through $\prec^{\mathcal{M}}$ and $\ll^{\mathcal{P}}$. In the two first cases, the conflict must be solved in the same way as ordinary local property conflicts.

Figure 12(e) illustrate the latter case and raises the question of local property inheritance of the global property `p` in the local class B_2. The classes B_1 and A_2 are incomparable by $\prec^{\mathcal{M}}$ or $\ll^{\mathcal{P}}$, but the intuitive vision considering refinement as an incremental modification of classes gives to B_2 the method p defined in B_1. This intuition agrees with the definition of $\mathcal{Q}$, where the local property p defined in B_1 is the local property for B (18).

4.4 Static Typing

4.4.1 Property Conformance.

With redefinition or inheritance of a local property `l` in a class c, it is necessary to check that this one conforms to the local properties defined in the superclasses of C_m. In the example in Fig. 12(a), it should be checked that local property `p` inherited in B_2 (i.e. that defined in B_1) conforms with the one defined in A_2. Conformity between local properties can take several forms according to the considered language: arity, static types of result and parameters, declared exceptions, contracts, etc.

Since the refinement semantic is not a specialization semantic, the covariant refinement policy of [10] does not apply: on the contrary, class refinement may be used to extend a method behavior by making it accepting more argument values (parameter type contravariance) and giving more result values (return type contravariance). In accordance to the type theory, parameter type contravariance is safe whereas return type contravariance is unsafe. Hence, within a safe typing framework, refinement must use a strict invariance return type rule.

4.4.2 Constructors.

In statically typed languages without class refinement, polymorphism does not apply to instance constructors: the static type used predetermines the dynamic type of created instances to come. Thus, in the language specifications, the particular methods with instance constructor role are either not inherited (Java or C++) or inherited but without their instance constructor role (Eiffel).

With class refinement, it is a bit different since the dynamic type of newly instancied object is statically unknown, since the local classes statically handled by the module can be refined in possible submodules. Thus, during a class refinement, on one hand, constructors must be fully inheritable. On the other hand, the refining classes must make sure that the constructors introduced into the refined classes remain coherent (possibly in redefining them).

5 Related Works

Modular open classes [7] (for MultiJava an extension to Java) proposes compilation units, similar to the modules presented in this present article. They are provided with a dependence relation via the keyword `require`. MultiJava makes it possible to extend existing classes by adding functions by an ad hoc syntax whereas, in opposition to our approach, method redefinition, attribute addition or declaration of Java interface implementation are not allowed. Nevertheless, MultiJava is compatible with separate compilation and dynamic loading. It also proposes an implementation of multimethods.

Classboxes [4] introduce a notion of 'box of classes' (kind of modules) for Smalltalk. Classes can be extended by adding or redefining methods and attributes in classboxes while controlling the visibility of these additions since these changes have only local impacts: message sending answers are both determined by the receiver and the *classbox*. This method sending mechanism is called *local rebinding*. Thus, in opposition to the contribution of our article, class amendments made in a *classbox* are applied only to this *classbox* and to *classboxes* which require it. Hence, message sending from others *classboxes* are not affected by the modification.

Package merge is a mechanism proposed by UML 2.0 [23]. It permits an incremental definition of packages. This mechanism is defined independently of package import. As for property multiple inheritance, multiple merge is not clearly specified and inherent conflicts are not discussed.

Difference-based modules [14] (for MixJuice a language based on Java) proposes modules in dependence relation and refinement of classes authorising method (re)definition, attribute addition and declaration of Java interface implementation. In the case of multiple dependencies between modules, global property conflicts are solved by explicit designation; local property conflicts are solved by linearization. The approach is compatible with separate compilation but does not allow dynamic loading. In contrast with our proposition, MixJuice neither reveals the analogy between classes and modules nor analyzes the various conflict configurations and their resolutions. The other main difference with our proposition is that classes are in single inheritance. Therefore adding super-classes to existing classes is not possible.

Virtual classes [18] of Beta introduces a mechanism where a inner class c_0 (called virtual) is defined in an outer class c_1, then c_0 can be extended (refined) in any outer class c_2 subclass of c_1. Using the vocabulary of our proposition, outer classes are modules, and inner classes are classes. The main difference with our proposition is that instances of a virtual class and instances of its redefinition are distinct and can exist simultaneously in a same program.

Higher-order hierarchies [13] are quite similar to our modules. Motivations are identical and isomorphism of the metamodels of classes (order 1) and modules (order 2) would allow any order generalization. It is however not our goal and we do not push the class metaphor so far: finally, as [28] says it, our modules are not classes, given that they do not have instances. Moreover, technically, both approaches are quite different concerning multiple inheritance, which is unavoidable as we saw it. *Higher-order hierarchies* proposes a combination of completely ordered *mixins* whereas our proposal relies on the interpretation of multiple inheritance in the metamodel. Since [12], it seems to us the only good way to understand multiple inheritance.

6 Conclusion and Perspective

We proposed in this article, *class refinement*, an extension of the object-oriented model for statically typed languages with multiple inheritance. It provides a concept of module and a mechanism of refinement between classes—i.e. modules can modify classes imported from their supermodules. This approach allows an explicit separation of concerns, therefore a better reusability of object oriented programs. Contrary to some other proposition, it requires only a light syntactic addition—i.e. a rudimentary module language to express that a module depends on another module and that a class (re)definition belongs to a module[8].

Although the use of modules and classes are fundamentally different [28], our proposal is based on a strict structural analogy between these two concepts since they are described by similar metamodels[9]. The metamodels enhances the distinction between local and global entities, properties or classes. This distinction allows a consistent conflict management which respects the semantics of class specialization.

6.1 Implementation limitation and perspective.

Implementation of programs using class refinement is almost similar to implementation of any object-oriented program. Global techniques à la SmartEiffel [8] may be used. The content of the classes of the program results from their successive refinements.

However, modules are good candidates for being compilation units in a separate compilation scheme. The PRM compiler prototype use the separate compilation scheme we developed in [25]. In a local phase, modules are separately compiled and produce incomplete executable code (it contains unresolved symbols). Such a separate module compilation is quite independent of other modules: only the metamodel of supermodules is needed, and this phase can be recursively done. In a link phase, compiled modules composing the program are gathered, their symbols are resolved, then a complete executable is produced.

This compilation scheme is incompatible with dynamic loading of modules since it implies that compiled modules of programs (executable code and metamodel) are entirely known before execution, as module dependence and class refinement are statically computed. Moreover class refinement is not a realistic approach with dynamic loading, i.e. refinement of classes during the execution of a program, since it requires instance modification and needs implementation techniques similar to instance migration ones.

6.2 Modeling limitation and perspective.

The main limitation of our proposition is that it does not support *revision* of models: like class specialization, class refinement is monotonous. For instance, class refinement cannot be used to suppress properties and specialization relation, to unify global properties nor to split a class into two classes. Although suppression is not clearly useful, global property unification would be an answer to some conflicts, with the proviso that one correctly formalizes it. But class splitting is clearly out of bounds of refinement.

[8] This one can be implicit if modules are source files and local classes are defined in these files.

[9] "Import is inheritance—why we (still) need both." to counter-paraphrase [28].

However, the language specification perspective is to answer remaining open questions. For instance, refinement and visibility (*public*, *private*, etc.) or refinement and genericity (parameterized classes).

References

[1] E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In Madsen [19], pages 133–152.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Combining feature-oriented and aspect-oriented programming to support software evolution. In *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, 2005.

[3] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for Dylan. In P. Cointe, editor, *Proc. ECOOP'96*, LNCS 1098, pages 69–82. Springer-Verlag, 1996.

[4] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *JMLC 2003 (Joint Modular Languages Conference)*, volume 2789, pages 122–131, 2003.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP'90*, SIGPLAN Notices, 25(10). ACM Press, 1990.

[6] S. Chiba. Javassist — a reflection-based programming wizard for Java. In *Proc. of ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symetric multiple dispatch for Java. In *Proc. OOPSLA'00*, SIGPLAN Notices, 35(10), pages 130–145. ACM Press, 2000.

[8] D. Colnet and O. Zendra. 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, pages 341–350. IEEE Computer Society, 1999.

[9] R. Ducournau. *Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual*. Sema Group, Montrouge, France, 1991.

[10] R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, LNCS 2426, pages 3–12. Springer-Verlag, 2002.

[11] R. Ducournau, M. Habib, M. Huchard, and M.-L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA'94*, pages 164–175. ACM Press, 1994.

[12] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3):309–345, 1995.

[13] E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proc. ECOOP'2003*, LNCS 2743, pages 303–329. Springer-Verlag, 2003.

[14] Y. Ichisugi and A. Tanaka. Difference-based modules: A class-independent module mechanism. In J. M. Troya, editor, *Proc. ECOOP'2002*, LNCS, pages 62–88. Springer-Verlag, 2002.

[15] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP'2001*, LNCS 2072, pages 327–355. Springer-Verlag, 2001.

[17] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. ECOOP'97*, LNCS 1241, pages 220–242. Springer-Verlag, 1997.

[18] O. Madsen and B. Møller-Pedersen. Virtual classes. a powerful mechanism in object-oriented programming. In *Proc. OOPSLA'89*, pages 397–406, New Orleans, 1989. ACM Press.

[19] O. L. Madsen, editor. *Proceedings of the 6th European Conference on Object-Oriented Programming, ECOOP'92*, LNCS 615. Springer-Verlag, 1992.

[20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor. Prentice Hall International, Hemel Hempstead, UK, 1988.

[21] B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1992.

[22] B. Meyer. Overloading vs. object technology. *Journal of Object-Oriented Programming*, 14(5):3–7, October/November 2001.

[23] OMG. Unified Modeling Language 2.0 superstructure specification. Technical report, Object Management Group, 2004.

[24] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, 2001.

[25] J. Privat and R. Ducournau. Link-time static analysis for efficient separate compilation of object-oriented languages. In M. Ernst and T. Jensen, editors, *Program Analysis for Software Tools and Engineering*, pages 29–36, 2005.

[26] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In E. Jul, editor, *Proc. ECOOP'98*, LNCS 1445, pages 550–570. Springer-Verlag, 1998.

[27] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading (MA), USA, 1986.

[28] C. A. Szyperski. Import is not inheritance — why we need both: Modules and classes. In Madsen [19], pages 19–32.