PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 34/05

Aspect-Oriented Composition of Design Patterns: A Quantitative Assessment

Nélio Alessandro Azevedo Cacho
Eduardo Magno Lages Figueiredo
Cláudio Nogueira Sant'Anna
Alessandro Fabricio Garcia
Thaís Vasconcelos Batista
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 34/05 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena October, 2005

Aspect-Oriented Composition of Design Patterns: a Quantitative Assessment *

Nélio Alessandro Azevedo Cacho[1] Eduardo Magno Lages Figueiredo
Cláudio Nogueira Sant'Anna Alessandro Fabricio Garcia[2]
Thaís Vasconcelos Batista[1] Carlos José Pereira de Lucena

[1]Computer Science Department – Federal University of Rio Grande do Norte (UFRN)
[2]Computing Department, Lancaster University, UK

cacho@consiste.dimap.ufrn.br, {emagno,claudios,lucena}@inf.puc-rio.br
a.garcia@lancaster.ac.uk, thais@ufrnet.br

Abstract. Pattern composition has been shown as a challenge to applying design patterns in real software systems. One of the main problems is that the implementations of multiple design patterns in a system are not limited to affect the application classes. They also crosscut each other in multiple heterogeneous ways so that their separation and composition are far from being trivial. In this context, it is of paramount importance to systematically verify whether aspect-oriented programming (AOP) supports improved composability of design patterns. This paper presents a systematic investigation on how AOP scales up to deal with modularization of pattern-specific concerns in the presence of pattern interactions. We have made both qualitative and quantitative assessments of 62 pair-wise compositions taken from 3 medium-sized systems implemented in Java and AspectJ programming languages. Our analysis has also included the evaluation of compositions involving more than two patterns. The assessment was based on four fundamental software attributes, namely separation of concerns, coupling, cohesion, and conciseness.

Keywords: Design patterns, aspect-oriented programming, composability, empirical studies, metrics.

Resumo. Composição de padrões tem se mostrado um desafio no uso de padrões em sistemas reais de software. Um dos maiores problemas é que a implementação de múltiplos padrões de projeto em um sistema não se limita a afetar as classes da aplicação. Em muitos casos, os padrões interagem entre eles de forma tão heterogênea que sua separação e composição não é trivial. Neste sentido, é importante uma verificação sistemática de como a programação orientada a aspectos (POA) pode suportar a composição de padrões de projeto. Este documento apresenta uma investigação sistemática de como os mecanismos de AOP auxiliam a modularidade de interesses específicos na presença de interações entre padrões. Neste estudo foram feitos tanto avaliações qualitativas quanto quantitativas para 62 pares de composição de padrões extraídas de 3 sistemas de tamanho médio implementados em Java e AspectJ. Também foram analisadas composições que envolvem mais de dois padrões para verificar o quanto às soluções são escaláveis. A avaliação é baseada em quatro atributos fundamentais da Engenharia de Software: separação de interesses, acoplamento, coesão e concisão.

Palavras-chave: Padrões de Projeto, Programação Orientada a Aspectos, Composição, Estudo Empírico, Métricas.

* This work has been partially supported by CAPES for Eduardo Figueiredo and by CNPq for Cláudio Sant'Anna. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0.

In charge for publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

Table of Contents

1 Introduction

Design patterns are present in every real system we build in order to assure stringent modularity principles, such as low coupling, high cohesion, conciseness, and separation of concerns. A design pattern assigns *roles* to their participant classes, which define the functionality of the participants in the pattern context [14]. The application of design patterns in complex systems is often a result of the composition of two or more pattern roles rather than their instantiations in an isolated manner. Their object-oriented (OO) implementations can be composed in different ways [8, 31], ranging from a simple method invocation between the patterns' roles to the sharing of one or more classes by the pattern roles.

Recent systematic studies [12, 15] have shown that a number of design patterns involve crosscutting concerns in the relationship between the pattern roles and participant classes in each instance of the pattern. The pattern roles often crosscut several business classes in a software system. OO abstractions are not able to localize these pattern-specific concerns and tend to lead to programs with poor modularity attributes [12]. The situation is even worst in real OO designs; pattern implementations typically crosscut the implementations of other pattern roles as they need to be composed. This means that pattern roles are scattered and tangled to each other and through the participant classes, which in turn leads to the pattern instances to get lost [27] or degenerate [8] in the system. Also this crosscutting phenomenon potentially has negative impacts on quality attributes of both pattern-specific and application-specific implementations.

In this context, it is of paramount importance to systematically verify whether AOP approaches (such as [20]) support improved composability of design patterns. It requires an investigation on how AOP scales up to deal with modularization of crosscutting concerns relative to pattern roles in the presence of pattern interactions. To the best of our knowledge, there is no systematic evaluation of the effects of AOP on pattern compositions. Up to now, qualitative and quantitative studies involving the "aspectization" of design patterns [12, 13, 14, 15] focused on the separation of a particular design pattern and its participant classes in the application. Although these first assessments provided initial interesting results, the implementation and evaluation of the pattern instances were isolated from each other. They were not taken in the context of existing software systems where interactions between pattern implementations are recurring and intricate. This problem seriously constrains the extrapolation of the results and limits our understanding in how aspect-oriented (AO) solutions scale up to cope with realistic scenarios involving complex pattern instances and different composition contexts. Hence, a number of questions remain unaddressed:

(i) does AOP promote improved pattern composability?

(ii) to what extent separation of pattern roles is preserved in realistic contexts involving pattern compositions?

(iii) what are the positive and negative influences of aspectized pattern compositions on fundamental software attributes, such as coupling, cohesion, and conciseness?

(iv) what does happen in more intricate compositions involving 3 or more patterns? How much do the aspect-oriented implementations scale in these cases?

To explore these issues, we have performed an empirical study where we have assessed OO and AO solutions that implement compositions of the Gang-of-Four (GoF) patterns. It includes the quantitative and qualitative assessments of 62 pair-wise compositions taken from 3 medium-sized systems implemented in Java [17] and AspectJ [2] programming languages. Our analysis has also included the evaluation of compositions involving more than two patterns. The assessments were based on 4 basic modularity attributes, namely separation of concerns (SoC), coupling, cohesion, and conciseness. This paper also correlates our study findings with the results from previous studies, contributing to an improved body of knowledge on the scalability of AOP for addressing the crosscutting property of patterns in a multitude of composition scenarios.

The remainder of this paper is organized as follows. Section 2 presents our study setting. Section 3 presents the study results with respect to the investigated modularity attributes. These results are interpreted and discussed in Section 4, which also points out some constraints of our study. Section 5 introduces some related work. Section 6 includes some concluding remarks and future work.

2 Study Format

This section first puts our study in perspective of previous work (Section 2.1). It also describes our categorization defined for pattern compositions (Section 2.2), and the procedures and metrics used to support our quantitative assessment (Section 2.3.2).

2.1 Modularizing Design Patterns

There are two kinds of pattern roles [15]: defining and superimposed roles. A *defining* role defines a participant class completely. In other words, classes playing a defining role have no functionality outside the pattern. The unique role of the Façade pattern [9] is an example of defining role. A *superimposed* role can be assigned to participant classes that have functionality outside of the pattern. An example of superimposed role is the Colleague role of the Mediator pattern [9], since a participant class playing this role usually has functionality not related to the pattern. General-purpose design patterns, such as GoF patterns, exhibit crosscutting concerns [9, 12]. For example, consider the *Mediator* and *Colleague* roles that are defined in the Mediator pattern. Some operations that change a *Colleague* must trigger updates to the corresponding *Mediator*; in other words, the act of updating crosscuts one or more operation in each *Colleague* in the pattern [12]. Mediator is also usually a superimposed role.

In this context, systematic studies have investigated the effects of AOP on the modularization of design patterns. Hannemann and Kiczales (HK) [15] have undertaken a qualitative study in which they have developed and compared Java [17] and AspectJ [2] implementations of the 23 GoF patterns [9]. The basic idea was the identification of the common part of several patterns and the isolation of their implementations in aspectual modules. Some of these modules were defined as reusable aspects that are extended in order to instantiate the pattern for a specific application. Our previous work [12] focused on the quantitative assessment of the HK implementations for the GoF patterns. Both studies have used modularity attributes as assessment criteria.

However, for each of the 23 patterns these initial studies used a *very simple* example that made use of the pattern. The pattern instances were not taken from realistic software systems. More importantly, AOP has not been assessed in the context of pattern compositions. The HK study alleged that 15 aspectized patterns had superior “composition transparency”. However, this property only captured a narrow view of pattern composability; it is related to the ease of composition between the general code of a single pattern and their multiple instances [15]. There are additional studies (e.g. [22, 23]) related to the aspectization of GoF patterns, but they focus on some specific patterns. They also do not assess how AOP scale up to deal with crosscutting concerns in the presence of pattern interactions.

2.2 Pattern Composition Categories

Pattern compositions involve the combination of their roles. Figure 1 shows an OO design slice of an OpenOrb-compliant middleware system [3] in which a number of GoF patterns are used and combined to achieve the middleware requirements of high customizability and adaptability. In the figure, each number represents a specific pattern, and the numbers are associated with methods and attributes. The number attachment indicates that the associated method or attribute is part of the implementation of the corresponding pattern. For example, the implementation of the Decorator pattern (represented by number 1) includes the attribute bind and affects several methods, such as `makeRequest()`, `breakBind()`, `rebind()`, `checkPreMethods()`, and `checkPosMethods()`.

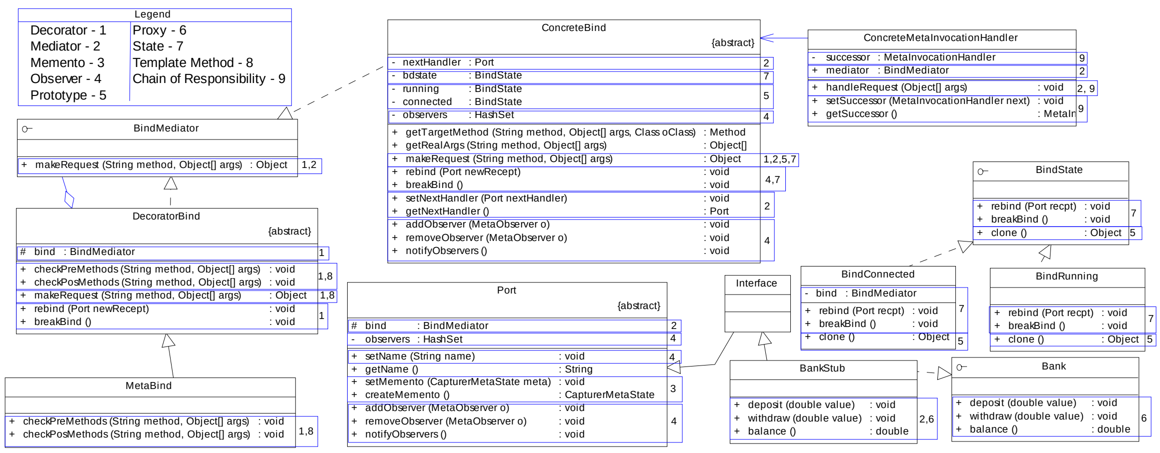


Figure 1: An OO design slice of the OpenOrb component model [3]: tangling and scattering of pattern-related concerns.

An analysis of Figure 1 also makes it evident that most of the pattern-related concerns are scattered over the system classes, as commonly happens in realistic applications. The roles of the State pattern affect various methods in the classes `ConcreteBind`, `BindConnected`, and `BindRunning`. Pattern roles are also tangled with each other and with middleware-specific concerns. For example, the methods `rebind()` and `breakBind()` in the class `ConcreteBind` implement middleware-specific functionalities, but also incorporate code of the Observer and State patterns. In this context, there are different ways in which the patterns interact with each other. The pattern compositions vary from a simple method invocation between the patterns' roles to the sharing of one or more classes by the pattern roles. The pair-wise interactions of OO pattern implementations investigated in our study are classified in 4 categories as described in the following.

Invocation-based composition. The implementations of the two composed patterns, namely P1 and P2, are disjoint and they have no class in common. The roles of P1 and P2 are only connected through one or more method calls. This is the simplest form of pattern composition. The combination of Chain of Responsibility (CoR) and Prototype is included in this category. Figure 1 illustrates the realization of this combination for a middleware implementation. Note that the CoR and Prototype patterns are affecting different classes. Their composition is based only on a call to the method `ConcreteBind.makeRequest()` (implementing the Prototype pattern) within the method `handleRequest()` of the class `ConcreteMetaInvocationHandler` (participant of the CoR pattern).

Class-level interlacing. The implementations of patterns P1 and P2 have one or more classes in common. The roles of P1 and P2 are implemented by different sets of methods and attributes in these shared classes. In other words, the involved patterns have coinciding participant classes, but there is no common method or attribute implementing roles of both patterns. As a result, the pattern implementations have been *interlaced* (or tangled) at the class level. Examples of this composition category in Figure 1 are Memento with Observer, and Prototype with Observer. For example, the class `Port` belongs to both Memento and Observer implementations, but it has no method that contains code relative to both patterns.

Method-level interlacing. Differently from class-level interlacing, the implementations of patterns P1 and P2 have one or more methods in common. Different pieces of code in these methods are dedicated to implement roles of both P1 and P2. Hence the pattern implementations are interlaced at the method level. Both method- and class-level interlacing produce tangling of concerns, but at different levels of abstraction. Some examples of this kind of pattern composition appear in Figure 1: Mediator with CoR, and Mediator with Prototype. They can be easily detected in Figure 1 as the numbers of the patterns in the composition appear attached to a same method. For instance, the method `ConcreteBind.makeRequest()` is mostly dedicated to the implementation of the *Mediator* role. However, it also contains code relative to the Prototype pattern.

Overlapping. The implementations of patterns P1 and P2 share one or more statements, attributes, methods, and classes. This combination style is different from method-level interlacing because here the shared elements are entirely part of roles in both patterns; in the previous case, the parts of the code in the common method implementing the pattern roles are disjoint. An example of overlapping in Figure 1 is the combination of Decorator with Mediator: the interface `BindMediator` and the method `ConcreteBind.makeRequest()` are part of both pattern structures. It may be the case that we have a complete overlapping in the sense that the implementation of a given pattern is entirely contained by the other pattern. The Decorator pattern, for instance, contains the implementation of Template Method (Figure 1).

2.3 Assessment Procedures

Our study has focused on the assessment of compositions involving all the 23 GoF design patterns. We have used three medium-sized software systems as case studies: (1) an OpenOrb-compliant middleware system [3], (2) a measurement tool [6], and (3) an agent-based application for supporting paper submission and selection processes [10, 11]. Tables 2, 3, 4 and 5 present the compositions and from which system is extracted each one. These systems were selected for several reasons. First, their OO implementations are largely based on design patterns due to their requirements of maintainability, evolvavility, and reusability. The second reason is the heterogeneity of the pattern compositions found in these systems. Third, they encompass different characteristics, diverse domains, and different degrees of complexity in terms of pattern instances and their compositional nature. The following subsections describe how we have obtained and assessed the pattern compositions.

2.3.1 Selection and implementation of pattern compositions

We have reengineered the existing Java implementations of the first two systems with AspectJ in order to produce the AO versions of those systems. For the third case study, we have reused both existing Java and AspectJ implementations [10, 11]. In both OO and AO solutions, we have tried to maximize the separation of each pattern with respect to both the second pattern in the combination and the application-specific concerns. While implementing the AspectJ versions, we have also aimed at preserving the use of the original versions of the pattern implementations [15] as much as possible in order to correlate the results of this study with previous ones [12,15]. However, due to each application's specificities, in some cases we needed to carry out minor modifications in the original AspectJ implementations [15] while trying to achieve the intended pattern modularization. Moreover, in other cases we needed to rely on a different AspectJ version because the application context required a specific pattern variant.

A tally of 62 compositions was chosen in the 3 case studies. Most of these compositions are documented through the GoF pattern catalogue [9]. Each pattern participated at least in 2 compositions, and each composition category (Section 2.2) involved the minimum of 9 different composition instances. The measurement process was preceded by the isolation of each composition instance from the application implementation so that we could perform the proper measurements.

In order to compare the two implementations of the compositions, we had to ensure that both Java and AspectJ versions were implementing the same functionalities. Therefore, some minor modifications were realized in the code of the patterns. Examples of such kinds of changes were: (i) to add or remove a functionality – a method, a class or an aspect – in the aspect-oriented (or object-oriented) implementation of the composed patterns in order to ensure the equivalence between the two versions; we decided to add or remove a functionality to the implementation by evaluating its relevance for the pattern implementation; and (ii) to ensure that both versions were using the same coding styles due to the fine granularity of our metrics.

2.3.2 Measurement process

The quantitative assessment was based on the application of a metrics suite [10, 24] to the 62 compositions. These metrics are useful to capture important modularity dimensions in the pattern compositions, namely separation of concerns, coupling, cohesion, and size. The coupling, cohesion, and size metrics are extensions of traditional and OO metrics in order to be applied in a paradigm-independent way, and support the generation of comparable results between Java and AspectJ solutions. The metrics suite also encompasses new metrics for measuring separation of concerns. The separation of concerns metrics measure the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advices), and lines of code.

The used metrics are briefly described in Table 1; an extensive explanation and justification about them are out of the scope of this work and can be found at [10, 24]. Table 1 presents a brief definition of each metric, and associates them with the attributes measured by each one. These metrics have already been extensively used and proved to be useful quality indicators in several studies [7, 10, 11, 12, 14, 26]. We have applied the chosen metrics to both Java and AspectJ versions. We analyzed the results, and also compared them with the results gathered in the two main previous studies (Section 4.1).

In the measurement process, the data was partially gathered by our own measurement tool [6]. It supports all the metrics, except the metrics of separation of concerns (CDC, CDO, and CDLOC). The data collection of the separation of concerns metrics was preceded by the shadowing of every class, interface and aspect in both implementations of the pattern compositions. Their code was shadowed according to the pattern roles that they implement. We treated each design pattern as a concern in order to investigate its crosscutting structure in pattern compositions. After the shadowing, the data of the separation of concerns metrics (CDC, CDO, and CDLOC) was manually collected.

Table 1: The Metrics Suite [7, 18].

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components (CDC)	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.
	Concern Diffusion over Operations (CDO)	Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern Diffusions over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Coupling	Depth Inheritance Tree	
	Coupling Between Components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled.
	Number of Children (NOC)	Counts how many children a class or aspect has.
Cohesion	Lack of Cohesion in Operations (LCOO)	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same instance variable.
Size	Lines of Code (LOC)	Counts the lines of code.
	Number of Attributes (NOA)	Counts the number of attributes of each class or aspect.
	Weighted Operations per Component (WOC)	Counts the number of methods and advices of each class or aspect and the number of its parameters.

3 Results

This section presents the measurement results for the 62 compositions, which are grouped in the respective composition categories. For each category, we focus first on the presentation of results related to separation of concerns. Afterwards, we show how the aspectization of the pattern compositions impacted on the other software attributes. In order to illustrate the results, some specific compositions are used as representatives in the following subsections. The discussion about the interplay among all the results is concentrated in Section 4. Section 4 also discusses the relationships between our study's results and the conclusions obtained in previous case studies (Section 2.1).

Graphics are used to represent the data gathered in the measurement process. The graphic Y-axis presents the absolute values gathered by the metrics. Each pair of bars is attached to a percentage value, which represents the difference between the AO and OO results. A positive percentage means that the AO implementation was superior, while a negative percentage means that the AO implementation was inferior. These graphics support an analysis of how both solutions for the pattern compositions affect the selected measures. There are two kinds of graphics: (i) the graphics for SoC measures, and (ii) the graphics for the coupling, cohesion, and size attributes. The first one shows how each pattern was isolated in the pattern composition in terms of the 3 SoC measures. The results shown in the other graphics were gathered according to the entire composition point of view; that is, they represent the tally of metric values associated with all the classes and aspects involved in the pattern composition as a whole.

3.1 Invocation-based composition

For this category, 9 compositions were investigated (Table 2). The AO solution was superior in terms of SoC for 5 compositions against 1 of the OO solution. For 3 compositions no difference was observed. In general, the interactions between the patterns in this category are not addressed in the aspectization process. Their implementations are basically disjoint, with no shared method or class. The method calls that connect them typically were not part of the functionality defined by the superimposed roles and usually were not aspectized. As a result, the composition quality for this category largely depended on the patterns being combined. Considering the compositions analyzed, if at least one pattern presents good results in its individual aspectization, the consequence is an overall improvement of separation of concerns in terms of the whole composition implementation. Similarly, the OO solution is better when the composition contains patterns that do not achieve satisfactory modularization in AO implementations.

Table 2: Invocation-based pattern compositions.

Composition		System
Pattern A	**Pattern B**	
Singleton	Iterator	Middleware
Façade	Singleton	Middleware
Façade	Memento	Middleware
Command	Builder	Middleware
CoR	Prototype	Middleware
Abstract Factory	State	Measurement Tool
Interpreter	Iterator	Measurement Tool
Interpreter	State	Measurement Tool
Proxy	Interpreter	Measurement Tool

For example, the successful aspectization of the Interpreter pattern has a positive influence on the general result of its composition with the State pattern. As illustrated in Figure 2, this composition presents better results in terms of the number of transition points (CDLOC) and also in the diffusion over components (CDC). This result is due to the effectiveness of AspectJ mechanisms to localize the Interpreter pattern. This AO solution transfers the methods in charge of performing interpretations from the classes to the Interpreter aspect. As a consequence, the number of operations is not reduced (CDO) the diffusion over components (CDC) is reduced from 14 to 1 and the number of transition points is reduced from 26 to 2 (Figure 2).

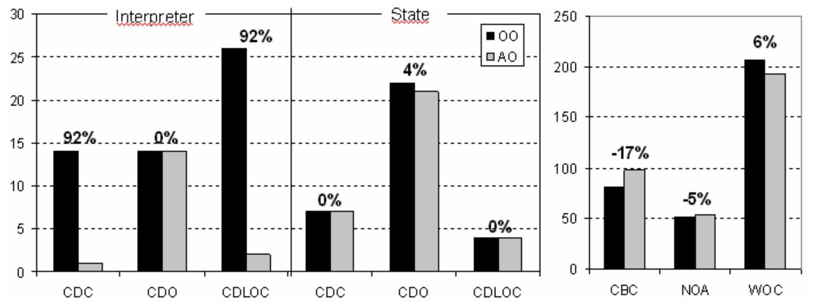


Figure 2: Results of an invocation-based composition: Interpreter with State.

The invocations among the patterns involved in the composition can be aspectized or not; it depends on the patterns involved in the composition. There were only 3 cases where the inter-pattern invocations were isolated in the aspects. We have observed that when the invocations are necessarily transferred to the aspect code, the overall coupling of the composition tended to be worse. This problem has happened in the AspectJ implementation of the combination of Interpreter with State (Figure 2). Similarly to the SoC measures, the other metrics, NOA and WOC, depended on each pattern involved in the composition due to the loose connection between the patterns.

3.2 Class-level interlacing

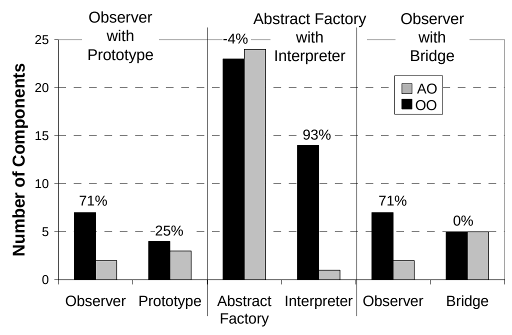
For the 12 compositions investigated in this category, all AspectJ solutions have in general shown significant superiority in terms of SoC measures. The compositions in this category are shown in Table 3. The improvements come primarily of disentangling the pattern concerns in the shared classes. The Java implementations of those classes typically include significant code from both patterns in addition to the business-related concerns. As a result, the OO solutions exhibited inferior separation of concerns as they encompass classes with mixed concerns. Moreover the shared classes in Java implementations presented low cohesion as their internal operations have a weaker coupling between them. Figures 3 and 4 respectively illustrate the SoC and cohesion superiorities of AO solutions through example pattern compositions in this category.

With respect to separation of concerns, the pattern compositions can be further classified in 3 groups. The first group includes the compositions where the two patterns were aspectized, and the AspectJ implementations of both of them have presented better separation of concerns. The combination involving Observer and Prototype (Figure 3) is a representative of this situation: the roles of both patterns were better localized in terms of components (CDC), operations (CDO), and transition points (CDLOC). In some measures, the superiority of AspectJ was higher than 20% for both patterns involved in the composition.

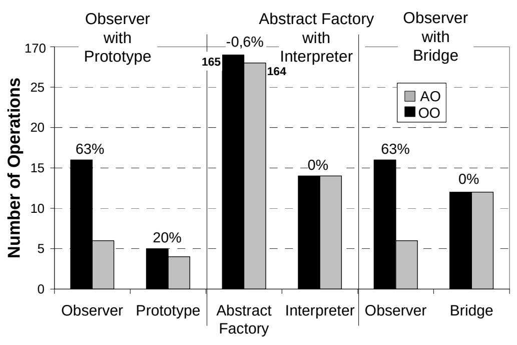
Table 3: Pattern compositions with class-level interlacing.

Composition		System
Pattern A	Pattern B	
CoR	Observer	Middleware
Proxy	Singleton	Middleware
Mediator	Observer	Middleware
Observer	Memento	Middleware
Observer	Prototype	Middleware
Observer	Strategy	Middleware
Observer	Template Method	Middleware
Observer	Visitor	Middleware
Observer	Bridge	Middleware
Decorator	Observer	Middleware
Abstract Factory	Interpreter	Measurement Tool
Proxy	Singleton	Measurement Tool

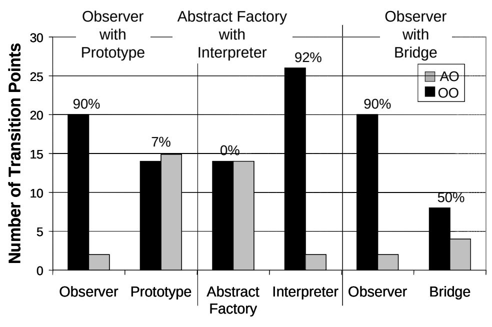
The second group of compositions also encompassed the aspectization of both patterns being composed. However, the AspectJ implementations have only shown SoC improvements for one of the composed patterns, as is the case of the combination of Abstract Factory with Interpreter. Benefits were observed only in the AO solution of the Interpreter (Figure 3). Finally, the compositions in the third group involved the aspectization of only one in each pair-wise composition. For example, the Observer pattern was the sole aspectized pattern in the composition with the Bridge pattern. The aspectized pattern was typically responsible for improvements in the separation of pattern-specific concerns in the composition (Figure 3). However, the AspectJ superiority for the Interpreter pattern has decisively contributed to the overall SoC of the composition. Specific constraints in the application implementation or in the pattern combinations were the reasons for not aspectizing one of the patterns (see Section 4.1.4 for further details).



(a) CDC measures



(b) CDO measures



(c) CDLOC measures

Figure 3: Separation of concerns in pattern compositions with class-level interlacing.

Interestingly, for all the 3 groups mentioned above, the AspectJ pattern implementations that presented improved SoC are exactly the ones that have exhibited superior results when analyzed in isolation in the previous studies [12, 15]. It means that the modularity of their AO solutions have scaled well in more complicate pattern instances and in the presence of pattern interactions with class-level interlacing. The AspectJ implementations of specific patterns that did not show improvements (e.g. Abstract Factory in Figure 3) also confirmed results observed in our previous assessments [12].

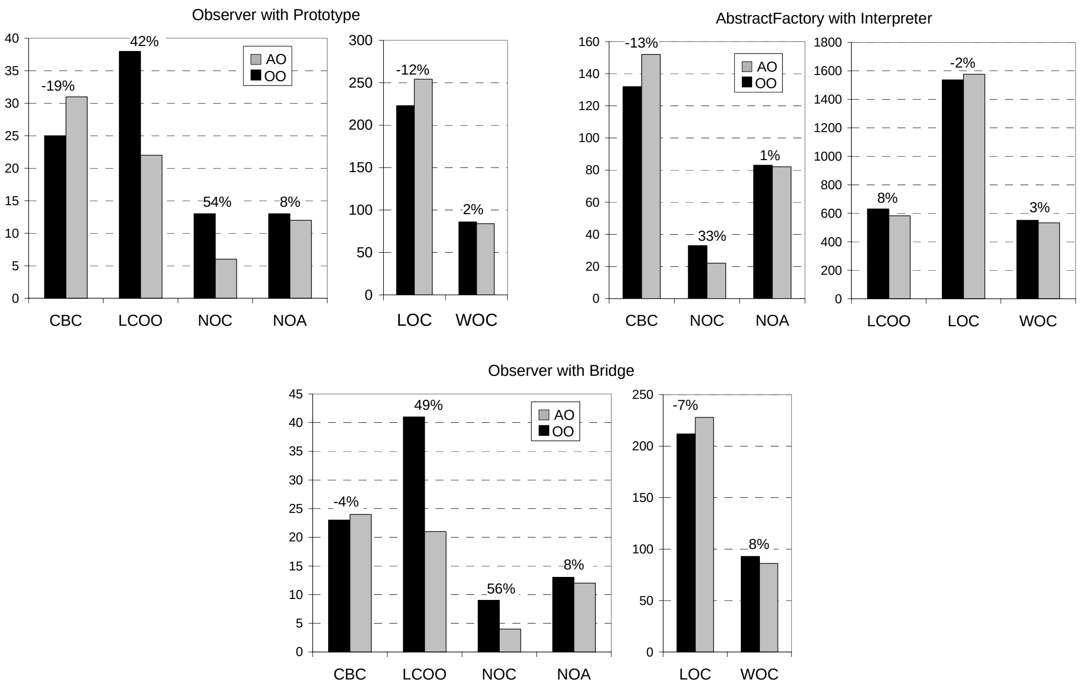


Figure 4: Pattern compositions with class-level interlacing: coupling, cohesion, and size measures.

Considering the other measures, the pattern combinations with class-level interlacing did not repeat the results in the previous composition category. As discussed before, invocation-based compositions (Section 3.1) have revealed much more influence of the two combined patterns in the overall results of the respective compositions. The main reason is that they encompass loose pattern interaction (i.e. simple method calls). Differing from those compositions, the outcomes here for all the 12 combinations with class-level interlacing were very similar, independently from the patterns taking part in the compositions. The presence of tight coupling between the composed patterns (i.e. one or more classes in common) led to similar benefits in the AspectJ implementations (Figure 4): (i) higher cohesion (LCOO metric), (ii) reduced number of attributes (NOA metric), and (iii) reduced number of operations and parameters (WOC metric). These recurring benefits are due to the reduction of tangling and scattering relative to the pattern-specific concerns in the classes shared by the composed patterns. The AspectJ implementations have also presented similar drawbacks in this category: more lines of code and stronger coupling (CBC metric). However, these drawbacks will be discussed in Section 4.1.3 because similar findings were obtained for these measures through all the composition categories.

3.3 Method-level interlacing

We have analyzed 17 compositions with method-level interlacing (Table 4), which we have classified in 4 groups according to the similarities in the measurement outcomes. The first group involved three combinations: Mediator with CoR, Observer with

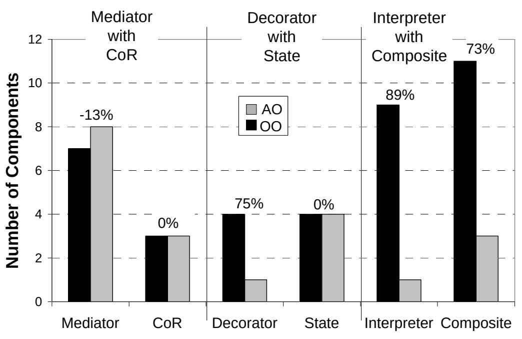
Composite, and CoR with Strategy. The SoC measures for this group have shown no benefits in favor of AspectJ. These compositions involve an interesting scenario: the AspectJ implementation of one pattern explicitly interferes in the separation of concerns relative to the second pattern. The aspect of the first pattern inevitably contains code of the second one. It directly contributed to no SoC improvements even in the combinations involving AspectJ pattern implementations qualified as superior in previous studies [12, 15], such as Observer, CoR, and Composite.

Table 4: Pattern compositions with method-level interlacing.

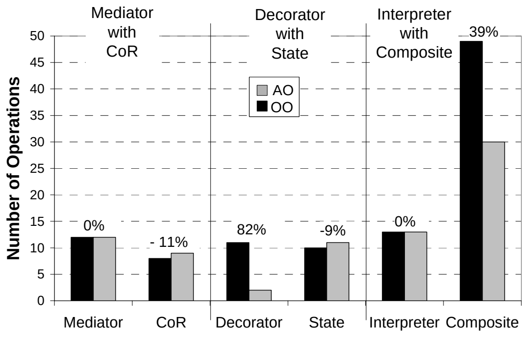
Composition		System
Pattern A	**Pattern B**	
Bridge	Composite	Middleware
Bridge	Visitor	Middleware
Bridge	CoR	Middleware
CoR	Strategy	Middleware
Mediator	CoR	Middleware
State	Observer	Middleware
Decorator	Prototype	Middleware
Decorator	State	Middleware
Mediator	Prototype	Middleware
Mediator	State	Middleware
Observer	Composite	Middleware
Prototype	Strategy	Middleware
Prototype	Template Method	Middleware
State	Strategy	Middleware
State	Template Method	Middleware
Factory Method	Memento	Middleware
Interpreter	Composite	Measurement Tool

For example, the Java implementation of the method `ConcreteMetaInvocationHandler.handleRequest()` have code of both Mediator and CoR patterns (Figure 1). The aspectization of the CoR pattern [15] implements this method as an inter-type declaration. As a result, the CoR aspect also contains code of the Mediator pattern, thereby leading to the increment of one in the CDC measure (Figure 5a). This means the Mediator code is inevitably scattered over an additional component in the AspectJ solution for this combination. Even if we try to refactor the method `handleRequest()`, it still will contain a method call that is part of the Mediator behavior. As we will discuss in Section 4.1.4, the Mediator pattern has not been aspectized in this combination. The size, coupling, and cohesion measures for this group have varied from composition to composition.

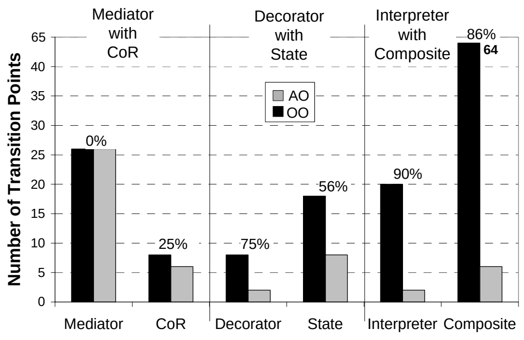
The second group includes only the composition Interpreter with Composite. To some extent, this combination presents a characteristic similar to the previous group: the Interpreter aspect is forced to have two calls to the Composite aspect, which are implemented using `aspectOf`. Nevertheless, the overall separation of concerns of the AO solution for this combination is superior (Figure 5). The Java implementation of Interpreter classes was highly tangled with Composite code, which is effectively modularized in the AspectJ implementation. In addition those two calls located at the Interpreter aspect are in two different classes of the Java solution, contributing to the AspectJ victory. As Figure 6 shows, this composition has revealed a high cohesion (LCOO), fewer attributes (NOA), and fewer operations and parameters (WOC).



(a) CDC measures



(b) CDO measures



(c) CDLOC measures

Figure 5: Separation of concerns in pattern compositions with method-level interlacing.

The third group is formed by 9 combinations: Decorator with Prototype, Decorator with State, Mediator with Prototype, Mediator with State, Prototype with Strategy, State with Strategy, Prototype with Template Method, and State with Template Method. In these cases, the AspectJ implementation of one pattern does not interfere in the separation of concerns relative to the second pattern. An explicit separation of concerns was achieved between the two patterns because each aspect isolates all the crosscutting code relative to the corresponding pattern. Their aspectization also separates the code of the two patterns that is mixed in the methods they have in common in the Java implementation.

For instance, the implementation of the method `ConcreteBind.makeRequest()` includes pieces of code from both Decorator and State patterns (Figure 1). The AO solution of this combination modularizes those pieces of code in the advices of the aspects implementing each pattern. Both aspects define the execution of the method `makeRequest()` as join point of interest. This method is not aspectized because it is part of the Mediator pattern, which has not been refactored as an aspect in the middleware case study (Section 4.1.4). In fact, we can see in Figure 6 that the AspectJ solution presents improved locality of pattern-specific concerns for this combination. However, since both aspects in this combination are affecting the same join point, an order in their executions needs to be established using `declare precedence`. This feature introduces additional coupling in the AO solution (Figure 6). Although with distinct implementation strategies, similar results are obtained for the other 8 pattern compositions in this group.

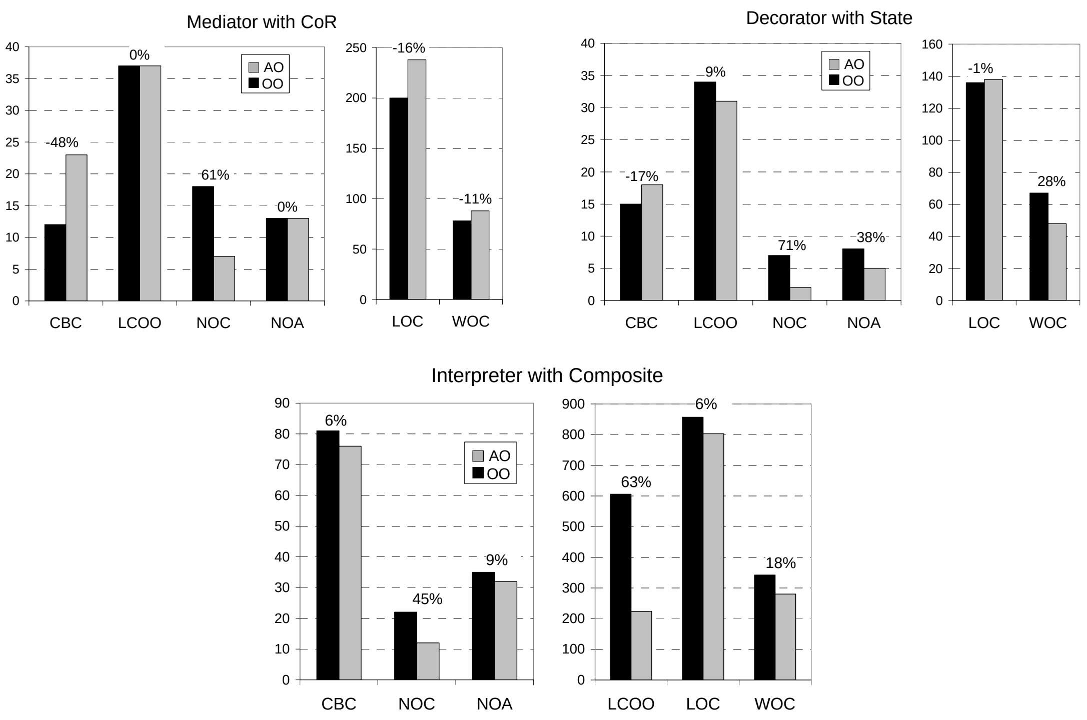


Figure 6: Pattern compositions with method-level interlacing: coupling, cohesion, and size measures.

The last group in this category encompasses 3 different compositions involving the Bridge pattern, and the combination of Factory Method with Memento. In these combinations, one of the patterns has not been modularized as aspects in the AspectJ implementation. The aspect of the second pattern modularizes its code that was tangled in the "interlaced" method. In this way, the separation of concerns and cohesion for the compositions in this group were notoriously improved.

3.4 Overlapping

We have analyzed 24 compositions with overlapping which are presented in Table 5. We have classified in 4 groups according to the similarities of the overlapping nature: attribute overlapping, statement overlapping, partial class overlapping and total or complete overlapping. We have not found any pattern combination involving method overlapping in the 62 compositions analyzed. Table 5 also shows which pattern compositions were classified in each group.

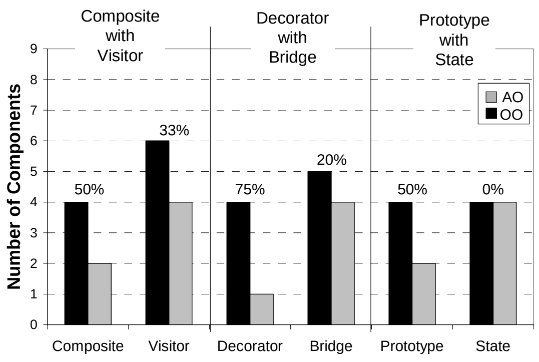
Attribute overlapping occurs when two or more patterns share a same attribute. In the 3 investigated compositions with attribute overlapping, the shared attribute was transferred to one of the aspects and accessed by the other pattern aspect through the invocation of `aspectOf`. As a consequence, the separation of pattern roles is increased, but the coupling is stronger. For instance, in the composition of Composite and Visitor patterns, an attribute is responsible for maintaining the reference to the composite object, and it also plays the *ObjectStructure* role of the Visitor pattern. It means that a same attribute is used in a different way for each pattern. In the aspectization process, the attribute is maintained in the Visitor pattern and is removed of the Composite and replaced by an invocation to `aspectOf`. In spite of the stronger coupling, this strategy led to a stronger cohesion (LCOO) and the reduction of operations and parameters (WOC) in the AO implementation. These positive results are illustrated in Figure 8. In the other 2 compositions with attribute overlapping, the results were not similar because there were so many inter-aspect invocations due to the shared attribute. This problem led to a reduction of separations of concerns (in terms of the three measures) and an augment on the coupling between components. In Figure 7, which depicts the composition of Composite and Visitor, the influence of attribute overlapping is lower because there is only one invocation of `aspectOf`.

Table 5: Pattern compositions with overlapping.

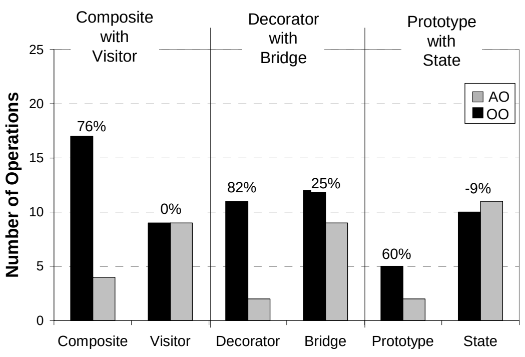
Composition		System
Pattern A	**Pattern B**	
Factory Method	CoR	Middleware
Flyweight	Command	Middleware
Command	Proxy	Middleware
Decorator	Bridge	Middleware
Decorator	Strategy	Middleware
Bridge	Factory Method	Middleware
Factory Method	Command	Middleware
Factory Method	Observer	Middleware
Factory Method	Visitor	Middleware
Factory Method	Composite	Middleware
Factory Method	Flyweight	Middleware
Flyweight	Adapter	Middleware
Mediator	Decorator	Middleware
Mediator	Strategy	Middleware
Mediator	Template Method	Middleware
Proxy	Adapter	Middleware
Proxy	Builder	Middleware
Proxy	Flyweight	Middleware
Template Method	Bridge	Middleware
Composite	Visitor	Middleware
State	Prototype	Middleware
Decorator	Template Method	Middleware
Proxy	Composite	Measurement Tool
Strategy	Template Method	Agent Application

We have analyzed 4 compositions with statement overlapping, in which the involved patterns share, at least, one statement. In the combination of Prototype and State a given statement creates a clone and also modifies the state of a variable that represents the current state. We have observed for the 4 compositions that, in general, the aspectization leads to the augment of separation of concerns and tends to increase the coupling between the patterns. This stronger coupling is because aspect code is intermingled with the invocations to the other pattern aspect. Thus, similarly to the attribute overlapping, the coupling between components is increased after aspectization due to the inter-aspect coupling in addition to: (i) the natural coupling between the business classes, and (ii) the coupling between the classes and the aspects. Figure 8 shows the augment of coupling, lack of cohesion and WOC in the combination of Prototype with State.

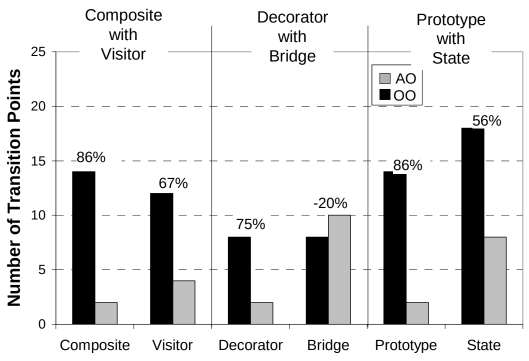
In the 17 other compositions occurs a class overlapping. This overlapping occurs when at least one class is shared by patterns involved in a composition. This overlapping can be total or partial. The total overlapping (3 compositions) is when a pattern is completely contained in another pattern. The composition of the Template Method and the Decorator patterns is an example of total overlapping (Figure 1) where the *Decorator* and *ConcreteDecorator* roles of the Decorator pattern contain the *AbstractClass* and *ConcreteClass* roles of the Template Method. The aspectization process of the dominant pattern (Decorator) removed the subordinated pattern (Template method). This feature impacts on the SoC metrics (Figure 7) with the reduction of CDC and also the number of transition points. Coupling, LOC and WOC are also reduced (Figure 8).



(a) CDC measures



(b) CDO measures



(c) CDLOC measures

Figure 7: Separation of concerns in pattern compositions with overlapping.

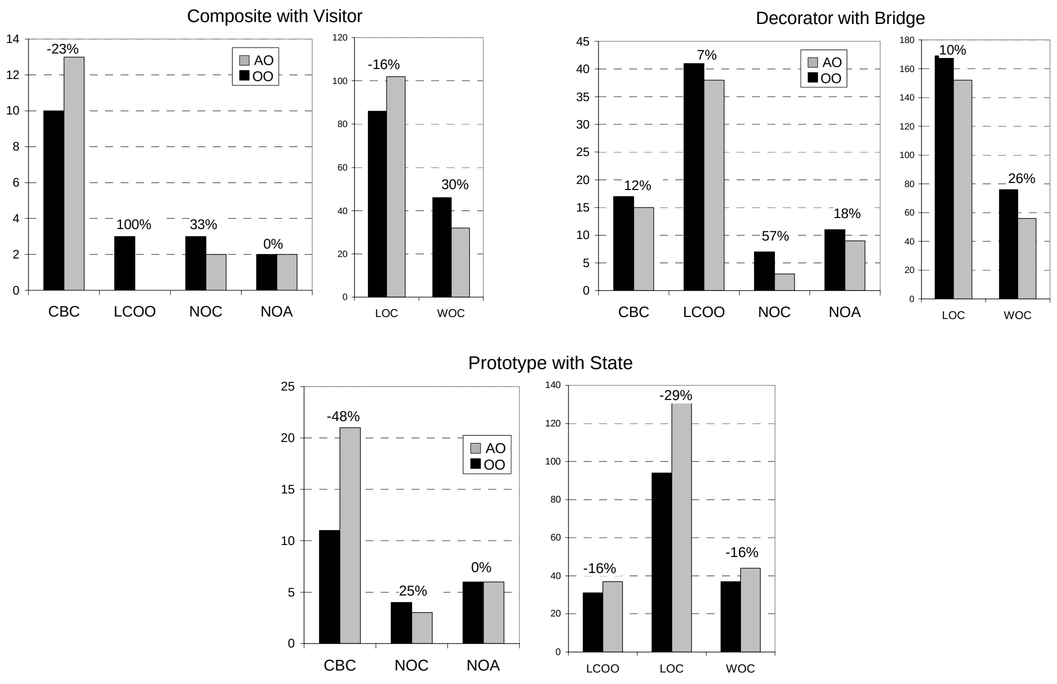


Figure 8: Pattern compositions with overlapping: coupling, cohesion, and size measures

With respect to the partial class overlapping, 14 compositions were found in our case studies. In these combinations, part of the involved patterns share one or more classes. For instance, the composition of Decorator with Bridge shared two classes that played two Bridge roles (*Implementor* and *ConcreteImplementor*) and two Decorator roles (*Decorator* and *ConcreteDecorator*). The original classes of the Bridge pattern shared by the Decorator pattern are transferred to the Decorator aspect and "removed" from the application. In general, the aspectization process of compositions with partial class overlapping, the aspectization of a pattern implies that the second pattern is not aspectized. Similarly to other 13 combinations, the aspectization of the Decorator pattern combined with the Bridge pattern resulted in improved SoC and also a slightly reduction of coupling, LOC and WOC.

4 Discussions

This section provides a more general analysis of the previously observed results in Section 3, and discussions about the constraints on the validity of our empirical evaluation.

4.1 General Analysis

This section introduces a more qualitative analysis by discussing the four questions raised in the introduction and using the collected quantitative data (Section 3) as the basis. When appropriate, we also correlate through the sections below the findings in our evaluation with claims and results from previous systematic case studies [12, 15] (Section 2.1).

4.1.1 Does AOP enhance pattern composability?

For the context of this study, we consider that a pattern implementation has a "good" composability if it can be directly reused and smoothly extended to different composition contexts. A transparent pattern composition would require only concretizing abstract aspects and extending abstract pointcuts and methods. The core implementation of the pattern roles should not be aware of the composition specificities. Moreover the aspectization of the pattern composition should not impact negatively on the modularity attributes. The results reported in Section 3 suggest that the success or failure for supporting a straightforward composition aspectization depended basically on two key complementary factors: (i) the suitability of AOP to modularize a given design pattern, and (ii) the intricacies of the pattern composition instance at hand. The predominance of a factor over the other depended on the composition category.

The first factor predominated when we had two patterns taking part in an invocation-based composition (Section 3.1) or simply having one or more classes in common (Section 3.2). There is a loose connection between the involved patterns in these cases. As a result, the composition transparency and the achieved modularity degree depended mostly on the adequacy of AspectJ mechanisms to isolate the crosscutting concerns relative to each separate pattern. The particularities of compositions based on invocations and class-level interlacing did not impose major problems. For most of the investigated compositions, we were able to use the reusable pattern implementations as proposed in [15] without changes to the core structure of the patterns. We have used in few cases a different implementation, but because in such situations the application circumstances required pattern variants. In fact, most measures for the patterns participating in such compositions in this study were similar to the measures we have obtained for each individual pattern in our previous quantitative study [12].

However, an important finding of this study is that the presence of more intricate relationships between two patterns can hinder a smooth composition process and, sometimes, affect negatively the modularity attributes of the patterns being composed. As reported in Sections 3.3 and 3.4, some pattern compositions with intra-method interlacing and overlapping required additional restructuring steps in the original pattern implementations as proposed in [15]. For example, sometimes we need to change the original structure of the pattern and use inter-type declarations in one pattern aspect so that the second pattern aspect can be properly combined with the former. This means that the aspectual design of the pattern needed to be aware of the composition context and the generic pattern aspect could not be directly reused.

This scenario happened in the AspectJ implementation of State with Observer for the middleware case study. The Observer aspect was interested in events related to invocations of methods of the State aspect. These steps were often required even for patterns which have been qualified as reusable and/or pluggable [15] in the previous case studies, such as State, Adapter, and Strategy. Such refactoring steps do not necessarily led to negative effects on the modularity attributes of the pattern composition. Indeed a number of cases indicated that the AspectJ solutions were superior, such as the composition of State with Observer mentioned above.

However, as discussed through Section 3, the aspectization of some specific compositions with strong coupling between the patterns can bring modularity problems. For example, there were compositions where a pattern aspect inevitably contained code of the second one, thereby affecting negatively the overall separation of concerns (Section 3.3). Moreover the aspectization of different types of overlapping-based compositions tended to present SoC improvements, but also typically resulted in additional couplings between the modules participating in the composition (Section 3.4). Interestingly, some cases involved design patterns that have shown superiority for AspectJ implementations in previous studies [12, 15], such as CoR and Adapter. Section 4.1.4 shows that some additional issues can emerge in compositions involving more than two patterns.

4.1.2 Separation of roles in pattern compositions

Section 3 has focused the discussion of the results for separation of concerns under the composition point of view. This section summarizes and discusses the results for separation of concerns in terms of each design pattern. Previous evaluations [12, 15] have centered the assessment on individual instances of design patterns. In this study, our goal is to understand to what extent separation of each pattern roles is preserved in contexts involving pattern compositions.

Table 6 summarizes the findings on separation of concerns (SoC) for each GoF design pattern. The rows of the table present all 23 design patterns, while the columns show the main SoC conclusions of the other experiments and this study. The second and third columns respectively describe the SoC results (locality) of the HK study [15] and of our first study [12]. The fourth column summarizes the SoC findings related to each pattern in our composition-oriented evaluation. The last two columns are respectively concerned with the number of compositions each pattern participates and if the used version follows the original implementation proposed by Hannemann and Kiczales [15]. In second column's cells, the value "yes" means that the corresponding AspectJ pattern implementation was qualified with a good localization in terms of its roles. With respect to the second and third columns, we have classified an AspectJ (labeled "AO") or Java (labeled "OO") solution as superior when it has achieved better results for separation of concerns. In the third column, the conclusion was based on the analysis of how many compositions a given OO or AO solution were better when compared with the results of the other solution. The AspectJ solutions that achieved the best results (more than 35% in all SoC measures) for all compositions are marked with the symbol "+" in the third column. Finally, the symbol "*" in the last column means that the pattern was not aspectized in this study (Section 4.1.4).

Table 6: Overview of the main findings of the three studies.

Pattern	Previous Study			This Study	
	HK Study (Locality)	**Our First Study (SoC)**	**Separation of Concerns**	**Number of Composition**	**Original Implementations**
Abstract Factory	no	OO	=	2	yes
Adapter	yes	AO	=	2	yes
Bridge	no	OO	=	7	no*
Builder	no	OO	=	2	yes
CoR	yes	AO	=	6	yes
Command	yes	AO	=	4	yes
Composite	yes	AO	AO^+	6	yes
Decorator	yes	AO	AO^+	7	yes
Façade	Same Implementation for Java and AspectJ			2	yes
Factory Method	no	OO	OO	8	yes
Flyweight	yes	OO	=	4	yes
Interpreter	no	=	AO	4	no
Iterator	yes	AO	AO	4	yes
Mediator	yes	AO	=	7	no*
Memento	yes	AO	AO	3	yes
Observer	yes	AO	AO^+	11	yes
Prototype	yes	AO	AO	7	yes
Proxy	yes	AO	AO	8	yes
Singleton	yes	AO	AO	4	yes
State	yes	=	AO	8	yes
Strategy	yes	AO	AO	7	no
Template Method	yes	OO	AO	7	no*
Visitor	yes	AO	AO	4	yes

Table 6 shows that only one design pattern has clearly presented superior separation of pattern-related concerns in Java implementations. Factory Method pattern provided better results in separation of concern for OO version and this finding confirms our first study result (column 2). In addition, the AspectJ implementations of 13 patterns in this study have shown better results in terms of separation of concerns, and 9 patterns presented similar (or not conclusive) results in both OO and AO implementations. For the 13 patterns that have shown SoC superiority in AspectJ implementations, only two (Interpreter and State) refuted findings of our previous study, when they were analyzed in isolation. In the previous study, they were classified as "similar results in both OO and AO implementations". However, the Interpreter solution used in this study was different from original HK implementation, which was also used in our previous evaluation. Due to different application constraints, all `interpret()` methods were moved to the aspect and introduced to classes by inter-type mechanisms (Section 3.1).

The State pattern had its modularity improved in most of AO compositions (6 against 2). The two cases where no SoC improvement was detected, the AspectJ implementations were exactly the same as proposed by the HK study. In the HK implementations, the state transitions are located in Java classes that play the State role, which is usually defining (Section 2.1). For example, this result was observed in the combination of State with Interpreter (Section 3.1). Differently from the original HK implementation, the state transitions in the other 6 implementations occur in the classes that play the Context role in the State pattern [9]. This role is typically superimposed, and the state transitions in those classes result in high tangling and scattering of the pattern implementation. In these cases, the aspectization process is also effective to remove the state transitions from the business classes and other pattern implementations, reducing the tangling and scattering. The AspectJ superiority for these State instances can be observed in combinations appearing in Figures 5 and 7, where the difference is higher than 56 % for the CDLOC measures. The Java implementation is superior in the CDO measures, but it is because the State aspects have an additional operation. In addition, the difference is lower than 10 %, which can be considered as insignificant.

The 9 patterns with similar results for AO e OO versions (represented by "=" in column 4) can be classified in two main situations. The first situation includes three patterns - Bridge, Mediator and Template Method, which were not aspectized because application constraints or composition particularities (Section 4.1.4). In the second case, the pattern aspectization has not produced SoC improvements in almost all the compositions those patterns participated. This category included: Abstract Factory, Adapter, Builder, CoR, Command, and Flyweight. We have identified 3 main factors that determine the negative performance of AspectJ for modularizing those patterns: (i) the composition particularities that the pattern participated, (ii) the pattern instance size, i.e. the number of classes playing the pattern roles, and (iii) the aspectization approach. The CoR pattern, for instance, is an example in which the performance was influenced by the compositions it took part. The method-level interlacing composition "CoR with Strategy" have shown no benefits in favor of AspectJ because the aspectization of Strategy brings code of the CoR pattern to its aspect (Section 3.3). The Command pattern has also not exhibited an improved separation of concerns in the AspectJ implementations. In this case, the main factor was the instance size. There were not too many classes in the application playing the roles *Command*, *Receiver* and *Invoker*. The Abstract Factory, Builder and Flyweight patterns have presented no modularity improvements, confirming the findings of previous case studies (columns 2 and 3 of Table 6).

4.1.3 Coupling, cohesion, and size

Based on the results presented in Section 3, we have observed that the measures relative to cohesion (LCOO), complexity of operations (WOC), and number of attributes (NOA) also depend both on the composition category and on the involved patterns. In general, the AO solutions were superior in terms of NOC measures, since the use of AspectJ reduces the overuse of inheritance mechanisms. However, as illustrated in Figures 4, 6, and 8, most measures indicated that AspectJ implementations resulted in higher coupling (CBC) and more lines of code (LOC) than the respective Java implementations.

However, a careful analysis of the implementations show that these higher CBC and LOC values for AO solutions in general are related to presence of generic aspects in several AspectJ pattern implementations, which have the intention of making the pattern solutions more reusable. As several investigated compositions involve few participant classes playing the pattern roles, the presence of generic aspects artificially has lead to higher values for LOC and CBC. This effect was more evident when we compared the composition instances taken from the middleware implementation with the composition instances obtained from the agent-based application and the measurement tool. The former ones often involve few participant classes while the other ones typically consist of several participant classes. For example, the composition Interpreter with Composite (Figure 6) was taken from the measurement tool and has exhibited favorable LOC and CBC values for the AspectJ implementation.

Nevertheless, it is important to highlight that in several cases a higher CBC value was in fact a clear indicator of stronger coupling in the AspectJ solution. For example, this problem happened in some invocation-based compositions when the inter-pattern invocations were inevitably transferred to the code of the aspects (Section 3.1). As there was an implicit connection between the base classes, the aspect-class and inter-aspect dependencies just introduced new sources of coupling in the composition implementation. Similar coupling problems were identified in compositions with intra-method interlacing (Section 3.3) and overlapping (Section 3.4).

4.1.4 Scalability of AOP in Complex Compositions

The previous sections focused on discussing how the aspectization of pair-wise compositions impacts different modularity attributes. This section discusses how AOP scaled in compositions involving a greater number of patterns in terms of such modularity attributes and pattern composability. We have implemented and analyzed different compositions with 3, 4, 5, 6, and 7 patterns, such as the one represented in Figure 1. In general, we have observed that the measures tended to be similar in these more complex compositions, especially when they mostly involved invocation-based and intra-class compositions. However, we have detected some problems when the combination included a high incidence intra-method interlacings and overlappings.

In some situations, these problems hindered the aspectization of certain design patterns, such as Proxy and Mediator. The aspectization of certain patterns, as proposed in [15], can cause some design conflicts. As a consequence, it is necessary to carefully analyze which patterns should be aspectized and which patterns should not. Consider for example the composition of Proxy, Flyweight, and Adapter patterns. The *Adapter*, *FlyweightFactory*, and *RealSubject* roles have a class in common. The HK implementation of the Flyweight pattern suggests the transformation of *FlyweightFactory* into an aspect. On the other hand, the Adapter implementation requires the removal of the Adapter class, and the use of inter-type declarations to insert its methods in the class is playing the *Adaptee* role. The Proxy pattern acts as a client in this composition. With the divergence of these two suggestions, it is necessary to choose which pattern should be aspectized and if the Proxy pattern will invoke the aspect represented by *FlyweightFactory* or will invoke a method defined in the *Adaptee*. In the implementation of the middleware system we have chosen to aspectize the Adapter pattern because this approach would reduce the coupling among the elements of the composition. The decision of aspectizing the Adapter pattern made the aspectization of Flyweight impossible. This also reveals that the results of the aspectization of Proxy with Flyweight and Proxy with Adapter are different from the aspectization of the combination of such three patterns: Proxy, Flyweight and

Adapter. As a result, in evolution scenarios, these pair-wise combinations may need to be restructured in the need of adding a third pattern.

In some cases, the aspectization of a given design pattern in complex compositions has not been revealed as a good design option according to application requirements. Consider the example illustrated in Figure 1 involving the Mediator pattern as a central design element. In this case, the Mediator and Proxy patterns are combined to implement the *LocalBind* mechanism of OpenOrb. The Proxy pattern is used to maintain the contracts defined between the components. In this example the `BankStub` class is used as a Proxy to access the implementations defined by the `Bank` *Subject*. The Mediator pattern supports the dynamic adaptation mechanism of OpenOrb. In this case, the `BankStub` class also plays the *Colleague* role by inheriting the bind attribute from the `Interface` and `Port` classes. This attribute supports the invocation of `bindlocal()` of the `ConcreteBind` class. The aspectization of these two patterns would require the definition of an advice to handle each method provided by the client's interfaces. In addition, all invocations would be forwarded to the abstract Mediator aspect. As a result, the Mediator pattern has not been aspectized because this strategy would insert a bottleneck in the invocation of the middleware platform and, as a consequence, the performance would be reduced.

4.2 Study Constraints

The use of the GoF patterns could be pointed out as a constraint in our experimental evaluation. However, we have focused first on this pattern catalogue for two main reasons. First, they are domain-independent and widely-used solutions. Second, this strategy allowed us to compare our results with previous case studies that exploited these patterns, and understand how the pattern implementations scaled in the presence of pattern interactions. As previous work has not systematically investigated the influence of AOP on pattern composability, we believe this study improves the current knowledge base about the aspectization of these general-purpose patterns. Other researchers can reuse our study format as a basis for further studies intended to investigate other design patterns.

There are a number of other existing metrics and other modularity dimensions that we could exploited in our study. We have to decide to focus on the metrics described in Section 2.3.2 because they have already been proved to be useful in several previous case studies to be useful quality indicators in several case studies [7, 10, 11, 12, 26]. In fact, despite the well-known limitations of these metrics, as already discussed in [12], they complement each other and are very useful when analyzed together. In addition, there is no way in a single study to explore all the possible measures. For every possible metrics suite that you take, there will be always some dimensions that will remain uncovered. In addition, future case studies can use additional metrics and assess the pattern compositions in terms of different modularity dimensions.

It is also important to notice that the scope of our experience is limited to: (i) the patterns selected for this comparative study, (ii) the specific Java and AspectJ implementations mostly based on the GoF book [9] and the HK study [15], (iii) the Java and AspectJ programming languages, (iv) the composition categories described in Section 2.2, and (v) our 3 case studies. Although our study covers a huge number of pattern compositions and different composition categories, it obviously does not cover all the composition possibilities. For instance, it does not exploit method overlapping. However, we believe that this first empirical study on the aspectization of pattern compositions provided interesting evidences about benefits and drawbacks the use of AO abstractions might bring, as discussed in Sections 3 and 4.1.

5 Related Work

Related work can be categorized into two groups: those related to pattern composability and those that empirically investigates aspectization of multiple crosscutting concerns. However, none of them investigates the impact of AOP on the aspectization of pattern compositions in the light of fundamental software attributes.

5.1 Pattern Composability

There are several ways of classifying relationships between design patterns according to different purposes [27, 31]. Zimmer [31] have proposed a classification of the relationships between the GoF design patterns. The following categories were proposed: (i) X uses Y, (ii) X is similar to Y, (ex.: Abstract Factory, Prototype and Builder deal with object creation; Glue and Mediator decouple objects); and (iii) X can be combined with Y (a Factory Method is typically called in a Template Method; Composite and Decorator are often used together). However, this is a higher-level classification used with the purpose of improving the documentation of pattern languages. The classification used in this work was focused on the composition of the implementations for pattern solutions; indeed, it was abstracted from pattern realizations in several real system implementations [3, 7, 11] and our own extensive experience on pattern compositions.

Murali et al [22] discusses the use of design patterns and AOP in a middleware implementation. They claim that the combination of AOP and design patterns lead to many benefits to the middleware in terms of reusabilility, modularity, and adaptability. Rouvellou et al. [23] have discussed how middleware modularity can be improved by using separation of concerns strategies. They have theoretically stated that the entanglement between middleware components is low when they are highly separable. However, these authors do not apply any metrics to assess the implementation and do not give an empirical support for such conclusions. In addition, they do not analyze the composition of patterns used in the implementation.

5.2 Other Empirical Studies

There are a number of quantitative studies (such as [7, 11, 12, 14]) that apply the same metrics used in this work. However, none of them applies the metrics upon pattern compositions. Godil and Jacobsen [14] have applied the metrics to evaluate an aspect-oriented version of a database system refactored using the horizontal decomposition principle. Although they applied the metrics to assess an AO implementation, they do not use design patterns in such a system.

Soares [26] focuses on the investigation of AOP to modularize distribution and persistence concerns. He has used specific design patterns to implement Java and AspectJ versions of a web-based system. The author has concluded that the AspectJ implementation is better than the corresponding Java implementation. However, this study has also not assessed the suitability of AOP to isolate pattern implementations in the presence of intricate pattern interactions.

6 Conclusion

Since the publication of the first catalog containing the 23 Gang-of-Four patterns [9], design patterns have quickly been recognized to be useful and important in successful software development. It is well recognized that programming languages affect pattern implementation. Hence it is natural to explore the effect of AOP techniques on the implementation of the GoF patterns. This paper presented an empirical study that investigated the scalability of AOP for composing GoF design patterns. We have used 3 medium-sized systems implemented in Java and AspectJ, and evaluated 62 compositions in these systems.

We also compared this study results with the findings from previous studies, contributing to an improved body of knowledge on the scalability of AOP. We defined a categorization of pattern compositions, and the relationships with different ways of crosscutting. For each category, we analyzed the compositions, determined the aspectization approach, applied the metrics, and presented in this paper a depth analysis of the results. The study shows that the aspectization results depend on the patterns involved, the composition intricacies, and the application requirements. In some situations, the aspectization of the pattern composition is not straightforward and several design options need to be considered. Sometimes, it requires a global reasoning in order to understand that impact of each design option in the context of the whole system implementation. In order to extend the body of knowledge on the aspectization of design patterns, as a future work we intend to use other AO programming languages, such as Caesar [21] and Hyper/J [28], and apply the same metrics used in this work.

References

[1] Alencar, P. et al. **A Query-Based Approach for Aspect Measurement and Analysis**. TR CS-2004-13, School of Computer Science, Univ. of Waterloo, Canada, Feb 2004.

[2] **AspectJ Team. The AspectJ Guide**. http://eclipse.org/aspectj/.

[3] Blair, G., Costa, F., Saikoski, K., Parlavantzas. The Design and Implementation of Open ORB version 2. **IEEE Distributed Systems Online Journal**, 2(6), 2001.

[4] Chidamber, S. and Kemerer, C. A Metrics Suite for OO Design. **IEEE Transaction on Software Engineering.**,20-6, June 1994, 476-493.

[5] Fenton, N. and Pfleeger, S. **Software Metrics: A Rigorous Practical Approach**. London: PWS, 1997.

[6] Figueiredo, E., Garcia, A, Sant'Anna, C., Kulesza, U., Lucena, C. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. **Proceedings of the 9th ECOOP Workshop on Quantitative Approaches in OO Software Engineering (QAOOSE.05)**, Glasgow, July 2005.

[7] Filho, F., Rubira, C., Garcia, A. A Quantitative Study on the Aspectization of Exception Handling. **Proceedings of the ECOOP Workshop on Exception Handling in OO Systems**, in conjunction with the ECOOP'05 Conference, Glasgow, Scotland, July 2005.

[8] Florijn, G., Meijers, M., Winsen, P. van. Tool Support for Object-Oriented Patterns. **Proceedings of European Conference on Object-Oriented Programming (ECOOP),** 1997.

[9] Gamma, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[10] Garcia, A. **From Objects to Agents: An Aspect-Oriented Approach**. Doctoral Thesis, PUC-Rio, Rio de Janeiro, Brazil, April 2004.

[11] Garcia, A. et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. **In Software Engineering for Multi-Agent Systems II**, Springer, LNCS 2940, Jan 2004.

[12] Garcia, A. et al. Modularizing Patterns with Aspects: A Quantitative Study. **Proceedings of the 4th International Conference on Aspect-Oriented Software Development**, 2005, 3 - 14

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

[14] Godil, I., Jacobsen, H. Horizontal Decomposition of Prevayler. **In Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research 2005 (CASCON)**, Richmond Hill, Canada, October 2005.

[15] Hannemann, J., Kiczales, G. Design Pattern Implementation in Java and AspectJ. **Proceedings of Conference On Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'02).** Nov 2002, 161-173.

[16] Henderson-Sellers, B. **Object-Oriented Metrics: Measures of Complexity**. Prentice Hall, 1996.

[17] **Java Reference Documentation**. http://java.sun.com/reference/docs/index.html.

[18] Kersten, M. and Murphy, G. Atlas: A Case Study in Building a Web-based Learning Environment Using Aspect-Oriented Programming. **Proceedings of Conference On Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)**. November 1999.

[19] Kiczales, G. et al. Aspect-Oriented Programming. **Proceedings of European Conference on Object-Oriented Programming** ECOOP'97, LNCS 1241, Finland, June 1997, 220-242.

[20] Lopes, C. **D: A Language Framework for Distributed Programming**. PhD Thesis, Northeastern University, 1997.

[21] Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. In **Proceedings of the ACM International Conference on Aspect-Oriented Software Development (AOSD'05)**, Chicago, USA, pp. 90-99, (2005).

[22] Murali, T., Pawlak, R., Younessi, H. Applying Aspect Orientation to J2EE Business Tier Patterns. **Aspects, Components and Patterns for Infrastructure Software Workshop (AOSD2004)**, pp 55-61, 2004, Lancaster, UK

[23] Rouvellou, I., Sutton, S. and Tai, Stefan. Multidimensional Separation of Concerns in Middleware. **Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)**, Ireland, 2000.

[24] Sant'Anna, C. et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. **Proceedings of Brazilian Symposium on Software Engineering (SBES'03)**, Manaus, Brazil, Oct 2003, 19-34.

[25] Sant'Anna, C. et al. Design Patterns as Aspects: A Quantitative Assessment. **Proceedings of Brazilian Symposium on Software Engineering (SBES'04)**, Brazil, Oct 2004.

[26] Soares, S. **An Aspect-Oriented Implementation Method.** Doctoral Thesis, Federal Univ. of Pernambuco, Oct 2004.

[27] Soukup, J. **Implementing Patterns**. In: Coplien J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison-Wesley 1995, pp. 395-412.

[28] Tarr, P. et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. **Proceedings International Conference on Software Engineering (ICSE'99)**, May 1999, 107-119.

[29] Zhao, J. **Towards a Metrics Suite for Aspect-Oriented Software**. TR SE200213625,Inf. Proc. Society of Japan, 2002.

[30] Zhao, J. and Xu, B. Measuring Aspect Cohesion. **Proceedings Conference on Fundamental Approaches to Software Engineering (FASE'04)**, LNCS 2984, Barcelona, March 2004, 54-68.

[31] Zimmer, W. Relationships between Design Patterns. **Pattern Languages of Program Design**, pp. 345 – 364, 1995

[32] Zuse, H. **History of Software Measurement**. Available on-line at: irb.cs.tu-berlin.de/~zuse/metrics/History_00.html.

[33] Yacoub, S. and Ammar, H. **Composition of Design Patterns**. Addison Wesley, 2003.