Electronic Journal of SADIO

http://www.dc.uba.ar/sadio/ejs

vol. 2, no. 1, pp. 17-29 (1999)

Reifying Design Patterns as Metalevel Constructs

Claudia Marcos[1,2] Marcelo Campo[1] Alain Pirotte[2]

[1]Univ. Nacional del Centro Prov. Bs. As. - Fac. Ciencias Exactas
ISISTAN - Grupo de Objetos y Visualización,
San Martín 57, (7000) Tandil, Bs. As., Argentina

[2]Université catholique de Louvain - UCL
IAG - Institute d'Administration et Gestion
1 Place Doyens, (1348) Louvain-la-Neuve, Belgium

E-Mail: {cmarcos, mcampo}@exa.unicen.edu.ar, pirotte@info.ucl.ac.be

Abstract

A design pattern describes a structure of communicating components that solves a commonly occurring design problem. Designing with patterns offers the possibility of raising the abstraction level at which design is performed, with improvements in clarity, understanding, and facility of maintenance of applications. However, in their most common presentation, design patterns are informal pieces of design process, which application is not reflected in the operational system, and the potential advantages of a more principled design are not realized. This work proposes to organize design in such a way that pattern applications remain explicit in the operational systems. A reflective architecture is proposed, where patterns are reified as metalevel constructs.

Keyword: design patterns - metalevel architecture - metaobjects -computational reflection

1. Introduction

The continuous evolution in information technologies leads to building information systems with ever more ambitious requirements. A fundamental property of software, in order to satisfy its evolution, is its adaptability. A system is said to be adaptable when it can be modified to satisfy new requirements at a reasonable cost compared to that of the required to re-implementing the system. The construction of such systems, however, is not easy, mainly because it requires much design experience. Experience and knowledge of experimented designers are captured by *design patterns* [Gamma, 1995]. One of their objectives is also to convey knowledge and experience from expert to novice designers.

A design pattern is an abstract solution of a design problem. A pattern prescribes a generic organization of classes, their roles and collaborations, and the distribution of responsibilities among them. The solution given by a pattern is abstract in that, each time a pattern is used in an application, the elements prescribed by the pattern must be identified: the name of the application classes and the methods that represent the elements prescribed by the pattern must be specified. No single definitive list of patterns has been proposed yet. Several catalogues of patterns have been published, [Gamma, 1995], [Buschmann, 1996], [Tichy, 1997], and each one gives a different classification and description of patterns.

Patterns make possible to deal with designs in a higher level of abstraction. This raises the level in which designers communicate and discuss design decisions. In this sense, one of the potential benefits that the use of patterns brings to software development is the understanding and maintenance of designs.

However, to be effectively useful in the maintenance phase, this is necessary that they are well documented and reflected in the program code. Unfortunately, this not always happens. In the literature, several approaches were proposed to make design patterns visible for the maintenance phase. For example, in [Campo, 1997] and [Lange, 1995] works design patterns are recovered from application code. In these works, particular characteristics of each pattern have been identified. These characteristics allow the identification of the potential patterns used in an application. The definition of pattern characteristics is a very hard task. It has been very difficult to make a differentiation among the characteristics of similar patterns. Furthermore, for some patterns, it was not possible to find them. To solve this problem, the authors took into account specific aspects of a particular implementation language. This allows finding more potential patterns but it depends on a specific programming language.

Another approach is the construction of a tool for automatically generating pattern code from information supplied by the designer [Eden, 1997a], [Eden, 1997b], [Budinsky, 1996], and [Bosh, 1996]. These tools introduce new notations by which the user can specify the patterns to be applied in the application. The tool uses this information to generate the code corresponding to those patterns in a specific programming language. Each time a pattern needs to be incorporated or deleted, the application code must be regenerated. This can produce errors in the classes affected by the pattern. Moreover, when new patterns are added those tools must be modified.

In this work we present an alternative approach to this problem based on a reflective model that allows the representation of design patterns as an explicit part of applications. A reflective metalevel architecture allows the representation of patterns as *first-class entities.* This architecture is composed of two levels: base level and metalevel. The metalevel represents design patterns and the base level contains the information of the specific application that is under development. A metaobject class at the metalevel represents each pattern. When a pattern is used in an application, the association between base level and metalevel is established. At run-time, the metalevel manipulates the objects at the base level according to the architecture defined by the patterns used in the application.

The rest of this paper is organized as follows. In the next section the main characteristics of the reflective metalevel architecture for the representation of design patterns are described. Section 3 presents, as an illustrative example, the representation of the *Composite* design pattern with the reflective architecture proposed. Section 4 presents the design of the reflective architecture. Finally, in Section 5 some conclusions are presented.

2. Reflective Architecture for Design Patterns

The design of a software artifact is an integral part of the essence of such artifact. However, particularly in object oriented systems, implementation languages do not provide means to reflect design decisions directly in the code. This limitation brings many well-known problems during the maintenance phase. These problems are due, at least in part, to the fact that is necessary to recognize that the design *lives* inside the running system, defining the way such system behaves at runtime. Statically, on the other hand, the design also defines how the different components of the system are structured to implement such behavior. In other words, the design can be seen as a meta-model of the software system, which defines how such software is structured and behaves. If a mean to represent and implement such a meta-model at runtime were available then an explicit trace of the involved design structures can be deduced from the code itself.

Taking into account this view, we propose a reflective model that allows representing design patterns as an explicit part of applications. In this model, patterns are reified as metalevel constructs, which provide the essential control structure that drives the program behavior during runtime.

Reflection is the capability of a computational system to reason about and act upon itself [Maes, 1987]. A reflective system incorporates data representing static and dynamic aspects of itself; this activity is called *reification.* This self-representation makes it possible for the system to answer questions about and support actions on it. In a reflective architecture, a computational system is viewed as incorporating an object part and a reflective part. The task of the object computation is to solve problems and return information about an external domain, while the task of the reflective is to solve problems and return information about the system itself.

The reflective architecture proposed has two levels: *metalevel,* or *reflective level,* and *base level* (Fig.1). The metalevel allows representing design patterns and the base level contains the specific information of the application under development. Application classes, their methods, and relationships among classes are designed at this

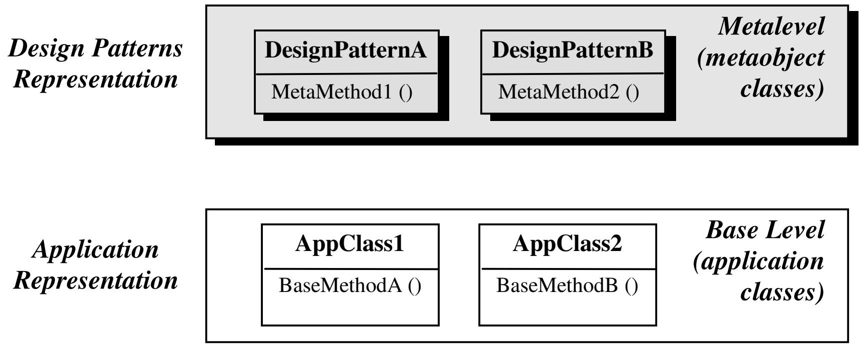

Fig.1 Representation of design patterns in the reflective architecture

level. At run-time, the metalevel manipulates the objects at the base level according to the architecture defined by the patterns used in the application.

Designing the proposed reflective architecture involves decisions about the representation of patterns at the metalevel, the association between base level and metalevel, and the behavior of the reflective architecture, called *reflection mechanism*.

- ***Representation of Design Patterns at the Metalevel***. A pattern can be see as a fine-grained framework. It prescribes a template of the control structure to solve a design problem, leaving the implementation of some methods to the programmer. A metaobject class [Maes, 1988] at the metalevel (Fig. 1) represents this template prescribed by the pattern. The metaobject classes representing patterns are implemented independently from the base level classes and can be reused for the construction of different applications.

- ***Association between Base Level and Metalevel.*** When a design pattern is used in an application the metaobject corresponding to the pattern is created. Then, the association between base level and metalevel is established (Fig.2). This association can be established between a class, a method, or an object at the base level and the metaobject at the metalevel.

- ***Reflection Mechanism.*** At execution time, when an object at the base level receives a message, e.g. M1 (Fig.2), the reflection mechanism intercepts the message. Then, the reflection mechanism redirects the thread of control to the associated metaobject of the object at the base level, for example, invoking the M2 method. At the metalevel, the metaobject does its corresponding computation and, when its execution finishes, the reflection mechanism decided whether or not return the thread of control to the method at the base level.

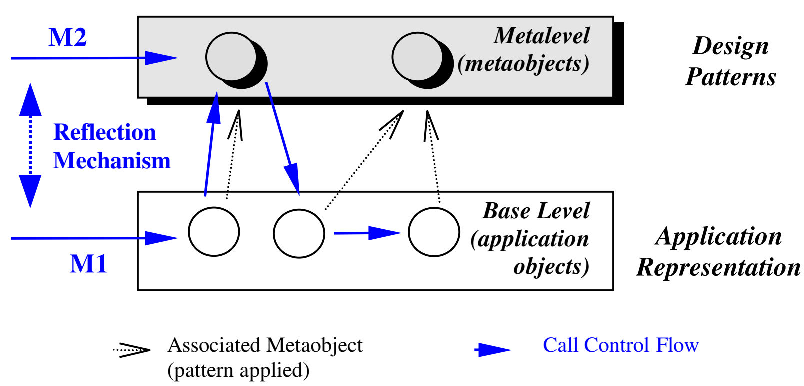

Fig.2 Reflective architecture behavior

3. An Example

In the previous section, a reflective metalevel architecture based on metaobjects for the representation of design patterns has been presented. In this section, we illustrate our approach by representing the *Composite* design pattern [Gamma, 1995]. The *Composite* design pattern composes objects into tree structures to represent part-whole hierarchies. It defines the way in which simple and compound objects can be treated uniformly.

For example, consider a graphic editor that allows the user to make figures, such as lines or rectangles. The user can group figures to form a complex figure, which can be used for the construction of another complex figure. The *Composite* design pattern can be used to describe how to use recursive composition, in such a way that simple and complex figures can be treated uniformly. This pattern prescribes an abstract class to represent both kinds of objects: simple and complex ones. In the graphic editor problem, this abstract class is the *Graphic* class (Fig.3)[1]. This class defines specific operations for all graphic objects, such as *Draw*. The subclasses of *Graphic* that represent simple objects, *Line*, *Rectangle*, implement the *Draw* method to draw simple figures. The subclass of *Graphic* representing complex figures, *Figure*, implements the *Draw* method invoking this operation to all the objects that compound it. The *Figure* class defines the *components* variable, which maintains the information of its components.

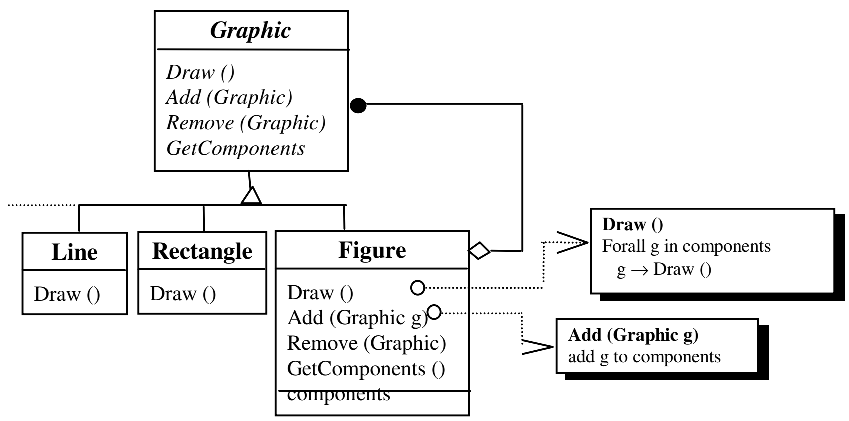

Fig.3 Application of the *Composite* design pattern

This pattern can be used to represent recursive part-whole hierarchies whose clients are be able to treat all objects in the composite structure uniformly and ignore the difference between simple and composite objects.

[1] In this document, the design of classes is based on OMT diagrams [Rumbaugh91]. Classes are represented as rectangles, with the class name in bold type at the top, and objects as circles. Elements at the metalevel are designed with a shadow and in a grey colour. The pseudocode of methods is represented by a shadow rectangle.

3.1 Metaobject Class for Representing the Composite Design Pattern

We represent the *Composite* design pattern through the metaobject class *MOComposite*. The objective of this class is to appropriately dispatch an operation requested for a complex object to all the components of the object.

MOComposite class defines four methods: *MMComposite*, *MMAdd*, *MMRemove*, *MMActualizeComponents*, and *MMGetComponents*.

MMGetComponents returns the components of a complex object. This method calls the *GetComponents* method at the base level, which returns the components of an object. The information of which are the components of a complex object is maintained in the *components* variable defined in the class representing complex objects. *MMActualizeComponents* actualize the components of an object at the base level with a new value. The *MMComposite* method recovers the components of the complex object received as parameter invoking the *MMGetComponents* method. Then, *MMComposite* dispatches the operation received as parameter to each component at the base level. At the end, the thread of control is returned to the base level.

The *MMAdd* and *MMRemove* methods incorporate and delete, respectively, a component to a complex object. To obtain and to actualize the components of the complex object, these methods invoke the *MMGetComponents* and the *MMActualizeComponents* methods respectively.

3.2 Association between Base Level and Metalevel

To use the *Composite* pattern using the reflective architecture, it is necessary to specify which application classes represent the classes prescribed by the pattern, i.e., which classes are involved in the pattern. For example, for the graphic editor problem (Fig.3), the *Graphic* class represents figures; *Line* and *Rectangle* subclasses represent simple figures, i.e., simple objects; and the *Figure* subclass represents complex figures, i.e., complex objects. Then, the name of the operation has to be specified, in this case *Draw*.

When those application classes, involved by the pattern, have been designed, the association between base level and metalevel is established. This association is established between the *Figure* class at the base level and an object of the *MOComposite* class, i.e., a metaobject, at the metalevel. For example, in the following way:

moComposite → new MOComposite
creates the metaobject moComposite *as an instance of the* MOComposite *class*
classreflection (Figure, moComposite)
creates the association between the Figure *class and the metaobject* moComposite

In the implementation phase, the programmer has to implement only the *Draw* method of the *Line* and *Rectangle* classes and the *GetComponents* method of the *Figure* class. It is necessary to implement this last method because the structure of the *components* variable can not be the same in different applications of the pattern. The methods defined in the *Figure* class are involved in the reflection mechanism and they do not need any implementation.

3.3 Reflection Mechanism

Consider the creation of the *myline* and *myrectangle* simple objects and the complex object *complexfigure* by the user. Then, the simple objects *myline* and *myrectangle* are added as components of the *complexfigure* object.

omplexfigure $\rightarrow$ Add (myline)

adds the object myline *to the complex structure* complexfigure

complexfigure $\rightarrow$ Add (myrectangle)

adds the object myrectangle to the complex structure complexfigure

When the *complexfigure* object receives the *Draw* message (1) (Fig.4), the thread of control is redirected to the *MMComposite* method (2) at the metalevel. The *MMComposite* method calls the *MMGetComponents* method (3) to recover the components of the *complexfigure* object. This method consults the *components* variable of the *complexfigure* object and returns its components: *myline* and *myrectangle* objects. Then, the *MMComposite* invokes the *Draw* method on *myline* (4) and on *myrectangle* (5). At the end, the thread of control is returned to the *Draw* method of *complexfigure* at the base level, which finalizes its execution.

If the *myline* or *myrectangle* objects receive the *Draw* message, the reflection mechanism does not redirect the thread of control to the metalevel.

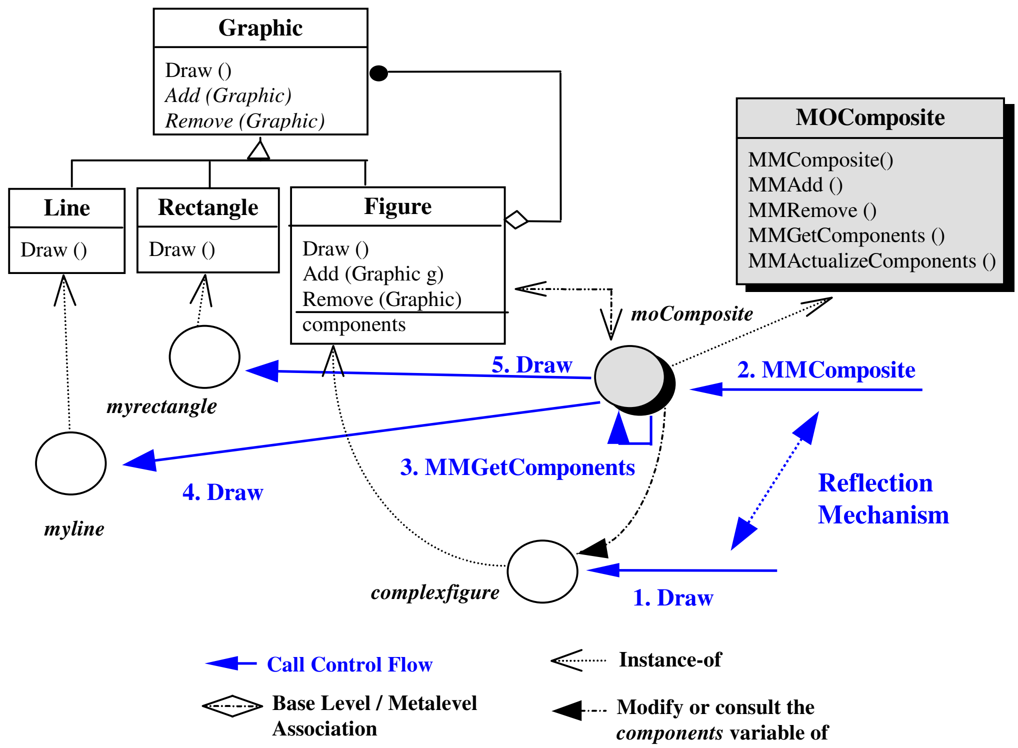

Fig.4 Reflection mechanism when the *Draw* method is invoked

4. Metaobject Protocol

Most current programming languages do not directly support the metaobject concept, but a basic support can be incorporated implementing the following facilities [Ferber, 1989]:

- Representation of system information as data.
- Association between objects at the base level with their metaobjects.
- Metaobject activation when an object at the base level is invoked.

The implementation of those functions in a programming language is called a *Metaobject Protocol* (MOP), for the language. The first aspect describes how and which data will be represented at the metalevel, is related to the reification process. The other two aspects are related to the mechanisms that define how the association between both levels is implemented and how the reflection mechanism is activated. There are two different ways to implement these aspects [Maes, 1988]: i) the responsibility of the activation of the reflection process can be assigned to the object at the base level, which has the information of the associated metaobjects, or ii) it can be assigned to the system.

We use the CLOS programming language [Kiczalzes, 1991] to implement the proposed reflective architecture. The general model for this reflective architecture can be divided in three models (Fig.5): *Reflection Mechanism Model*, which implements the reflection mechanism; *Design Patterns Model,* in which patterns are represented by metaobject classes; and *Application Model,* where the application classes are defined.

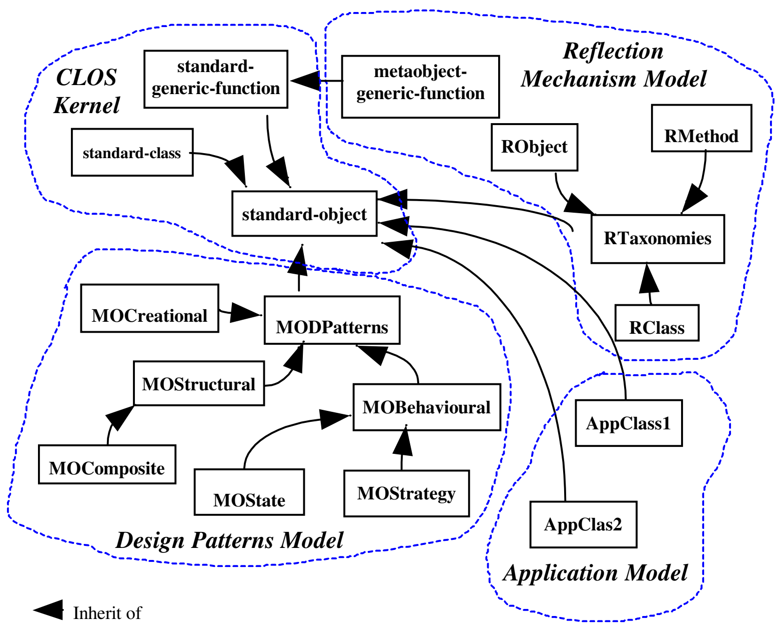

Fig.5 Reflective Architecture Model

4.1 Design Patterns Model

This model represents metaobject classes. The abstract class *MODPatterns* represents all patterns supported by the reflective architecture (Fig.6). Each design pattern has its corresponding subclass in this hierarchy. For example, the *Composite* design pattern is represented by the *MOComposite* class, the *State* pattern by the *MOState* class, and the *Strategy* by the *MOStrategy* class.

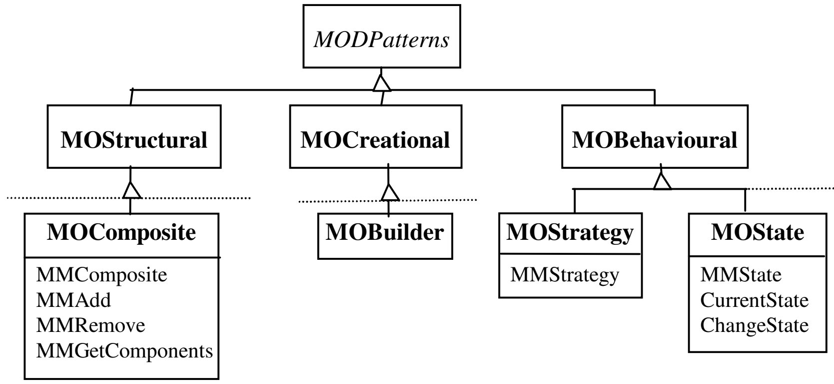

Fig.6 Design patterns metaobject classes hierarchy

4.1.1 Reflection Mechanism Model

The reflection mechanism model represents three strategies for conducting the reflection mechanism, that is the detail of when the thread of control is redirected to the metalevel: class reflection, object reflection, and method reflection [Marcos, 1997]. This model maintains information about which objects, methods, or classes of the base level are involved in the reflection mechanism and which are the associated metaobjects. The abstract class *RTaxonomies* (Fig.7) represents the possible strategies. For each strategy, it is necessary to maintain common information: the associated metaobject and the metamethod to which the reflection mechanism will redirect the thread of control at the metalevel. Additionally, for the class reflection, the information of the reflected class at the base level is required. For the object reflection, the object involved in the reflection mechanism, that is the reflected object, is maintained. For the method reflection, it is necessary to maintain the information of the reflected method and the class at the base level that defines this method.

For implementing message interception in CLOS it is necessary to introduce

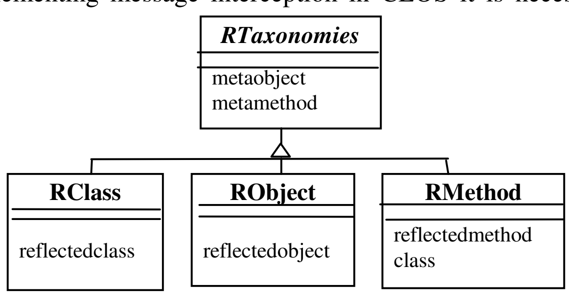

Fig.7 Reflection information hierarchy

changes to the kernel language. The most important change concerns to the definition of the primitive message-passing function, which checks if there is a metaobject bound to the receiver object. If there is one, the message is delegated to the metaobject. In CLOS-like languages, reflection takes place via generic functions. This mechanism must be implemented by redefining the method selection and the effective method construction machinery, which is defined on generic function classes. The method that has this responsibility is the compute-discriminating-function method. For this reason, we have redefined the compute-discriminating-function method, which finds the associated metaobjects and changes the thread of control to the corresponding metaobject at the metalevel. The associated-metaobject-of method returns a list of *RTaxonomies* objects with the information of the associated metaobjects of an object. Then the analize-metainfo method is invoked. This function traverses the list of metainformation and changes the control to the corresponding metaobject. Then, the default message interpretation is used whether or not the receiver object has an associated metaobject.

```
(defmethod compute-discriminating-function ((mogf mobject-generic-function) args)
    (let ((normal-dfun (call-next-method))
        (list-metaobj (send associated-metaobject-of parameters)))
        ;; finds the corresponding metainformation
        (send analize-metainfo mogf list-metaobj args)
        ;; analize the metainformation and change the control to the metalevel
        (apply normal-dfun args)
    )
)

(defmethod analize-metainfo ((mogf mobject-generic-function) metainfo args)
    (cond ((not (null metainfo))
        (send call-metaobject mogf (car metainfo) args)
        ; ; calls the corresponding metamethod at the metalevel
        (send analize-metainfo mogf (cdr metainfo) args)
        ; ;  does a recursive invocation for the rest of the associated metaobjects
        )))
```

Only the methods belonging to a reflection mechanism, i.e., the reflected methods, or all methods defined in a reflected class, are analyzed by the compute-discriminating-function method. To support this, we change the class of the generic function corresponding to those methods to the *mobject-generic-function* class. When the association between a method at the base level and a metaobject is established in a method reflection case, the reflection-method method is invoked. This function creates an instance of the *RMethod* class, with the reflective information, and changes the class of the corresponding generic function to the *mobject-generic-function* class.

```
(defmethod reflection-method (method class  mobject mmethod methname)
    (progn
     (make-instance 'RMethod
            :method method
            :class class
            :metaobject mobject
```

```
                    :metamethod mmethod)
            ; ; makes an instance of the RMethod class
        (ensure-generic-function methname
                    :lambda-list (method-lambda-list method)
                    :generic-function-class (find-class 'mobject-generic-
function))))
    ; ; changes the class of the corresponding generic-function
```

5. Conclusion

In this work a reflective architecture based on metaobjects for the representation of design patterns has been presented. In this architecture the metalevel represents the design patterns template of the control structure and the base level represents the specific elements of an application. The reflective level should manipulate the objects at the base level according to the architecture defined by the design patterns.

Each design pattern has been analyzed and its template of the control structure is represented by a metaobject class at the metalevel. When a pattern is used in an application the association between the base and metalevel is done. At run-time, when an object receives a message the reflection mechanism will take the decision whether or not redirects the thread of control to the metalevel, according to the applied pattern.

By the use of metaobjects for the representation of the design patterns it would be possible to maintain traceability to preserve patterns information for helping in the understanding and maintenance of applications. The representation of the design pattern template of the control structure is by metaobjects classes allows reusing this structure whenever the patterns are used in the same or different applications. If a new pattern is needed in an application, it is possible to incorporate dynamically a new design pattern. To do this, it is necessary to identify the pattern that we want to use and the application classes involved in the pattern, and codify some methods prescribed by the pattern.

6. Bibliography

[Beck, 1994] Beck K., and Johnson R. Patterns Generate Architectures. In Tokoro M., Pareschi R., editors. *Proc. of the 8th European Conference on Object-Oriented Programming,, ECOOP `94*. Bologna, Italy, July 1994, Lecture Notes in Computer Science 821, Springer-Verlag, Berlin Heidelberg New York, 1994.

[Bosch, 1996] Bosch J. Language Support for Design Patterns. *In Proc. of Technology of Object-Oriented Languages and Systems, Tools Europe'96.* pages 197-210. Paris, France. February 1996.

[Budinsky, 1996] Budinsky F., Finnie M., Vlissides J., and Yu P. *Automatic Code Generation from Design Patterns.* http://www.almaden.ibm.com/journal/sj/budin/budinsky.html

[Buschmann, 1996] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal Michael. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons, 1996.

[Campo, 1997] Campo M., Marcos C., and Ortigosa A. Framework Understanding and Design Patterns: A Reverse Engineering Approach. *In Proc. of Ninth International Conference on Software Engineering and Knowledge Engineering, SEKE'97.* Madrid, Spain. June 1997.

[Coad, 1992] Coad P. Object Oriented Patterns. *Communication of the ACM*. September 1992. Vol. 35 No 9.

[Cointe, 1987] Cointe P. Metaclasses are First Class: the ObjVlisp Model. In N.K. Meyrowitz, editor. *Proc. of the 2nd Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA'87*. ACM SIGPLAN Notices 2212. Orlando, Florida. December 1987.

[Coplien, 1995] Coplien J. and Schmidt, editors. *Patterns Languages of Program Design.* Addison-Wesley, 1995.

[Eden, 1997a] Eden A., Gil J., and Yehudai A. Automating the Application of Design Patterns. pages 44-46. *Report on Object Analysis and Design ROAD*. May 1997.

[Eden, 1997b] Eden A., Gil J., and Yehudai A. Precise Specification and Automatic Application of Design Patterns. *In Proc. of Automated Software Engineering, ASE'97.*

[Ferber, 1989] Ferber J. Computational Reflection in Class Based Object Oriented Languages. In Meyrowitz N.K., editor. *Proc. of the 4th Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'89*. ACM SIGPLAN Notices 2410. New Orleans, Louisiana. October 1989.

[Gamma, 1995] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns. Elements of Reusable Object Oriented Software*. Addison Wesley 1994.

[Johnson, 1997] Johnson R. Frameworks = Components + Patterns. *Communication of the ACM*. Volume 40, Number 10, pages 39-42. October 1997.

[Kiczalzes, 1991] G. Kiczales, J. des Rivihres, D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991

[Lange, 1995] Lange D. and Yuichi N. Interactive Visualization of Design Patterns can Help in Framework Understanding. In Wirfs-Brock R. editor. *Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'95*. ACM SIGPLAN Notices 3010. Austin, Texas. October 1995.

[Maes, 1988] Maes P. Issues in Computational Reflection. In Maes P. and Nardi D., editors. *Metalevel Architecture and Reflection*, pages 21-35. Amsterdam. Elsevier Science 1988.

[Marcos, 1997] Marcos C. *Design Patterns as First-Class Entities*. Technical Report TR-97/29, IAG-QANT, Université catholique de Louvain, Belgium, December 1997. Epreuve de confirmation de la thèse pour le Doctorat en Sciences Appliquées.

[Pree, 1994] Pree W. *Design Pattern for Object Oriented Development*. Addison Wesley 1994.

[Riehle, 1996] Riehle D. and Züllighoven H. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems 2,* 1, 1996.

[Rumbaugh, 1991] Rumbaugh J., Blaha M., Permerlani W., Eddy F., and Lorenson W. *Object-Oriented Modelling and Design*. Prentice-Hall. Englewood Cliffs, NJ 1991.

[Tichy, 1997] Tichy W. *Essential Software Design Patterns*. University of Karsruhe. http://wwwipd.ira.uka.de/~tichy/patterns/overview.html