Design Patterns as an Explicit Part of Applications

Claudia Marcos*† and Marcelo Campo*

*Univ. Nacional del Centro Prov. Bs. As. - Fac. Ciencias Exactas
ISISTAN - Grupo de Objetos y Visualización,
Paraje Arroyo Seco - Campus Universitario, (7000) Tandil, Bs. As., Argentina

†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

1. Design Patterns

Design patterns are becoming increasingly popular as mechanisms to describe general solutions to design problems that can be reused in the construction of different applications. Gamma et al. [Gamma95] define design patterns as *descriptions of communicating object classes that are customized to solve a general problem in a particular context.* The basic motivation behind the pattern idea resides in the fact that similar design problems recur in different context [Cockburn96, Pree94, Riehle96, Buschmann96]. The main goal of patterns is to solve a specific design problem allowing flexibility of evolution.

Patterns make possible to talk, as well as to think about designs on a higher level of abstraction. Instead of thinking in terms of individual classes and their behavior, it is possible to start to think in terms of collaborating classes, their relationship and responsibilities. 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.

Essentially, a design pattern expresses a design intent, suggesting a generic organization of classes and distribution of responsibilities among them, that solve a design problem. If a user has a design problem and knows which pattern intent to solve his problem, and what classes and methods the pattern prescribes. Then, the user can apply this pattern in his design making more reusable ones. Once the design has been finished, all methods prescribed by the pattern must be implemented. If a pattern was used more than one time in the same application, its methods must be implemented each time.

2. The Problem

To be the patterns effectively useful in the maintenance phase, it is necessary that they are well documented and reflected in the program code. Unfortunately, this not always happens mainly because the pattern implementation depends on the facilities provided by the target programming language.

In catalogues, the pattern implementation is often given by several informal recommendations, in terms of examples or template code, mainly in C++ and Smalltalk. Some languages can support a pattern directly by means of constructs or mechanisms of the language itself. For example, the pattern Singleton in Modula-3 through the module construct, or the

Iterator pattern in CLU. On the other hand, the same pattern can be distorted or complicated because of the lack of a supporting language construct or mechanism. For example, the pattern Proxy in C++, because of C++ is a strongly typed language, or the Adapter in Smalltalk, because of the pattern Adapter needs multiple inheritance. Furthermore, the same pattern can be implemented in very different ways in the same programming language. For example, in the pattern Observer a class can have the responsibility of taking the decision to notify a modification and another class can have the responsibility of notifying the modification to all its dependents. Another alternative is that the same class can have both responsibilities, that is, to take the decision and to notify the modification to all its dependents.

In this way, with the usual programming languages, there is no easy way to keep a trace in the application code that patterns were applied during design. This causes valuable information to be lost during maintenance of application programs. Maintenance consists in assessing and adapting an operational system according to changes in the environment of the system and in the needs that the system is supposed to fulfill. Maintenance is best conduced when all the information produced during the system life cycle is available, in particular which patterns were applied during design. Unfortunately, the current practice in that maintenance is most often based on the application code of the operational system and not in the application design. In this way, the advantages given by patterns in the design phase, like understanding and communicating design, are not applied in the maintenance phase.

Several works have been developed to maintain design patterns in the application code. Design patterns are recovered from application code in [Campo97] and [Lange95] works. In these works, particular characteristics of each pattern have been identified that allow the identification of the potential patterns used in an application. These characteristics are compared with an existing application and the potential patterns used in the application are identified. The definition of pattern characteristics is a very hard task, for example, it has been very difficult to make a differentiation among the characteristics of similar patterns (such as, Composite and Decorator, or State and Strategy). Furthermore, for some patterns, it was not possible to find them, so 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 drawback is that these works assume that the application to be analyzed was made using patterns, on the contrary case any pattern would be found.

Another approach to maintain a trace of patterns in the application code is the construction of a tool for automatically generating pattern code from information supplied by the designer [Eden97, Budinsky96, Bosh96]. 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. This solution has some drawbacks, 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 in a tool it must be modified, this can produce errors in applications already created with the tool.

The proposition to represent each design pattern with a special class, called *Pattern class,* implemented in C++ was presented in [Soukup92]. This class is a friend class and has the same pattern logic and behavior as the pattern, the main purpose of the *Pattern* class is the interface. However, only few pattern classes are represented and is uncertain whether all patterns can be implemented as this kind of class. The class that represents a pattern manipulates the internal object structure of the classes that form the pattern violating the encapsulation. If the attribute name of a class is changed the corresponding *Pattern* class must be changed to maintain consistency.

The definition of several general-purpose language constructors and mechanisms that would simplify design patterns implementation was proposed in [Baumgartener96]. In this work each design pattern has been examined in detail, taking into account its implementation in different programming languages. Using this analysis the authors describe the main characteristics that a programming language must have in order to simplify the pattern implementation. Unfortunately, a programming language with these characteristics does not currently exist.

In all cases, any solution should consider 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 metamodel of the software system, which defines how such software is structured and behaves. If a mean to represent and implement such a metamodel at runtime were available then an explicit trace of the involved design structures could be deduced from the code itself. In this sense the use of computational reflection techniques appears as a reasonable alternative.

3. Our Approach - Reflective Architecture for the Representation of Design Patterns

In this work 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.

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

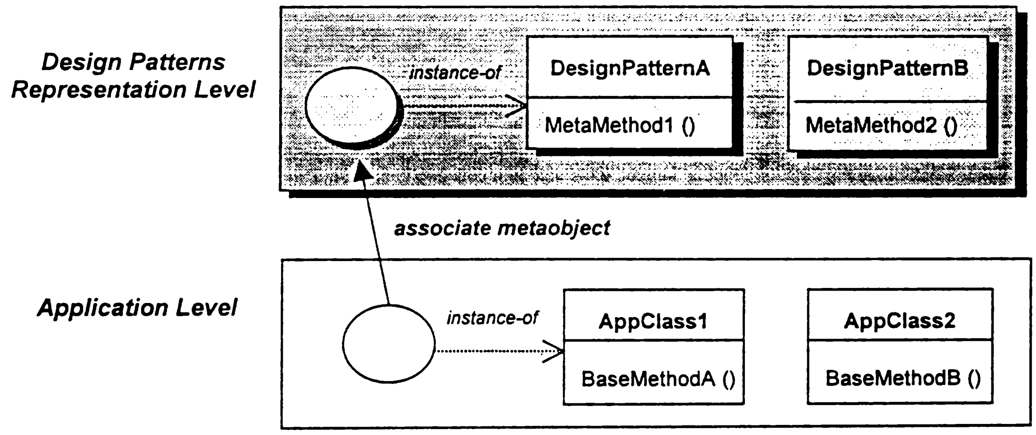

Figure. 1 Reflective architecture for the representation of design patterns.

When a pattern is used in an application the association between the base and metalevel is done. At run-time, when an object at the base level receives a message the reflection mechanism takes the decision whether redirects the thread of control to the metalevel, according to the applied pattern.

A pattern can be seen as a fine-grained framework, which prescribe a template of the control structure to solve a design problem. This control structure is represented by a metaobject class [Maes88] at the metalevel. Several design patterns have been analyzed, such as Gamma's and Buschman's patterns and another patterns proposed in the literature (e.g., the Backup pattern [Subranmanian97]), and their template of the control structure represented by a metaobject class at the metalevel. By means of the use of metaobject classes for representing design patterns it is possible to maintain traceability in order to preserve patterns information for helping in the understanding and maintenance of applications. Metaobject classes can be reused whenever the patterns are used in the same or different applications. If a new pattern is necessary in an existing application to increase its functionality, it is possible to incorporate the new pattern dynamically. 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.

The reflective architecture supports several patterns but not all patterns proposed in the literature, however it is possible to need a pattern not represented yet in the reflective architecture for the construction of a new application. There are mechanisms to introduce a new pattern in the reflective architecture, it is only necessary to identify the template of the control structure of the new pattern and represent it in a new metaobject class.

Some patterns are naturally formulated using one strategy where all methods and objects of a class go through a reflective behavior: we call this strategy *class reflection*. For other patterns, it is sufficient to apply a reflective behavior to some methods of the class: we call this strategy *method reflection*. Still other patterns are naturally formulated by having some objects of a class go through the reflective behavior: we call this strategy *object reflection*. These strategies define a *reflection taxonomy*.

The reflective architecture and taxonomy was implemented in the CLOS [Kiczalzes91] programming language by the introduction of some changes in its kernel. The most important change concerns to the re-definition of the primitive message-passing function, which checks if there is a metaobject bound to the target object [Marcos99]. If there is one, the message is delegated to the metaobject. The taxonomy can be used in the construction of several systems which need reflective characteristics.

4. References

[Baumgartner96]	Baumgartner G., Laüfer K., and Russo V. *On the Interaction of Object-Oriented Design Patterns and Programming Languages.* Technical Report CSD-TRS-96-020. Department of Computer Sciences Purdue University. February 29, 1996.
[Bosch96]	Bosch J. Language Support for Design Patterns. In *Proc. of Technology of Object-Oriented Languages and Systems, Tools Europe'96*. Paris, France February 1996. Pretince-Hall. http://www.pt.hk-r.se/~bosh.
[Budinsky96]	Budinsky F., Finnie M., Vlissides J., and Yu P. *Automatic Code Generation from Design Patterns*. IBM System Journal Vol. 35 No 2 1996. http://www.almaden.ibm.com/journal/sj/budin/budinsky.html
[Buschmann96]	Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal Michael. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons 1996.

[Campo97] Campo M., Marcos C., and Ortigosa A. Framework Comprehension and Design Patterns: A Reverse Engineering Approach. In *Proc. of the Ninth International Conference on Software Engineering and Knowledge Engineering SEKE'97,* Madrid, Spain, June 18-20.

[Cockburn96] Cockburn A. The Interaction of Social Issues and Software Architecture. *Communication of the ACM.* October 1996, Vol. 39 No.10. pp. 40-49.

[Eden97] Eden A. and Yehudai A. Patterns of the Agenda. Published in LSDF'97, Workshop in conjunction with ECOOP'97 European Conference on Object Oriented Programming Ireland July 1997.

[Gamma95] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns. Elements of Reusable Object Oriented Software.* Addison Wesley 1995.

[Kiczalzes91] G. Kiczales, J. des Rivihres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Lange95] 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*, October 1995, Austin, Texas, ACM SIGPLAN Notices 3010. pp. 342-357

[Maes87] Maes P. Concepts and Experiments in Computational Reflection. In Meyrowitz N.K., editor, *Proc. of the 2nd Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA'87*, December 1987, Orlando, Florida, ACM SIGPLAN Notices 2212.

[Marcos99] Marcos C., Campo M., and Pirotte A. *Extending CLOS to Support a Reflection Taxonomy.* Technical Report YEROOS TR-99/04, IAG-QANT, Université catholique de Louvain, Belgium, March 1999.

[Pree94] Pree W. *Design Pattern for Object Oriented Development.* Addison Wesley 1994.

[Riehle96] Riehle D. and Züllighoven H. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems* 2, 1, 1996.

[Soukup95] Soukup J. Implementing Patterns. In: J. O. Coplien, and D. C. Schmidt editors. *Pattern Languages of Program Design.* Chapter 20, pp 395-412, Addison Wesley 1995. http://www.codefarms.com/papers/patterns.html

[Subramanian96] Subramanian S. and Tsai W. Backup Pattern: Designing Redundancy in Object-Oriented Software. *In Pattern Languages of Program Design 2.* Chapter 13, pp. 207-225, Addison-Wesley 1996.