Conquering Fine-Grained Blends of Design Patterns

L. Sabatucci[1], A.Garcia[2], N. Cacho[2], M. Cossentino[3], and S. Gaglio[1]

[1] Dip. Ingegneria Informatica, University of Palermo, Italy
sabatucci@csai.unipa.it, gaglio@unipa.it
[2] Computing Departement, Lancaster University, United Kingdom
a.garcia@lancaster.ac.uk, n.cacho@lancaster.ac.uk
[3] ICAR-CNR, Consiglio Nazionale delle Ricerche, Palermo, Italy
cossentino@pa.icar.cnr.it

Abstract. The reuse of design patterns in realistic software systems is often a result of blending multiple pattern elements together rather than instantiating them in an isolated manner. Pattern blends can manifest in heterogeneous ways, typically including overlaps and interlaces of inner class members taking part in the patterns implementation. The explicit description of pattern compositions is the key for (i) documenting the structure and the behavior of blended patterns and, (ii) more importantly, supporting the reuse of composite patterns across different software projects. In this context, this paper proposes a fine-grained composition language for describing varying blends of design patterns based on their structural and behavioural semantics. Pattern compositions are specified using an expressive, albeit simple, set of operators that allow for *unifying*, *conjoining*, *concealing* and *externalizing* pattern elements. The reusability and expressiveness of the proposed language are assessed through its application to 32 compositions of GoF patterns recurrently appearing in three different case studies: the OpenOrb middleware, and the JHotDraw and JUnit frameworks.

1 Introduction

Even though design patterns have been widely accepted by industrial and academic organizations, their definition and reuse still impose deep concerns on contemporary software engineers. The pivotal difficulty stems from the fact that pattern solutions are largely sensitive to different contexts where they are reused, especially on how they are combined with each other [1, 5]. Patterns often need to be documented as pair-wise blends of patterns' responsibilities rather than as individual and intact entities. This phenomenon has been recurrently identified in the design of product lines [8], middleware systems [6], and domain-specific frameworks [7]. Not surprisingly there is an increasing empirical evidence that a considerable proportion of code clones in real-world software projects are related to variants of pattern blends [2, 14].

Effective reuse of composite patterns is far from being trivial for several reasons. The symbiotic application of design patterns results in the intricate twine

of pattern participants and the target application [12]. It involves multiple forms of pattern blends, ranging from conservative combinations of pattern elements to different overlaps of fine-grained pattern responsibilities materialized, for instance, by particular actions, attributes or events [7]. Pattern composites usually entail significant morphs of the original pattern solutions through the conjunction or merge of structural and behavioral elements. They should be systematically documented so that they can be unambiguously instantiated, traced and reused within and across software projects. The lack of explicit documentation for recurring compound patterns leads to design rationale being irrecoverable [3]. In fact, pattern composition support has been recognized to be a key element for the usability of pattern languages and underlying development tools [4, 9].

However, after twelve years the Gang-of-Four (GoF) pattern catalogue [10] has been published, effective support for documenting recurring composite patterns is still lacking. One of the main gaps is that pattern composition has been restricted to coarse-grained documentation approaches [18, 21] which do not address structural and behavioural blends of inner participant members [12]. Even though contemporary programming techniques, such as aspect-oriented programming [15] and subject-oriented programming [17], have brought advanced mechanisms for enabling improved pattern composability [13], empirical evidence shows that they do not scale much for coping with modular treatment of pattern composites [7, 11]. Also, emerging model weaving techniques are not tailored to composing different forms of pattern overlaps and interlaces.

In this context, the contribution of this paper is threefold. First, it presents a classification of pattern blends (Section 2) using a real middleware system to their illustration. Second, a design approach is proposed for addressing those varying forms of pattern blends (Section 3). We define a design language for describing fine-grained pattern compositions based on their structural and behavioral semantics. An expressive and simple set of operators is used for *unifying*, *conjoining*, *concealing* and *externalizing* pattern elements (Section 4). Third, the proposed approach is assessed through its application to different open source applications, the *OpenOrb* middleware, and the JHotDraw and JUnit frameworks (Section 5). Our analysis is based on the reuse and expressiveness evaluation of 32 GoF pattern compositions. We also discuss the novel features of our technique on the light of a comparison with existing work (Section 6). Some concluding remarks are reported in Section 7.

2 Heterogeneous Pattern Blends

This Section presents an analysis of heterogeneous forms of pattern blending, which are commonly found in real multi-pattern software systems (Section 2.1). This analysis allows us to provide a classification of the different forms in which design patterns are blended (Section 2.2).

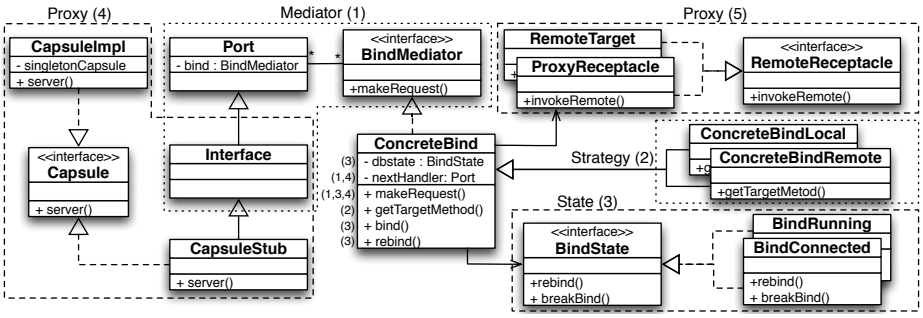

Fig. 1. Design slice of the OpenOrb middleware

2.1 Case Study: A Reflective Middleware

Figure 1 shows a design slice of an *OpenOrb*-compliant reflective middleware system [7] in which 21 classical design patterns [10] are used and combined to achieve the middleware requirements of customizability and adaptability [6]. A number of methods and attributes were omitted for simplification reasons. In Figure 1, each number represents a specific pattern, and these numbers are associated with methods and attributes in the ContreteBind class. The goal is to illustrate how various pattern realizations affect internal members of a single class. The attachment of a number implies that the respective method or attribute is part of the implementation of the corresponding pattern.

For instance, the implementation of the *Mediator* pattern (represented by number 1) includes: (i) all methods defined in the Port, BindMediator, Interface classes and (ii) the attribute nextHandler and the method makeRequest in the class ContreteBind. As a result, a single pattern is blended with other four patterns (Figure 1): *State*, *Strategy*, and two instances of *Proxy*. Figure 1 is a mere representative example of the difficulty in understanding and reusing pattern compositions in realistic scenarios. Table 1 lists the total number of per-pattern compositions for the three case studies used in our empirical evaluation (Section 5).

Table 1. Overview of pattern blends in Open-Orb design

Pattern	Over.	Cons.	Pattern	Over.	Cons.	Pattern	Over.	Cons.
Adapter	2	1	Facade	0	2	Prototype	6	1
Bridge	7	2	Factory Method	8	4	Proxy	6	2
Builder	1	1	Flyweight	4	2	Singleton	2	2
CoR	5	1	Iterator	0	2	State	4	0
Command	3	1	Mediator	7	3	Strategy	7	0
Composite	6	2	Memento	2	1	Template Method	6	0
Decorator	6	1	Observer	11	2	Visitor	3	1

2.2 Blending Categories

According to recent observations in the literature [12, 21], there are different forms for combining design patterns. Alternatively, pattern blends can be classified in two top-level categories: conservative blends and overlapping blends. A conservative combination maintains separate the structure of the involved patterns, so that original elements are always identifiable in the model, thereby creating loose relationships among the elements of the involved patterns. For instance, Figure 1 shows that the combination of *Proxy(5)* and *Mediator(1)* preserves an intact core structure of both the patterns. They are combined, in a behavioral fashion, through simple method calls from *Mediator* to *Proxy* elements.

Pattern overlapping occurs when pattern elements are merged in order to obtain a unified structure and behavior. However, it is not easy to separate the contributions of each pattern in overlapping blends. The reason is that the pattern-blending process involves the partial or full juxtaposition of two or more pattern elements. For instance, Figure 1 depicts three instances of pattern overlaps, listed in the three last rows of Table 2. Overlaps of patterns usually encompass compositions of fine-grained pattern elements, such as methods and attributes. These cases necessarily involve a tight coupling between patterns participating in the composition. For instance, the combination *Mediator(1)* and *Proxy(4)* requires the sharing of the makeRequest method to realize participants defined in both patterns (Figure 1) .

Table 2. Hybrid pattern blends in the reflective middleware system

Pattern Name	Combined with	Category
Proxy(5)	Mediator	Conservative
Proxy(4)	Mediator	Overlap (method overlap)
Proxy(4)	Observer	Overlap (method interlace)
Proxy(4)	Singleton	Overlap (class interlace)

Overlaps occur when realizations of patterns *P1* and *P2* share one or more statements, attributes, methods or entire classes. Conservative blends are realized with a loose coupling between some classes of the patterns *P1* and *P2* using a temporal reference among them. For instance, in Figure 1), the overlap involving (*Proxy(4)* and *Mediator(1)* is used to implement the connection between the *proxy* and the *real subject* participants of the *Proxy* pattern. This is useful when many *proxy* and *real subject* objects exist, and the designer wants to implement a flexible mechanism to define how these elements have to interact, thus avoiding the direct invocation. The *Mediator* pattern fits this requirement by assigning the responsibility of coordinating a set of colleagues to to the mediator. Therefore, this blending requires the unification of some responsibilities of the two patterns:

the *proxy* and *real subject* participants must also be colleagues referring to the same *mediator*.

Besides, the second instance (*Proxy(5)* and *Mediator(1)* in Figure 1) allows providing a reference to an object located in a different machine. This composition uses a different approach between the same two patterns in order to realize the coordination process encapsulated in the *mediator* participant. Here the mediator object (ConcreteBind) needs a RemoteTarget within its mediation process; these two objects are separated by using a *Proxy* pattern. The *mediator* participant only requires a reference to the *proxy* object thus the structure of the resulting pattern maintains unchanged both the *Mediator* and the *Proxy* original structures.

3 Defining Composable Patterns

The remaining sections describe a fine-grained design approach to support the pattern solution definition, the composition process and the pattern instantiation in the system. The pattern definition process is discussed in this Section, whereas the composition technique is discussed in Section 4. The instantiation phase is only briefly introduced due to space constraints. Our approach subsumes a pattern description language and it is based on a set of constituents that can be combined in order to define the structure and the behavior of the pattern solution. The language makes it possible to refer to (i) elements used to define the pattern and (ii) elements of the programming language for implementing the solution. Table 3 introduces the definitions of the elements in the pattern description language.

Table 3. Categories of elements used for describing a pattern

Terms	Definition
Pattern Description Element (PDE)	An atomic constituent of a pattern that describes the structure or the behavior of the solution. They are: (i) participants, (ii) composables, (iii) events and (iv) actions.
Language Element (LE)	Element of the target programming language used for implementing the pattern. In this paper we have used Java for implementing the case study, so the LEs are: classes, attributes, methods, constructors, interfaces and the like.
Affected System Element (ASE)	Element of the system that is influenced by the pattern application. A typical example of ASE is a business class that is assigned to a participant of the pattern. Its structure is modified because it must be compliant with PDE constraints.

The definition of pattern elements encompasses alternant levels of stability: some PDEs (Pattern Description Elements) are precisely described and do not

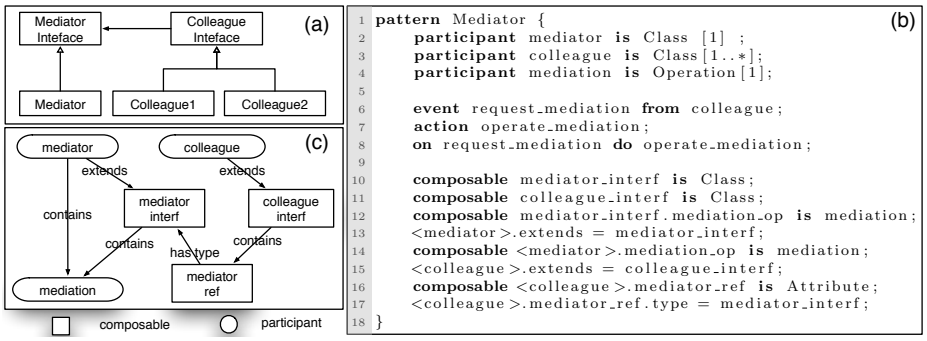

Fig. 2. Formalization of the *Mediator* pattern. (a) The classical structure from [10]. (b) A slice of code used to describe the solution. (c) pattern semantic description diagram.

require further details through the pattern instantiation, whereas some others are sketched and their concrete definition is delayed to the pattern instantiation phase. The latter means that the structure and behaviour of those pattern elements are volatile and their final definition depends on the application context and the other patterns to which they are going to be composed. This kind of PDE supports the generalization and reuse of patterns in very distinct contexts where the nature of the problem may be different. Along this section the *Mediator* pattern [10] (Fig 2.a) is used to illustrate the pattern description language.

Table 4. Summary of the main properties of the Pattern Description Elements

PDE	Category	Description
Participant	static	Participants are abstract elements to which it is possible to assign responsibilities. In the instantiation phase LEs must be assigned to each participant. Assigning a feature to a participant means to define a constraint for those entities.
Composable	static	Composables are concrete internal elements of the pattern structure to which it is possible to assign responsibilities. They provide means for introducing a constant element in the structure. A composable does not require a further customization in the instantiation phase.
Event	dynamic	An event encapsulates an abstract circumstance that can be used as trigger for generating a specific behavior involving one or more static elements. The definition of the context that generates an event must be completed in the instantiation phase.
Action	dynamic	An action is an atomic piece of behavior that expresses a specific collaboration involving static elements. Actions are abstract elements that must be refined in the instantiation phase.

3.1 Static Pattern Description

The description of the pattern structure comprises participants and composables (see Table 4). Both of them can be used to assign pattern responsibilities. The main difference is that a composable is a concrete element that will be added to the system, whereas a participant is only a placeholder for an ASE (Affected System Element). Both a participant and a composable own a type, which refers to a LE (Language Element).

The *Mediator* pattern description (Fig 2.b, lines 2-4) includes the colleague, the mediator classes and the mediation method as participants. Therefore, not only classes can be defined as participants: here a participant method is defined. The type of a participant indicates what kind of ASE can be assigned to the participant. For instance the Interface class can be a colleague, the ConcreteBind class can be the mediator and the makeRequest method can realize the mediation. Participants are also marked with multiplicities (at the end of each expression) that are constraints for the number of ASEs allowed. In the example only one mediator and a mediation are allowed, whereas many colleagues may exist.

Composables are concrete elements that introduce a standard structure in the system in which the pattern is going to be instantiated. Several composables are part of the *Mediator* pattern (Fig 2.b, lines 10-12, 14 and 16). For example, the colleague_interf and the mediator_interf are two classes of the mediator structure that do not depend on the specific application context. During the pattern instantiation phase all the composable elements become elements of the system. A relationship between a composable and a participant represents a constraint. The *Mediator* pattern description (Fig 2.b) encompass the identification of some relationships. For instance, at line 13, a colleague_interf is defined as a superclass for all colleague classes. Since the colleague is a participant, all its ASEs will be imposed to inherit from the colleague_interf.

Fig 2.c provides a pattern semantic description diagram for the *Mediator* pattern. This kind of diagram is conceived to show the structure of a pattern solution as a typed graph. It is an UML class diagram where a graphical stereotype notation is used in order to obtain a concise diagram. Participants are shown by using ovals, whereas composables are shown by using boxes. Relationships are used to link these elements, thereby creating a graph. The diagram focuses on the relationships among participants and composables underlining the semantics that is behind the pattern. This representation is a good instrument to discuss the composition operators (Section 4).

3.2 Dynamic Pattern Description

The description of the behavior of a pattern comprises two PDEs: events and actions. Their use allows for the behavioral description of the pattern semantics. An event encapsulates an abstract circumstance that is the cause of triggering

```
1 pattern MediatedProxy composes Mediator, Proxy {                    (a)
2     participant realcolleague unifies realsubject with colleague;
3     participant proxy unifies proxy with colleague;
4     composable subject unifies colleague_interf with subject;
5     event request unifies proxy_request with request_mediation;
6     participant mediator_interface externalizes mediator_interf;
7 }
```

```
1 pattern ProxyMediationPolicy composes Mediator, Proxy {             (b)
2     composable proxy_mediation conjoins mediator with proxy;
3     event request conjoins request_mediation with proxy_request;
4     composable default_request conceals request;
5 }
```

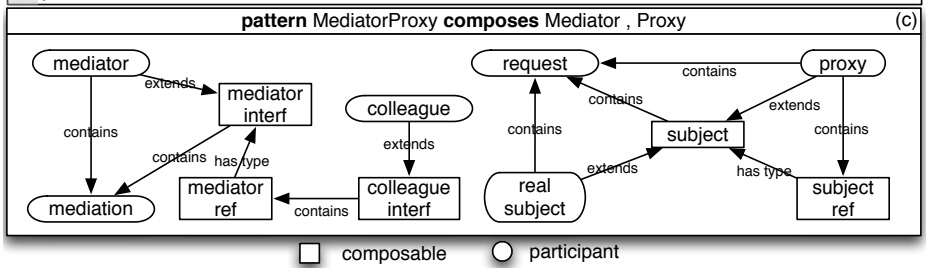

Fig. 3. Example of two different styles of compositions between *Mediator* and *Proxy* patterns. (a) Slice of code for an overlapping composition (*MediatedProxy*). (b) Slice of code for a conservative composition (*ProxyMediationPolicy*). (c) Pattern semantic description diagram showing a generic composition of *Mediator* with *Proxy*, before applying any operators.

a specific behavior, involving one or more static elements. The *Mediator* pattern description (Fig 2, line 6) includes an example of event definition. The request_mediation is an event that can be originated by a colleague. This event expresses the need of a colleague to communicate with another colleague. The specifications of the conditions that generate an event largely depends on the specific problem. For example the exact moment in which a colleague needs to communicate can not be predicted because it tends to be an application specific decision. Thus conditions are defined in the pattern instantiation phase.

Together with events, actions have a fundamental role in the definition of the behavior. An action encapsulates what happens when an event occurs. Actions are related to events by using cause-effect relationships. The *Mediator* pattern (Fig 2.b) the operate_mediation action is defined at line 7 and it is connected to the request_mediation event at line 8. Actions, as well as events, are abstract elements that require to be detailed in the pattern instantiation phase.

4 Operators for Pattern Composition

This section presents the operators for pattern composition based on the fine-grained pattern elements (Section 3). Along all this section we use two examples of composition between the *Mediator* and the *Proxy* patterns (Section 2.2).

The *MediatedProxy* (Fig 3.a) is an overlapping composition, whereas the *ProxyMediationPolicy* (Fig 3.b) is a conservative composition. In general terms, the composition process between a couple of patterns *P1* and *P2* creates a new pattern *P3* that contains all the PDEs of *P1* and *P2*. Fig 3.c shows the result of the composition, before the use of any operators.

4.1 Static Pattern Blending

The static operators can be used in order to modify the structure of the pattern solution, represented by a graph in the pattern semantic description diagram. Table 5 presents a summary of all the static composition operators.

Table 5. Summary of the static operators and their effects

Operator	Rationale
Unification	The unification is used to express overlapping compositions. The rationale behind this operator is to operate fusions of couples of static elements with a consequent merging of responsibilities. The result is to overlap the structure of two patterns using the two elements as pivot for the operation. This produces strong changes in the resulting pattern structure.
Conjunction	The conjunction operates a conservative pattern blending. The rationale behind this operator is to create a synergy among the responsibilities of two patterns, by maintaining them separated. The two elements are linked by a new element, introduced in the structure. Only marginal changes are visible in the resulting structure of the involved patterns, promoting the traceability of the involved elements.
Concealing	This unary operator has been conceived to modify the nature of a participant into a composable. The responsibilities assigned to a participant are imposed to the elements of the system that participate to the pattern. Concealing a participant means that all its responsibilities are delegated to a composable. They are no more visible outside the pattern. The visible effects of this operation are i) to allow mixed composition (unification and conjunction) among participants and composables ii) to internally set some responsibilities in order to assign a standard behavior and iii) to narrow the complexity of the pattern.
Promotion	This unary operator has been conceived to modify the nature of a composable into a participant. The rational behind this operator is to delay the assignment of these responsibilities till the instantiation phase, exactly like for participants. The visible effects of this operation are i) to allow mixed composition (unification and conjunction) among participants and composables and ii) to change the standard behavior of a pattern, by delegating some aspects of its structure to elements of the system.

Static Unification. The unification operator is used to express overlapping compositions producing strong changes in the resulting pattern: the elements that are unified represent the pivot points for the overlap. The unification can be applied to two operands that must refer to the same PLE and the same LE. The new element will receive all the features of its originators, and these will no more be present in the structure.

Figure 4.a/b show two unifications of participants. The effect is the creation of two new participants, RealColleague and Proxy that get all the relationships

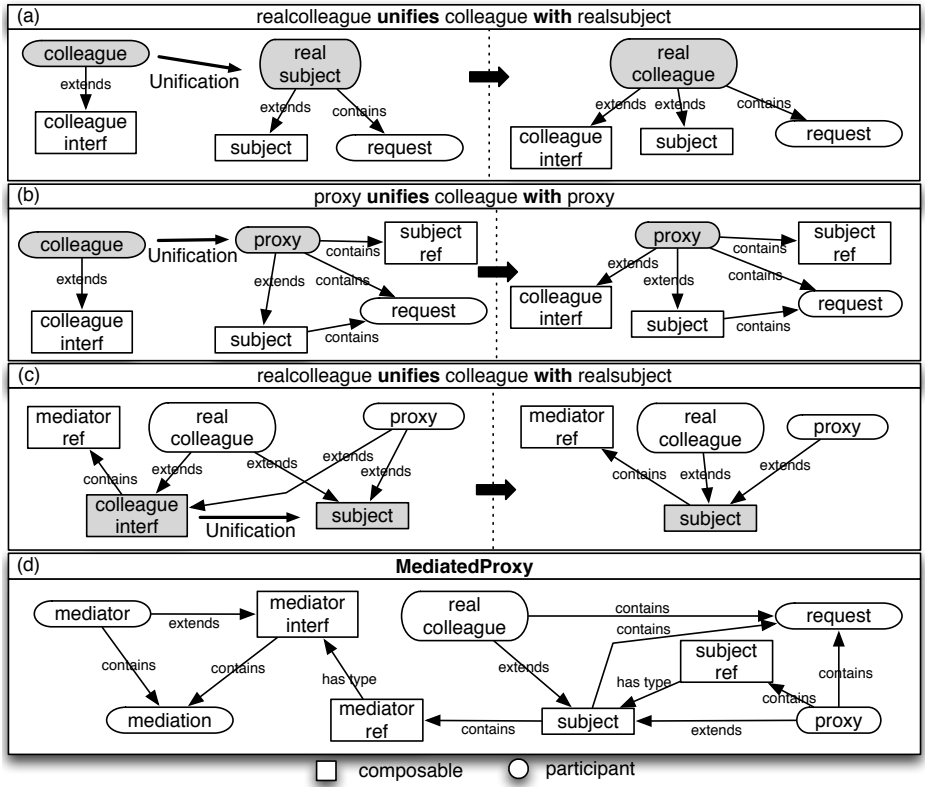

Fig. 4. Effects of the static unification in the *MediatedProxy* pattern. (a-b) Unification of participants. (c) Unification of composables. (d) Final structure for the *MediatedProxy* after the use of the operators.

that their originators prescribed in the original pattern description. The aim of these two unifications is to create a pattern with the characteristics of a *Proxy*, where both the proxy and realsubject participants are also colleagues of a *Mediator* structure, so they can communicate by using a mediator.

When the unification is applied to two participants, the new participant has a multiplicity that is the intersection of the two original's ones. For instance the unification of a participant with multiplicity [0,2] with a participant [1,*] generates a participant [1,2]. Operations in Figure 4.a/b generate a composition problem when the pattern implementation target is an object-oriented programming language (even though it is easily realisable with aspect-oriented languages). After the unification, both the RealColleague and the Proxy are involved in a multiple inheritance. Therefore, the unification of the colleague_interf composable with the subject composable solves this problem. The effect of this operation is shown in Figure 4.c and the final structure is shown in Figure 4.d.

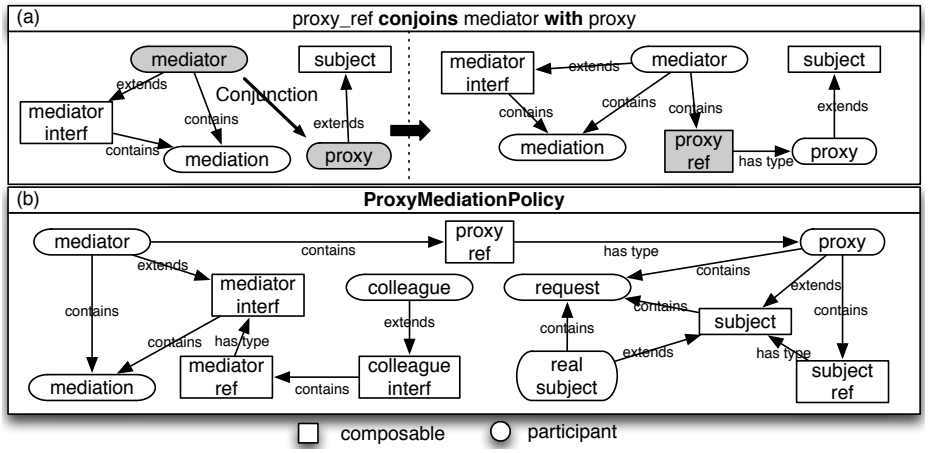

Fig. 5. Effects of the static conjunction in the *ProxyMediationPolicy* pattern. (a) conjunction of participants. (b) Final structure for the pattern after the use of the operator.

Static Conjunction. The conjunction supports a conservative pattern blending. Only marginal changes are visible to the structure of the involved patterns, promoting the traceability of the involved elements. The operands continue to exist after the operation. The visible effect is the creation of a new element that is responsible for connecting the two ones in order to realize their collaboration. The language does not put any constraints on the nature of the two operands that is possible to conjoin (as for the unification). They can be indifferently participant and composable elements but syntactic rules of the programming language must be kept.

Figure 5 illustrates the conservative composition *ProxyMediationPolicy*. The rationale is to create a synergy between the two patterns, without modifying their standard behavior. This is obtained by conjoining the mediator with the proxy participants. The operator introduces a proxy_ref attribute in the mediator class, that refers to a proxy object. The new pattern has all the characteristics of a *Mediator*, which uses the *Proxy* inside the mediation process.

Externalization and Concealing. These two unary operators are conceived in order to modify the nature of composables and participants. The externalization is applicable only to a composable, changing it to a participant of the pattern. After this operation, ASEs can be assigned to the new participant. The twofold goal of externalization is to (i) allow for the unification of a composable with a participant, and (ii) delegate some responsibilities (originally delineated inside the pattern) to ASEs. An example of externalization is shown in Figure 6.a, where the operator is applied to the mediator_interf composable. The result is the creation of a new participant, named mediator_interface replacing the com-

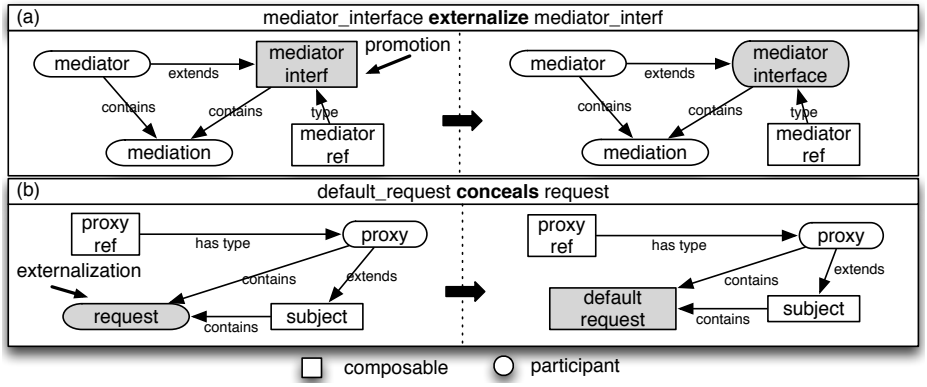

Fig. 6. (a) Effect of the externalization of a participant in the *MediatedProxy*. (b) Effect of the concealing of a composable in the *ProxyMediationPolicy*.

posable.

The concealing operator modifies the nature of a participant delegating its responsibilities to an composable of the structure. It becomes a fixed element of the structure and it does not requires further detailing, in the instantiation phase. The aims of concealing are to: (i) allow unification between a participant with a composable, and (ii) specialize the pattern, setting some responsibilities. An example of concealing is shown in Figure 6.b, where the operator is applied to the request participant with the introduction of a default_request composable. The latter is a standard method for executing the *proxy* request by the mediator.

4.2 Dynamic Pattern Blending

Dynamic composition operators complete the language for pattern blends. Only two operators are contained in this category, the unification and the conjunction. Both of them work on pattern events, and as consequence, on their associated actions.

Event Unification. As already illustrated in Section 3, pattern description defines events and actions as expressions of the pattern behavior. The effect of unifying two events is the creation of a new event in the pattern, whereas the two original ones do not exist anymore. The new event is responsible to trigger all the events of its originators. This operation, therefore, produces a blend of the flows of actions related to the two involved patterns. After this blending, new actions can be added to the flow of events, and the order of execution of the existing ones can be rearranged according to new needs.

Figure 7.a shows an unification of events related to the *MediatedProxy* pattern: the request_mediation is unified with the proxy_request. The new pattern uses

Table 6. Summary of the dynamic operators and their effects

Operator	Rationale
Unification	This operator produces a new event that is responsible to trigger all the actions of the original two events. The visible effect is a correspondent blending of their flow of actions. All the actions of the two events are triggered by the new event with a possible rearrangement of the original order or an overlapping effect.
Conjunction	The concatenation is used to create a sequence of the two flows of events. One event disappears since all its actions are triggered by the end of the execution of the actions of the first event. The visible effect is to link the execution of a flow of actions to the execution of another flow of actions. These two flows are executed sequentially, when the new event is triggered.

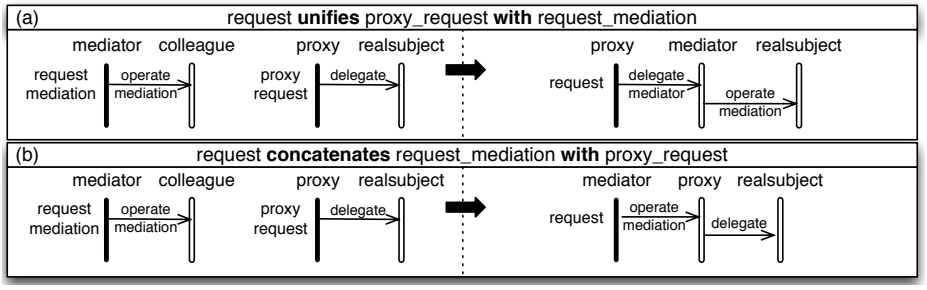

Fig. 7. (a) Effect of the event unification in the *MediatedProxy* pattern. (b) Effect of the event conjunction in the *ProxyMediationPolicy* pattern

the *Mediator* logic to allow the communication between the proxy and the realsubject of the *Proxy* pattern. Therefore the delegate action is executed by using the operate_mediation. After the unification a new event request is the trigger for these actions.

Event Conjunction. As for the unification, the effect of the conjunction of two events is the creation of a new event in the pattern, whereas the two original ones do not exist anymore. The difference is that this operation maintains unchanged the flows of actions considering them as atomic blocks of behavior. The new event is responsible to trigger the first flow of actions. The execution of the second flow of actions is triggered after the end of execution of the first one.

In Fig 7.b a conjunction of events (for the the *ProxyMediationPolicy* pattern) is shown: the request_mediation is concatenated to the proxy_request since the mediation algorithm of the *Mediator* uses a remote invocation encapsulated in the *Proxy*. Therefore the delegate action is executed as a consequence of the execution of the operate_mediation.

5 Evaluation and Lessons Learned

This section discusses some results obtained by the application of our pattern composition language to the three different case studies: OpenOrb, JHotDraw and JUnit. We have chosen these applications because they are from heterogeneous application domains and, as a result, the feasibility of our language constructs and composition operators can be assessed in different contexts. OpenOrb is a pattern-based middleware application that was already introduced and discussed in Section 2: we have considered for our studies the core part of the system consisting of 133 classes. JHotDraw is an open-source software conceived for drawing 2D graphics. It was built with a massive use of design patterns as exercise for demonstrating the reuse of design patterns. JUnit is an open source testing software written in Java. It encompasses a pattern-oriented framework design with variabilities for the organization of software tests and testing graphical interfaces.

5.1 Empirical Procedures

The activities executed for the assessment of the proposed language are the following: (i) analysis of the patterns and their blends that have been previously applied to the three target systems; (ii) once we identified a couple of interacting patterns we have analyzed the motivations of their collaboration, by identifying the blend intent and, finally, (iii) we have analyzed how to aggregate the responsibilities of these patterns by using our language. We have been able to represent 20 patterns from the GoF's catalogue and to combine them by using 30 pair-wise compositions, and 2 compositions involving more than two patterns. Finally we have instantiated 62 patterns (some of them more than once) in the OpenOrb case study.

Table 7 presents the results of these activities and consists of blocks of columns. The first block reports the name of the patterns involved in each composition. This is the name we use for referring to the composition itself. The second block reports the reuse of the pattern composition for each of the three case studies (OpenOrb, JHotDraw and JUnit): the check in a cell means that this pattern composition has been reused in the case study.

The third block, named *type*, represents a classification for each of the involved patterns. The categories we considered derive from the GoF's work: B indicates Behavioural patterns, S indicates Structural patterns and C stands for Creational patterns. Behavioral patterns are oriented to problems focused on algorithms, distribution of responsibilities and communication between the classes and the objects of the solution. Structural patterns are dedicated to problems where the structure and the organization of the involved elements are fundamental. Creational patterns describe solutions for promoting the independence of the system from the object creation process. We have encountered 5 categories of compositions: B-B, B-S, B-C, C-S and S-S.

Table 7. Results obtained by the pattern composition language applications

Composition	Case Study			Type	Participant			Composable			Event		Elements		Bending Category
	OpenOrb	JHotDraw	Junit		Unification	Conjunction	Concealing	Unification	Conjunction	Externalization	Unification	Conjunction	before	after	
Command+Composite	√	√	√	B-B	3			1		1		1	4+5	6	OVER
Command+Visitor	√		√	B-B	2		3	4			1	1	5+5	6	OVER
Iterator+Strategy	√			B-B	1		1	1					4+3	4	OVER
Mediator+State	√			B-B	1		1	1				1	4+5	8	CONS
Mediator+Strategy	√			B-B	2		1	2			1		3+4	4	OVER
Mediator+Strategy+ TemplateMethod	√			B-B	4		2	3			1	1	2+3+4	4	OVER
Memento+State	√			B-B	1		1	1					3+3	5	CONS
Observer+State	√	√		B-B	1			1					4+3	6	CONS
Observer+Strategy		√		B-B	3			2				1	4+3	4	OVER
State+Strategy		√		B-B	3			1			1		3+3	3	OVER
Strategy+TemplateMethod	√	√		B-B	2		1	1				1	2+3	3	OVER
Builder+Command	√		√	B-C	1		2		1		2		5+4	8	CONS
FactoryMethod+State		√		B-C	2			1			1		4+3	5	OVER
Prototype+State	√	√		B-C	2			1					2+2	2	OVER
Prototype+Strategy		√		B-C	3			1			1		3+3	3	OVER
Adapter + Command	√	√	√	B-S	1							1	3+5	7	CONS
Adapter+Observer		√	√	B-S	1			1			1		4+4	5	OVER
Adapter+Strategy		√		B-S		1				1		1	3+3	6	CONS
Bridge+Command	√			B-S	2			1			2		4+5	8	CONS
Command+Proxy	√			B-S	4			1			1	1	5+3	6	OVER
Composite+Iterator	√		√	B-S		1	1						4+4	7	CONS
Composite+Visitor	√		√	B-S	2			1				1	6+4	7	OVER
Decorator+(Mediator+ Strategy+ TemplateMethod)	√			B-S	1		1	1				1	(4)+4	7	CONS
Decorator+Mediator	√			B-S	2			1				1	4+4	6	OVER
Decorator+TemplateMethod	√	√		B-S	2		1	1			1		2+4	4	OVER
Flyweight+Mediator	√			B-S	1		1	1				1	5+4	7	OVER
Mediator+Proxy	√			B-S		1						1	3+4	8	CONS
Adapter+Prototype		√		C-S	1		1				1		4+3	5	CONS
FactoryMethod+Proxy	√			C-S	3			1			1	1	4+3	6	CONS
Proxy+Singleton	√			C-S	1								4+1	4	OVER
Adapter+Composite		√	√	S-S					1			1	4+3	7	CONS
Composite+Decorator		√	√	S-S	3			1					4+3	4	OVER
Operator Employment					**55**	**3**	**17**	**31**	**2**	**2**	**15**	**16**			

The following three column blocks of Table 7 report the employment of operators which have been used for realizing the compositions. These sections show how many times each operator has been used. The most frequently used operator is the unification, used in 84 different static compositions. The block named *elements* reports some information about the static structure of the patterns. The column *before* contains the number of classes constituting the solution of the patterns involved in the compositions. The column *after* indicates the number of classes of the new pattern, obtained by the composition. As we discussed before, our design language allows for specification of hybrid compositions that are neither full overlap nor full conservative. This value gives useful indication for classifying a composition in these two classical categories: when the value in the column *after* is nearer to the sum of the two values contained in the column

before then the composition might be considered conservative. The reason being that every class of the original pattern structures is maintained in the new composed one. Otherwise the composition might be considered as overlapping. This categorization is reported in the last block, named *Composition Category*. The following subsections discuss the results reported in Table 7 in terms of our language expressiveness and reusability.

5.2 Expressiveness of Heterogeneous Pattern Blends

We have observed that the expressiveness of our pattern blending language is widely related to the fine-grained nature of the composition operators (Section 4). In fact, the operators covered all the pattern compositions emerging in our three case studies, with some exceptional cases being discussed later. The static unification was employed 86 times, while dynamic unification was applied 15 times. The conjunction was used 19 times for static elements and 16 times for dynamic ones. We have also employed the concealing operator 17 times, against the low usage of the externalization (used just in only 2 compositions). These results provided evidence that our innovation operators for pattern blending were effective to support the reuse of the structure and the behavior of patterns adapting them to new intents and contexts, in order to compose them with other patterns.

In addition, the language has shown to be suitable for expressing both conservative and overlapping pattern blends. We have formalized 13 conservative compositions and 19 overlapping compositions in the three case studies. Several hybrid pattern compositions have been obtained with a conjunct use of different type of operators. For instance, the *Command+Builder* conservative combination has been obtained with 1 unification and 1 conjunction (for the static part) and 2 unifications for the dynamic part. The *Factory Method+Proxy* conservative composition involves the use of unification and conjunction for the dynamic part, and 4 unifications for the static part. This outcome provided us with evidence of the language scalability to deal with complex pattern blends.

On the other hand, we have also learned some possible further enhancements to our pattern composition language. For instance, some difficulties we observed in the description of the *Chain of Responsibility* and *Facade*. The problem in representing the *Chain of Responsibility* is related to the implicit relationship among the participants *predecessors* and *successor*. In instantiation phase, apart to assign the classes to these participants, it is necessary to specify the order of these classes in the chain. We are getting around this limitation by instantiating the pattern more than once, every time with only one *predecessor* and one *successor*. Therefore, we are also studying a way to delay the definition of some elements of the pattern in order to better handle these intricate scenarios.

5.3 Reusability of Pattern Blends

This subsection illustrates the reusability of pattern compositions across single and multiple software projects.

Table 8. Effects of the operators on the reusability

Operator	Effect on Reuse	Rationale
participant unification	negative	The unification of participants concentrates the responsibilities of PDEs thus LEs must own more characteristics in order to be suitable for participating to the pattern. Figure 4.a/b puts in evidence that after the composition the new participants are more intricate than the originator's ones. For instance the *real colleague* participant requires LEs that must be at the same time *colleague* (that need to communicate) and *real subject* (for example remote objects of the system). This reduces the class of problems for which the pattern is suitable. This may produces a difficult in reusing the new pattern.
participant conjunction	positive or negative	The conjunction of participants increments the number of responsibilities without concentrating them. LEs must own the same characteristics than before the composition. However more participants are required in order to instantiate the pattern. Figure 5.b shows that participants are exactly the same than in the originator. The conjunction may have both positive and negative effects on the reusability.
event unification	positive	The unification produces a new event that encapsulates and reorganizes the actions of the two originators. Figure 7.a shows a dynamic unification used to create a *proxy* invocation by using a *mediator*. The identity of the two flow of activities is lost but it is possible to express a more complicate and specific behavior in according to new needs, with a positive impact on the reusability.
event conjunction	negative	Beside, the conjunction produces a collaboration where the two flows of events maintain their identity. Figure 7.b shows a dynamic conjunction with the aim to introduce a *proxy* invocation inside the *mediation* process. The context where this kind of behavior is useful is reduced, so this operator may have negative effects on the pattern reusability.
concealing	negative	The concealing operator increase the number of responsibilities that are solved in a standard way, reducing the global complexity of the pattern. The number of ASEs that participate to the pattern decreases, and the pattern may become easier to reuse. For instance, the concealing operated in Figure 6.b introduce a standard method for executing the proxy request. Anyway composables encompass a static structure, that could be specific for a class of problems but unsuitable for another ones.
externali-zation	positive	The externalization operator increases the number of total responsibilities that must be assigned to ASEs, thus creating a more flexible structure. For instance, Figure 6.a shows the externalization of the *mediator_interf* class that, after the operation, can be assigned to an ASE and enriched with other attributes and methods. This has the counter effect to (i) increase the complexity of the pattern and, at the same time, (ii) delegating some responsibilities to ASEs, to increase its reusability in different contexts.

Reuse of Pattern Blends. We have also analyzed to what extent each pattern composition has been reused across the three case studies. The evidence of composition reuse was especially high thanks to the fact the analyzed applications are from very different domains. The result of this experiment is that 13 over 32 pattern compositions have been reused in more than one case study. Two of these combinations, namely (*Adapter+Command* and *Command+Composite*), have been reused in all the three case studies.

We noticed that: (i) surprisingly, overlapping compositions revealed easier to reuse against conservative compositions; (ii) the reuse seems independent from the column *type*: we have reused compositions containing Behavioral, Structural and Creational patterns; and (iii) compositions from the Structural-Structural

category revealed easiest to reuse, while the Creational-Structural category revealed hardest to reuse.

The justification for these outcomes can be based in two directions: the heterogeneity of the case studies and the nature of the involved patterns. A general consideration may be that it is easier to reuse a pattern when it is more abstract and adaptable to several circumstances. Therefore during the composition, some patterns greatly reduce their generality, becoming more difficult to employ in different contexts. For instance, one of the common patterns that typically is used in many contexts is the *Observer*. Table 7 reveals that it was used in only three compositions and only two of these were reused in more case studies. Since the language expressiveness is perfectly suitable for describing this pattern, we imagine that this peculiarity depends by some intrinsic unrevealed features of the pattern itself.

Intuitively, the composition process often creates patterns to solve classes of problems considerably different from the original ones. On the other hand, the new class of problems is not totally independent from the original ones. For instance, the *Observer* pattern has been defined to avoid tight coupled objects involved in one-to-many dependencies. Also the dynamics of the *Observer* is well defined: when the state of the *subject* changes, it notifies all its *observers*. The *Observer+Strategy* composition has been defined to inform the *observer* class when a *subject* class changes its strategy to operate. This composition adds some constraints to the feasibility of the new pattern: the strategy is a dynamic feature of a class, therefore the pattern becomes useful only when we want to decouple dynamic relationships, whereas the *Observer* is suitable both for static and dynamic relationships. Therefore this composition (*Observer+Strategy*) becomes specific for a class of problems that is a subset of the *Observer*'s class of problems. The pattern reusability is naturally reduced.

On the Intricacies of Pattern Composition Reuse. The observation above does not necessarily hold for every kind of composition. The rule to determine how a pattern composition is reusable seems very difficult to pinpoint. In order to discover what kind of dependency could exist between pattern composition and reuse, Table 8 provides an overview of the effects of each operator on the reusability.

These considerations can be useful in order to analyze why some categories of pattern compositions are easier to reuse in comparison to other ones. For instance, the Behavioral-Structural and Behavioral-Behavioral compositions seem easier to reuse than Behavioral-Creational and Creational-Structural compositions. In order to detect the causes of this difference, we have sorted the patterns of Table 7 according to their categories and we have considered how many times each operator has been used in each category. The result is interesting: the static unification and the static conjunction have been used with the same frequency in all the categories. The concrete difference is the use of the externalization operator, the event unification and event conjunction. Both B-S and B-B categories indicate a greater use of event unification and externalization, which, according

to our considerations, have no negative impact on the reusability. Besides the two categories B-C and C-S encompass a greater use of conjunction of events, which may reduce reusability. In conclusion from our observation, the dynamic part seems to have a greater influence on the reusability of pattern compositions than the static elements.

6 Related Work

Different approaches have been proposed for documenting pattern blends: role composition [12, 18], UML-based composition [21], temporal logic composition [16] and aspect-oriented composition [6, 7, 13, 19].

In [18] role diagrams are used for implementing and documenting object collaboration patterns. These roles are different from our participants, since only classes can play roles in these patterns, whereas every entity can be a participant (including methods and attributes).

In [21] an UML approach for composing patterns is proposed. The main limitations of this work compared to ours are: (i) this approach is suitable only for OO languages, whereas our target programming language can be any, and (ii) the UML approach is mainly focused on the static composition, whereas we introduce operators for dynamic blends.

Mikkonen [16] proposes a formal approach for composing patterns, which focuses on cooperation of behavioral layers. Their composition is based on one operator only: the multiple inheritance. The limitation is twofold: the verbosity and heavyweight precision it requires in defining and combining the pattern elements. It is not clear how this techniques address hybrid pattern compositions and, thereby hindering the effective feasibility and reusability of the compositions in realistic contexts as investigated in our study.

In [20] a generic composition technique for merging dynamic structures is proposed. This technique is based on state charts, but it can be extended to other UML diagrams. The process considers the diagram meta-model in order to build a graph, to which it is possible to apply several transformations. The approach is similar to our pattern semantic description diagrams. The differences are: (i) our operators generate transformations both for the structure and the behavior of a pattern which is essential for pattern composition descriptions, and (ii) the operators are high-level design instruments than can be easily managed in the system development phase by designers.

7 Conclusion and Future Work

This paper presented an innovative composition technique for describing blends of design patterns, based on their own static and dynamic semantics. The language has been conceived for dealing with composition, presenting a set of operators to manage different pattern blending styles. The peculiarity of the approach is the fine grained level chosen for fronting with the composition, which makes

it possible to combine overlapping and conservative combinations of pattern elements in the resulting composite pattern. We have applied our approach in three real-life case studies, obtaining encouraging results in terms of reusability and expressiveness. Future work includes the refinement of a visual notation for representing the composition, and the concluding our ongoing tool implementation for the pattern composition process.

References

1. C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*, volume 2 of *Center for Environmental Structure Series*. Oxford University Press, New York, NY, 1977.
2. H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes*, 30(5):156–165, 2005.
3. J. Bosch. Specifying frameworks and design patterns as architectural fragments. In *Proceedings of TOOLS '98*, page 268, Washington, DC, USA, 1998. IEEE Computer Society.
4. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, 1996.
5. F. Buschmann and R. Meunier. *A System of Patterns*. ACM Press Addison-Wesley Publishing Co., New York, NY, USA, 1995.
6. N. Cacho, T. Batista, A. Garcia, C. Sant'Anna, and G. Blair. Improving modularity of reflective middleware with aspect-oriented programming. In *Proceedings of SEM '06*, pages 31–38, New York, NY, USA, 2006. ACM Press.
7. N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *Proceedings of AOSD '06*, pages 109–121, New York, NY, USA, 2006. ACM Press.
8. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
9. A. H. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *Proceedings of ASE '97*, page 143, Washington, DC, USA, 1997. IEEE Computer Society.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.
11. A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of AOSD '05*, pages 3–14, New York, NY, USA, 2005. ACM Press.
12. I. Hammouda and K. Koskimies. An approach for structural pattern composition. In *Proceedings of SC 2007*, Braga, Portugal, March 2007.
13. J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of OOPSLA '02*, pages 161–173, New York, NY, USA, 2002. ACM Press.
14. C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *First International Symposium on Empirical Software Engineering and Measurement, 2007 (ESEM).*, 0-21 Sept 2007.

15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, Jyvaskyla, Finland, June 9-13 1997. Springer.
16. T. Mikkonen. Formalizing design patterns. In *Proceedings of ICSE '98*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society.
17. H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA '95*, pages 235–250, New York, NY, USA, 1995. ACM Press.
18. D. Riehle. Describing and composing patterns using role diagrams. In K.-U. Mätzel and H.-P. Frei, editors, *1996 Ubilab Conference*, pages 137–152, Zürich, Germany, June 1996.
19. A. L. Santos, A. Lopes, and K. Koskimies. Framework specialization aspects. In *Proceedings of AOSD '07*, pages 14–24, New York, NY, USA, 2007. ACM Press.
20. J. Whittle, A. Moreira, J. Araújo, P. Jayarama, A. Elkhodary, and R. Rabbi. An expressive aspect composition language for uml state diagrams. In *Model Driven Engineering Languages and Systems*, pages 514–528, 2007.
21. S. M. Yacoub and H. H. Ammar. Uml support for designing software systems as a composition of design patterns. In *Proceedings of UML'01*, pages 149–165, London, UK, 2001. Springer-Verlag.