Enhancing Design Pattern Flexiblity with Reflection: Reflective Design Patterns

Dimitrios Theotokis[1], Anya Sotiropoulou[1], Georgios Gyftodimos[2]

[1] Department of Computer Science and Technology
University of Peloponnese, 22100 Tripolis, Greece
[2] Department of Informatics and Telecommunications,
University of Athens, 15784 Athens, Greece
Emails: dtheo@uop.gr, anya@uop.gr, geogyf@di.uoa.gr

Abstract. Design Patterns were proposed as a mean to resolve various issues that occur from inheritance in object-oriented systems and in particular issues regarding software evolution as well as to provide a standardised approach for object-oriented software design. Based on the notions of aggregation/parametrisation plus inheritance, design patterns do not really resolve all the problems for which they were invented for, e.g. incremental software evolution and reuse, but simply shift them at a higher level of abstraction since design patterns, despite their claim, do not promote as a loose coupling as required for incremental software evolution. In this paper we present an advancement of design patterns, namely reflective adaptive design patterns, which enables software evolution to occur without being hindered by the problems of inheritance and the issues that occur from parametrisation/aggregation irrespective of the use of inheritance. The proposed alternative finds is corner stone on both traditional design patterns and reflective language and provides a flexible approach towards design-patterns-based software evolution in which loose coupling and high cohesion are guaranteed.

1 Introduction

The evolution of software designs and software has been in the centre of attention for many researchers and practitioners in recent years. Many approaches have been proposed for this purpose including, but not limited to, design patterns [3], subject-oriented programming [5], aspect-oriented programming [6], role-based programming [16, 19–21] etc.

What is central to all approaches employed towards software evolution, is the need to overcome the various problems that occur from inheritance, particularly when it is used as a means for software evolution, through the use of subclasses. Core to inheritance is the notion of a statically defined relationship between two classes, namely the super class and the sub-class. Bound during the compilation phase this relationship hinders the evolution of a sub-class, since the developer is faced with inheritance related drawbacks, such as the common ancestors dillema [8], the homonymous attributes problems [2, 10] amongst others.

Design patterns were originally proposed as a vehicle for constructing better and more flexible designs, on the one hand, and reusable and evolvable code on the other.

Be that as it may, fundamental to the principle of design patterns are the notions of aggregation and/or parametrisation, complemented with inheritance. In other words, embedded to design patterns are statically bound references of participating objects: an actuality that prevents the transparent evolution of software designs and code in a manner that is free of editive changes, that is, changes that affect existing code.

In this paper we propose an enhancement to design patterns based on the notions of object-bound polymorphism (O-bound polymorphism) and reflection. According to this approach, references to participating objects are bound to their most general superclass, and delegation between objects is achieved during runtime, dynamically, using reflection instead of message passing.

The rest of this paper is organised as follows: in section 2 a brief description of design patterns is given so that the issues that hinder evolution can be identified. Next, in section 3 the notions underlying reflection and reflective languages are presented. Reflective design patterns are presented in section 4, while in section 5 a comparison between traditional and reflective design patterns is given, highlighting the benefits of reflective design patterns with respect to software evolution. Finally, section 6 concludes the paper and suggests future research dimensions. It must be stressed at this point that the proposed approach — that of reflective design patterns — is useful particularly when software evolution combined with reuse are of interest.

2 Design Patterns

According to Christofer Alexander [1] "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to the problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Although Alexander was referring to architectural design, what he claims is also true about object-oriented design patterns. Software solutions are expressed in terms of objects and interfaces instead of architectural elements, such as walls and floors, but central to both kinds of patterns is a solution to a problem in a context. According to [3] a pattern has four essential elements:

1. The name of the pattern. The pattern name is a handle we can use to describe a design problem, its solution, and consequences in a word or two.
2. The problem, which describes when to apply the pattern. It explains the problem and its contexts.
3. The solution describes the elements that make-up the design, their relationships, responsibilities, and collaborations. The solution does not describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, a pattern describes an abstraction of a design problem and how a general arrangement of elements (classes and objects) solves it.
4. The consequences are the results and trade-offs of applying the pattern.

Gamma et al. [3] proposed a classification of design patterns according to their intent. Under this classification design patterns are separated into three categories, namely Creational, Structural and Behavioural. In the context of this paper we deal only

with behavioural design patterns, since these are used for software evolution and in particular behavioural evolution, i.e. incremental behavioural changes of an existing design or implementation.

2.1 Structure of Design Patterns

According to Gamma [3] design patterns are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context. The mechanism that is widely used in design patterns is delegation. Delegation is used when an object needs to share or specialise behaviour with or without the use of inheritance. Each design pattern consists of an object, or set of objects, known as receiver, and an entity that dispatches messages, also known as delegates. Communication between receiver and delegate is achieved via message passing. The entities involved in the message passing are the receiver, which receives the message and forwards it to the second entity, the delegate, which handles the message. In order for the message to be handled correctly, the receiver passes to the delegate a reference to itself. For instance, figure 1, Client and Colleague are delegates, while Mediator is the receiver. Similarly, subject is the delegate and ObserverC is the receiver.

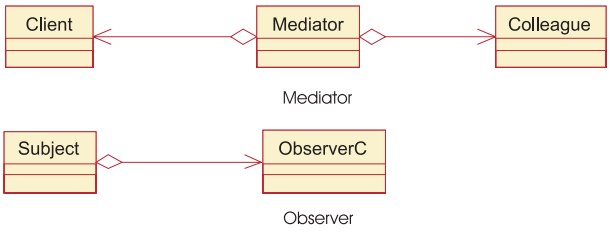

Fig. 1. Mediator and Observer patterns

It follows, that key to the structure of design patterns are the notions of aggregation and/or parametrisation complemented in some cases with inheritance. To elaborate further on this figure, 1 illustrates two representative behavioural design patterns, namely Mediator and Observer. The Mediator design pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and permits to vary their interaction independently. For example, the Mediator design pattern could be employed as a way of enforcing loose coupling between two widgets in a graphical user interface that need to interact, instead of providing each widget with the reference of the other one.

The Observer design pattern defines an one-to-many dependency between objects, so that when one object changes state all its dependents are notified and updated automatically. Assuming a widget that presents a chart for some data and a widget that controls the type of chart (histogram, pie chart, area chart etc.) the former, presentation widget, is the observer and the latter, controller widget, is the subject.

In both cases, the receiver contains references to the delegate entities. In the case of the Mediator design pattern the mediator object contains references of both the colleague and the client, while in the case of the observer design pattern the subject contains a reference to the observer. Further, both the Client and the Colleague objects refer to the Mediator and the ObserverC object refers to the Subject object The implementation of these design patterns is depicted in figure 2. What is important to notice in both implementations is the hard-coded reference to the delegates both in the attributes and methods part of the respective classes, shown in lines 2, 4–6, 11, 15–16, 20, 21, 23–26, 30, 32–33, 38, and 40 in figure 2.

By hard coding these references it becomes clear that evolution becomes less transparent since it requires editive changes, when the delegates change. The obvious change is that of the type of the delegate in the receiver both. This will affect both the attributes, as well as the methods' bodies, where explicit reference to the delegate object is made.

Of course one could argue that by making both the receiver and the delegates abstract classes and sub-classing them according to specific needs, as these occur, one can transparently evolve the design. However, this is only a first impression. A problem that arises from such an approach is that a new class needs to be created every time a new requirement arises, a practice that leads to class explosion. Moreover, this approach does not really promote loose coupling, since the participating parts still hold references of each other and still rely on parameter passing to achieve delegation. Indeed, if the base class of the inheritance graph exhibited by the objects participating in the design pattern changes, their need for editive changes becomes necessary. Finally, and according to Szyperski [18], a major disadvantage occurs by the use of inheritance in this manner. This disadvantage is known as "implicit self-recursion" or "processing of a common self". In other words, if an object is an instance of a class, then it has exactly one identity: its "self" reference, even if its class inherits from many other classes. However, in the case of design patterns there is not a common self between the receiver and the delegate object, since they belong to different class hierarchies. The difference in question, does not show until a self-recursive invocation is studied. In the case of implementation inheritance, control in a self-recursive invocation always returns to the last overriding version of any method. That is, control can return from any of the evolved superclasses back to any of the involved sub-classes. This is different in the case of delegation. Once control has been passed from the receiver to the delegate self-invocations stay within the delegate objects, which belong to a different class hierarchy than that of the object which originally delegated the message. The solution to this problem could be the design of delegate objects with inter-references to objects participating in the pattern and the use of parametrisation in order to achieve true self-recursive invocations. But this makes the situation very complex especially in the context of unanticipated software evolution.

3 Reflection and Reflective Languages

A programming environment is called a meta-level architecture when the implementational structures of the meta-system are available for inspection and/or manipulation

```
1  class Delegate1 {
2      private Mediator mediator;
3
4      public void method1() {mediator.method1(...);}
5      public void method2() {mediator.method2(...);}
6      public void method3(...) {...}
7      public void method4(...) {...}
8  }
9
10 class Delegate2 {
11     private Mediator mediator;
12
13     public void method1(...) {...}
14     public void method2(...) {...}
15     public void method3() {mediator.method3(...);}
16     public void method4() {mediator.method3(...);}
17 }
18
19 class Mediator {
20     private Delegate1 delegate1;
21     private Delegate2 delegate2;
22
23     public void method1(...) {delegate2.method1(...);}
24     public void method2(...) {delegate2.method2(...);}
25     public void method3(...) {delegate1.method3(...);}
26     public void method4(...) {delegate1.method4(...);}
27 }
28
29 class Subject {
30     private Observer observer;
31
32     public method1() {observer.notify(...);}
33     public method2() {observer.notify(...);}
34     ...
35 }
36
37 class Observer {
38     private Subject subject;
39
40     public void notify(...) {...}
41 }
```

Fig. 2. Pattern implementation

by the programmer or another computational system [11, 17]. The above definition does not make any commitment about "how much" and "in which way" can be inspected and manipulated. For instance, a programming language may provide restricted access to inspect and control the stack or the variable binding environment. These facilities are generally the basis for implementation debuggers. At the other extreme, the whole meta-program of the language processor may be available such that it can be edited by the programmers. An example of such a language is Prolog. In between, the two extremes "lies" the definition of an open implementation, as a system that provides an additional interface to controlled or structured access to its meta-level [7, 13]. The meta-level interface specifies points where the user can provide alternative implementations of the meta-system.

Reflective systems are special cases of meta-level architectures where the computational system is given access to the own meta-system. Smith [15] defines a reflective system as one that is constructed not only to reason about an external world in virtue of comprising a process formally interpreting representations of that world, but also is made to reason about itself in virtue of comprising a process formally manipulating representations of its own operations and structures. The main difference between a meta-level architecture and a reflective architecture is that a meta-level architecture only provides static access to the representations of a computational system, while a reflective architecture also provides dynamic access to this representation. In other words, in a reflective system the base and meta-level are causally connected to each other.

A program can be given access to its meta-system by extending the programming language with reflective operators [17]. A language that provides reflective operators is called a reflective programming language. In other words, all systems written in a reflective language can access a causally connected representation of themselves. A program that uses reflective operators is called a reflective program. Thus, a reflective program is both a base-program and a meta-program at the same time, since it can reason about and act upon the representation of the external domain, as well as that of the computational system described by itself.

Reflection techniques allow the implementation of a language to be exposed in a way that satisfies two important criteria. First, the access to this implementation must occur at an appropriately high level of abstraction. Second, the access must be effective in the sense that the meta- and base-systems are causally connected, i.e., manipulations of the system's representation must actually change the language behaviour.

4 Reflective Design Patterns

In [?,9] reflective design patterns have been proposed for the implementation of fault tolerant systems and for automating software design patterns using a case-based reasoning approach. In these works the use of the Guaraná meta-object protocol [12] is employed to provide a means for developing reflective design patterns. The use of an explicit meta-level for the realisation of reflective design patterns, also proposed by [14], renders the development of reflective design patterns independent of language support for reflection. However, it adds a layer of abstraction that complexes matters

and makes design and implementation less intuitive, since one has to develop not only the reflective design pattern, but also its meta-level in order to support it.

Before proceeding in describing the underlying concept of reflective design patterns it is essential to provide the requirements for their realisation in a computational system.

The concept of reflective design patterns finds its foundations in two concepts. First, the provision for reflective techniques by the computational system in which they are deployed like those provided by Java or Smalltalk. Equally important are: (a) the presence of a common super-class, or meta-class in terms of Smalltalk, sub-classes or instances of which are all the classes in the computational system and (b) a common super-class or meta-class, sub-classes of which are all class instances. The former provides the means to introspect any given class's or instance's structure, while

```
1  class Mediator {
2     protected Object client;
3     protected Object colleague;
4
5     Mediator(Object client, Object colleague) {
6        this.client = client;
7        this.colleague = colleague;
8     }
9
10    void invokeClientMethod(String methodName,
11                            Class [] formalArguments,
12                            Object [] parameters) {
13        Class clientClass = client.getClass();
14        Method classMethod = null;
15        Class [] exceptionsThrownByMethod = null;
16        try {
17            classMethod  =
18                  clientClass.getDeclaredMethod(methodName,
19                                                formalArguments);
20            exceptionsThrownByMethod = classMethod.getExceptionTypes();
21            classMethod.invoke(client, parameters);
22        }
23        catch(Exception e) {
24             System.out.println(e.toString());
25             System.out.println(methodName);
26             System.out.println("Method " + methodName + "threw " +
27                                e.getClass().getName() + " because " +
28                                ((Throwable )e).getCause());
29         }
30    }
31
```

Fig. 3. Reflective pattern implementation

```
32    void invokeColleagueMethod(String methodName,
33                               Class [] formalArguments,
34                               Object [] parameters) {
35        Class clientClass = colleague.getClass();
36        Method classMethod = null;
37        Class [] exceptionsThrownByMethod = null;
38        try {
39            classMethod  =
40                  clientClass.getDeclaredMethod(methodName,
41                                                formalArguments);
42            exceptionsThrownByMethod = classMethod.getExceptionTypes();
43            classMethod.invoke(colleague, parameters);
44        }
45        catch(Exception e) {
46            System.out.println(e.toString());
47            System.out.println(methodName);
48            System.out.println("Method " + methodName + "threw " +
49                               e.getClass().getName() + " because " +
50                               ((Throwable )e).getCause());
51        }
52    }
53
54    void setClient(Object client) {
55       this.client = client;
56    }
57
58    void setColleague(Object colleague) {
59       this.colleague = colleague;
60    }
61 }
```

Fig. 4. Reflective pattern implementation (continued)

the latter provides a common and abstract point of reference for all participating entities in the computational system, including the computational system itself.

Having in mind these two requirements what follows is the implementation of the Mediator design pattern, in terms of the reflective capabilities provided by Java, as illustrated in figure 4 and a discussion that highlights the benefits that emerge from its use compared with its traditional counterpart.

4.1 Structure of a Reflective Design Pattern

Following the principles of traditional design patterns, reflective ones "inherit" the notions of aggregation and/or parametrisation, but unlike the former, the latter do so in a most abstract way: the receiver contains references of the most abstract representation of its delegates. In our case, each delegate is typed as an instance of its most

general supper-class i.e. the Object class (lines 2,3 figure 4). This provides the flexibility required to evolve software in a transparent way, as any instance of any class can be represented as an instance of the Object class, according to the principles underlying the O-bounded polymorphism, also known as coercion polymorphism. Thus the delegates participating in the pattern may be instances of any class and, more importantly, can change during runtime using the "`setClient`" and "`setColleague`" methods, provided by the mediator. This characteristic alone does not solve the problem in question. What is now required is to "remove" explicit and hard-coded references to method calls (message passing - lines 23–26 figure 4) to each delegate and consequently the types of their associated formal arguments. In doing so and in contrast to its traditional counterpart, the Reflective Mediator only provides two methods, namely "`invokeColleagueMethod`" and "`invokeClientMethod`" (figure 4), each of which define three formal arguments: an array of class objects that contains the types of the method's formal arguments, an array of objects that contains the values corresponding to the method's formal arguments, and a string representing the method name to be invoked.

4.2 How do Reflective Design Patterns Work?

As in the case with traditional design patterns, reflective ones operate on the principle of delegation through message passing. The delegate delegates a message to the receiver, which in turn delegates to the receiving delegate. In traditional design patterns this is achieved by explicitly calling the required method, passing to it the appropriate values for its formal arguments. In reflective design patterns this is not the case. The delegate "constructs" the method call for the method it wishes to invoke, which is located in the receiving delegate and calls the receiver invokeClientMethod or invokeColleagueMethod, which processes the "constructed" method call and dispatches the message to its actual recipient. Thus, the calling delegate provides the method name, the types of the formal arguments and their corresponding values for the method in the receiving delegate and dispatches the message in question to the receiver for further processing. Upon reception of the message the receiver determines if there exist such a method in the receiving delegate and if so dispatches the message to it with the values provided by the calling delegate (lines 13–28 and 34–43, figure 4).

Determining if the method exists in the receiving delegate is achieved by introspecting with reflection the structure of the receiving delegate's class. If such a method does not exist a call back to the calling delegate is dispatched informing it so. It becomes evident that each delegate need to posses knowledge of the structure (contract) of the receiving delegate in order to "construct" a method call. This again is achieved through reflection. Consequently the only hard-coded knowledge that each delegate requires to posses is the method names comprising the structure of the receiver, namely invokeClientMethod and invokeColleagueMethod. As such, each delegate can evolve independently of each other without affecting the structure or function of the pattern.

5 Comparison of Design Patterns and Reflective Design Patterns

While traditional design patterns promote robust designs and design and code reuse, they do not deal effectively with unanticipated design and software evolution because of the structural infrastructure. Both the receiver and delegate need to posses knowledge of each others' internal structure. Further, they heavily rely on the use of parameters to achieve their function. These parameters are hard-coded to their structure thus leaving little room for maneuvering when evolution requirements arise. Even when complemented by class hierarchies these problems are not resolved due to the recursive self-invocation problem, which, unlike in the case of class hierarchies, is absent in design patterns [18].

The use of reflection in the receiver entity of a design pattern enables the loose coupling required for transparent software evolution when unanticipated changes occur. Key to reflective design patterns is the absence of hard-coded references within the method code of both the delegates and the receiver. The only knowledge that delegates need to posses is the contract exposed by other delegates so that they can dispatch the appropriate request through the receiver. Provided that the contract is maintained the delegates can evolve without requiring any editive changes that would mainly affect the types of the formal argument of their methods.

The main difference between traditional design patterns and reflective ones lies in the absence of hard-coded processing logic within the receiver. In contrast with traditional design patterns, in reflective ones the receiver only provides a meta-level delegation mechanism that simply acts as a viaduct between two or more delegates. The difference in the structure of the receiver entity in both approaches can be clearly seen in the respective implementations provided in figures 2 and 4. While in figure 2 there is an explicit implementation for each of the functions performed by each delegate (lines 4–7 and 13–16) in figure 4 there is only one method that handles all requests for message passing to a delegate, namely invokeClientMethod and invokeColleagueMethod. There are no hard-coded references nor explicit types defined, elements that hinder transparent evolution. Instead, the types of arguments are represented as an array of Class instances, each specifying a particular argument type. Moreover, and in line with traditional design patterns, reflective ones abide to the notions of aggregation and parametrisation. The only difference is that in the latter case we deal with the most general level of abstraction, a fact that does not require rigid design decisions. What is interesting to note is that in the proposed approach there is also a significant reduction of the required code for achieving the task in hand. However, this gained flexibility has its trade-off in code readability and understanding, as well as performance.

5.1 Advantages and Disadvantages of Reflective Design Patterns

Although reflective design patterns promote reuse and evolution in a more transparent way than their traditional counterparts, they expose some problems related to their use:

First, they require a deeper understanding of the task in hand, since there is no direct delegation of messages, but instead one is faced with meta-level delegation that

admittedly can be confusing. Second, they depend on a given language's reflective capabilities. Some languages provide little inherent support for reflection and as a result the developer must provide support for them. Finally, due to the meta-level delegation employed by reflective design patterns there is an evident performance degradation, since method lookup must be performed dynamically, twice: once to identify the method to be invoked and once to select the appropriate method abiding to the rules of F-bounded polymorphism. In order to evaluate the performance degradation introduced by reflective design patterns a set of benchmarks where carried out using the Mediator design pattern. Table 1 illustrates the results of benchmarks. Each test was performed 100 times and the average of the resulting values excluding the best and worst cases was used. Three different tests were carried out, involving 10 000, 100 000 and 1 000 000 invocations, between each delegate and the receiver. Clearly, one can see that reflective design patterns impose a performance degradation by a factor of about 20. Thus, the use of reflective design patterns should be employed in domains where their traditional counterparts do not provide the flexibility required. Table 2 presents various domains where reflective design patterns are suitable despite the aforementioned performance degradation.

Number of invocations	**Traditional D.P.**	**Reflective D.P.**
10 000	5.16	98.44
100 000	45.78	1030.15
1 000 000	456.41	10072.66

Table 1. Traditional vs. Reflective Design Patterns: Performance issues

Domain of application	**Traditional D.P.**	**Reflective D.P.**
Rigid Design Decision	YES	NO
Occurence of unanticipated changes	NO	YES
Reuse	YES	YES†
Runtime software evolution	NO	YES
Defered design decisions	NO	YES
Tailorable information systems	NO	YES

†Reflective design patterns are better suited for reuse, since they can be reused as is, without any modifications.

Table 2. Traditional vs. Reflective Design Patterns: Applicability

In any case, one must weight the benefits provided by reflective design patterns with respect to software evolution. It is clear that in reflective design patterns delegates and receivers are more cohesed and less coupled and thus promote better and more robust designs, which can evolve transparently without requiring editive changes in

existing code. This alone, it is our belief, that is an advantage that over-shadows the aforementioned disadvantages. Table 3 summarises the benefits of both traditional and reflective design patterns.

Criterion	**Traditional D.P.**	**Reflective D.P.**
Readability	Good	Average‡
Understanding	Easy	Difficult‡
Language support	All languages supporting pointers and references	Explicit language support for reflection
Ease of development	Medium	Hard
Reuse	Medium	High
Implicit design decision	Yes	No
Software evolution	Restricted	Unrestricted
Coupling	Tight	Loose

‡This is the case for the unaccustomed with Reflective D.P. user. Once the user becomes familiar with the concept of reflection and introspection readability and understandability are increased.

Table 3. Traditional vs. Reflective Design Patterns: Comparison

6 Conclusions

The proposed variation of design patterns provides a vehicle that allows for better reuse and software evolution due to the enhanced loose coupling between the delegate and receiver entities in a design pattern. Avoiding to hard-code in any sense the inter-references between these two entities leads to truly decoupled communicating objects, a requirement fundamental to software evolution and in particular when unanticipated changes come to play. Further, this approach demotes scattering and tangling of code, a fact that itself promotes evolution in a transparent way.

One of the issues that require further investigation is the performance degradation associated with meta-level delegation. In this light, the use of dynamic proxies may be employed in order to minimise method lookup during the initial invocation of a method. Consequent invocation will be treated through proxies created during the initial invocation thus speeding-up the entire process.

References

1. C. Alexander, S. Isikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. *A Pattern Language*. Oxford University Press, New York, (1977)
2. B. Carré, J.-M. Geib. The Point of View Notion for Multiple Inheritance. *Proceedings Joint OOPSLA-ECOOP Conference*, (1990)
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, (1994)

4. P. Gomes, F.C. Pereira, P. Paiva, N. Seco, P. Carreiro, J.L. Ferreira, C. Bento. Using CBR for Automation of Software Design Patterns. *Proceedings 6th European Conference in Advances in Case-Based Reasoning (ECCBR)* Scotland, UK, (2002) 534-548
5. W. Harrison, H. Ossher, Subject-oriented Programming (a Critique of Pure Objects). *Proceedings 8th OOPSLA Conference*, (1993) 411-428
6. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. M. Loingtier, J. Irwin. Aspect-oriented Programming (invited talk). *Proceedings ECOOP Conference*, Springer LNCS Vol.1241, (1997) 220-243
7. G. Kiczales. Towards a New Model of Abstraction for the Engineering of Software (invited talk). *Proceedings 9th OOPSLA Conference*, (1994) http://www.xerox.com/PARC/spl/eca/oi.html
8. J.-L. Knudsen. Name Collisions in Multiple Specification Hierarchies. *Proceedings 2nd ECOOP Conference*, Springer LNCS Vol.322, (1988) 93-109
9. L.L. Ferreira, C.M.F. Rubira. Reflective Design Patterns to Implement Fault Tolerance. *Proceedings OOPSLA 98 Workshop: Reflective Programming in C++ and Java*, Vancouver, BC., Canada, (1998) 81-85
10. M. Van Limberghen, T. Mens. Encapsulation and Composition as Orthogonal Operations on Mixins: a Solution to Multiple Inheritance Problems. *Object-Oriented Systems*, 3(1), (1996)
11. P. Maes. Concepts and Experiments in Computational —Reflection. *Proceedings 2nd ACM OOPSLA Conference*, (1987) 147-155
12. A. Oliva, L.E. Buzato. Composition of Meta-Objects in Guaraná. *Proceedings OOPSLA'98 Workshop: Reflective Programing in C++ and Java*, Vancouver, BC., Canada, (1998)
13. R. Rao. Implementational Reflection in SILICA. *Proceedings 6th ECOOP Conference*, Springer LNCS Vol.512, (1991) 251-267
14. Y. Sanada, R. Adams. Representing Design Patterns and Frameworks in UML — Towards a Comprehensive Approach. *Journal of Object Technology*, 1(2),:143-154 (2002), http://www.jot.fm/issues/issue_2002_02/article3.
15. B.C. Smit. Procedural Reflection in Programming Languages. Ph.D. Thesis. Massachussets Institute of Technology, Technical Report 272, MIT Laboratory of Computer Science, (1982)
16. F. Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data and Knowledge Engineering*, 35:83-106 (2000)
17. P. Steyaert. Open Design of Object-Oriented Language. A Foundation for Specialisable Reflective Language Frameworks. Ph.D. Thesis. Programming Technology Lab, Vrije Universiteit, Brussels (1994)
18. C. Szyperski. *Component Software. Beyond Object-Oriented Programming.* ACM and Addison-Wesley, (1998)
19. D. Theotokis: "Approaching Tailorability in Object-Oriented Information Systems through Behavioural Evolution and Behavioural Landscape Adaptability", *Journal of Applied System Studies*, Special Issue on Living, Evolutionary, Tailorable Information Systems: Development Issues and Advanced Applications (to appear)
20. D. Theotokis, G. Gyftodimos, P. Georgiadis. Atoms: a Methodology for Component Object-oriented Software Development. *Proceedings International OOIS Conference*, London UK, (1996) 226-242
21. D. Theotokis, G.-D. Kapos, C. Vassilakis, A. Sotiropoulou, G. Gyftodimos. Distributed information systems tailorability: a Component Approach. *Proceedings 7th IEEE FTDCS Workshop*, Cape Town, South Africa, (1999) 95-101