Category-Based Composition in Object-Oriented Languages

Paulo Jorge Lopes de Moura
Department of Informatics
University of Beira Interior
6201-001 Covilhã, Portugal
Phone: +351 275319700, Fax: +351 275319732
pmoura@noe.ubi.pt

Abstract. In pure object-oriented languages like Smalltalk or Java, sharing a method between a set of unrelated objects requires the use of either inheritance or composition mechanisms. These two solutions can be problematic and cumbersome. Using inheritance, shared methods must be stored in a common ancestor, implying the use of multi-inheritance or, in single-inheritance systems, root objects with large collections of methods and complex interfaces. The use of instance variables to implement composition implies a level of indirection with the consequent need of glue code whenever we need to add the composed object methods to the container's interface. Logtalk, a Prolog object-oriented extension, features *categories* as a new composition mechanism that solves these problems. Categories complement instance variable based composition and provide alternative solutions to multi-inheritance designs for single inheritance languages. Categories also provide several developing benefits such as incremental compilation and splitting of complex objects into more manageable and reusable components.

1 Introduction

Sometimes we want to define and reuse a set of methods and variables that, even if functionally cohesive, do not fit the notion of an object and only make sense when composed with other code to construct new objects. Take for example the Smalltalk [1] dependency mechanism that enables an object to notify a set of dependent objects of the occurrence of some relevant event. We may need this mechanism for some of our objects but certainly not for all. In Smalltalk, this is an all–or–nothing preposition. For example, in VisualWorks [2, 3] the code is contained in a parcel and is an optional feature that one can load/add to the base system. The dependent methods (and associated data structures) are always added to the root class, `Object`, making them available for all objects. We either have dependents support for all objects or for none. In other systems where the dependency code is included by default, it is also stored in the root class `Object`. There are two issues here: how do we encapsulate and how do we reuse such a set of methods? We seek a solution that enables us to add the methods that we want to reuse to the interface of only those objects that will use them, and at the same level as the object's locally defined methods. In hybrid languages like C++ [4], we can always write generic code like utility functions without encapsulating them inside objects. However, in pure object-oriented languages like Smalltalk or Java [5], all the code that we write must be encapsulated in some object. Sharing a method among a set of unrelated objects requires then the use of either inheritance or composition mechanisms.

1.1 Reuse via Inheritance

Using inheritance, shared methods must be stored in a common ancestor. If the chosen language supports multi-inheritance, we can encapsulate our methods in a parent class for of all the objects that need to inherit such methods. Implementation multi-inheritance usually causes no problem for independent sets of cohesive methods and variables. However, languages like Smalltalk only support single-inheritance[1] and others like Objective-C [6, 7] or Java only supports multi-inheritance of protocols or interfaces. Without stepping into the single- versus multi-inheritance controversy [8], we need a solution that can be adopted in single-inheritance languages like Smalltalk or Java.

With single-inheritance, shared methods must be added to some common ancestor object. The outcome is often root objects with large collections of methods and complex interfaces, despite the fact that most descendant objects never use most of the methods [9]. For example, the root class of VisualWorks 3.0 has 94 methods in 25 categories while Squeak 2.5 [10, 11], another Smalltalk system, has 96 methods in nine categories. The numbers for Java and Objective-C are better. In Java 1.2 [12] we have 12 methods in class `Object` but 35 methods in class `Class` (the root of the instantiation graph). In Apple's Objective-C frameworks [13] we have 38 instance and class methods in the root class `NSObject`. Single-inheritance may also force hierarchy relations that do not reflect the application domain but are rather workarounds for language limitations [14], although the problem can be mitigated by multi-inheritance of interfaces or protocols.

1.2 Reuse via Instance-Based Composition

Common object-oriented languages like Smalltalk, Objective-C, Java, or C++ lack native support for composition at the same level as inheritance. The customary solution to implement composition is to use an instance variable to hold a reference to an instance of the class that contains the methods we need to reuse. This implies a level of indirection when we want to add the composed object methods to the container's interface, with the consequent need of cumbersome glue code and some performance penalties. In addition, updating the composed object will not automatically update the container object's interface. Note that these drawbacks of what we call instance-based composition result from our need of a *different* kind of composition solution, not from any inherent problem of this reusing method.

[1] There are however some research Smalltalk systems that support multi-inheritance.

1.3 Reuse via Category-Based Composition

In this paper, we present the *category* concept, a new composition mechanism, as a possible solution for these problems, especially in the context of single-inheritance languages. Categories are implemented in the Logtalk 2.x system [15, 16, 17], an object-oriented extension to the programming language Prolog [18]. First, we start with a brief description of the Logtalk system. Second, we describe the category concept, its roots and its implementation in Logtalk, comparing it to related work. Third, several examples are presented. Last, we conclude by summarizing some cases where categories can be a useful tool and discuss some possible extensions of the category concept.

2 The Logtalk System

Logtalk 2.x is an Open Source [19] object-oriented extension to Prolog consisting of a pre-processor and a runtime engine. The pre-processor compiles Logtalk source files to Prolog code that is then compiled by the chosen Prolog compiler. Logtalk is a neutral object-oriented language, supporting both prototypes and classes in the same application. We can have any number of object hierarchies, including reflective designs, and freely exchange messages between any kind of objects. Logtalk supports separation of interface from implementation through the definition of protocols (similar to Objective-C protocols or Java interfaces), that can be implemented by either prototypes or classes. Data hiding is ensured by the implementation of public, protected, and private inheritance along with public, protected, and private predicates. Multiple inheritance is supported for both protocol and implementation. Other major features include parametric objects; event-driven programming; instance-defined methods; dynamic binding; static and dynamic predicates; automatic generation of documentation files in XML format; and categories, the subject of this paper. Logtalk is compatible with any operating system running a Prolog compiler complying with the Prolog ISO Standard [20]. It currently includes configuration files for nineteen commercial and academic Prolog compilers. Logtalk 1.0 has released in February 17, 1995, while the first public (non-beta) 2.x version has been released in February 9, 1999[2]. The current distribution includes a standard library, several examples, user and reference manuals, and a programming tutorial. Logtalk users are developing applications ranging from power network fault diagnosis [21] to multi-agents systems, web interfaces, expert systems, OODBMS prototypes, CAD systems, and several academic research projects on OOP [22] and knowledge representation languages [23]. Logtalk is also being used to teach object-oriented programming to

[2] Although the two systems share the same name and broad concepts, they are not compatible and are designed with different sets of goals.

undergraduate students. As a Prolog extension, Logtalk can be used as an interpreted, interactive, object-oriented programming language.

Using Logtalk/Prolog as a research tool in object-oriented programming has several advantages but also some drawbacks. Prolog provides an excellent tool for prototyping new ideas in object-orientation, testified by the amount of research being done in object-oriented logic programming systems [24, 25]. Code can be interpreted either as procedures or as data, enabling easy meta-programming. Interactive programming environments allow quick prototyping of new ideas. However, because Prolog is a logic programming language, it also implies an additional effort when translating Logtalk concepts and ideas to the more common jargon of languages like Smalltalk, C++ or Java. Prolog programs are made of predicate directives and definitions. Predicate definitions describe what we know to be true about the application domain. Indeed, Prolog programs can be seen as executable specifications. Predicate directives describe predicate properties, determining how predicate definitions are compiled. For those not familiar with logic programming and Prolog in particular, in this paper we may equate Prolog/Logtalk predicates with C++ object members. That is, predicates can play the role of both methods and variables. Mutable state is usually represented by dynamic predicates, i.e. predicates whose definition may be changed during runtime.

2.1 Logtalk Basics

One of the Logtalk main goals is to provide Prolog with predicate encapsulation capabilities. In particular, Logtalk objects must be able to supersede the current features of the current Prolog module system [26]. To ensure a smooth learning curve, Logtalk uses standard Prolog syntax with the addition of a few directives and message sending operators. The simplest example of a Logtalk object will be something like:

```
:- object(foo).

   :- public(bar/1).

   bar(1).
   bar(2).

:- end_object.
```

The first argument of the opening directive is the object identifier. By default, all object predicates are private. We use the `public/1` directive to declare public predicates. This is an example of a one-of-a-kind object. It does not belong to any object hierarchy or have any dependencies on other objects, categories, or protocols. We can put this object definition on a source file or create the object dynamically at runtime using the Logtalk built-in predicate `create_object/4`:

```
| ?- create_object(foo, [], [public(bar/1)], [bar(1), bar(2)]).
```

Either way, after compiling and loading the object or after creating it dynamically, we can try queries like:

```
| ?- foo::bar(X).

X = 1 ;
X = 2 ;
no (more) solutions
```

The `::/2` infix operator used above is our message sending operator. There are also complementary syntax constructs allowing us to send a set of messages to an object or the same message to a set of objects in a compact way.

Objects can also be defined by extending other objects, resulting in a prototype hierarchy. For example:

```
:- object(bar,
      extends(foo)).

      bar(X) :-
         ^^bar(X).
      bar(3).
      bar(4).

:- end_object.
```

The `^^/1` prefix operator enables us to call inherited definitions when redefining a predicate (similar to the Smalltalk *super* call):

```
| ?- bar::bar(X).

X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
no (more) solutions
```

Objects may also instantiate and/or specialize other objects, defining typical class-based hierarchies, with or without metaclasses. For example:

```
:- object(polygon,
      instantiates(metaclass),
      specializes(object)).

      :- public(area/1).
      :- public(perimeter/1).
      ...
```

```
        perimeter(P) :-
           ::nsides(N),
           ::slength(L),
           P is N*L.

        ...

    :- end_object.
```

The `::/1` prefix operator allow the sending of messages to *self*, that is, to the object that received the original message.

Predicate declarations can be contained inside objects or in independently defined protocols:

```
    :- protocol(polygon_interface).

        :- public(area/1).
        :- public(perimeter/1).
        ...

    :- end_object.
```

We can then rewrite the example above as:

```
    :- object(polygon,
        implements(polygon_interface),
        instantiates(metaclass),
        specializes(object)).

        ...

    :- end_object.
```

By default, inheritance is public. We can however restrict the scope of inherited predicates by prefixing the name of the instantiated, specialized, or extended object or the name of the implemented protocol with the keywords `public`, `protected` or `private`. For example:

```
    :- object(stack,
        implements(public::stack_protocol),
        extends(protected::list)).

        ...

    :- end_object.
```

This feature works similar to C++ restricted inheritance. Dynamic state can be represented by dynamic predicates, that is, predicates whose definition can be changed at runtime. Logtalk dynamic predicates can thus play the role of object variables in languages like C++ or Java. Instead of an assignment primitive, Logtalk defines a set of built-in methods that allow us to assert and retract clauses for dynamic predicates. Being methods, they always modify the predicate definition contained in the target of the assert/retract message, as long as the predicate is within the scope of the object that sends the corresponding message. For instance, assume that we add the following directive to our first example:

```
:- dynamic(bar/1).
```

If our code contains a call such as:

```
..., ::assertz(bar(5)), ...
```

This message will assert a new clause as the last one for the predicate `bar/1` in *self*. Assignment can be performed by first retracting all predicate clauses and then asserting the new definition:

```
...,
::retractall(bar(_)),
::assertz(bar(5)),
...,
```

Logtalk assert and retract built-in methods make features likes Smalltalk class variables trivial to implement. For further reading regarding Logtalk syntax, semantics, and remaining features please see [17]. Better yet, download the system from [16] and play with the included examples.

2.2 Logtalk Programming versus Traditional Object-Oriented Programming

Logtalk extends Prolog to objects, in a similar way as CLOS extends LISP or Objective-C extends C. There are however three significant differences. The first concerns the predicate notion that is inherited from Prolog. Predicates remove the dichotomy between state and behavior. A predicate states what is true about a domain. We can use it to represent either an attribute or a method but we are not forced to make such a distinction. A consequence is that we no longer need separate inheritance or scope rules for state and behavior. For example, methods can be defined in instances. State can be easily shared without the need of introducing concepts like class or shared variables. The second distinction concerns the nature of classes and objects. We are no longer constrained to define classes as static entities and instances as runtime-only objects. Logtalk source files can describe either classes or instances. Both types of objects can be either static or dynamically created at runtime. Third, Logtalk was designed as a neutral, unbiased language, to support both prototype and class-based programming. We can stick to traditional class-based designs or we can use prototypes. We may choose to implement full reflective systems or simply use our classes as instance factories. We can define a single hierarchy as in Smalltalk or Java or we can have multiple, independent, hierarchies like in C++. We can restrict ourselves to single inheritance or take advantage of multi-inheritance support. Contrast these features with languages like C++, Java, or Smalltalk that are defined as class-based languages, with a clear distinction between variables and methods, about what is static and what is dynamic, about what must be at achieved compile time or can be performed at runtime.

3 Category Concept and Implementation

The starting point for the Logtalk category concept comes from the Smalltalk-80 language where methods can be partitioned into named functional categories. However, Smalltalk-80 categories have only a documentation meaning, used to organize source code, and implemented by the language class browser. Logtalk extends this concept by making a category an encapsulation unit, at the same level as objects or protocols. The main idea is that we can compose a set of categories in order to define new objects, enabling code reuse without using inheritance or instance-variable based composition. Conversely, any object may be split in a set of categories. The splitting is straightforward and the code only requires elementary changes if predicates in one category need to call predicates in other category (because we are no longer calling code in the same encapsulation unit). Categories main purpose is the encapsulation of functional sets of predicates, serving as object building blocks. Categories are fully implemented in the current Logtalk release, providing the following properties:

1. Categories have the same encapsulation power as objects: a category may contain both predicates directives and definitions. A category may also implement one or more protocols.

2. Category predicates are reused by importing the category into an object. The predicates are virtually added to the object’s protocol, along with any local object predicates, without any code duplication.

3. Categories provide runtime transparency: predicates added via a category are inherited by all the descendants of the importing object and can be called, redefined or specialized like any other object predicate[3]. One important consequence of this property is that an object can be factored in categories without breaking its clients or its descendants.

[3] Nevertheless, Logtalk includes reflection methods that enable us to determine if a predicate is defined in either a category or an object.

4. An object may import one or more categories. Any number of objects can import a category. A category is always shared between all importing objects with no duplication of code.

5. A category may declare and use dynamic predicates. In this case, each importing object will have its own set of clauses for each dynamic predicate. This enables a category to define and manage (object) state.

6. An object can restrict the scope of imported category predicates by prefixing the category name with one of the keywords `public`, `protected`, or `private`, in a similar way to public, protected and private inheritance. By default, importation is public if the scope keyword is omitted.

7. Categories are compilation units; i.e., they are independently compiled from importing objects or implemented protocols, enabling incremental compilation.

8. There are no inheritance or importation mechanisms for categories. They can not inherit from, or be inherited by, other categories or objects. They can not also import, or be imported by, other categories. It is thus both meaningless and an error to send a message to a category.

9. Categories enable an object to be virtually assembled only when created or loaded to memory. By importing one or more categories, an object will have a distributed dictionary of predicates composed of its own dictionary and of the dictionaries of each imported category. An object may then be updated simply by updating an imported category, without any need to recompile it or to access its source code.

10. Both classes and prototypes can import a category at the same time; its implementation is independent of the implementation of either type of object. The use of categories is orthogonal to the choice of the most appropriated object concept, enabling the development of category libraries that can be reused in either prototype or class based designs.

11. Categories can be dynamically created and abolished at runtime (just like objects or protocols). Note however that runtime creation of new categories does not imply ant kind of instantiation process: categories are not objects. Instead, Logtalk uses the same code self-modifying features found in Prolog.

Categories can be seen as a dual concept of Logtalk protocols: protocols provide interface reuse, while categories enable implementation reuse without using inheritance. Both protocols and categories are intended to encapsulate cohesive data. Both are used as building blocks in the definition of new, possibly unrelated, objects, allowing finer grain reuse. Also, similar to a protocol, a category can be imported by several objects and an object can import several categories. However, while protocols can extend other protocols, a category can not be constructed as a composition of other categories. This can be seen as a limitation that constrains categories to be used as an enhanced virtual import mechanism, instead of a full blow separation of concerns or composition mechanism [27]. Nevertheless, despite its simplicity, categories enjoy several useful properties. But also because of its simplicity, categories are very easy to implement using current object compiling technology.

Conflicts may arise if two imported categories define the same predicate. This is akin to multi–inheritance conflicts but much simpler to spot and solve because categories do not inherit from other categories or objects. In addition, a category is mainly used to encapsulate a set of functionally cohesive predicates, thus minimizing the chances of name conflicts. The current Logtalk version uses a simple depth-first lookup when searching for a predicate, implicitly solving any possible name clashes.

If a category defines the same predicate as an object into which it is imported, the object predicate overrides the predicate definition defined in the category. Note however that categories are primarily object building blocks, not object re-factoring solutions, thus minimizing this kind of conflicts.

3.1 Related Concepts

The Flavors [28] system, an object-oriented extension to LISP [29], introduced the concept of mixins [30], a coding convention that uses abstract sub-classes to specialize behavior in parent classes. Mixins are combined, using multi-inheritance, with other mixins to build regular classes. Mixins, like Logtalk categories, often encapsulate a set of functionally cohesive methods and attributes. However, categories are reused by composition while mixins are reused through multi-inheritance. In a language that supports multi-inheritance, mixins enable flexible reusing without the need to introducing a new kind of entity. Another important difference is that, while mixins rely on specialization of parent methods (using the *call-next-method* primitive), categories do not need to depend on importing objects. Categories are often used to encapsulate independent, self-contained code, resulting in more flexible and powerful reusing mechanism. We can only use a mixin if the class that inherits the mixin also inherits a parent that defines the method specialized by the mixin. No such constraints exist in reusing Logtalk categories.

Regarding Smalltalk, most systems define interface primitives for loading and saving fragments of code. These primitives, historically named *FileIn* and *FileOut*, enable the programmer to add or remove methods and variables from a class. Recent Smalltalk implementations like VisualWorks improve upon this idea introducing the concept of *parcel*. The fragments of code or parcels correspond often to a category or a set of categories, giving a useful operational meaning to an otherwise documentation only concept of categories. However, a Smalltalk category is always associated with a specific class and can not be shared between two or more classes. Logtalk removes this restriction, generalizing the category concept to enable a category to be imported into any object.

The Objective-C language also implements a category concept but intended as a way to extend an existing class with new methods, even when the extended class source code is not available. It can be seen as an alternative to a sub-class. One can not however extend a class other than the one specified in the category declaration. This differs from Logtalk where categories are independent of objects and any category can be imported by any object. While Objective-C categories are designed to extend existing code, Logtalk categories are object building blocks. Although two different concepts, aiming at different goals, they share some important properties such as run-time transparency, encapsulation of related methods, incremental compilation, and easier maintenance of complex objects.

The Self [31, 32] prototype programming language defines a concept of *traits* prototypes that are used to store common behavior, playing a similar role to classes in class–based languages. One drawback of traits is that although they are objects, they cannot answer most messages because the corresponding methods need access to slots only available in descendant prototypes [33]. Logtalk categories can play the role of traits as a way to store shared methods with the advantage that is not possible to send a message to a category.

As we have written in the previous section, Logtalk categories are an evolution of the Smalltalk-80 functional category concept, taking what is essentially a browser documentation feature and transforming it in a code reuse language mechanism. As such, it compares favorably to other reuse mechanisms at the same level as mixins, multi-inheritance, or instance-based composition. It does not intend however to compete with higher level solutions to composition of separation of concerns issues like aspect-oriented programming [34], subject-oriented programming [35], or binary component adaptation [36] among others.

3.2 Implementation

The current Logtalk version contains a full implementation of all the properties of the category concept as described in this paper. The system also includes several examples of the use of categories, some of them presented here in the next session. It should be noted that the Prolog/Logtalk features of interpreting code either as data or executable procedures, combined with easy conversion between data and code, arguably makes it easier to implement features like categories when compared to languages like C++, Smalltalk, or Java. The Logtalk source files can be downloaded from the Logtalk web site to close examine of the implementation details.

Objects that import categories and/or implement protocols have a distributed dictionary of predicates. That is, in addition to a list of local predicates, objects have links, defined at compile time, to the dictionaries of imported categories and implemented protocols. These links are always searched before inheritance links. Categories (and protocols) are compiled such that the encapsulated code can be shared and used by several objects (either prototypes or instances/classes) at the same time. This is accomplished in two steps. First, category predicates are compiled like object predicates, with extended arguments for execution context information. This extended arguments include *self* (object that received the original message), *this* (object, importing the category, that virtually contains the predicate under execution), and *sender* (object that has sent the original message). Second, at runtime, the object–category dictionary links propagate the current execution context to the category predicates, enabling them to be used like they have been defined in the importing object. In the case of dynamic predicates (that is, predicates whose definition can be modified at runtime), the implementation of the predefined methods that allow us to add, change and delete definitions ensure that each importing object will have its own set of definitions.

An important point is the performance cost of adding categories to an object-oriented language. If all imported categories only contain predicate directives, then performance should be similar to languages like Objective-C or Java that implement multi-inheritance of protocols. In the more common situation where a category contains both predicate directives and definitions, searching for a predicate can require looking inside an imported category. This has a small performance penalty that is proportional to the number of imported categories and results from the need to access several encapsulation units. Because categories can provide alternative solutions to the use of multi-inheritance (see the `points` example in the next section), we should also compare the costs of these two reusing methods. While an inheritance link may lead to several other inheritance links, following an imported category link implies only one level of indirection when searching an importing object predicate dictionary: a category does not inherit or import code from other categories or objects. This ensures that a design using single-inheritance and categories has a more predicable and better method lookup performance than an equivalent multi-inheritance solution.

3.3 Syntax

In order to make the examples in the next section easier to grasp, let us briefly describe the Logtalk syntax for defining and using categories, complementing the introduction given in the previous session. To define a new category we use a starting directive, either `category/1`[4] or `category/2`, and an ending directive, `end_category/0`, to encapsulate the predicates directives and clauses:

```
:- category(Ctg).

    ...

:- end_ category.
```

[4] In Prolog and in Logtalk directives and predicates are identified by <name>/<number of arguments>.

If a category implements one or more protocols, we use the `category/2` opening directive. Protocols implemented by a category are listed after the category name:

```
:- category(Ctg,
      implements(Ptc1, Ptc2, ...)).

      ...

:- end_ category.
```

To import a set of categories into an object we write:

```
:- object(Obj,
      imports(Ctg1, Ctg2, ...)).

      ...

:- end_object.
```

We can restrict the scope of the imported category predicates by using a scope keyword (either `public`, `protected`, or `private`). For example:

```
:- object(Obj,
      imports(protected::Ctg)).

      ...

:- end_object.
```

By default, if the scope keyword is omitted, category importation is public.

4 Examples

The following examples compare instance- with category-based composition and illustrate how categories can be used for sharing code between selected objects, providing alternative solutions to multi-inheritance. The current Logtalk distribution contains several more examples of the use of categories.

4.1 Splitting an Object in Categories

Let us start with the most basic benefits of categories: code documentation and organization. Most Smalltalk implementations already classify methods in several functional categories. In these cases, splitting a class using the Logtalk category concept is a trivial job. Only elementary code changes may be needed if a method in one category needs to call a method in other category. Let us turn instead our attention to Java. Take for example the class `Float`, contained in the `java.lang` package [12]. This class declares twenty-three new methods that can be easily classified in four categories named `constructors`, `comparing`, `testing` and `converting` as follows:

```
constructors
Float(double value)
Float(float value)
Float(String s)

comparing
compareTo(Object o)
compareTo(Float anotherFloat)
equals(Object obj)

testing
isNaN(),isNaN(float v)
isInfinite(),isInfinite(float v)

converting
hashCode()
toString(), toString(float f)
valueOf(String s)
...
```

The immediate benefit is to the programmers browsing the class to locate a method to perform a specific type of service, and results from the simple fact of classifying the methods in appropriated functional categories. The Java docs present them in alphabetical order, handy only if we are looking for the details of a known method. It should be noted however that, just as a class hierarchy implicitly reflects a specific classification point-of-view over a set of objects, there is usually more than one way to split a set of methods into functional categories.

4.2 Categories as a Complementary Composition Tool

The category concept here presented provides a composition mechanism different from what we may call instance-variable composition. Usually, composition is accomplished by storing references to other objects in instance variables. Comparing the two mechanisms shows they are complementary, addressing different needs, rather than competing ways of doing composition. Instance-variable based composition is mainly used to implement part-of hierarchies. The implied level of indirection is often used to our advantage to control which methods (if any) are made available to the clients of the container object. By contrast, the methods imported from a category are transparently used and are conceptually at the same level of any object-defined method. Instance-variable composition also implies the creation of new objects every time a container object is instantiated and, therefore, a policy to control the process. It is however free from name clashes that may affect multi-inheritance or category-based composition solutions.

Let us start by defining a category that implements a set of predicates for handling a dictionary of attributes. We will need public predicates to set, get, and delete attributes, and a private dynamic predicate to store the dictionary entries. Let us name these predicates `set_attribute/2` and `get_attribute/2`, for getting and setting an attribute value, `del_attribute/2` for deleting attributes, and `attr_/2`, for storing the attribute–value pairs:

```
:- category(attributes).

    :- public(set_attribute/2).    % set a pair attr-value
    :- public(get_attribute/2).    % test/get a pair attr-value
    :- public(del_attribute/2).    % delete a pair attr-value
    :- private(attr_/2).           % attributes storage
    :- dynamic(attr_/2).

    set_attribute(Attr, Value):-
       ::retractall(attr_(Attr, _)),
       ::assertz(attr_(Attr, Value)).

    get_attribute(Attr, Value):-
       ::attr_(Attr, Value).

    del_attribute(Attr, Value):-
       ::retract(attr_(Attr, Value)).

:- end_category.
```

If needed, we can put the predicate directives inside a protocol that will be implemented by the category:

```
:- category(attributes,
    implements(attributes_protocol)).

    ...

:- end_category.
```

We reuse the category predicates by importing them into an object:

```
:- object(person,
    imports(attributes)).

    ...

:- end_object.
```

After compiling and loading this object and our category, we can now try queries like:

```
| ?- person::(set_attribute(name, paulo), set_attribute(gender,
male)).
yes

| ?- person::get_attribute(Attr, Value).
Attribute = name, Value = paulo ;
Attribute = gender, Value = male ;
no
```

Note that the attributes category interface is now part of the person object interface. Most object-oriented programming language libraries provide dictionary classes that we can reuse in our applications, either by multi-inheritance or by composition. In this example, multi-inheritance would result in viewing person as a kind of dictionary, hardly an elegant solution. With instance-based composition, glue code will be needed to add the desired dictionary methods to the public interface of person. Category-based composition thus provides an alternative solution without any of these problems. Moreover, the resulting category could be reused in other places of our application or in other applications.

To further illustrate the differences between instance variable-based and category-based composition let us give another example using input/output operations. Assuming a stream-based input/output model, any object may need to define, redirect, open, read, write or close new streams. Using a category-based approach, we may start by defining a `streaming` category, containing predicates to maintain a dictionary of currently defined streams:

```
:- category(streaming).

      :- public(stream/2).     % test/get pair name-stream
      :- public(set_stream/2). % define a new pair name-stream
      ...

:- end_category.
```

This category may also implement common stream operations. Any object whose interface must include stream input/output operations can then import this category:

```
:- object(an_object,
      imports(streaming)).

      ...

:- end_object.
```

This way we are able, for example, to query the object (or its descendants) about the streams it defines by using messages like:

```
| ?- an_object::stream(Name, Stream).
```

If an instance variable-based solution is preferred or needed, we can still reuse the `streaming` category by first importing the category into a class:

```
:- object(streams,
      imports(streaming),
      instantiates(class),     % some suitable metaclass
      specilizes(object)).     % some suitable inheritance root

      ...

:- end_ object.
```

We can now store instances of this object in instance-variables of any object that needs to perform stream input/output operations. For example:

```
:- object(an_object,
      imports(attributes)).    % from the previous example

      init :-
         streams::new(Strs),
         ::set_attribute(streams, Strs),
         ...

      ...

:- end_ object.
```

However, the streaming methods can no longer be used directly:

```
| ?- an_object::set_stream(Name, Stream).

uncaught exception:
error(existence_error(
      predicate_declaration, set_stream(Name, Stream)),
      an_object::set_stream(Name, Stream), user)
```

Messages like this will now generate unknown message exceptions because the streaming protocol is no longer part of the object protocol.

4.3 Hierarchy Relations

One of the Logtalk companion examples defines a set of categories implementing methods for inspecting hierarchy relations between objects. Some methods can be defined for both prototype and class hierarchies and can be abstracted in a common protocol:

```
:- protocol(hierarchyp).

      :- public(leaf/1).       % test/get an hierarchy leaf
      :- public(leaves/1).     % get list of all hierarchy leaves
      ...

:- end_protocol.
```

For prototype hierarchies, we can define methods such as `parent/1`, `ancestor/1`, or `descendant/1`:

```
:- category(p_hierarchy,
      implements(hierarchyp)).

      :- public(ancestor/1).   % test/get an ancestor prototype
      :- public(descendant/1). % test/get a descendant prototype
      :- public(parent/1).     % test/get a parent prototype
      ...

:- end_category.
```

While for instance/class relations we can have methods like `class/1`, `superclass/1`, or `instance/1`:

```
:- category(ic_hierarchy,
      implements(hierarchyp)).

      :- public(class/1).      % test/get an instance class
      :- public(instance/1).   % test/get a class instance
      :- public(superclass/1). % test/get a class superclass
      ...

:- end_category.
```

Although these methods are potentially useful for any object, most objects will never use them. Most applications do not need to perform reflective computations. In languages like Smalltalk or Java, this kind of methods must be added to the root object in order to be available for any object that may need them. By encapsulating these methods in a category, they can be added to the interface of only those objects that really need them.

4.4 Monitoring Category

Besides integrating logic and object-oriented programming, Logtalk also supports event-driven programming where an event is generated every time a message is exchanged between objects. Any object may act as a monitor for a registered event. A minimal monitor protocol consists only of a callback method, but more sophisticated behavior is possible. For instance, an object may need to keep a dictionary of events that can be modified, activated, and suspended. However, not all application objects will act as monitors and, among those they do, some may only need basic behavior. Encapsulating the monitor methods in a root object will ensure the requirement that any object may perform a monitor role but will also just clutter the interface of non-monitor objects. Moreover, not all applications use event-driven programming. Defining a monitoring category solves these problems easily:

```
:- category(monitoring).

      :- public(activate/0).    % activate events, start monitoring
      :- public(add_event/4).   % define a new event
      :- public(del_event/4).   % delete a defined event
      :- public(event/4).       % test/get a defined event
      :- public(reset/0).       % stop and delete all events
      :- public(suspend/0).     % suspend monitoring
      ...

:- end_category.
```

Any object that needs more complex monitor behavior just needs to import this category:

```
:- object(my_monitor,
      imports(monitoring)).

      ...

:- end_object.
```

Alternatively, the root of any sub-hierarchy of monitor objects may import the category. This way, objects that will never perform the role of monitors will not need to inherit a set of useless methods. Applications that do not use event–driven programming will not need to include code that will never be called.

4.5 Points

This example shows how categories may be used as an alternative to multi-inheritance solutions[5]. The description of the original problem can be found in [37]. Assume that we want to represent points in a two-dimensional space. We can start by creating a `point` class defining a method `move/2` to translate a point to a new position, and a method `print/0` that outputs the current position[6]:

```
:- object(point,
      instantiates(class),
      specializes(object)).

      :- public(move/2).        % move point to a new position
      :- public(position/2).    % test/get point position
      :- public(print/0).       % output current point position

      :- private(xy_/2).        % point position storage
      :- dynamic(xy_/2).

      move(X, Y) :-
         ::retractall(xy_(_, _)),
         ::assertz(xy_(X, Y)).

      position(X, Y) :-
         ::xy_(X, Y).

      print :-
         self(Self),
         ::xy_(X, Y),
         writeq(Self), write(' @ '), write((X, Y)), nl.

:- end_object.
```

From this base class, we want to derive two sub-classes: `bd_point` and `hst_point`. Instances of `bd_point` can only move around in a restricted area. Instances of `hst_point` remember its previous positions. The new classes are easily defined by specialization of the `move/2` and `print/0` methods:

```
:- object(bd_point,
      instantiates(class),
      specializes(point)).

      :- private(bds_/3).       % coordinate bounds storage
      :- dynamic(bds_/3).
```

[5] The full source code of this example is available with the current Logtalk release.

[6] To save space, code related to instance initialization is omitted in this example.

```
      move(X, Y) :-
         ::bds_(x, MinX, MaxX),
         X >= MinX, X =< MaxX,
         ::bds_(y, MinY, MaxY),
         Y >= MinY, Y =< MaxY,
         ^^move(X, Y).

      print :-
         ::bds_(x, MinX, MaxX),
         writeq(bds(x)), write(': '), write((MinX, MaxX)), nl,
         ::bds_(y, MinY, MaxY),
         writeq(bds(y)), write(': '), write((MinY, MaxY)), nl,
         ^^print.

:- end_object.
```

Similar for the `hst_point` class:

```
:- object(hst_point,
      instantiates(class),
      specializes(point)).

      :- private(hst_/1).    % position history storage
      :- dynamic(hst_/1).

      move(X, Y) :-
         ::position(X0, Y0),
         ^^move(X, Y),
         ::retract(hst_(Hst)),
         ::assertz(hst_([(X0,Y0)| Hst])).

      print :-
         ::hst_(Hst),
         write('history: '), write(Hst), nl.
         ^^print.

:- end_object.
```

Now assume that we want to define another sub-class, named `bd_hst_point`, which combines the behavior of both `bd_point` and `hst_point`. This suggests a multiple inheritance solution: `bd_hst_point` clearly specialize both `bd_point` and `hst_point`, sub-classes of `point`. However, this solution, even if possible, hides several problems. The first obstacle is that the *bounded* and the *history* behavior are embedded in the specialization of methods `move/2` and `print/0`. Defining new methods such as `check_bds/2` and `print_bds/2` in class `bd_point` and `add_to_hst/2` and `print_hst/0` in class `hst_point` can easily solve this particular problem. A bigger problem is that the basic behavior for *moving* or *printing* a point is defined in class point. However, because the corresponding methods are redefined in classes bd_point and hst_point, how does one call the original definitions stored in point? Note that if the methods move/2 and print/0 are inherited from both hst_point and bd_point then a point will be moved and printed twice. If the inheritance is carried out, for each method, only from one of the superclasses, then we will be breaking the problem symmetry. The class bd_hst_point could build its own definitions of methods move/2 and print/0, adding to the inherited definitions from one of the superclasses the calls to the methods specific of the other superclass. However, this solution is also problematic. Let us assume that the method move/2 is inherited from class hst_point (by using some suitable *super* call). Then, any change on the definition of the same method in class bd_point will be ignored by bd_hst_point. In a large program, such problems can easily get unnoticed because the symmetry suggested by the multiple inheritance design is not reflected by the actual implementation. Such problem could be avoided by explicitly adding the class point as a base class for bd_hst_point. For example, in Eiffel we will need to rename (and discard!) the conflicting inherited methods of both base classes:

```
class
   bd_hst_point
inherit
   bd_point
            rename
                  move as bp_move,
                  print as bp_print
            end
   hst_point
      rename
                  move as hp_move,
                  print as hp_print
            end
   point
      redefine move, print end
feature
   print is
   do
            precursor
            print_bds
            print_hst
   end
   ...
end
```

This solution could also be implemented in C++ using virtual base classes:

```
class bd_hst_point
      : public virtual point, public bd_point, public hst_point {
   ...
   void print();
   ...
}

void bd_hst_point::print()
{
   point::print();
   bd_point::print();
   hst_point::print();
}
```

This way the class `point` will provide the basic behavior for the `move/2` and `print/0` methods. These two methods are redefined in order to include the needed calls to the methods inherited from classes `bd_point` and `hst_point` that implement the *bounded* and the *history* behavior.

In Logtalk, we can use categories to solve this problem in a clean and extensible way without using multi-inheritance. In order to do so, we will start by defining two new categories, `bd_coord` and `point_hst`. The `bd_coord` category will contain the methods associated with point coordinate bounds:

```
:- category(bd_coord).

      :- public(set_bds/3).     % store a coordinate bounds
      :- public(bds/3).         % test/get coordinate bounds
      :- public(check_bds/2).   % checks coordinate value
      :- public(print_bds/1).   % print a coordinate bounds

      :- private(bds_/3).       % coordinate bounds storage
      :- dynamic(bds_/3).

      set_bds(Coord, Min, Max) :-
         ::retractall(bds_(Coord, _, _)),
         ::assertz(bds_(Coord, Min, Max)).

      bds(Coord, Min, Max) :-
         ::bds_(Coord, Min, Max).

      check_bds(Coord, Value) :-
         ::bds_(Coord, Min, Max),
         Value >= Min, Value =< Max.
```

```
        print_bds(Coord) :-
           ::bds_(Coord, Min, Max),
           writeq(bds(Coord)), write(': '), write((Min, Max)), nl.

  :- end_category.
```

The methods for storing previous point positions will be encapsulated in the `point_hst` category:

```
  :- category(point_hst).

        :- public(add_to_hst/1).     % store a point position
        :- public(init_hst/1).       % initialize position history
        :- public(hst/1).            % get the point history
        :- public(print_hst/0).      % print the point history

        :- private(hst_/1).          % position history storage
        :- dynamic(hst_/1).

        add_to_hst(Pos) :-
           ::retract(hst_(Hst)),
           ::assertz(hst_([Pos| Hst])).

        init_hst(Hst) :-
           ::retractall(hst_(_)),
           ::assertz(hst_(Hst)).

        hst(Hst) :-
           ::hst_(Hst).

        print_hst :-
           ::hst_(Hst),
           write('history: '), write(Hst), nl.

  :- end_category.
```

Each one of the `bd_point`, `hst_point` and `bd_hst_point` classes will import the related categories in order to provide the intended behavior:

```
  :- object(bd_point,
        imports(bd_coord),
        instantiates(class),
        specializes(point)).

        move(X, Y) :-
           ::check_bds(x, X),
           ::check_bds(y, Y),
           ^^move(X, Y).
        print :-
           ::print_bds(x),
           ::print_bds(y),
           ^^print.

  :- end_object.
```

Likewise for the hst_point class:

```
    :- object(hst_point,
        imports(point_hst),
        instantiates(class),
        specializes(point)).

        move(X, Y) :-
            ::position(X0, Y0),
            ^^move(X, Y),
            ::add_to_hst((X0, Y0)).

        print :-
            ::print_hst,
            ^^print.

    :- end_object.
```

The bd_hst_point class will be defined as a point subclass, importing both point_hst and bd_coord categories:

```
    :- object(bd_hst_point,
        imports(bd_coord, point_hst),
        instantiates(class),
        specializes(point)).

        move(X, Y) :-
            ::check_bds(x, X),
            ::check_bds(y, Y),
            ::position(X0, Y0),
            ^^move(X, Y),
            ::add_to_hst((X0, Y0)).

        print :-
            ::print_bds(x),
            ::print_bds(y),
            ::print_hst,
            ^^print.

    :- end_object.
```

Note that the redefinition of our classes using the newly defined categories is transparent to the classes clients and descendants. Also, `bd_hst_point` is independent of both `bd_point` and `hst_point` yet it shares their behavior via the common imported categories. This solution can be easily extended if we need to add other point flavors besides coordinate bounds or position history. Using categories, we just import into an object each category implementing a desired flavor, no matter how many flavors and flavors combinations we may have. Contrast this with a multi-inheritance solution where each new flavor will need to be implemented as a new sub-class, possibly inheriting from several other flavor sub-classes, resulting in high levels of coupling between objects. We have thus found not only an alternative solution to the use of multi-inheritance but also a better one: a way to freely combine multiple orthogonal implementations without applying multi-inheritance mechanisms.

To end this example here are some possible messages sent using the Logtalk/Prolog top-level interpreter:

```
| ?- point::new(P,[xy-(1, 3)]), P::(print,  move(7, 4), print).

p1 @ (1, 3)

p1 @ (7, 4)

P = p1
yes
```

Similar messages but with bounds on coordinate values:

```
| ?- bd_point::new(P,[xy-(1, 3), bds(x)-(0, 13), bds(y)-(-7, 7)]),
P::(print, move(7, 4), print).

bds(x): 0,13
bds(y): -7,7
bp11 @ (1, 3)

bds(x): 0,13
bds(y): -7,7
bp2 @ (7, 4)

P = bp2
yes
```

Same problem but storing the history of past point positions:

```
| ?- hst_point::new(P,[xy-(1, 3)]), P::(print, move(7, 4), print).

history: []
hp3 @ (1, 3)
```

```
   history: [(1,3)]
   hp3 @ (7, 4)

   P = hp3
   yes
```

Same problem but with bounds on coordinate values and storing past positions:

```
   | ?- bd_hst_point::new(P,[xy-(1, 3), bds(x)-(0, 13), bds(y)-(-7,
7)]), P::(print, move(7, 4), print).

   bds(x): 0,13
   bds(y): -7,7
   history: []
   bhp4 @ (1, 3)

   bds(x): 0,13
   bds(y): -7,7
   history: [(1,3)]
   bhp4 @ (7, 4)

   P = bhp4
   yes
```

5 Conclusions

Logtalk categories are a simple and natural evolution of the original Smalltalk category concept, easily implemented using the same compilation techniques that we apply to objects. Despite its simplicity, categories enable good solutions for reusing sets of utility methods and multi-inheritance designs.

The category concept is being actively used in the development of the Logtalk standard library and is being evaluated by writing Logtalk applications. Logtalk is an object-oriented programming research language and, as such, a natural environment for trying out new ideas. However, adding a new feature to an established language must always be carefully pondered. We believe that the category concept here presented brings several important advantages to object-oriented languages and, in particular, to single inheritance languages. Summarizing our main results, categories can be used to:

· Provide alternative solutions to the use of multi-inheritance for single-inheritance languages. Even for multi-inheritance languages, categories enable elegant implementations that minimize object coupling.

· Complement instance variable based composition by providing a composition mechanism where composed methods are at the same level as the container object methods, with full run-time transparency. Category imported methods are called, redefined, and otherwise used like any container object method.

· Enable different, unrelated objects, to share and reuse methods without using inheritance. This way methods can be made available to only those objects that really need them, avoiding large root objects populated with code that most descendants will never use.

· Split complex objects into a set of more manageable components, each containing a functionally cohesive set of methods that can be independently developed, compiled, and reused. Besides the advantages of incremental compilation, categories also make it possible to update an object without accessing its full source code.

· Encapsulate code that does not fit the notion of, or does not make sense as, an object. Two good examples are the Smalltalk dependency mechanism and the Logtalk monitoring methods.

It is also important not to forget the benefits that are inherited from the original Smalltalk-80 concept of methods functional categories, regarding code documentation and organization. Tacked together, all these features promote cleaner and simpler object-oriented designs and help improve reuse across object-oriented applications.

6 Future Work

An interesting research path will be to implement categories in common languages like Java or Smalltalk, either by using a pre-processor approach like in Logtalk, or by modifying an existing compiler. A good candidate will be a language like Squeak, a Smalltalk system written using Smalltalk itself, making it easy to modify and try out new features. We expect categories to be easy to implement in dynamically type checked languages and, with some more work, in statically type checked languages like C++, retaining all the features listed in the conclusions.

Future work may also include extending the Logtalk concept of categories to include some of the features of Objective-C categories. More specifically, the possibility of augmenting a class protocol without modifying its source code. Note that Logtalk categories already enable us to update an object interface by updating an imported category, but the `imports` clause can not be added or changed at runtime. A possible solution may be to adopt a mechanism to establish an import link without requiring sources changes on either importing objects or imported categories. Another possible, more explicit and declarative solution, will be to allow an object to import a set of matching categories. If we extend the identity concept to allow category names with the same name but different arguments, we could then write:

```
:- object(an_obj,
      imports(a_ctg(_)).     % import all matching categories

      ...

:- end_object.
```

If we then have the following two matching categories:

```
:- category(a_ctg(foo)).

      :- public(foo/1).
      ...

:- end_category.

:- category(a_ctg(bar)).

      :- public(bar/1).
      ...

:- end_category.
```

Our object could then answer messages defined in all matching categories. For example:

```
| ?- an_obj::(foo(X), bar(Y)).
```

This solution is not difficult to implement but, as a language feature, it will probably be a source of misunderstandings because it uses the same syntax of parametric objects with a very different semantics. It is also not clear that its benefits will outweigh the added complexity.

References

1. Goldberg, A., Robson, D., *Smalltalk-80 The language and its implementation*, Addison-Wesley Series in Computer Science, 1983
2. ObjectShare, Inc., *VisualWorks Application Developer’s Guide*, VisualWorks Software Release 3.0, 1998
3. ObjectShare, Inc. web site: http://www.objectshare.com
4. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Series in Computer Science, 1991 (Second Edition)
5. Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley, 1996

6. Cox, B., Novobilski, A., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 2nd edition, 1992
7. Apple Computer Technical Documentation: *Object-Oriented Programming and the Objective-C Language* http://www.apple.com/developer/SWTechPubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm
8. Shan, Y., Cargil, T., Cox, B., Cook, W., Loomis, M., Snyder, A., *Is Multiple Inheritance Essential to OOP? (Panel)*, in Proceedings OOPLSLA 93, ACM
9. Taenzer, D., Ganti, M., Podar, S., *Problems in Object-Oriented Software Reuse*, in Proceedings ECOOP 89, British Computer Society Workshop Series, Cambridge University Press
10. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A., *Back to the Future – The Story of Squeak, A Practical Smalltalk Written in Itself*, in Proceedings OOPSLA 97, ACM
11. Squeak home page, http://www.squeak.org/
12. Java web site, http://www.javasoft.com
13. Apple Computer Technical Documentation: *MacOS X Server – Foundation Framework Classes*, 1999
14. Cook, W. R., *Interfaces and Specifications for the Smalltalk-80 Collection Classes*, in Proceedings OOPSLA 92, ACM
15. Moura, P., Costa, E., *Logtalk: Object-Oriented Programming in Prolog*, in Proceedings 2nd Portuguese Conference on Object-Oriented Technology, 3i Consultores, Lisboa, 1994
16. Logtalk web site, http://www.ci.uc.pt/logtalk/logtalk.html
17. Logtalk 2.6 Documentation, Technical Report DMI-2000/1, University of Beira Interior, Portugal, 2000
18. Clocksin, W.F., Mellish, C.S., *Programming in Prolog*, Springer Verlag, New York, 1981
19. Open Source web site, http://www.opensource.org/
20. ISO/IEC DIS 13211-1 International Standard – *Programming Language Prolog Part I: General Core*, 1995
21. Rayudu, R. K., Samarasinghe, S., Maharaj, A., *A Cooperative hybrid algorithm for fault diagnosis in power transmission*, 2000 IEEE Power Engineering Society Winter Meeting, January 2000, Singapore
22. Zimányi, E., *Implementing materialization in Logtalk*, Technical Report YEROOS TR-97/09, Laboratoire de Bases de Données, Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, Switzerland, April 1997
23. Chen, Y., Shieh, J. S., KSL: A Specification Language for Knowledge Organization and Representation, International Conference Intelligent Information Management Systems, Washington, D.C., U.S.A., 1996
24. Alexiev, V., A (Not Very Much) Annotated Bibliography on Integrating Object-Oriented and Logic Programming, URL: ftp://menaik.cs.ualberta.ca/pub/oolog/

25. Davison, A., A survey of logic programming-based object oriented languages, Tech Report 92/3, Dept. of Computer Science, The University of Melbourne, Australia, 1993
26. ISO/IEC DIS 13211-21 International Standard – *Programming Language Prolog Part II: Modules*, 2000
27. P. Tarr, H. Ossher, W. Harrison, S.M. Sutton, Jr., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, Proceedings of the International Conference on Software Engineering (ICSE'99), May 1999
28. Moon, D. *Object-Oriented Programming in Flavors*, in Proceedings OOPSLA 86:1-8, ACM
29. Steele Jr, G. L., *Common LISP: The Language*, Digital Press, Bedford, Massachusetts, 1984
30. Bracha, G., *Mixin-based Inheritance*, in Proceedings OOPLSLA 90, ACM
31. Ungar, D., Smith, R. B., *Self: The Power of Simplicity*, in Lisp And Symbolic Computation, 4, 3, Kluwer Academic Publishers, 1991
32. Smith, R. B., Ungar, D., *Programming as an Experience: The Inspiration of Self*, in Proceedings ECOOP 95:303-330, Lecture Notes in Computer Science, Vol. 952, Springer–Verlag, 1995
33. Taivalsaari, A., *Classes vs. Prototypes: Some Philosophical and Historical Observations*, in Prototype–Based Programming: Concepts, Languages and Applications, Springer Verlag, 1999
34. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., Irwin, J., *Aspect-Oriented Programming*, in Proceedings ECOOP'97, Lecture Notes in Computer Science, Springer-Verlag, June 1997
35. H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal, *Specifying Subject-Oriented Composition*, Theory and Practice of Object Systems, Vol. 2, Nº 3, Wiley & Sons, 1996
36. Keller, R., Hoelzle, U., *Binary Component Adaptation*, in Proceedings ECOOP'98, Lecture Notes on Computer Science, Springer Verlag, 1998
37. SICStus Objects: http://www.sics.se/isl/sicstus.html