A pattern system for aspect-oriented design

Ouafa Hachani
LSR-IMAG
Grenoble, France
+33 476 827 264
OuafaHachani@imag.fr

ABSTRACT
Aspect orientation can be used to evolve and improve object-oriented design patterns. However, the newly proposed patterns are generally specific to a particular aspect-oriented programming language (such as AspectJ, Hyper/J, etc). In order to mitigate this limit, we proposed a general aspect-oriented design modeling language that we used to express the aspect-oriented structures of all the GoF design patterns. This research led us to define a system of eight original patterns that capitalize expertise related to aspect-oriented design. The proposed patterns are coordinated and treated on a hierarchical basis that makes it possible to offer a method to carry out aspect-oriented design and programs with good quality. This paper presents the 8 new aspect-oriented design patterns.

Keywords
Aspect-orientation, aspect-oriented design, design patterns, aspect pattern description formalism, GoF patterns.

1. INTRODUCTION
Aspect-oriented design is a relatively young area, and design knowledge is expected to emerge as practice advances. Some aspect-oriented design refactorings, directives and guidelines have already emerged [3, 13, 19]. However, there is some other useful knowledge in software engineering, currently recommended for the design of reusable object-oriented systems, which may be affected by the aspect orientation. Design patterns are examples of such well-used knowledge. Several works are done on object-oriented design patterns and aspect-oriented programming. The motivation of these works, as well as our own work on this topic, is to provide aspect-oriented implementations of object-oriented design patterns. The proposed implementations have better properties: explicit separation of the base and the code induced by the pattern instantiation, more readability of the code and traceability of the patterns, ease of reusability, modularity and evolution of both the base code and the pattern instantiation. The newly proposed pattern solutions are nevertheless language-specific, because aspect-oriented programming models and languages still lack a consensus on their basic concepts and mechanisms.

In order to mitigate this limit, we proposed an approach based on meta-modeling and model transformations for expressing language-independent aspect-oriented designs, and we used it to express the aspect-oriented structures of all of the 23 GoF patterns. This led us, by studying the analogies appearing in the obtained pattern structures, to isolate 8 new aspect-oriented patterns. The main contribution of this paper resides in the identification, specification and organisation of these new patterns, as well as, in the proposition of an aspect pattern description formalism. Though the patterns presented in this paper derive from the GoF design patterns, they aim therefore to be general-purpose. Moreover, as the 23 GoF patterns illustrate a variety of designs that provided us with a rich source of design knowledge, the new patterns capitalize useful expertise that allows aspect-oriented design and program with good software engineering quality attributes.

The rest of this paper is organized as follows. Section 2 presents the background of our work in order to easily understand the new patterns. Section 3 introduces the aspect pattern description formalism which is illustrated by the GoF *Strategy* pattern. Section 4 introduce the 8 new aspect patterns and details their relations. Section 5 considers some related works; it provides a discussion and serves as a first validation effort. Section 6 concludes this paper.

2. BACKGROUND
The effort leading to the identification of the 8 new aspect-oriented patterns documented in this paper was carried out through studies based on our previous work: the review of the 23 GoF design patterns in the light of aspect-orientation and the proposal of an aspect-oriented modeling language. We propose here to briefly recall this research work.

2.1 Aspect orientation and object-oriented design patterns
Object-oriented design patterns [1, 6, 7] are known to be helpful to design reusable components or, more generally, software. However, since they are generally defined by collaborations between several classes, it is difficult to identify where and how they have been applied in the source code of a large piece of software. Moreover, we have more precisely shown that several problems are related to the use of design patterns in their original object-oriented solutions: *confusion*, *indirection*, *breaching of encapsulation* and *inheritance* problems [9]. These four problems can be further considered as special cases of the two recurrent problems of “code scattering” and “code tangling” [18] that are addressed by several aspect-oriented

programming models and languages. Indeed, aspect-oriented concepts and mechanisms allow for new program designs that are out of reach of strict object-orientation and can (not surprisingly) improve the structures and implementations that were initially proposed in object-oriented design patterns. In this way, several works such as [14, 16, 20, 21, 22] have been made to evolve some existing object-oriented design patterns into aspect-oriented design patterns. These works, as well as our own work [9, 10, 11] on this topic, mainly intend to provide aspect-oriented implementations of object-oriented design patterns. Without harming the benefits of the initial patterns, such evolved design patterns have several additional benefits coming from the use of aspect-oriented mechanisms and techniques: explicit separation of the base code and the pattern instantiation, ease of evolution, less dependencies, traceability of patterns.

For instance we have worked on AspectJ [17] and Hyper/J [23] implementations[1] for the 23 GoF patterns [7] and explained how these solutions can avoid the patterns problems [10]. For this purpose we did not systematically mimic the original object-oriented structure (as proposed in [14]), but we rather were inspired by the patterns intents. As our main goal in this work was to improve the traceability of design patterns, we chose to end with exactly one aspect (respectively one hyperslice) for each pattern instantiation, so that it be easy to identify them in the code. This also improves readability, traceability, adaptability and evolution of both the code relative to the pattern and the one relative to the classes on which it is applied. Furthermore, the number of participants involved in a pattern is significantly reduced.

To illustrate the approach we proposed above, we explain below how AspectJ (the AOP programming language) can be used to implement and improve the GoF *Strategy* pattern. We first provide a concise comprehensive overview of what AOP is: we briefly recall its basic concepts.

2.1.1 Aspect-oriented programming key concepts

AOP [18] aims at organizing programs by decomposing them into *aspects* and classes. An aspect is a software decomposition unit that encapsulates a concern which is transversal to the application considered split into classes. "Code scattering" and "code tangling" problems may be resolved with the help of such a dual decomposition in aspects and classes: one can define as many aspects related to a class as it is necessary to put in them the code that would be otherwise mixed with the code which is very relevant to the class, and an aspect can hold all the code that would be otherwise scattered in several classes.

Interacting between classes and aspects can be defined using *join points* and *pointcuts* (*i.e.* collections of joint points), some points of the execution flow of an application. One can for instance consider as join points, in AspectJ [17], calls to an operation, return from its execution, or read or write access to an attribute [17]. Crosscutting codes that execute when an application reaches a join point are defined within *advices*. Aspects can also affect the static type hierarchy of program classes. They can add new operations and attributes to a class (*introduction*), or declare that a class extends a new super-class [17] (*parent declarations*). Composition of aspects and components is called *weaving* [18] and it generally takes place at compile time. To delay this composition until compilation, rather than to have it done at code writing, reduces the coupling of aspects and classes and provides new reuse perspectives.

2.1.2 AspectJ implementation of Strategy

Strategy's intent is to define a family of interchangeable encapsulated algorithms [7]. In other words, it allows giving polymorphic definitions of a method for the instances of the same class. Figure 1 shows the object-oriented structure of this pattern.

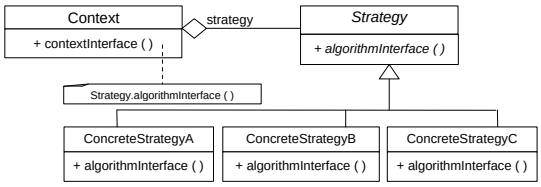


Figure 1. *Strategy* object-oriented structure [7].

Each subclass of the Strategy abstract class holds a different definition of a method named algorithmInterface() that is supposed to be applied on instances of Context. We propose to define a hook operation algorithmInterface() within the Context class (line 0 - Figure 2), that does nothing by default. We then propose to gather the various definitions of the polymorphic behavior (named defaultAlgorithm(), algorithm1(), algorithm2(),… lines 3-6 in Figure 2) in one crosscutting concern ContextStrategies for each instance of the *Strategy* pattern.

```
0  privileged aspect ContextStrategies pertarget  ( target (Context) && call (void Context.algorithmInterface ())) {
1      private int Context.strategy =  0;
2      public Context.new(int strategy) { this.strategy=strategy ; }
3      private defaultAlgorithm() { // the default algorithm }
4      private algorithm1() { … };
5      …
6      private algorithmn() { … };
7      pointcut performAlgorithm (Context c): target (c) && execution (void Context.algorithmInterface ())
8      void around (Context c): performAlgorithm (c){
9          switch (c.strategy) {
10             case 1: algorithm1(); break;
11             case 2: algorithm2(); break;
12             …
13             default: defaultAlgorithm(); }
14     }
15 }
```

Figure 2. Outline of the AspectJ code for the ContextStrategies aspect [7].

[1] Source files for these implementations, along with other materials (detailed specifications of the meta-models, models of the GoF patterns) related to the work we present in this paper are available from [27].

ContextStrategies also introduces an attribute strategy (line 1) and a parameterized constructor Context(int stg) (line 2) in the Context class. The strategy attribute is used for the internal representation of the chosen strategy in each instance of Context. A named pointcut PerformAlgorithm (line 7) intercepts all calls to the Context's algorithmInterface() method, in order to replace each of its invocations by the invocation of the appropriate algorithm. An advice (lines 8-14) is then defined so that it invokes, in its turn, one of the definitions of the polymorphic behavior, depending on the actual value of the strategy attribute hold by the receiver.

2.2 A general meta-model for aspect-oriented design modeling

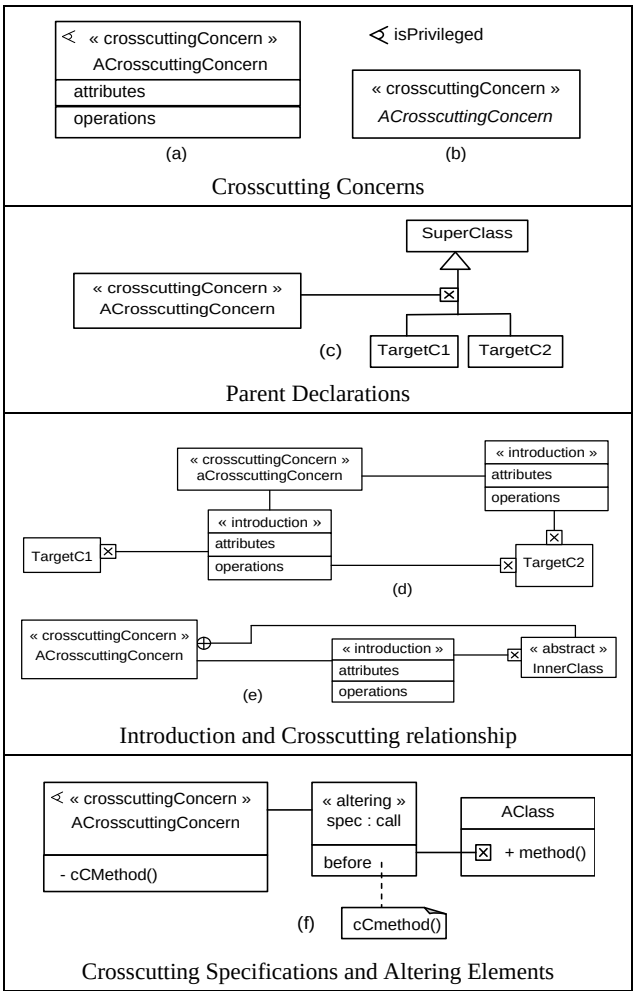
Aspect orientation can be used to improve object-oriented design patterns. However, due to a certain lack of consensus on what are the basic aspect-oriented concepts and mechanisms and the diversity of the aspect-oriented programming languages, most of the aspect-oriented design modeling languages proposed today are specific to a particular programming technique (AOP [2], Composition patterns [4], Aspectual collaborations [15]…) or language (AspectJ [25], [26], Hyper/J [24]…). This makes it difficult to express new pattern structures in a way that is not dependent from a specific programming language. We argue that a more abstract design modeling language is needed to fully express design patterns in a programming language independent manner.

In [12] we propose an extension of the UML meta-model for aspect-oriented concepts and relations that allows for the expression of language-independent aspect-oriented design patterns. We worked out this general meta-model by identifying the common concepts and relations of both AspectJ/UML [12] and HyperJ/UML [12], two specific meta-models that we propose respectively to AspectJ and Hyper/J. Transformation rules can then be applied for migrating models that are instances of the general meta-model to the instances of one of the specific meta-models [12]. We applied the general meta-model on the 23 GoF design patterns and noticed similarities between most of their aspect-oriented structures. The analysis of these similarities led us to identify new aspect-oriented design patterns that are organized in a smaller set of 8 patterns. This encouraged us to work further on the categorization and description of such "aspect aware" design patterns. We propose here to briefly recall the concrete syntax of Aspect/UML that we use to describe the aspect pattern structures. Table 1 summarizes the main presentation elements of the concrete syntax: graphical presentations of Aspect/UML model elements.

2.2.1 Crosscutting Concern

Crosscutting concerns are units encapsulating concerns that cannot be modularized using traditional programming techniques. Like classes in UML, each crosscutting concern is represented as a rectangle with three compartments including respectively its name, attributes and operations (cf. Table 1(a)). The name of a crosscutting concern is yet prefixed by the stereotype «crosscuttingConcern». In addition, a new symbol representing an eye can be used to indicate that a crosscutting concern is privileged (cf. Table 1(a)). A crosscutting concern may be drawn showing only its name; it can also be declared as abstract (Table 1(b)).

Table 1. Aspect/UML notation



2.2.2 Parent Declaration

Parent declarations may add super classes to one or more existing ones, by using generalization/specialization relationships. As well as, they can add several realization relationships between one interface and one or more existing classes. We choose to represent parent declarations by simply drawing crosscutting relationships between the crosscutting concern that declares them, and the added generalization/specialization or realization relationships, as shown in Table 1(c).

2.2.3 Introduction and Crosscutting relation

Crosscutting concerns can introduce one or more features (attributes and operations) in several target types. To represent such introductions, we choose to gather them by target. Each introductions group is rendered as a rectangle with three compartments, including respectively the stereotype «introduction» (introductions are anonymous), the newly added attributes, and operations (Table 1(d)). It is attached to its crosscutting concern by using a simple solid line, and to its target type by using a solid line with a square that contains a cross pointing the target. Such a line represents the crosscutting relationship. Table 1(e) shows some examples of introductions in an inner class.

2.2.4 Crosscutting Specification and Altering Element

Crosscutting specifications indicate when (i.e. *call* or *initialization*) and where (i.e. target behavioural features) altering elements (i.e. features of crosscutting concerns that affect the behaviour of base classifier) have to take places. Each crosscutting specification and its associated altering elements are represented together as rectangle with two compartments. The first compartment represents the crosscutting specification; it contains the stereotype «altering» as well as the description of the crosscutting specification in the form:

[crosscuttingSpecification_name]:
crosscuttingSpecification_type
with crosscuttingSpecification_type::=call|initialization.

Crosscutting specifications are attached to their crosscutting concerns with a simple solid line and to their target behavioural features through their altering elements (cf. Table 1(f)). A crosscutting specification may be declared abstract. In such a case it is rendered by its name in italics and an undefined type. The second compartment of an «altering» bloc specifies the set of the altering elements (cf. Table 1(f)). The syntax we retain to describe the altering elements is as follows.

alteringElement_type ["(" arguments ")"][':' return_type]
with arguments ::= argument_name ':' argument_ type
[',' arguments]
alteringElement_type ::= before | after | combination |
replacement | narrowing

3. AN ASPECT PATTERN DESCRIPTION FORMALISM

In order to retain complete description of aspect-oriented design patterns (in a way similar to the description formalism used in [7]), we adapted P-Sigma [5], a general purpose pattern description formalism that was defined in our research team, into AP-Sigma. We just extend it with some keywords directly related to aspect-orientation that we use for the description of the pattern's intent and forces. The main objective of such formalism is to normalize patterns description in order to ease their reuse and allow organizing them by characterizing their relations. AP-Sigma is composed of three parts: *Interface*, *Realization* and *Relations*. Those parts are detailed below with some excerpts of the description of *Strategy*.

3.1 Pattern Interface

A pattern interface is composed of five items aimed to ease pattern selection. Table 2 illustrates these items.

Table 2. Interface of the *Strategy* pattern

Identifier	*Strategy*
Classification	object ^ alter ^ replacement ^ instantiation
Problem	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Context	*Applicability.* Use the strategy pattern when: - many related classes differ only in their behavior, - we need different variants of an algorithm…
Forces	*Forces.* Ease the adaptation and evolution of algorithm… *Qualities*. readability ^ encapsulation ^ evolution

3.1.1 Identifier

The identifier item holds the name of the pattern.

3.1.2 Classification

The classification item defines the pattern intent through a collection of domain keywords (terms of the application domain of the patterns). It provides an intuitive domain classification. It is constructed as a logical expression. Example keywords are: (1) *Class* and *Object*, specify whether the pattern applies to classes or objects, (2) *Add* and *Alter*, indicates if the pattern deal with the structures of base classes or with their behaviors. If the impact type is *Alter* we also use the following keywords for extra precision: *before*, *after*, *combination*, *narrowing* or *replacement*, in addition to *instantiation* and *execution* keywords that specify the dynamic context of altering.

3.1.3 Problem

The problem item specifies in details the problem addressed by the pattern.

3.1.4 Context

The pattern context is subdivided into two fields: *Applicability* and *Pre-condition*. The former field identifies, the typical situations in which the pattern can be applied but that are special cases of the problem addressed by the pattern. The latter field specifies the eventual pre-condition that have to be verified prior to the application of the pattern. Such pre-conditions are in general defined by models.

3.1.5 Forces

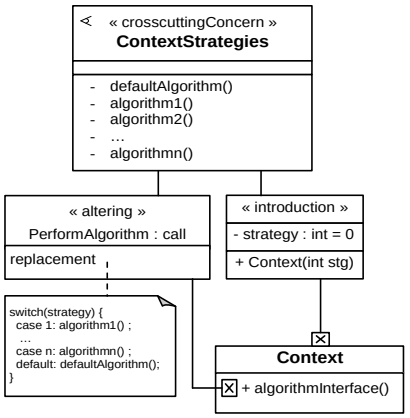
The pattern forces item consists of two fields. It mainly specifies the pattern contributions through a collection of

quality criteria. The two fields are: *Forces* and *Qualities*. *Forces* is a text field that discusses the pros and cons of applying the pattern. *Qualities* holds a logical expression based on one or more criteria denoting the intended benefits of the pattern solution (i.e. *code reuse*, *traceability*…).

3.2 Pattern Realization

While the interface part of a pattern description focus more on the problem description, the realization part is devoted to the description of the pattern solution. Its four items are detailed below (see also Table 3 for the *Strategy*'s solution description).

Table 3. Realization of the *Strategy* pattern

Solution	*Solution*. Define in a *crosscutting concern*: - an *introduction* of an attribute used for the internal representation of the actual strategy, to the class Context, - a second *introduction* to the same class of a parameterized constructor to set the actual strategy, - the different variants of the considered behavior, - an *altering element* that overrides the impacted behavioral feature with the adequate variant, - a *crosscutting specification* specifying the impacted operation in the class Context. *Class/Aspect diagram*  **Figure 3. Strategy aspect-oriented structure.**
Application case	An example of the pattern application illustrated with the corresponding instances diagram and the sample code.
Consequences	One of the main advantages of this pattern solution is that the common code of the different variants of the behavior is gathered out in a common sub-operations/methods...

3.2.1 Solution

The solution item consists of two fields (*Solution* and *Concern Diagram*) describing the pattern solution in terms of the result. The first field is a text, while the second is an instance diagram of the Aspect/UML meta-model.

3.2.2 Application case

The application case item gives an example of the pattern application. It is optional, but recommended in order to facilitate the understanding of the pattern solution.

3.2.3 Consequences

The consequences item is a text that discusses the consequences induced by the patterns application.

3.2.4 Alternatives

The alternatives item specifies the possible alternative solutions.

3.3 Pattern Relations

The pattern relations part is composed of three items corresponding to the three types of relations between patterns: *uses*, *refines* and *alternativeOf*. For each relation, the item holds a list of related patterns. In the case of *Strategy* we can identify, for example, a *refines* relation with *Class Polymorphic Behavior* (see section 4.1) and two *uses* relations with *Add Features* and *Alter Bahaviors* (see section 4.3).

4. ABSTRACTING GOF PATTERNS INTO ASPECT-ORIENTED PATTERNS

While expressing the newly proposed aspect-oriented structures of the GoF design patterns, based on the implementations that were proposed in both our previous work and those of [14], with respect to our general meta-model, we have noticed similarities between several pattern structures. This is not very surprising in the sense that differences between related artefacts tend to disappear when described at higher level of abstraction. Indeed a pattern structure expressed as an instance diagram of the general meta-model is more abstract than those expressed in instance diagrams of language-specific meta-models. We also noticed that appearing similarities were due to the use of aspect-orientation as they were not so obvious in the corresponding strictly object-oriented structures. We thus begin to compare more thoroughly the aspect-oriented structures we have in order to abstract the 23 existing design patterns to 8 more general ones. As examples, we detail here four of the newly proposed Aspect patterns: *Class Polymorphic Behavior*, *Class/Instance Polymorphic Behavior with Standalone Classes* and *Add New Role*. After that, we briefly describe the four others while presenting all relations that exist between the whole 8 patterns.

4.1 Class Polymorphic Behavior

We consider here the *Class Polymorphic Behavior* (CPB) pattern that we identify as an abstraction of the GoF *Strategy*, *Template Method*, *Factory Method* and *Abstract Factory* patterns when considered in an aspect-oriented context.

The problem addressed by the *Strategy* pattern is defined by the problem item in table 2. Its aspect-oriented structure is as well specified in figure 3 (see table 3). The *Template Method* pattern's intent is to define the skeleton of an algorithm in an operation, while deporting some parts of it into subclasses. *Template Method* lets subclasses redefine certain parts of the algorithm without changing the algorithm's structure [7]. The figure 4 shows the aspect-oriented structure that we propose for this pattern.

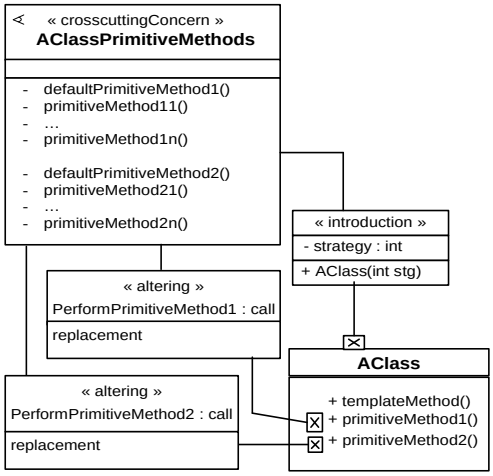


Figure 4. *Template Method* aspect-oriented structure.

Although the two patterns address different problems, by comparing their proposed aspect-oriented structures we can observe several similarities. These similarities also concern the aspect-oriented structures that are respectively proposed for the GoF *Factory Method* and *Abstract Factory* patterns. *Factory Method* defines an interface for creating an object, but lets subclasses decide which class to instantiate [7]. *Abstract Factory* provides an interface for creating families of related or dependent objects without specifying their concrete classes [7]. Figures 5 and 6 give the aspect-oriented structures that we propose respectively for *Factory Method* and *Abstract Factory*.

In the four structures (figures 3, 4, 5 and 6) we can distinguish the following elements:

- a crosscutting concern and a context class,
- an introduction of an attribute specifying the appropriate behavior of each context class instance,
- an introduction of a new parameterized constructor for setting the introduced attribute,
- different variants of the different affected operations,
- one or more crosscutting specifications specifying the affected operations,
- one or more altering element that have to perform in the place of the affected behavioral operations.

All these similarities occult the different intents of these four patterns, making it possible to define a more general aspect-oriented design pattern that address a more general problem: "give a polymorphic behavior to the instances of a given context class, while keeping unchangeable the context class definition". This is the problem addressed by the *Class Polymorphic Behavior* pattern that we propose. Table 4 briefly describes this pattern with respect to the AP-Sigma formalism.

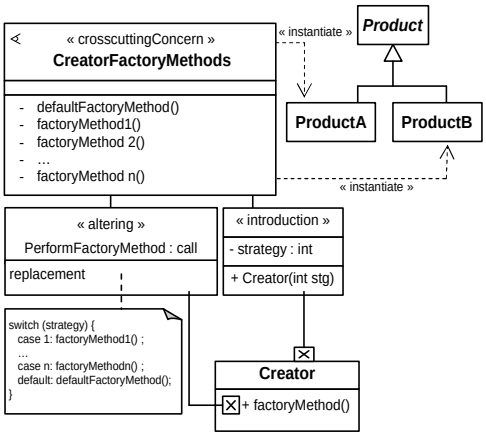


Figure 5. *Factory Method* aspect-oriented structure.

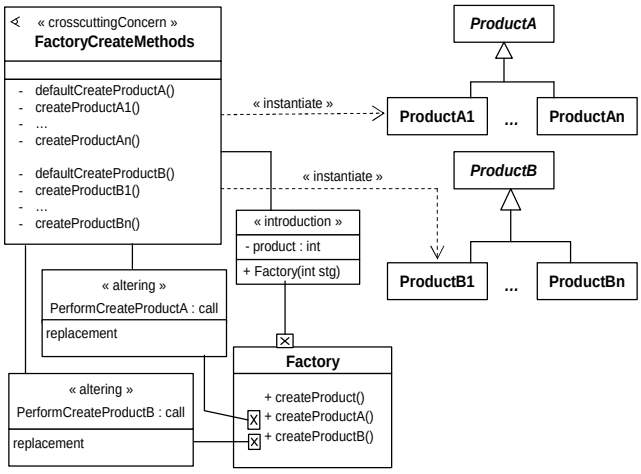


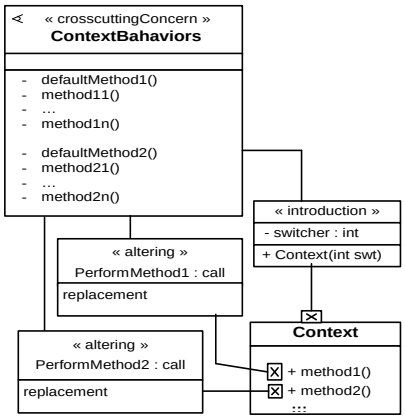
Figure 6. *Abstract Factory* aspect-oriented structure.

4.2 Instance/Class Polymorphic Behavior with Standalone Classes

4.2.1 Class Polymorphic Behavior with Standalone Classes

Figure 7 shows the aspect-oriented structure of the GoF *Builder* pattern. *Builder* proposes to separate the construction of a complex object from its representation so that the same construction process can create different representations [7].

Table 4. Description of the *Class Polymorphic Behavior pattern*

Identifier	*Class Polymorphic Behavior*
Classification	object ^ alter ^ replacement ^ instantiation
Problem	Give a polymorphic behavior to the instances of a given context class, while keeping unchangeable the class definition. The involved behavior can either concern one or more operations of the context class. Each instance of the context class has to be configured, at the instantiation time, with the appropriate variant of the polymorphic behavior.
Context	*Applicability*. Use the *Class Polymorphic Behavior* pattern in the following cases: - a given class must offer different variants of a specific behavior for its instances. Such behavior cans either involves one or more hook operations of the context class (Hook operations are concrete operations that provide default definition for the involved polymorphic behavior. A hook operation often does nothing by default). - we need different variants of a given behavior.
Forces	- *Let the definition of the context class safe*. The context class defines the default behavior of its involved operations. All the other variants are gathered out and encapsulated separately within a crosscutting concern. - *An alternative to sub-classing.* Inheritance offers another way to support all various behaviors of the involved hook operations. You can always directly subclass the context class to define its different behaviors within its subclasses. But, this increases the number of classes in the system. In addition, this alternative mixes the hook operation's various methods with the primary class's concern, making the class hard to understand, to maintain and to extend. The same way, the different various methods are scattering all over the subclasses so what they are not easy to maintain. Encapsulating all these specific behaviors in a crosscutting concern lets you maintain and extend independently the class's primary concern and the various methods of the hook operations. - *The Various behaviors are encapsulated separately, allowing gathering out their common functionalities.* The various behaviors are completely defined and encapsulated within the crosscutting concern. This makes it easy to gather out all common functionalities of the different behaviors.
Solution	*Solution*. 1. Identify which hook operations (method1(), method2()…) of the Context class are designed for altering. 2. Define a *crosscutting concern* (ContextBehaviors) that has to include the definition of an attribute *introduction* (switcher) and a parameterised constructor *introduction* (Context (int swt)), in the Context class. The switcher attribute is used for the internal representation of the appropriate variant of the involved behavior for each instance of the Context class. The parameterised constructor has to take the value to assign to the switcher attribute at the instantiation time. 3. For each hook operation that must be altered: - gather out and encapsulate within the crosscutting concern all various definitions of the hook operation (defaultMethodk(), methodk1(), methodk2()…), devoted for all specific Context's instances. - define within the crosscutting concern, a *crosscutting specification* (PerformMethodk…) that has to intercept all calls to the hook operation. - define as well a replacement *altering element* based on the crosscutting specification. This altering element has to replace every invocation of the hook operation with the invocation of exactly one of its various definitions, depending on the assigned value of the switcher attribute hold by the receiver instance. *Class/Aspect Diagram.* 
Consequences	- Common sub-behaviors of the various definitions of the polymorphic behavior can be gathered out in common sub-operations.

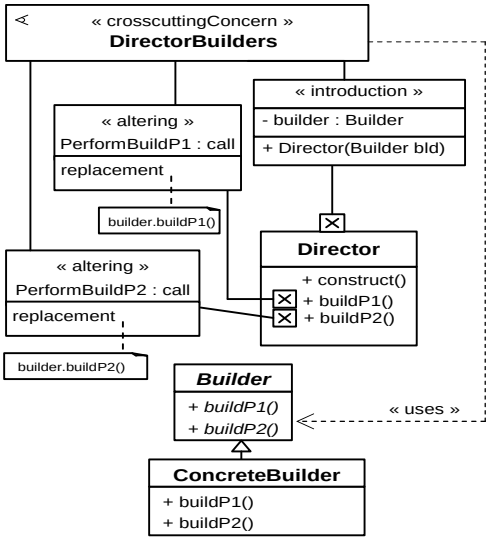


Figure 7. *Builder* aspect-oriented structure.

The GoF *Command* pattern proposes to encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations [7]. Figure 8 gives the aspect-oriented structure of the *Command* pattern.

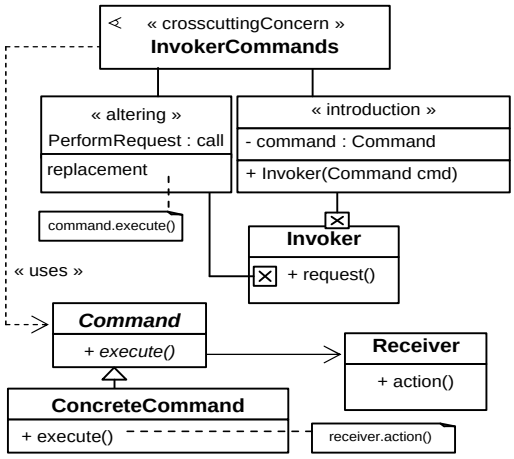


Figure 8. *Command* aspect-oriented structure.

While comparing the aspect-oriented structures of these two patterns, we can also observe several similarities. In fact, *Builder* and *Command* propose in general to mainly adapt one or more operations (buildP1(), buildP2()... and request()) of a context class (Director in *Builder* and Invoker in *Command*), while giving a polymorphic behavior to the instances of this class. To do this, they propose, in opposite to the previous four patterns, to gather out the various alternatives of the impacted operations in several behavioral classes (ConcreteBuilder in the *Builder* pattern and ConcreteCommand in the *Command* pattern). They propose then to associate a behavioral delegate object (i.e. an instance of a certain behavioral class) with each instance of the context class at the instantiation time, thereby each instance can automatically forward all the requests from their clients to its delegate object. The collaborations between the context class's instances and their delegate behavioral objects are possible thanks to an attribute introduction (builder in *Builder* and command in *Command*), as well as an introduction of a new parameterised constructor (Director(Builder bld) in *Builder* and Invoker(Command cmd) in *Command*). Crosscutting specifications (PerformRequest, PerformBuildP1…) intercept all calls to the context class's affected operations, in order to replace each of their invocations by the invocation of the appropriate variant hold by the delegate objects. Altering elements are then defined so that they invoke, in their turns, one of the definitions of the polymorphic behavior, depending on the actual delegate object.

All these similarities make it possible to define a new aspect-oriented design pattern, the *Class Polymorphic Behavior with Standalone Classes* (CPB-SC). Figure 9 shows the aspect-oriented structure of this newly identified pattern (see [27] for a complete description of this pattern). Note that *Class Polymorphic Behavior with Standalone Classes* proposes an alternative solution to the *Class Polymorphic Behavior* pattern, with different forces and consequences. We thus identify an *alternativeOf* relationship between these two patterns.

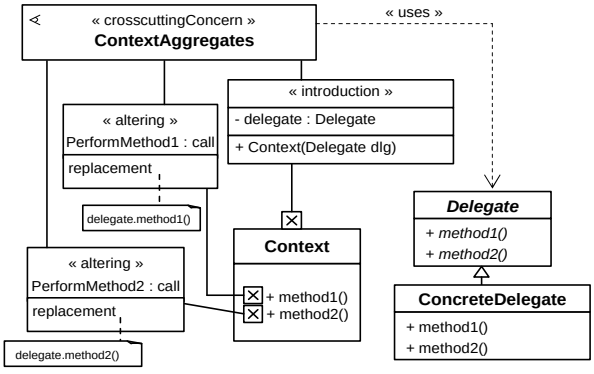


Figure 9. *Class Polymorphic Behavior with Standalone Classes* aspect-oriented structure.

4.2.2 Instance Polymorphic Behavior with Standalone Classes

As we proceeded for *Command* and *Builder*, we considered in the same way *State* and *Bridge* patterns to define the *Instance Polymorphic Behavior with Standalone Classes* (IPB-SC) pattern (see figure 10).

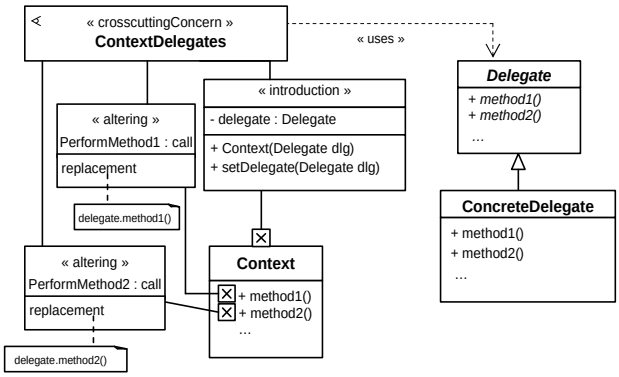


Figure 10. ***Instance Polymorphic Behavior with Standalone Classes*** **aspect-oriented structure.**

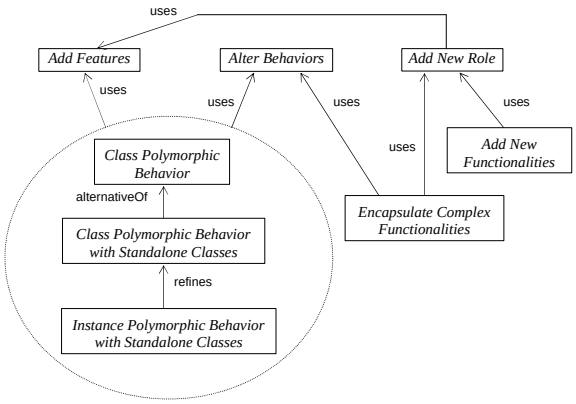


Figure 11. Cartography of the 8 new aspect-oriented patterns.

State and *Bridge* propose, as well, to give a polymorphic behavior to the instances of a given context class, while gathering out and encapsulating all various definitions of the involved behavior in additional standalone behavioral classes. They make it possible, however, to change the delegate object of a given instance during its execution. To do this, the aspect-oriented structures of the two considered patterns propose to define new setter operations (setState() in *State* and setImplementor() in *Bridge*) designed to be introduced in the context classes (Context in *State* and Abstraction in *Bridge*). Such operations make it possible to change the delegate objects at the execution time. *State* and *Bridge* differ however by the way in which the behavioral delegate objects are changed. *State* pattern proposes, for example, to define the criteria for delegate state object transitions in the crosscutting concern. This is possible since such criteria are fixed in advance, whereas they depend on client in the case of the *Bridge* pattern and they can't therefore be implemented within the crosscutting concern.

4.3 Other patterns

To cover the GoF pattern catalogue, we also analyzed all the aspect-oriented structures of the remainder patterns. We mainly identified three "primitive" patterns: *Add Features*, *Alter Behaviors* and *Add New Role*. We also defined two other patterns: *Add New Functionalities* and *Encapsulate Complex Functionality* that are added to the patterns that we detailed into 4.1 and 4.2, as well as, to the three "primitive" ones. Figure 11 outlines all relations that exist between the whole 8 new Aspect patterns.

We propose in what follows to briefly describe the last five new patterns. We mainly clarify their problems, contexts, and often their solutions, as well as we specify the GoF patterns from which they result.

4.3.1 Add Features (AF)

Add Features allows adding new properties (i.e. attributes) and/or behaviors (i.e. operations) to a given concrete or abstract class, while keeping unchangeable its definition. We use such a pattern when a non-central concern has to interfere with a given class, dictating that one or more new features have to be added to the former class without changing its definition. We defined the pattern by abstracting similarities that exist between the GoF *Adapter* and *Visitor* patterns, which are mainly and exclusively concerned by the *Add Features*'s intention.

4.3.2 Alter Behaviors (AB)

Alter Behaviors adapts the behavior of a given class that must interfere with a concern other than its primary concern, by dynamically and transparently altering one or more of its involved behavioral features (i.e. operations). The class's definition must be unchanged. We identified *Alter Behaviors* by considering commonalities that exist between the GoF *Singleton*, *Proxy* and *Decorator* patterns, which address the same problem that the *Alter Behaviors* pattern.

4.3.3 Add New Role (ANR)

Add New Role allows adding common behavior (that can take the form of one or more behavioral features) and/or properties (i.e. attributes) to one or more different classes without changing their definitions. Figure 12 shows the aspect-oriented structure of the pattern.

We obtained this pattern by abstracting similarities that exist between the GoF *Composite* and *Prototype* patterns. *Add New Role* pattern is designed to be used when different objects should have common features from a certain perspective or a subjective view on their system. Note that the needed additional features are not intrinsic to the involved classes and their objects. To do this, *Add New Role* proposes to define within a crosscutting concern an abstract class (Role) that is designed to be inherited by all of the involved classes (Context1, Context2…). The needed

common features are therefore introduced into the abstract super-class, by using the *Add Features* pattern. Added behavioral features can then be eventually redefined by several concrete operations that have to be introduced into the involved classes.

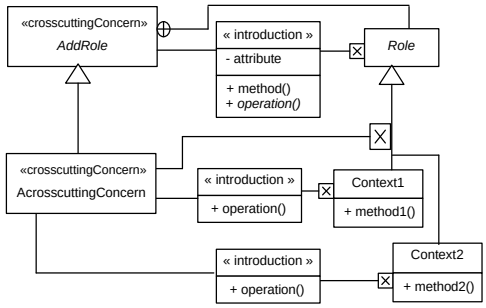


Figure 12. Add New Role aspect-oriented structure.

4.3.4 Add New Functionalities (ANF)

Add New functionalities proposes to add new functionalities to one or more existing classes, without modifying their primary behaviors. This is the implied intention, for example, of the GoF *Memento* and *Iterator* patterns. Each of the aspect-oriented structures of these two patterns is mainly based on the definition of new additional concrete class that holds the needed functionalities for the existing one. Such a class is defined within a crosscutting concern that has to abstract coupling between the newly added class and the existing one. To do this, *Add New Functionalities* uses the *Add New Role* pattern for defining two additional abstract classes that are designed to be inherited respectively by the two former classes.

4.3.5 Encapsulate Complex Functionality (ECF)

Encapsulate Complex Functionality proposes to carry out a complex functionality defined by collaborations between several existing classes that play different roles in the context of this functionality. This is the implied intention of the GoF patterns that mainly deal with collaborations between several classes, such as, *Chain of Responsibility*, *Mediator*, *Observer* and *Flyweight*. All aspect-oriented structures of the four patterns use the *Add New Role* pattern in order to extend the involved classes, and provide them all features they need to complete their common complex functionality. They also use the *Alter Behaviors* pattern, thereby altering the basic behaviors of the concerned classes and completely define the collaboration protocol of the needed functionality.

5. DISCUSSION AND ANALYSIS OF THE 8 NEW PROPOSED PATTERNS

We presented throughout the previous sections 8 new Aspect patterns: 3 “primitive” patterns and 5 “advanced” ones, all of them worked out from the aspect-oriented structures of the GoF design patterns [7]. The 8 new patterns aim therefore to be general-purpose rather than case specific or pattern specific. Still, what about the relevance of their addressed problems and proposed solutions?

A certain shortage of available Aspect applications makes it difficult to find out in which extent the newly proposed patterns are used in aspect-oriented software developments. In order to mitigate this problem, we propose here to identify all relations that exist between the newly proposed patterns and all of the GoF design patterns. In fact, although each one of the 8 new Aspect pattern was exclusively abstracted from some of the GoF design patterns (in particular from those who are primarily concerned by its specific intention), some of the newly proposed patterns (especially the primitive ones) have close connections with almost all of the 23 GoF patterns. Indeed, such example relations make it possible to concrete the utility of the new patterns, and to confirm the importance of their problems and solutions. Table 5 shows all relations that exist between the 8 new patterns and the aspect-oriented structures of the GoF patterns that we propose in our own work (line 1), but also those proposed by [14] (line 2). Note that the 3 “primitive” patterns are largely used by almost all of the 23 GoF patterns, whereas, the 5 “advanced” ones are less related to the 23 GoF patterns because they deal with more specific problems (contrary to the 3 “primitive” patterns).

In the same way, we have considered some other related works which deal with discovering new problems and solutions that are useful in aspect-oriented design and implementation. We mainly aim to analyze the relations that exist between these problems and solutions and the 8 new patterns. Table 6 summarizes, for instance, the relations that exist between 5 of our newly proposed patterns and the refactorings of [19], as well as the directives of [3].

[19] proposes to review traditional object-oriented codes in the light of aspect-orientation. They propose a collection of aspect-oriented refactorings covering both the extraction of aspects from object-oriented legacy code and the subsequent tidying up of the resulting aspects. Our *Add Features*, *Alter Behaviors*, *Add New Role* and *Add New Functionalities* patterns belong to the set of the proposed refactorings. They are in fact used by most of all the refactorings in order to completely or partially resolve their problems. *Move Field from Class to Inter-type* and *Move Method from Class to Inter-type* use, for example, the *Add Features* pattern. The *Extract Fragment into Advice* uses however the *Alter Behaviors* pattern, while *Extract Features into Aspect* uses in its solution both the *Add Features* and *Alter Behaviors* patterns. Besides, *Extract Inner Class to Standalone* uses in it turn the *Add New Functionalities* pattern. Finally, *Generalise Target Type with Marker Interface* proposes the same solution that our *Add New Role pattern*.

Table 5. Relations between the newly proposed patterns and the aspect-oriented structures of GoF patterns

Patterns	*AF*	*AB*	*ANR*	*CPB*	*CPB-SC*	*IPB-SC*	*ANF*	*ECF*
Abstract Factory	uses	uses		uses				
	uses			alternativeOf	alternativeOf			
Builder	uses	uses			uses			
	uses			alternativeOf	alternativeOf			
Factory Method	uses	uses		uses				
	uses			alternativeOf	alternativeOf			
Prototype	uses		uses					
	uses		alternativeOf					
Singleton	uses	uses						
		uses	uses					
Adapter	uses/may refines							
	refines							
Bridge	uses	uses				Uses		
	uses							
Composite	uses		uses					
	uses		alternativeOf					
Decorator	may uses	uses						
	may uses	uses						
Flyweight	uses	uses	uses					uses
	uses							alternativeOf
Proxy	may uses	uses	uses					
	may uses	uses	uses					
Chain of Responsibility	uses	uses	uses					uses
	uses	uses						alternativeOf
Command	uses	uses	uses		uses			
	uses	uses	uses			alternativeOf		
Interpreter	uses							
	uses							
Iterator	uses		uses				uses	
	uses						alternativeOf	
Mediator	uses	uses	uses					uses
	uses	uses						alternativeOf
Memento	uses		uses				uses	
	uses						alternativeOf	
Observer	uses	uses	uses					uses
	uses	uses						alternativeOf
State	uses	uses				Uses		
	uses	uses						
Strategy	uses	uses		uses				
	uses	uses	uses			alternativeOf		
Template Method	uses	uses		uses				
	uses			alternativeOf	alternativeOf			
Visitor	uses							
	uses		uses					

Table 6. Relations between 5 of the newly proposed patterns, the directives of [3] and the refactorings of the [19]

	Patterns	*AF*	*AB*	*ANR*	*ANF*	*ECF*
Directives of [3]	***Aspects for Collaboration***					alternativeOf
	Aspects for Evolution			extends		
	Aspects for Views	uses				
	Aspects and Obliviousness	uses	uses			
Refactorings of [19]	***Move Field from Class to Inter-type***	uses				
	Move Method from Class to Inter-type	uses				
	Extract Fragment into Advice		uses			
	Extract Feature into Aspects	uses	uses			
	Extract Inner Class to Standalone				uses	
	Generalise Target Type with Marker Interface			uses		

[3] propose some preliminary directives for good aspect-oriented design, based on the aSideML language [2]. The directives they propose are mainly derived from their AspectJ applications (such as Portalware MAS [8]), but also from several other applications. We can distinguish for example the *Aspects for Collaboration* directive, which proposes to encapsulate crosscutting concerns that are defined as collaborations between several classes within an aspect. Note that this directive deals thus with the same problem that *Encapsulate Complex Functionality*, it proposes however an alternative solution to this pattern. Another directive that we can consider here is *Aspects for Evolution*; it proposes to extend an existing class by adding new features so that this class can play a new role. Our *Add New Role* pattern refines the intention of this directive; it allows in fact adding new role to several classes at the same time. He can thus be used to address the problem of the *Aspects for Evolution* directive. The *Aspects for Views* directive proposes to offer several interfaces to a given class, which must be used by different clients in different contexts. This directive uses our *Add Features* pattern in its solution. Finally we can consider the *Aspects and Obliviousness* directive that uses in its solution the *Alter Behaviors* and *Add Features* patterns.

All these relations help to approve the utility of the whole patterns that we propose. We do not claim however these patterns are complete, but we believe they extend the existing proposals [2, 4] thus contributing to further mature the Aspect approach. The analysis of all of the aspect-oriented structures we considered in this section allowed us, in addition, to identify new alternative solutions to some of the 8 newly proposed patterns advancing thereby the new pattern's descriptions (see [27] for a complete description of all of the 8 proposed patterns).

6. CONCLUSION

In order to validate our aspect-oriented modeling approach, we used it for expressing and providing aspect language-independent structures of the 23 GoF patterns. These structures, being more general than their corresponding language-specific ones, helped us in identifying new 8 aspect patterns that are more general than are those of [7]. We then proposed an aspect pattern description formalism that we have used to completely describe the newly proposed patterns and organize them in a patterns system (interested readers may eventually check more complete descriptions in [27] as well as get more details on the abstract and concrete syntaxes, well-formedness rules and semantics of our three UML extensions). It is no wonder that the number of significant patterns is reduced. This comes from the higher abstraction level of the new aspect GoF pattern structures description.

We think that the proposed patterns can be considered as a first step in identifying new aspect-oriented design patterns, though they are exclusively based on the review of the object-oriented design patterns and not on analysis of existing aspect-oriented software developments. In fact, we argue that the identification of the recurring problems and their possible solutions in aspect-oriented design remains a significant work. We think consequently, in particular, that other aspect-oriented design patterns remain to be discovered. The identification of such Aspect patterns requires a different approach from that which we adopted, such as detailed and exhaustive analysis of several Aspect systems, that still today difficult because of the low number of existing aspect-oriented applications. Our approach being however based on the transformation of Object patterns, the newly proposed Aspect patterns can therefore be used within the evolution of an object-oriented design towards an aspect-oriented design. Such an evolution makes it possible to mitigate the consequences of "code scattering" and "code tangling" problems that characterize Object systems, in order to facilitate their evolution and to increase their reusability. Now that we have thoroughly defined a system of aspect-oriented design patterns, we still need to integrate the new patterns in an engineering method. We think, in fact, that it is important to produce new process patterns so that our proposals facilitate the use of the newly introduced aspect-oriented design patterns.

7. REFERENCES

[1] Buschman, F. What is a pattern?. *Object Expert, vol. 1, n°3*, (1996), 17–18.

[2] Chavez C., A Model Driven Approach to Aspect-Oriented Design. PhD thesis, Brazil, 2004.

[3] Chavez C., Lucena C. Guidelines for Aspect-Oriented Design. *Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'04)*, Brasília. Anais do Primeiro, 2004.

[4] Clarke S. Composition of Object-Oriented Software Design Models. PhD Thesis, Dublin City University, 2001.

[5] Conte A., Fredj M., Hassine I., Giraudin J.P., Rieu D. A Tool and a Formalism to Design and Apply Patterns. *In Proceedings of the 8th international conference OOIS 2002*, Montpelier, France, September 2002.

[6] Gamma, E. *Object-Oriented Software Developments based on ET++: Design Patterns, Class Library, Tools*. PhD thesis, University of Zürich, 1991.

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

[8] Garcia A., Silva V., Chavez C., Lucena C., "Engineering Muti-Agent Systems with Patterns and Aspects". Journal of the Brazilian Computer Society, 2002, 8(1):57-72.

[9] Hachani, O. *Utilisation de la programmation par aspects dans l'implémentation de patrons de conception*. Mémoire de DEA, Université Grenoble 1, 2002.

[10] Hachani, O., Bardou, D.: Using Aspect-Oriented Programming for Design Patterns Implementation. *OOIS 2002 Workshop on Reuse in Object-Oriented Information Systems Design*, 2002.

[11] Hachani O., Bardou, D. : On Aspect-oriented Technology and Object-Oriented Design Patterns. *ECOOP 2003 Workshop on Analysis of Aspect-Oriented Software*, 2003.

[12] Hachani, O. Gaining language independency in AOD though meta-modeling and model transformation. *2èmes journées sur l'ingénierie dirigée par les modèles (IDM)*, 2006.

[13] Hanenberg S., Oberschulte C., Unland R. Refactoring of Aspect-Oriented Software. Net.ObjectDays 2003, Erfurt, Germany, September 2003.

[14] Hannemann, J., Kiczales, G. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 2002 ACM SIGPLAN Conference on OOPSLA 2002*, SIGPLAN Notices, Vol. 37, N°11, ACM (2002), 161–173.

[15] Hermann, S. Composable design with UFA. *1st International workshop on Aspect-Oriented Modeling with UML in AOSD 2002*, Enschede, The Nethetlands, April 2002.

[16] Hirschfeld, R., Lämmel, R., Wagner, M. Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. *The 3rd German Workshop on Aspect-Oriented Software Development* (AOSD-GI 2003), 2003.

[17] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. An Overview of AspectJ. In *Proceedings of ECOOP 2001*, LNCS, Vol. 2072, Springer (2001), 327–353.

[18] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS, Vol. 1241, Springer (1997), 220–242.

[19] Monteiro M.P.. Catalogue of Refactorings for AspectJ. Technical Report UM-DI-GECSD-200402, Universidade do Minho, Portugal, December 2004.

[20] Noda, N., Kishi, T. Implementing Design Patterns Using Advanced Separation of Concerns. In *OOPSLA 2001 Workshop on AsoC in OOS*, 2001.

[21] Nordberg, M.E. Aspect-Oriented Dependency Inversion. *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[22] Nordberg, M.E. Aspect-Oriented Indirection – Beyond Object-Oriented Design Patterns. *OOPSLA 2001 Workshop "Beyond Design: Patterns (mis)used"*, 2001.

[23] Ossher H., Tarr P.L. Hyper/J™: Multi-dimensional separation of concerns for Java™, In *Proceedings of the ICSE 2000, International Conference on Software Engineering*, Limerick, Ireland, June, 2000.

[24] Philippow I., Riebisch M., Boellert K. The Hyper/UML Approach for Feature Based Software Design. *UML'03 4th Workshop on Aspect-Oriented Modeling with UML*, 2003.

[25] Stein D. *An Aspect-Oriented Design Model Based on AspectJ and UML*. Master Thesis, University of Essen, Germany, 2002.

[26] Suzuki, J., Yamamoto, Y. Extending UML with Aspects: Aspect Support in the Design Phase. *ECOOP'99 Workshop on Aspect-Oriented Programming*, 1999.

[27] http://www-lsr.imag.fr/Les.Personnes/Ouafa.Hachani/works.html.