Modularizing Design Patterns with Aspects: A Quantitative Study

Alessandro Garcia[1], Cláudio Sant'Anna[2], Eduardo Figueiredo[2], Uirá Kulesza[2], Carlos Lucena[2], and Arndt von Staa[2]

[1] Lancaster University, Computing Department, InfoLab 21, Lancaster - United Kingdom
garciaa@comp.lancs.ac.uk
[2] PUC-Rio, Computer Science Department, LES, SoC+Agents Group, Rua Marques de São Vicente, 225 - 22453 - 900, Rio de Janeiro, RJ, Brazil
{claudios, emagno, uira, lucena, arndt}@inf.puc-rio.br

Abstract. Design patterns offer flexible solutions to common problems in software development. Recent studies have shown that several design patterns involve crosscutting concerns. Unfortunately, object-oriented (OO) abstractions are often not able to modularize those crosscutting concerns, which in turn compromise the system reusability and maintainability. Hence, it is important verifying whether aspect-oriented approaches support improved modularization of crosscutting concerns relative to design patterns. Ideally, quantitative studies should be performed to compare OO and aspect-oriented implementations of classical patterns with respect to fundamental software engineering attributes, such as coupling and cohesion. This paper presents a quantitative study that compares Java and AspectJ solutions for the 23 Gang-of-Four patterns. We have used stringent software attributes as the assessment criteria. We have found that most aspect-oriented solutions improve separation of pattern-related concerns, although only four aspect-oriented implementations have exhibited significant reuse. This paper also discusses the scalability of the analyzed solutions with respect to separation of concerns, and the determination of a predictive model for the modularization of design patterns with aspects.

1 Introduction

Since the introduction of the first software pattern catalog containing the 23 Gang-of-Four (GoF) patterns [9], design patterns have quickly been recognized to be important and useful in real software development. A design pattern describes a proven solution to a design problem with the goal of assuring reusable and maintainable solutions. Patterns assign roles to their participants, which define the functionality of the participants in the pattern context. However, a number of design patterns involve crosscutting concerns in the relationship between the pattern roles and participant classes in each instance of the pattern [15]. The implementation of the pattern roles often crosscuts several classes in a software system. Moreover, recent studies [11, 12, 15] have shown that object-oriented (OO) abstractions are not able to isolate these pattern-specific concerns and tend to lead to programs with poor modularity. In this context, it is important to systematically verify whether aspect-oriented approaches

A. Rashid and M. Aksit (Eds.): Transactions on AOSD I, LNCS 3880, pp. 36 – 74, 2006.

[22, 33] support improved modularization of the crosscutting concerns relative to the patterns.

To the best of our knowledge, Hannemann and Kiczales [15] have developed the only systematic study that explicitly investigated the use of aspect-oriented programming (AOP) to implement classical design patterns. They performed a preliminary study in which they developed and compared Java [20] and AspectJ [2] implementations of the GoF patterns. Their findings have shown that AspectJ implementations improve the modularity of most patterns. However, these improvements were based on some attributes that are not well known in software engineering, such as composability and (un)pluggability. This study has also not investigated the scalability of both object-oriented and aspect-oriented solutions. Moreover, this study was based only on a qualitative assessment and empirical data is missing. To solve this problem, this previous study should be replicated and supplemented by quantitative case studies in order to improve our knowledge body about the use of aspects for addressing the crosscutting property of design patterns.

This paper presents quantitative assessments of Java and AspectJ implementations for the 23 GoF patterns. Our study is based on well-known software engineering attributes such as separation of concerns, coupling, cohesion and size. We have found that most aspect-oriented solutions improved the separation of pattern-related concerns. In addition, we have found that:

(i) The use of AOP helped to improve the coupling and cohesion of some pattern implementations.
(ii) The "aspectization" of design patterns reduced the number of attributes of 10 patterns, and decreased the number of operations and respective parameters of 12 patterns.
(iii) Only four design patterns implemented in AspectJ have exhibited significant reuse.
(iv) The relationships between pattern roles and application-specific concerns are sometimes so intense that it seems not trivial to separate those roles in aspects.
(v) The use of coupling, cohesion and size measures was helpful to assist the detection of opportunities for aspect-oriented refactoring of design patterns.

We have also analyzed the influence of AspectJ solutions on inheritance coupling. In addition, we discuss the scalability of both aspect-oriented and object-oriented solutions, and the determination of a predictive model for the aspectization of design patterns. As each design pattern usually has different variants and is heterogeneously instantiated through distinct applications [9], we also present some discussions about the particularities of the AspectJ implementations of the patterns used in this study. This information is useful to any software engineer, specially those who wish to replicate our experiment. Finally, we summarize how the findings of our study confirm or contradict the claims presented in the Hannemann and Kiczales' work [15].

The remainder of this paper is organized as follows. Section 2 presents our study setting, while giving a brief description of Hannemann and Kiczales' study. Section 3 presents the study results with respect to separation of concerns, and Sect. 4 presents the study results in terms of coupling, cohesion and size attributes. These results are

interpreted and discussed in Sect. 5, in which a broader analysis is drawn. Section 6 introduces some related work. Section 7 includes some concluding remarks and directions for future work.

2 Study Setting

This section describes the configuration of our empirical study. As this study is directly related to Hannemann and Kiczales' work, the goals and conclusions of that study are presented in Sect. 2.1. Section 2.2 uses the Mediator pattern to illustrate the crosscutting property of some design patterns. Section 2.3 introduces the metrics used in the evaluation process, and Sect. 2.4 describes our assessment procedures.

2.1 Hannemann and Kiczales' Study

Several design patterns exhibit crosscutting concerns [15]. In this context, Hannemann and Kiczales (HK) have undertaken a study in which they have developed and compared Java [20] and AspectJ [2] implementations of the 23 GoF design patterns [9]. They claim that programming languages affect pattern implementation. Hence it is natural to explore the effect of aspect-oriented programming (AOP) techniques on the implementation of the GoF patterns. For each of the 23 GoF patterns, they developed a representative example that makes use of the pattern and implemented the example in both Java and AspectJ.

Design patterns assign roles to their participants; for example, the Mediator and Colleague roles are defined in the Mediator pattern. A number of GoF patterns involve crosscutting structures in the relationship between roles and classes in each instance of the pattern [15]. For instance, 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 operations in each Colleague in the pattern.

Two kinds of pattern roles are identified in the HK study, which are called *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 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, since a participant class playing this role usually has functionality not related to the pattern. These kinds of roles are used by the authors to analyze the crosscutting structure of design patterns.

In the HK study, the goal of the AspectJ implementations is to modularize the pattern roles. The authors have reported that modularity improvements were reached in 17 of the 23 cases, and 12 aspect-oriented pattern implementations resulted in improved reuse. The degree of improvement with AOP has varied according to each pattern implementation. The next section discusses these improvements and crosscutting pattern structures in terms of the Mediator pattern.

2.2 Example: The Mediator Pattern

The intent of the Mediator pattern is to define an object that encapsulates how a set of objects interact [9]. The Mediator pattern defines two roles, Mediator and Colleague, to their participant classes. The Mediator role has the responsibility for controlling and coordinating the interactions of a group of objects. The Colleague role represents the objects that need to communicate with each other. Hannemann and Kiczales [15] present a simple example of the Mediator pattern in the context of a Java Swing application. In such a system the Mediator pattern is used to manage the communication between two kinds of graphical user interfaces components. A `Label` class plays the Mediator role of the pattern, and a `Button` class plays the Colleague role.

Figure 1 depicts the class diagram of the OO implementation of the Mediator pattern. The interfaces `GUIMediator` and `GUIColleague` are defined to realize the roles of the Mediator pattern. Specific application classes must implement these interfaces based on the role that they need to play. In the example presented, the `Button` class implements the `GUIColleague` interface. The `Label` class implements the interface `GUIMediator` in order to manage the actions to be executed when buttons are clicked. Figure 1 also illustrates how the OO implementation of the Mediator pattern is spread across the code of the application classes. The shadowed attributes and methods represent code necessary to implement the Colleague role of the Mediator pattern in the application context.

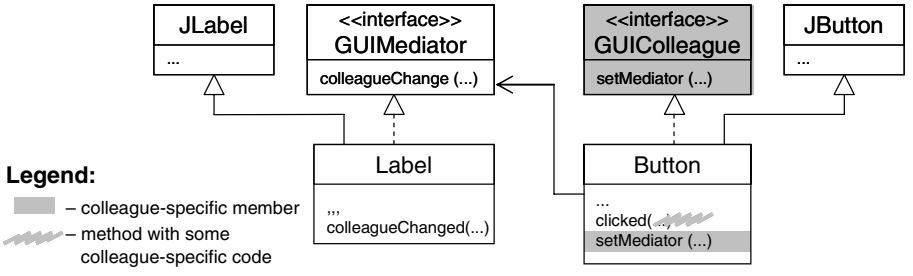


Fig. 1. The OO design of the mediator pattern

Figure 2 illustrates the source code of the `Button` class. The necessary elements to implement the Colleague role are shadowed. The `Button` class implements the `GUIColleague` interface (line 2), defines an attribute to reference a mediator (line 3), and implements the respective `setMediator()` method (lines 5–7). Moreover, the `clicked()` method of the `Button` class defines the functionality to communicate with the mediator (line 20).

In their study, Hannemann and Kiczales identified the generic part of several design patterns and isolated their implementation by defining “abstract reusable aspects”. These aspects are reused and extended in order to instantiate the pattern for a specific application. In the AspectJ solution of the Mediator pattern, for example,

the code for implementing the pattern is textually localized in two categories of aspects: (i) the `MediatorProtocol` abstract aspect that encapsulates the common part to all potential instantiations of the pattern, and (ii) concrete extensions of the abstract aspect that instantiate the pattern for specific contexts.

```
01 public class Button extends JButton
02                     implements GUIColleague {
03   private GUIMediator mediator;
04
05   public void setMediator(GUIMediator mediator) {
06     this.mediator = mediator;
07   }
08
09   public Button(String name) {
10     super(name);
11     this.setActionCommand(name);
12     this.addActionListener( new ActionListener() {
13       public void actionPerformed(ActionEvent e) {
14         clicked();
15       }
16     });
17   }
18
19   public void clicked() {
20     mediator.colleagueChanged(this);
21   }
22 }
```

Fig. 2. The Button class of the OO implementation

Figure 3 presents the reusable `MediatorProtocol` abstract aspect. Code related to the Colleague role is shadowed. Both roles are realized as protected inner interfaces named `Mediator` and `Colleague` (line 3 and line 7, respectively). Concrete extensions of the `MediatorProtocol` aspect assign the roles to particular classes. Implementation of the mapping from Colleague to Mediator is realized using a weak hash map that stores for each colleague its respective mediator (line 9). Changes to the Colleague–Mediator mapping can be realized via the public `setMediator()` method (lines 16–18). The `MediatorProtocol` aspect also defines an abstract pointcut named `change` and an abstract method named `notifyMediator()`. The former specifies points in the execution (joinpoints) of colleague objects where a communication with the mediator object needs to be established. The latter defines the functionality to be executed by a Mediator object when a change to a Colleague occurs. These abstract elements must be concretized by the `MediatorProtocol` subaspects. Finally, the communication protocol between Mediator and Colleague is implemented by an after advice (lines 22–24) in terms of the `change` pointcut and the `notifyMediator()` method.

As we can see, in the AspectJ implementation of the Mediator pattern, all code pertaining to the relationship between Mediators and Colleagues is moved into aspects. In this way, code for implementing the pattern is textually localized in aspects, instead of being spread across the participant classes. Moreover, the abstract aspect code can be reused by all pattern instances.

```
01 public abstract aspect MediatorProtocol {
02
03   protected interface Mediator { }
04
05   protected abstract void notifyMediator(Colleague c, Mediator m);
06
07   protected interface Colleague { }
08
09   private WeakHashMap mappingColleagueToMediator = new WeakHashMap();
10
11   private Mediator getMediator(Colleague c) {
12     Mediator mediator = (Mediator) mappingColleagueToMediator.get(c);
13     return mediator;
14   }
15
16   public void setMediator(Colleague c, Mediator m) {
17     mappingColleagueToMediator.put(c, m);
18   }
19
20   protected abstract pointcut change(Colleague c);
21
22   after(Colleague c): change(c) {
23     notifyMediator(c, getMediator(c));
24   }
25 }
```

Fig. 3. The MediatorProtocol aspect

2.3 The Metrics

In our study, a suite of metrics for separation of concerns, coupling, cohesion and size [29] was selected to evaluate Hannemann and Kiczales' pattern implementations. These metrics have already been used in five different studies [8, 10, 11, 19, 31], where the measures have been proved to be effective quality indicators. Most of them have been automated in our own measurement tool [7]. This metrics suite was defined based on the reuse and refinement of some classical and OO metrics [5, 6]. The original definitions of the OO metrics [5] were extended to be applied in a paradigm-independent way, thereby supporting the generation of comparable results. The metrics suite also encompasses new metrics for measuring separation of concerns [10, 29]. Table 1 presents a brief definition of each metric and associates them with the attributes measured by each one.

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 more directly a concern maps to the design and code elements, the fewer elements are affected by the concern, and the better modularized the system is. The suite is composed of three metrics for separation of concerns: (i) concern diffusion over components (CDC), (ii) concern diffusion over operations (CDO), and (iii) concern diffusion over lines of code (CDLOC).

In order to better understand these metrics, consider the OO example of the Mediator pattern, shown in Fig. 1 (Sect. 2.2). In that example, there is code relative to the Colleague role in the `GUIColleague` interface and in the shadowed methods of the `Button` class. In other words, the Colleague concern is implemented by one interface and one class. Therefore, the value of the CDC metric for this

concern is two. Similarly, the value of the CDO metric for the Colleague role is three, since this concern is implemented by the one method of the `GUIColleague` interface and the two shadowed methods of the `Button` class. Figure 2 shows the shadowing of the `Button` class in detail.

The CDLOC metric allows us to measure the number of transition points for each concern through the lines of code. A transition point is the place in the code where there is a "concern switch". CDLOC is measured by shadowing lines of code in the application classes related to the specific concern that you are interested in investigating. After that, it is necessary to count the number of transitions points through the source code of every shadowed class. In the example presented in Fig. 2, the `Button` class was shadowed in order to make it possible to measure the value of CDLOC for the Colleague concern. The value of CDLOC is four in that case, since that is the number of transition points through the source code of the `Button` class.

Table 1. The metrics suite

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 diffusion 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	Coupling between components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled
	Depth inheritance tree (DIT)	Counts how far down in the inheritance hierarchy a class or aspect is declared
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

Our suite also includes two metrics for assessing coupling from different viewpoints: coupling between components (CBC) and depth of inheritance tree (DIT). Coupling among system components has long been regarded as a major contributor to the system complexity. Coupling is an indication of the strength of interconnections between the components in a system. Highly coupled systems have strong

interconnections, with program units largely dependent on each other. Excessive coupling is not desirable, since it is detrimental to modular design. CBC is defined for a component (class or aspect) as a tally of the number of other components to which it is coupled. DIT is concerned with inheritance coupling. DIT is defined as the maximum length from a node to the root of the tree. It counts how far down the inheritance hierarchy a class or aspect is declared. DIT is an extension of the traditional OO metric [5] with the same name that also considers the inheritance between aspects [10, 29].

The suite of metrics encompasses one metric for cohesion, called lack of cohesion in operations (LCOO). This metric measures the lack of cohesion of a component by counting the amount of method/advice pairs that do not access the same instance variable [10, 29]. A low LCOO value indicates high closeness on the relationships between internal component operations (i.e., high cohesion), which is a desirable situation. On the other hand, low-cohesive components suggest an inappropriate design, because each of them involves the encapsulation of unrelated module entities, which should not be kept together in the same modular unit[3].

The software size measures the length of a software system's design and code [6]. Size metrics are concerned with different perspectives of the system size. The metrics suite encompasses three size metrics: (i) lines of code (LOC), (ii) number of attributes (NOA), and (iii) weighted operations per component (WOC). In general, the higher the size, the more complex the system is. LOC counts the lines of code in the system implementation, while NOA captures the number of attributes in each aspect or class. WOC measures are obtained by counting the number of parameters of the operation. The metric treats advice and methods of aspects in the same way that the corresponding OO metric [5] treats methods of classes.

2.4 Assessment Procedures

Replication of software engineering experiments is one of the main mechanisms to enable us to improve our understanding of existing techniques. In our study, we have used the same Java and AspectJ implementations of the HK study so that we could explicitly correlate our empirical results with the ones from this previous study. The AspectJ implementations basically followed the strategies described in [15], where abstract reusable aspects (Sect. 2.2) were defined when possible. It was not particularly feasible to define a reusable aspect for the patterns Abstract Factory, Factory Method, Template Method, Builder, and Bridge; aspects were used to isolate the pattern roles while providing support for multiple inheritance, which is not supported in Java. The Façade implementations are the same in AspectJ and Java.

As Hannemann and Kiczales have mostly chosen the default version of the patterns, no major decisions needed to be taken in the Java implementations of the patterns since the pattern implementations are already explicitly documented in the GoF book. This procedure was important to guarantee that the Java versions were good enough to enable fair comparisons with the AspectJ counterparts. The only major change done in both implementations of the patterns was that abstract classes

defined in the patterns were replaced with interfaces, as often happens in realistic applications. The idea is to allow the business classes to extend application-specific abstract classes in addition to the interfaces of the pattern. In few cases, they have chosen specific variants of the patterns in the Java implementations, but the design differences with respect to the main version of the pattern are also documented in the GoF catalogue. In addition, the AspectJ solutions implemented those same variants. The implementation of nondefault versions of the patterns only happened in two cases: the Singleton pattern (variant exploring specialization of singletons), and the Adapter pattern (variant called *Object Adapter* [9]). Refer to [1, 15] for further details about the design pattern implementations, and respective decisions and constraints.

In order to compare the two implementations of the patterns, we had to ensure that both versions of each pattern were implementing the same functionalities. Therefore, some minor modifications were realized in the original code [1] 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 pattern 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. Another kind of change was (ii) to ensure that both versions were using the same coding styles.

Afterwards, we changed both Java and AspectJ implementations of the 23 GoF patterns to add new participant classes to play pattern roles. For instance, in the Mediator pattern implementation, four classes playing the role of Colleague were added, as the `Button` class in Fig. 1 (Sect. 2.2); furthermore, four classes playing the role of Mediator were added, as the `Label` class in Fig. 1. These changes were introduced because the HK implementations encompass few classes per role (in most cases only one). Hence we have decided to add more participant classes in order to investigate the pattern crosscutting structure and the scalability of both OO and AO solutions. Table 2 presents the superimposed roles of each studied pattern and the participant classes introduced to each pattern implementation example. Finally, we have applied the chosen metrics to the changed code. We analyzed the results after the changes, comparing with the results gathered from the original code (i.e., before the changes).

In the measurement process, the data was partially gathered by the CASE tool Together 6.0 [34]. It supports some metrics: LOC, NOA, WOC (WMPC2 in Together), CBC (CBO in Together), LCOO (LOCOM1 in Together) and DIT (DOIH in Together). The data collection of the separation of concerns metrics (CDC, CDO and CDLOC) was preceded by the shadowing of every class, interface and aspect in both implementations of the patterns. Their code was shadowed according to the role of the pattern that they implement. Like the HK study, we treated each pattern role as a concern, because the roles are the primary sources of crosscutting structures. Figures 2 and 3 exemplify the shadowing of some classes and aspects in both Java and AspectJ implementations of the Mediator pattern by considering the Colleague role of this pattern. After the shadowing, the data of the separation of concerns metrics (CDC, CDO, and CDLOC) was manually collected.

Table 2. The design patterns, their superimposed roles and the respective changes

Design patterns	Superimposed roles	Introduced changes
Abstract Factory	–	4 Factories
Adapter	Adaptee	4 Adaptee methods
Bridge	–	2 Abstractions and 2 implementors
Builder	–	4 Builders
Chain of Responsibility (CoR)	Handler	4 Handlers
Command	Commanding, Receiver	4 Commands and 2 invokers
Composite	Composite, Leaf	2 Composites and 2 leafs
Decorator	Component	4 Decorators
Façade	–	No change
Factory Method	–	4 Creators
Flyweight	Flyweight	4 Flyweights
Interpreter	–	4 Expressions
Iterator	Aggregate	2 Iterators and 2 aggregates
Mediator	Mediator, Colleague	4 Mediators and 4 colleagues
Memento	Originator	2 Mementos and 2 originators
Observer	Subject, Observer	4 Observers and 4 subjects
Prototype	Prototype	4 Prototypes
Proxy	Proxy	4 Proxies and 2 real subjects
Singleton	Singleton	4 Singletons and 4 subclasses
State	Context	4 States
Strategy	Context	4 Strategies and 4 contexts
Template Method	AbstractClass, ConcreteClass	4 Concrete classes
Visitor	Element	4 Elements and 2 visitors

3 Results: Separation of Concerns

This section and Sect. 4 present the results of the measurement process. The data have been collected based on the set of defined metrics (Sect. 2.3). The goal is to describe the results through the application of the metrics before and after the selected changes (Sect. 2.4). The presentation of the measurement outcomes is broken into two parts. This section focuses on the analysis of to what extent the aspect-oriented (AO) and object-oriented (OO) solutions[1] provide support for the separation of pattern-related concerns. Section 4 presents the results with respect to coupling, cohesion and size. The discussion about the interplay among all the results is concentrated in Sect. 5. Section 5 also presents other relevant discussions, such as the relationships between our study's results and the conclusions obtained in the HK study.

Graphics are used to represent the data gathered in the measurement process. The resulting graphics present the gathered data *before* and *after* the changes applied to the pattern implementation (Sect. 2.4). 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

[1] From herein, we will use the terms "aspect-oriented solutions" and "object-oriented solutions" to refer to, respectively, the Aspect solutions and Java solutions.

an analysis of how the introduction of new classes and aspects affect both solutions with respect to the selected metrics. The results shown in the graphics were gathered according to the pattern point of view; that is, they represent the tally of metric values associated with all the classes and aspects for each pattern implementation.

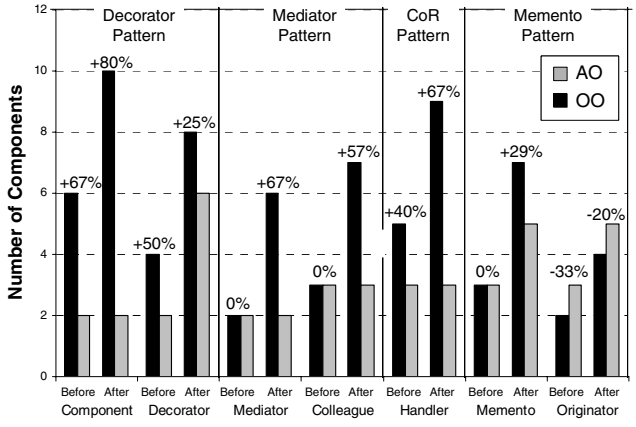
For separation of concerns, we have verified the separation of each role of the patterns on the basis of the three metrics defined for this purpose (Sect. 2.3). For example, the isolation of the Mediator and Colleague roles was analyzed in the implementations of the Mediator pattern, while the modularization of the Context and State roles was investigated in the implementations of the State pattern. According the data gathered, the investigated patterns can be classified into 3 groups. Group 1 represents the patterns that the aspect-oriented solution provided better results (Sect. 3.1). Group 2 represents the patterns in which the OO solutions have shown as superior (Sect. 3.2). Group 3 involves the patterns in which the use of aspects did not impact the results (Sect. 3.3).

3.1 Group 1: Increased Separation

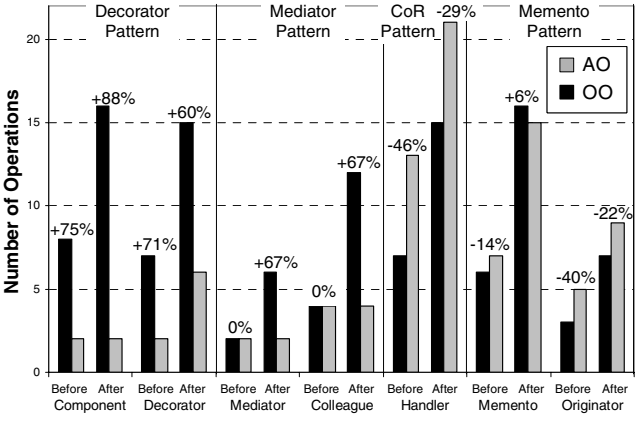
The first group encompasses all the patterns that aspect-oriented implementations exhibited better separation of concerns. This group includes the following list of 14 patterns: Decorator, Adapter, Prototype, Visitor, Proxy, Singleton, Mediator, Composite, Observer, Command, Iterator, CoR (Chain of Responsibility), Strategy and Memento. This list is decreasingly ordered by the measures for separation of concerns, starting from the design pattern that presents the best results for the aspect-oriented solution, the Decorator pattern.

Figures 4 and 5 depict the overall results for the AO and OO solutions based on the metrics. The figures only present a representative set of the patterns in this group. Note that the graphics present the measures before and after the execution of the changes. Figure 4a presents the CDC results, i.e., to what extent the pattern roles are isolated through the system components in both solutions. Figure 4b presents the CDO results, the degree of separation of the pattern roles through the system operations. Figure 5 illustrates the CDLOC measures – the tally of concern switches (transition points) through the lines of code.

Most of these graphics show significant differences in favor of the aspect-based solutions. These solutions require fewer components and operations than OO solutions to express these concerns. In addition, they require fewer switches between role concerns, and between role concerns and application concerns. An analysis of Figs. 4 and 5 show that the best improvements come primarily from isolating the pattern roles into the aspects. For example, the definition of the Component role required eight classes, while only two modular units were necessary to encapsulate this concern before the changes (Fig. 4a). It is equivalent to 67% in favor of the AO design for the Decorator pattern. In fact, most superimposed roles were better modularized in the AO solution, such as Mediator (8 against 2), Colleague (7 against 3), and Handler (9 against 3). The results were similar when analyzing separation of concerns over operations (Fig. 4b) and lines of code (Fig. 5). In addition, we can also observe that good results are achieved on the modularization of some defining roles, such as Decorator.



(a) Concern diffusion over components



(b) Concern diffusion over operations

Fig. 4. Separation of concerns over components and operations (Group 1)

After a careful analysis of Figs. 4 and 5, we come to the conclusion that after the changes most AOP implementations isolated the roles 25% or higher than the OO implementations. There are some cases where the difference is even more striking — the superiority of AOP exceeds 70%. In some cases, such as the Colleague role, the AO solution is even better before the incorporation of new components. This problem happens in the OO solution because several operation implementations are intermingled with role-specific code. For example, the code associated with the control and coordination of the interobject interactions (Mediator pattern – Sect. 2.2) is amalgamated with the basic functionality of the application classes. It increases the number of transition points and the number of components and operations that deal with pattern-specific concerns.

The results also show that the overall performance of the AO solutions gradually improves as new components are introduced into the system. It means that as more components are included into an OO system, more role-related code is replicated through the system components. Thus a gradual improvement takes place in the AO solutions of the patterns. The series of small introduced changes (Sect. 2.4) affects negatively the performance of the OO solution and positively the AO solution. The changes lead to the degradation of the OO modularization of the pattern-related concerns. This observation provides evidence of the effectiveness of AO abstractions for segregating crosscutting structures for the patterns in this group.

Among the list of 14 patterns mentioned above, the first six are the patterns that achieved the best results: Decorator, Adapter, Prototype, Visitor, Proxy and Singleton. These patterns have several similar characteristics. They presented superior results for the AO solution both before and after the introduced changes. This means that the AO implementations of these patterns are superior even in simple pattern instances, i.e., circumstances where there are few application classes playing the pattern roles. In fact, the role-specific concerns are easier to separate in these patterns because the AspectJ constructs directly simplify the implementation of most of these patterns, namely Decorator, Adapter, Visitor and Proxy. As a result, the implementation of these patterns completely disappears [15], requiring fewer classes and operations to address the isolation of the roles. All these six patterns have another common characteristic: they either involve no reusable aspect (Decorator and Adapter) or involve very simple reusable aspects (Prototype, Visitor, Proxy, Singleton).

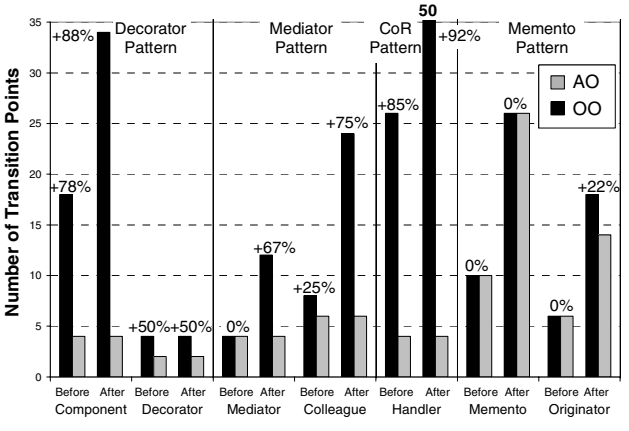


Fig. 5. Concern diffusion over LOC (Group 1)

The Decorator pattern is the representative of this kind of patterns in Figs. 4 and 5. Note that the AO solution for this pattern exhibits meaningful advantages on the modularization of both roles from all the perspectives: numbers of components (CDC), operations (CDO) and transition points (CDLOC). One additional observation is that these numbers remain unaltered as the change scenarios are applied to the AO implementation. For example, the absolute number of operations and components for specifying the Component role is the same before and after the scenarios in the AO design. The changes do not affect the measures. It demonstrates how well the AO

abstractions localize these pattern roles. In addition, after the scenarios are applied, the absolute difference on the measures between AO and OO implementations tends to be higher in favor of the AO solutions than before the change scenarios.

The following five patterns in Group 1 – Mediator, Composite, Observer, Command and Iterator – expressed similar results. They manifested improved separation of concerns only after the introduced changes. In general, the use of aspects led to inferior or equivalent results before the application of the changes, but led to substantially superior outcomes after the changes. It happens because the AO implementations of these patterns involve generic aspects that are richer; they encapsulate more operations and LOC than the simple reusable aspects defined for the four patterns mentioned before in this group. In this way, the benefit of improved locality is observed in the AO solutions of these patterns only when complex instances of the patterns are used. The more pattern code can be captured in a reusable aspect, the less has to be duplicated in the participant classes.

The Mediator pattern represents these five patterns in Figs. 4 and 5. Note that after the changes, the isolation of the Mediator and Colleague roles with aspects was 60% higher than the OO solution for all the metrics. This is an interesting fact given that in these cases the values were equivalent in both OO and AO solutions before the implementation of the changes. The definition of the Colleague role required 12 classes, while only four aspects were able to encapsulate this concern. This result was similar in the other four patterns, i.e., absolute number of components (CDC) did not vary after the modifications in the AO solutions. This reflects the suitability of aspects for the complete separation of the roles associated with the five patterns. When new classes are introduced, they do not need to implement pattern-related code.

Finally, there were three AO solutions in this group (CoR, Strategy, and Memento) that, although provided overall improvements in the isolation of the roles, presented some negative results in terms of a specific measure. Figures 4 and 5 illustrate two examples: CoR and Memento. The AO implementation of CoR has fewer components (Fig. 4a) and transition points (Fig. 5) both before and after the changes. However, it has more operations involved in the implementation of the pattern role (Fig. 4b). The AO solution of Memento isolates well the Memento role for most the metrics (CDC and CDO). However, although the implementation of the Originator role with aspects led to fewer transition points (Fig. 5), the same observation does not happen to number of operations and components (Fig. 4).

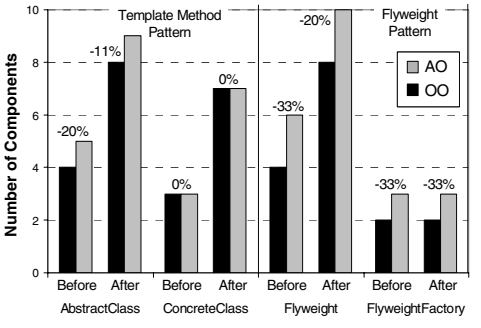
3.2 Group 2: Decreased Separation

The second group includes design patterns in which AO implementations exhibited decreased separation of concerns. This group includes six patterns, namely Template Method, Abstract Factory, Factory Method, Bridge, Builder and Flyweight. In fact, the AspectJ implementations of the first five are mainly meant to explore AOP as an alternative solution to multiple inheritance, replacing abstract classes with interfaces and thereby increasing implementation flexibility [15]. Figure 6 depicts the CDC, CDO and CDLOC measures of separation of concerns for the pattern implementations in this group.

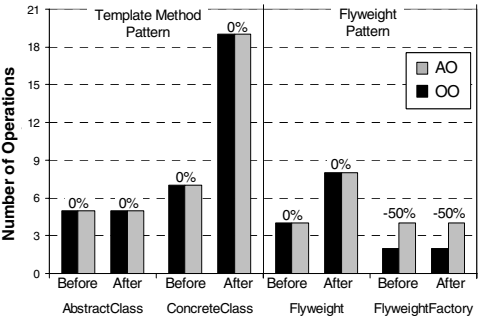
Although some measures presented similar results for the OO and AO solutions of these patterns, several measures presented differences in favor of OO

implementations. As the pattern roles are already nicely realized in OO, these patterns could not be given more modularized aspect-oriented implementations. Thus the use of aspects does not bring apparent gains to these pattern implementations regarding to separation of concerns. On the contrary, the OO implementations, in general, provided better results, mainly with respect to the CDC measures (Fig. 6a).

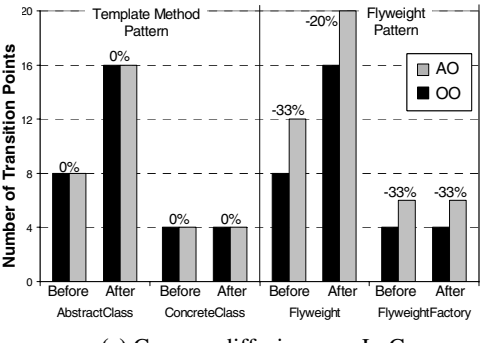
The main reason for this result is that all the patterns in this group, except the Flyweight, are structurally similar: they have an additional aspect to replace the



(a) Concern diffusion over components



(b) Concern diffusion over operations



(c) Concern diffusion over LoC

Fig. 6. Separation of concerns (Group 2)

abstract class mentioned in the GoF solution by interfaces without losing the ability to associated (default) implementations to their methods [15]. For example, the Template Method pattern has an additional aspect that attaches the template method and its implementation to a component that plays the AbstractClass role, thereby allowing it to be an interface. Although this kind of aspects makes the patterns more flexible, it does not improve the separation of the pattern-specific concerns.

The Flyweight pattern is an exception in this group. The OO design provided better results than the AO design for all the measures. The superiority of the OO solution reaches 33% for most of the measures. It happens because the AO solution does not help to separate a crosscutting structure relative to the pattern roles. In fact, the classes playing the Flyweight role are similar in both implementations. The aspects have no pointcuts and advices, and the generic `FlyweightProtocol` aspect could be implemented as a simpler class. As a result, the additional components and operations introduced by the AO solution decreases the separation of concerns since the roles implementation are scattered over more design elements.

3.3 Group 3: No Effect

This group includes three patterns: Façade, Interpreter, and State. Overall, no significant difference was detected in favor of a specific solution; the results were mostly similar for the AO and OO implementations of these patterns. The AO and OO implementations of the Façade pattern are identical. There were some minor differences, as in the State pattern, but they were irrelevant (less than 5%). The outcomes of this group were highly different from the ones obtained in Group 1 (Sect. 3.1) because the OO implementations of the patterns do not exhibit significant crosscutting structures. The role-related code in these patterns affects a very small number of methods.

4 Results: Coupling, Cohesion and Size

This section presents the coupling, cohesion and size measures. We used graphics to present the data obtained before and after the systematic changes (Sect. 2.4), similarly to the previous section. The results represent the tally of metric values associated with all the classes and aspects for each pattern implementation, except the DIT metric. The DIT results represent the maximum value of this metric through the whole pattern implementation. In other words, it represents the higher inheritance depth achieved in a given AspectJ or Java implementation. The patterns were classified into five groups according to the similarity in their measures.

4.1 Group 1: Better Results for AO

The first group includes the Composite, Observer, Adapter, Mediator and Visitor patterns, which presented meaningful improvements with respect to the attributes coupling, cohesion and size in the AO solution. In some cases, the improvement was higher than 50%. Figure 7 shows the graphics with results for the Mediator and Visitor patterns, which represent this group.

In the AO implementation of the Mediator pattern, the major improvements were achieved in the CBC, LCOO, NOA and WOC measures. The use of aspects led to a 17% reduction of CBC in relation to the OO design. This occurs because the Colleague classes are unaware of the Mediator class in the AO design (Sect. 2.2), while in the OO implementation each Colleague holds a reference to the Mediator. Thus, all the Colleague classes are coupled to the Mediator class. In the same way, the AO implementation of the Visitor pattern led to a 32% reduction after the changes. The reason is that the Visitor classes are coupled to all the Element classes in the OO implementation. These couplings are not necessary in the AO solution.

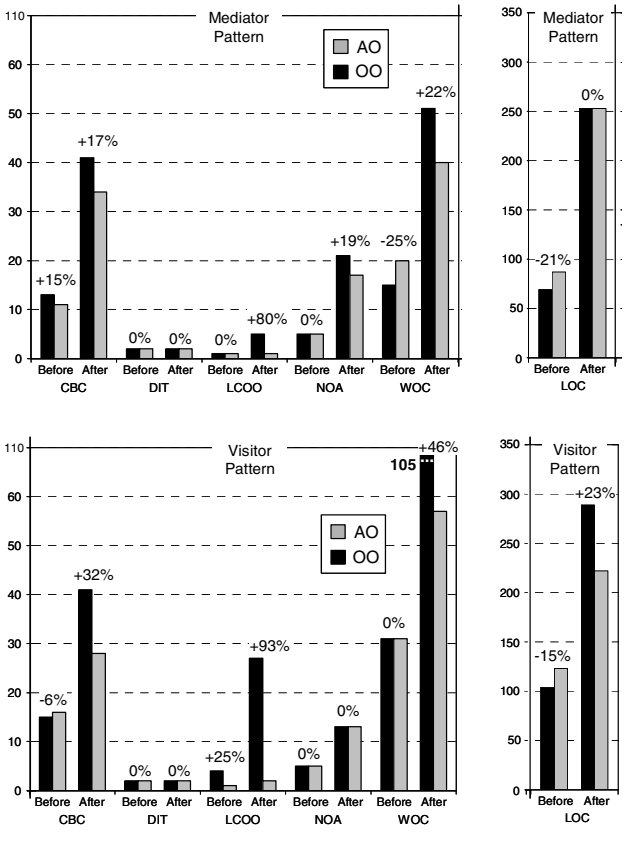


Fig. 7. The Mediator and Visitor patterns: coupling, cohesion and size (Group 1)

Note that inheritance was not affected by the use of aspects. The OO solution of the Mediator pattern used the interface implementation to define the Colleague and Mediator participants. The AO solution is based on specialization to define a concrete Mediator protocol (Sect. 2.2). As a result, the DIT was two for both solutions.

The AO solution was superior to the OO solution in terms of cohesion. The cohesion in the AO implementation was 80% higher than in the OO implementation because the Colleague and Mediator classes in the OO solution implement role-specific methods, which, in turn, are not related to the main functionality of the classes. An example is the `setMediator()` method, which is part of the Colleague role and is responsible for setting the Mediator reference (see Fig. 1). The AO design localizes these methods in the aspects that implement the roles, increasing the cohesion of both classes and aspects. Likewise, the OO solution of the Visitor pattern has a method defined in the Element classes to accept the Visitor objects. This method is not related to the main functionality of the Element classes and, therefore, does not access any attribute of these classes. In the AO solution, this method is moved to the aspect. Consequently, the cohesion of the Element classes in the OO implementation is inferior to the classes in the AO solution.

The number of attributes and weight of operations in the OO implementation of the Mediator pattern were, respectively, 19% and 22% higher than in the AO code after the introduction of new components. In the OO solution, each Colleague class needs both an attribute to hold the reference to its Mediator and a method to set this reference. These elements are not required in the Colleague classes of the aspect-oriented solution, because only the aspect controls the relationship between Colleagues and Mediators. A similar benefit was reached in the AO implementation of the other patterns in this group.

The coupling, cohesion and size improvements in the aspect-oriented solutions of the patterns in this group are directly related to the achieved separation of concerns for them (Sect. 3.1). The enhanced isolation of the pattern implementations directly contributed to (i) reduce the number of LOC, operations and attributes; (ii) improve the module cohesion by disentangling pattern-related concerns; and (iii) achieve reduced coupling (Fig. 7). For instance, as previously explained in this section, the coupling, cohesion and size of the Mediator pattern are improved because the pattern roles are better isolated in aspects and not spread over several classes. A similar result occurs in the other four patterns. For instance, in the Visitor pattern, the AO implementation solves the problem of code replication related to the implementation of the method that accepts the Visitor classes in every Element class. Hence after the changes the OO implementation had 23% more LOCs, and an inferior coupling in 46% (Fig. 7).

4.2 Group 2: Better Results for AO in Most Measures

This group encompasses the patterns in which AO solutions produced better results in most of the measures except in one metric. This group includes the Decorator, Proxy, Singleton and State patterns. The measures gathered from implementations of the Decorator, Proxy, Singleton were mostly similar. The AO implementation of these patterns showed improvements related to all metrics except the CBC metric. On the other hand, the AO solution of the State pattern did not show improvements only in the number of attributes. Figure 8 presents the results of the Decorator and State patterns as representative of this group.

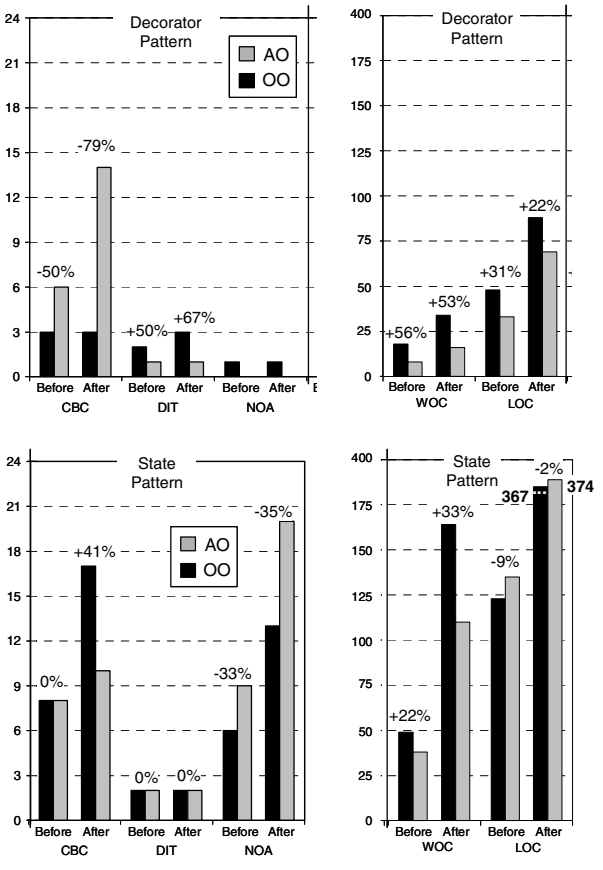


Fig. 8. The Decorator and State patterns: coupling and size (Group 2)

The AO implementations of the Decorator, Singleton and Proxy patterns manifest similar benefits to the patterns of Group 1 (Sect. 4.1). That is, the improvement in the separation of the pattern-specific code (Sect. 3.1) conducted to improvements in other attributes, such as, cohesion and size. However, as shown in Fig. 8 for the Decorator pattern, the CBC measures were inferior in the AO implementation: 50% and 79% before and after the changes, respectively. This problem occurs in the Decorator pattern because one of the Decorator aspects has to declare the precedence among all the Decorator aspects. Therefore, it is coupled to all the other aspects. In the Singleton pattern, there is an additional aspect per Singleton class. The coupling between the aspects and the Singleton classes increased the results of the CBC metric.

The measures concerning the State pattern provided peculiar results. Despite showing no improvements related to the separation of concerns metrics (Sect. 3.3), the AO implementation of the State pattern was superior in coupling, cohesion and weight of operations (Fig. 8). On the other hand, the OO implementation provided better results in two measures: NOA and LOC. The coupling in the OO solution is higher than in the AO solution because the classes representing the states are highly

coupled to each other. This problem is overcome by the AO solution because the aspects modularize the state transitions (Fig. 9), minimizing the coupling between the pattern participants. Figure 9 shows that the coupling in the OO solution is 7 because each State class needs to have references to the other State classes.

It is important to highlight that the definition of the State pattern [9] does not specify which pattern participant defines the criteria for state transitions. In this way, it is possible to isolate the state transitions even in the Java solution by moving them from the "state" classes to the "context" class (when the criteria are fixed). However, even though it is possible to isolate the transitions in the "context object", the transitions can be, in several cases, more naturally implemented in the state classes due to a number of conditions/constraints specific to the state classes. The AspectJ solution supports an improved modularization of the state transitions in this second case.

With respect to WOC measures, the OO solution produced more complex operations because all the methods on the State classes have an additional parameter to receive the Context object in order to implement the state transition. It is not required in the AO design because a central aspect is responsible for managing the transitions between states.

From the NOA point of view, the OO design was superior because the AO design has additional attributes in the aspects to hold references to the State elements. This difference increases as new State elements are added to the system (Fig. 8). In spite of the fact that the State classes in the AO implementation have fewer lines of code, the OO implementation as a whole provided fewer LOCs. This occurs because the aspect, which manages the state transitions, has a high number of LOCs since: (i) it holds references to all the State classes, and (ii) it has one additional advice associated with methods of State classes.

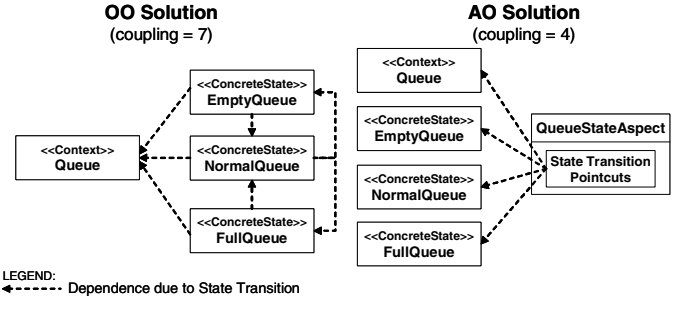


Fig. 9. Coupling in the state pattern: OO vs. AO

4.3 Group 3: Better Results for OO in Most Measures

This group includes the CoR, Command, Prototype and Strategy patterns. The measures gathered from the implementations of these patterns were similar in the sense that, in general, the OO implementations provided better or similar results. The AO solutions improved the results for only one size metric. The AO implementation

of the CoR, Command and Strategy patterns required fewer attributes than the OO implementation (NOA metric), while the AO solution of the Prototype pattern involved fewer operations (WOC metric).

The CoR pattern is the representative element of this group. Figure 10 shows the results for this pattern. Note that the OO implementation had 75% more attributes than the AO implementation after the inclusion of new Handler classes. Nevertheless, the AO implementation showed inferior results concerning lines of code and weight of operations. Moreover, there was insignificant difference between the two solutions in terms of the coupling metrics (CBC and DIT).

As shown in Sect. 3.1, these patterns benefit from the AO implementation in terms of separation of concerns. However, those benefits were not sufficient to improve most of the other quality attributes. For instance, the OO implementation of the CoR pattern requires the incorporation of an attribute to hold a reference to its successor in the Handler class. In the AO implementation, the chain of successors is localized in an aspect, removing the successor attribute from the Handler classes. As a consequence, the number of attributes was lower in the AO implementation. However, the amount of additional operations required in the aspect to handle the chain of successors negatively affected the LOC and WOC measures. Furthermore, due to the coupling between the aspect and all the Handler classes, the AO solution did not provided significant improvements (CBC metric). This phenomenon also happened in the other patterns of this group. For instance, in the AO implementation of the Prototype pattern, the methods to clone the Prototype classes were localized in an aspect and not replicated in all the Prototype classes. However, this design choice was only sufficient to reduce the weight of operations (WOC metric).

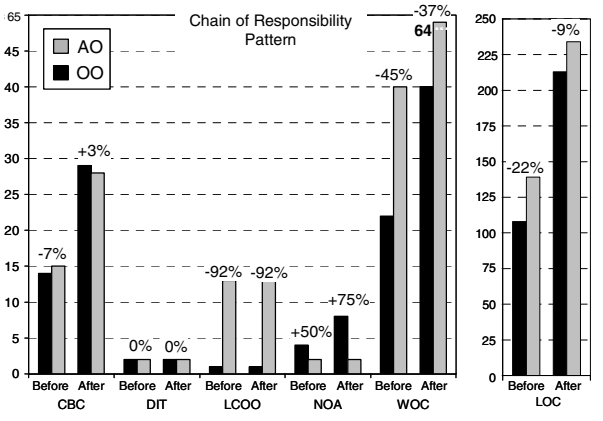


Fig. 10. The Chain of Responsibility pattern: coupling, cohesion and size (Group 3)

4.4 Group 4: Better Results for OO

The fourth group comprises the patterns that the AO implementation provided worse results related to coupling, cohesion, and size. This group includes the following list

of eight patterns: Template Method, Abstract Factory, Bridge, Interpreter, Factory Method, Builder, Memento and Flyweight. The Template Method and Memento patterns represent this group in Fig. 11.

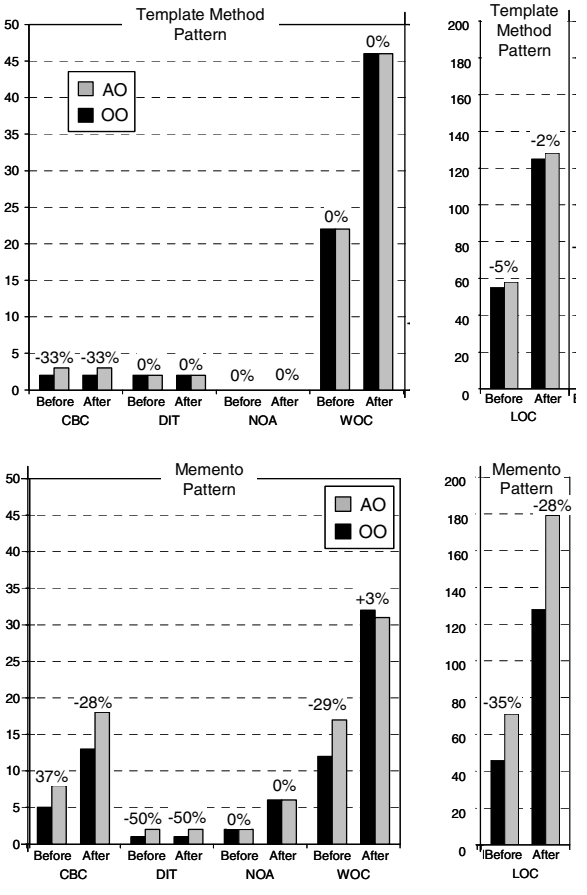


Fig. 11. The Template Method and Memento patterns: coupling and size (Group 4)

The measures of the Template Method, Abstract Factory, Bridge, Interpreter, Factory Method and Builder patterns exhibited minor differences in favor of the OO implementation. In fact, we have already mentioned in Sect. 3.2 that these patterns are already nicely realized in OO, and thus could not be given more modularized AO implementations. The AO implementation of the Template Method, for instance, showed higher coupling (33%) and more lines of code (5%) than the OO implementation. The other measures produced equal results for both solutions (see Fig. 11). This minor difference is due to the additional aspect which associates (default) implementation to the methods in the interface that plays the AbstractClass role.

The measures of the Flyweight and Memento patterns showed better results for the OO implementation. The AO implementation of the Flyweight pattern showed worse results mainly with respect to coupling. It is because an aspect is coupled to all Flyweight classes in order to introduce the Flyweight interface in them by means of the intertype declaration mechanism. The AO implementation of the Memento pattern showed the worst results when compared with the other AspectJ pattern implementations in this group. Removing the pattern-related code from the Originator classes and placing it in an aspect makes the design more complex. This is shown by the results of the CBC, DIT, WOC and LOC metric (see Fig. 11).

4.5 Group 5: No Effect

This group includes the Iterator and Façade patterns. The measures related to these patterns exhibited no significant difference in favor of a specific solution. The AO and OO implementations of the Façade pattern are essentially the same. In the AO implementation of the Iterator pattern, the method which returns a reverse iterator is removed from the Aggregate classes. These methods are localized in an aspect. However, the number of methods was not reduced since it was still necessary one method per Aggregate class. Therefore, in spite of showing better separation of concerns (Sect. 3.1), the AO implementation provided insignificant improvements in terms of coupling, cohesion and size.

5 Discussions

Empirical studies are the most effective way to supply evidence that may improve our understanding about software engineering phenomena [4, 23]. Although quantitative studies have some disadvantages [23], they are very useful because they boil a complex situation down to simple numbers that are easier to grasp and discuss. They supplement qualitative studies with empirical data. Quantitative studies investigating the implementation of design patterns as aspects are rare [15]. Most of the claims are supported by experience reports of practitioners, but there is a lack of quantitative research providing empirical evidence in favor of the claimed benefits. This section provides a more general analysis (Sect. 5.1) of the previously observed results in Sects. 3 and 4, some analysis of specific design patterns (Sect. 5.2), and discussions about the constraints on the validity of our empirical evaluation as well as lessons learned (Sect. 5.3).

5.1 General Analysis

This section presents an overall analysis of the results observed on the application of metrics for separation of concerns, coupling, cohesion and size. The general analysis also covers discussions on: the scalability of the pattern implementations (Sect. 5.1.2), the effects of the design pattern aspectization on different coupling dimensions (Sect. 5.1.4), reusability issues (Sect. 5.1.6), the interplay between these measures and a predictive model (Sect. 5.1.7), a comparative summary between this study's findings and the HK study's claims (Sect. 5.1.8), and the need for multidimensional assessments (Sect. 5.1.9).

5.1.1 Separable and Inseparable Concerns

Table 3 summarizes the findings on separation of concerns for each design pattern. This table complements the graphics presented in Sect. 3, which only shows the results for some representative patterns. The first three columns bring the gathered data for both AO and OO solutions with respect to all the three measures of separation of concerns: concern diffusion over components (CDC), concern diffusion over operations (CDO), and concern diffusion over lines of code (CDLOC). Table 3 focuses on the measures obtained after the changes (Sect. 2.4) introduced to the pattern implementations.

An additional goal of Table 3 is to provide a different perspective on the results obtained for separation of concerns. While the graphics in Sect. 3 show the measures in terms of each pattern role, Table 3 presents the values associated with the whole design pattern, i.e., the value shown in each cell represents the tally of the measures for all the roles of a design pattern. For example, consider the Mediator pattern: the graphic in Fig. 4a shows that, after the changes, the CDC measure for the Mediator role was 6 in the OO version against 2 of the AO version, and for the Colleague role was 7 in the OO version against 3 of the AO version. As a result, considering the two roles of the pattern, the final result indicates that the AO solution was superior – 5 against 13 of the OO solution, as illustrated in Table 3. This different perspective shows how the Java and AspectJ solutions were effective or not to modularize the pattern as a whole. It is worth recalling here that a higher value means that the implementation approach was inferior to modularize the pattern roles.

The last two columns of Table 3 are respectively concerned with the scalability criterion and with the indication of which implementation was superior. The scalability issue will be discussed in the next section. With respect to the last column, we have classified an AspectJ or Java solution as superior when it has achieved better results for most the measures when compared with the results of the other solution. The AspectJ solutions that achieved the best results, as discussed in Sect. 3.1, are marked with the symbol "+". The AspectJ implementations for these patterns were superior both before and after the introduced changes.

Table 3 shows that AspectJ implementations of 14 patterns have shown better results in terms of all the metrics for separation of concerns. In addition, the Java implementation of six patterns presented superior separation of roles (Sect. 3.2), and three patterns presented similar results in both implementations (Sect. 3.3). This observation provides evidence of the superior effectiveness of AO abstractions for segregating crosscutting structures relative to design patterns. Indeed, most of these results have confirmed the observations in the HK study in terms of the locality property.

However, the HK study also claimed that three additional patterns offered locality improvements in the respective AO implementations: Flyweight, State and Template Method. Our study's results somewhat contradict these claims (Table 3). The solution of patterns in Group 2 (Sect. 3.2), like Template Method, sounds to be natural in the OO fashion, and it does not seem reasonable or even possible to isolate the pattern roles into aspects. In fact, the AO solution of the Template Method is not aimed at improving the separation of the pattern roles, but increasing the pattern flexibility [15] (Sect. 3.2). The AO implementation of the Flyweight pattern is similar to the OO implementation with additional aspects that do not assist in the isolation of

crosscutting pattern-specific concerns (Sect. 3.2). The separation of concerns in the AO version of the State pattern helps to separate state transitions, but the differences in the measures are not significant (Sect. 3.3).

Table 3. Overall results for separation of concerns

	CDC		CDO		CDLOC		Scalability		Superior
Design pattern	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**solution**
Abstract Factory	14	16	35	35	34	34	No	No	OO
Adapter#	8	7	30	22	32	16	No	Yes	AO+
Bridge	12	13	24	26	16	16	No	No	OO
Builder	9	10	29	30	8	8	Yes	Yes	OO
CoR#	9	3	15	21	50	4	No	Yes	AO
Command#	17	11	23	16	38	21	No	Yes	AO
Composite#	18	9	149	28	70	48	No	No	AO
Decorator#	18	8	31	8	38	6	No	Yes	AO+
Façade	Same implementations for Java and AspectJ								
Factory Method	14	16	23	23	18	18	No	No	OO
Flyweight#	10	13	10	12	20	26	No	No	OO
Interpreter	13	13	26	26	38	38	No	No	=
Iterator#	10	6	20	20	18	14	No	No	AO
Mediator#	13	5	18	6	36	10	No	Yes	AO
Memento#	11	10	23	24	44	40	No	No	AO
Observer#	14	9	49	9	92	20	No	Yes	AO
Prototype#	7	3	7	2	30	8	No	Yes	AO+
Proxy#	11	11	38	19	8	2	No	Yes	AO+
Singleton#	6	6	6	1	6	2	Yes	Yes	AO+
State#	10	10	78	78	30	30	No	No	=
Strategy#	14	12	20	17	18	16	No	No	AO
Template Method#	15	16	24	24	20	20	No	No	OO
Visitor#	20	9	50	23	34	14	No	Yes	AO+
Success total	**6 vs. 12**		**5 vs. 11**		**1 vs. 14**		**2 vs. 11**		**6 vs. 14**

The design pattern contains one or two superimposed roles.
+ The AO solutions that achieved the best results.

An additional interesting observation in our study is that sometimes the pattern roles are expressed separately as aspects, but it remains nontrivial to specify how these separate aspects should be composed with the application classes into a simple manner. A lot of effort is required to compose the participant classes and the aspects that modularize the pattern roles. For example, the AO design of the Memento pattern provided better separation of the pattern-related concerns (Sect. 3.1). However, although the AO solution isolates the pattern roles in the aspects, it resulted in higher complexity in terms of coupling (CBC), inheritance (DIT) and lines of code (LOC), as described in Sect. 4.4. The same observation can be made for the Strategy and CoR patterns (Sect. 4.3). Hence, there are some cases where the separation of the pattern-related concerns leads to more complex design solutions.

The last line of Table 3 also counts how many patterns each solution was superior with respect to each metric (3 first cells), and in general terms (last cell). These values show that around 50% of the AO solutions have not shown improvements in terms of the CDO metric. In these cases, either the OO implementation required fewer operations to handle the pattern-related concerns than the AO implementation or they were similar. An analogous situation occurred in the CDC measures. The superiority of the AO solutions seems to be more compelling in the CDLOC measures: 14 against 1. The frequency of concern switches in the AspectJ implementations was drastically reduced. It means that there is a tendency on several AspectJ implementations to not reduce the number of operations implementing a concern. In general, it seems that the most recurring benefits come from *disentangling* the pattern-related concerns and other application concerns.

5.1.2 Scalability

As explained in Sect. 2.4, we changed both original Java and AspectJ implementations of the 23 patterns to investigate the scalability of those solutions to more complex instances of the patterns. In the context of this study, scalability is used to determine whether the introduction of the changes (described in Table 2) in a given implementation did not require modifying more components in that implementation than the number of elements introduced. In other words, we considered here a solution as scalable if the evolution of the implementation did not impact a number of modules that is higher than the number of modules being introduced.

We have used the CDLOC metric as the main mechanism to assess the scalability of the OO and AO versions. For example, Fig. 5 shows that the total number of concern switches for the implementation of the Mediator pattern, considering both roles before the changes, is 12 in the OO version and 10 in the AO version. After the changes, the number of switches remains 10 in the AO solution. However, it grows to 36 in the OO version, which is higher than the number of introduced changes (8 changes – i.e., 4 mediators and 4 colleagues). As a result, Table 3 indicates that the OO solution is not scalable, while the AO solution is considered scalable. In fact, the evolution of the AspectJ version occurred in a modular manner. All the separation of concerns measures, not only CDLOC, remained unaltered as the change scenarios were applied to the implementation, as illustrated in Figs. 4a, 4b and 5. The changes did not affect the measures. We have drawn a similar conclusion for the AO implementation of the Decorator pattern in Sect. 3.1; it is also ranked as scalable in opposite to the corresponding OO version.

Table 3 summarizes the scalability results for all the OO and AO solutions. Some AO solutions that were classified as superior did not achieve a good scalability. For the 14 AspectJ solutions that were considered as superior, 11 implementations were also classified as scalable. Only two Java solutions, Builder and Singleton, were effectively scalable with respect to the CDLOC measures. Although the AO solutions of the Composite, Iterator and Memento presented a better separation of the pattern roles than the respective OO solutions, they are not very scalable since they also require reasonable efforts to support the separation of the pattern roles. For instance, Fig. 5 illustrates this scalability problem for the Memento pattern. The CDLOC measures show that a number of extra changes were also required in the AspectJ version. A similar problem was detected for the Iterator and Composite. We do not

extensively reproduce all the detailed measurements here. The complete description of the data gathered is available at [28].

5.1.3 Reducing Coupling and Increasing Cohesion

Table 4 summarizes the conclusions related to coupling and cohesion for each design pattern. Like Table 3, it complements the graphics presented in Sect. 4, which shows only partial results. The first two columns respectively describe the results with respect to intercomponent coupling (CBC) and inheritance-related coupling (DIT) for both AO and OO solutions. The third column presents the gathered data for the cohesion metric (LCOO). Table 4 also concentrates on the description of the measures obtained after the changes.

Table 4. Overall results for coupling and cohesion

	CBC		DIT		LCOO		Superior
Design pattern	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**solution**
Abstract Factory	37	44	7	7	1	1	OO
Adapter#	5	5	2	1	–	–	AO
Bridge	17	18	2	2	0	0	OO
Builder	2	3	2	2	12	6	OO
CoR#	29	28	2	2	1	13	OO
Command#	21	34	7	7	3	4	OO
Composite#	47	23	2	2	463	82	AO
Decorator#	3	14	3	1	0	0	AO
Façade	Same implementations for Java and AspectJ						
Factory Method	22	24	2	2	3	0	OO
Flyweight#	11	17	2	2	0	1	OO
Interpreter	17	23	5	5	0	0	OO
Iterator#	12	13	2	2	0	0	=
Mediator#	41	34	2	2	5	1	AO
Memento#	13	18	1	2	0	0	OO
Observer#	45	40	2	2	80	30	AO
Prototype#	7	13	2	2	0	0	OO
Proxy#	11	39	2	2	0	0	AO
Singleton#	11	22	2	2	5	0	AO
State#	17	10	2	2	106	93	AO
Strategy#	18	32	2	2	–	–	OO
Template Method#	2	3	2	2	–	–	OO
Visitor#	41	28	2	2	27	2	AO
Success total	**15 vs. 6**		**1 vs. 2**		**3 vs. 8**		**12 vs. 9**

The design pattern contains one or two superimposed roles.

It is interesting to observe that the intercomponent coupling was weaker in 15 Java solutions against 6 AspectJ implementations. The DIT values were similar for both versions in most the measures. With respect to the cohesion metric, the AspectJ solutions achieved a better score: eight implementations were more cohesive against

only three Java implementations. As indicated in Table 4, it was not possible to measure the cohesion of a few solutions because either there was no attribute defined in those implementations or there were modules with a single method. As explained in Sect. 2.3, our selected cohesion metric captures the closeness between internal methods by checking accesses to the same attributes. Considering all the coupling and cohesion measures, only five AspectJ solutions clearly presented weaker coupling and stronger cohesion, namely Mediator, Observer, State, Visitor, and Composite.

Finally, based on Tables 3 and 4 and on the interplay of the results in Sects. 3 and 4, we can conclude that the use of aspects provided better coupling and cohesion results for the patterns with high interaction between the roles in their original definition. In fact, the Mediator, Observer, State, Visitor and Composite patterns are examples of this kind of pattern. The Mediator pattern, for instance, exhibits high inter-role interaction: each Colleague collaborates with the Mediator, which in turn collaborates with all the Colleagues. The use of AOP was useful to reduce the coupling between the participants in the pattern and to increase their cohesion, since the aspect code modularizes the collaboration protocol between the pattern roles. Figure 9 illustrates how the aspect was used to reduce the coupling of the OO solution of the State pattern. On the other hand, the use of aspects did not succeed for improving coupling and cohesion in the patterns whose roles are not highly interactive. This is the case for the Prototype and Strategy patterns and the patterns in Group 4, presented in Sect. 4.4.

5.1.4 Inheritance Coupling: A Different Perspective

Given the results obtained from the DIT measures, which did not show considerable differences between AspectJ and Java implementations, we have decided afterwards to use another classical metric: Number of Children (NOC) [5]. This measure counts the number of modules that extends a module using inheritance. Table 5 presents the NOC measures for the OO and AO versions of all the pattern implementations. It also compares the DIT values with the NOC values.

From the NOC point of view, it is clear that the use of AO abstractions significantly reduces the use of inheritance as extension mechanism. While AspectJ solutions tend to present a stronger intercomponent coupling (Sect. 5.1.3) since they heavily rely on pointcuts and advice to support the specification of extensions and refinements to the affected modules, the Java implementations tend to present a stronger inheritance coupling. This observation motivates the need for further empirical case studies that evaluate the trade-offs of using each of these different extension mechanisms with respect to distinct quality attributes, such as understandability, reusability, maintainability, and reliability.

5.1.5 Aspects and Size Attributes

The reduction in the program size in general decreases the likelihood of developers introducing errors into the system [25]. Table 6 presents the overall results for size-related measures in terms of each pattern. Section 4 presented the size results associated with coupling and cohesion. Table 6 brings a new view for our assessment because it classifies the pattern implementations only in terms of size-related programming efforts. The columns respectively present the results with respect to number of attributes (NOA), complexity of operations (WOC) and lines of code LOC).

Table 5. Results for two inheritance-related measures

	DIT		NOC		Superior
Design pattern	**OO**	**AO**	**OO**	**AO**	**solution**
Abstract Factory	7	7	6	6	=
Adapter#	2	1	1	0	AO
Bridge	2	2	8	8	=
Builder	2	2	6	6	=
CoR#	2	2	7	1	AO
Command#	7	7	6	1	AO
Composite#	2	2	6	1	AO
Decorator#	3	1	8	0	AO
Façade	Same implementations for Java and AspectJ				
Factory Method	2	2	6	6	=
Flyweight#	2	2	6	7	OO
Interpreter	5	5	9	9	=
Iterator#	2	2	6	3	AO
Mediator#	2	2	10	1	AO
Memento#	1	2	0	3	OO
Observer#	2	2	10	3	AO
Prototype#	2	2	6	1	AO
Proxy#	2	2	9	6	AO
Singleton#	2	2	5	10	OO
State#	2	2	7	7	=
Strategy#	2	2	6	1	AO
Template Method#	2	2	6	6	=
Visitor#	2	2	10	10	=
Success total	**1 vs. 2**		**3 vs. 11**		**3 vs. 11**

The design pattern contains one or two superimposed roles.

We have found that the use of aspects has a considerable impact on the size attributes of the pattern implementations In general, the AO solutions were superior with the exception of lines of code. For 7 of the patterns, the AO solutions had fewer LOC than the OO solutions, which were superior in 14 cases. However, for these 14 implementations, the difference was not relevant in several cases. In fact, the discrepancy was evident (i.e., more than 10%) only in 1 case: the Memento pattern (Table 6). For ten of the patterns, the AspectJ implementations had fewer attributes than the Java implementations. Only one OO solution was superior in terms of NOA. For 12 of the patterns, the AO implementation reduced the number of operations and respective parameters (WOC metric). The OO implementation provided better results for seven patterns with respect to the WOC metric.

The last column of Table 6 indicates which solution was superior for each pattern considering all the three size measures. Similarly to Tables 4 and 5, we have classified an AspectJ or Java solution as superior when it has achieved better results for most the measures when compared with the results of the other solution. We have only considered that an implementation was better than the other when the difference

between two values for the same metric was equal or higher than 10%. The last cell of Table 6 shows the final result: the AO solutions succeeded in ten cases against four for the OO solutions.

Table 6. Overall results for size measures

	NOA		WOC		LOC		Superior
Design pattern	**OO**	**AO**	**OO**	**AO**	**OO**	**AO**	**solution**
Abstract Factory	9	9	37	41	231	265	OO
Adapter#	3	1	34	32	67	61	AO
Bridge	1	1	40	44	156	161	OO
Builder	7	7	50	51	168	177	=
CoR#	8	2	40	64	213	234	=
Command#	6	4	26	29	198	206	=
Composite#	19	12	169	63	501	283	AO
Decorator#	1	0	34	16	88	69	AO
Façade	Same implementations for Java and AspectJ						
Factory Method	1	1	17	17	135	146	=
Flyweight#	7	7	30	36	119	132	OO
Interpreter	14	14	99	99	216	219	=
Iterator#	9	9	50	53	164	163	=
Mediator#	21	17	51	40	253	253	AO
Memento#	6	6	32	31	128	179	OO
Observer#	26	21	134	117	363	265	AO
Prototype#	6	6	38	33	142	147	AO
Proxy#	9	3	105	38	248	190	AO
Singleton#	30	26	25	21	238	251	AO
State#	13	20	164	110	367	374	=
Strategy#	5	1	62	58	251	264	AO
Template Method#	0	0	46	46	125	128	=
Visitor#	13	13	105	57	289	222	AO
Success total	**1 vs. 10**		**7 vs. 12**		**14 vs. 7**		**4 vs. 10**

The design pattern contains one or two superimposed roles.

5.1.6 Reusability Issues

The HK study observed reusability improvements in the AspectJ versions of 12 patterns by enabling a core part of the pattern implementation to be abstracted into reusable code (Sect. 2.2). In our study, expressive reusability was observed only in four patterns: Mediator, Observer, Composite and Visitor. These patterns were also qualified as reusable in the HK study and have several characteristics in common: (i) defined as reusable abstract aspects, (ii) improved separation of concerns (Sect. 3.1), (iii) low coupling – CBC – and high cohesion – LCOO (Sect. 4.1), and (vi) decreased values for the LOC and WOC measures as the changes are applied. Expressive reuse is evident when the extension or customization of existing components to include new functionalities requires the implementation of few lines of code, operations, attributes, classes and the like.

However, note that in our investigation the presence of generic abstract aspects has not necessarily led to improved reusability in several cases. The Flyweight, Command, CoR, Memento, Prototype, Singleton and Strategy patterns have abstract aspects and were ranked as "reusable" patterns in the HK study. In contrast, an analysis of the results presented in Sects. 3 and 4 leads to contrary conclusions for these patterns. In general, reusable elements lead to less programming effort by requiring fewer operations and lines of code to be written. However, the LOC and WOC measures of the AO implementations of these patterns were higher than in the respective OO implementations both before and after the changes. In fact, the abstract aspects associated with these patterns are very simple and do not enable a reasonable degree of reuse.

5.1.7 Superimposed Roles as a Predictive Model?

Determining when an AO technique is useful in a given context is a challenging task. The HK study has tried to establish a predictive model for helping the designers to decide when AspectJ should be used in design pattern implementations. According to this preceding study, the *presence of superimposed roles* (Sect. 2.1) seems to be a determining factor in such a decision-making process. Participant classes have their own functionalities outside the pattern scope in addition to the incorporation of pattern-related superimposed behavior. The OO version of the pattern implementation forces each of these classes to implement at least two concerns: the original responsibility and the pattern-specific behavior. The HK study claims that the AspectJ solution allows for the improved modularization of the superimposed roles.

Various flavors of our empirical study can be used to support or refute this claim, including the separation of concerns measures (Sect. 3), and the coupling, cohesion, and size measures (Sect. 4). In general, the results presented in Table 3 do not accredit this predictive model as absolute. In the table, the 17 patterns with superimposed roles are marked with "#". Some patterns that encompass superimposed roles achieved improved modularity in AspectJ implementations, namely Adapter, Decorator, Proxy, Visitor, Composite, Mediator, Singleton and Observer. Indeed, for seven of them (except the Composite and Iterator patterns), the AO solution has scaled up well (Table 3). However, seven of them did not reach convincing modularity improvements: Templated Method, Command, Flyweight, Memento, Strategy, CoR and Prototype. Moreover, the AspectJ version of the State pattern has not exhibited improved separation of concerns, when the aspectization of the Iterator pattern has presented poor coupling (CBC metric) and more complexity in the operation definitions (WOC metric). As a result, there is no evidence that the presence of superimposition should be considered as the sole determining factor to use AO abstractions to implement design patterns.

Analyzing simultaneously Tables 3 and 4 and according to the discussions in the previous subsections, it clearly seems that other important factors should be considered as part of a predictive model. Coupling and cohesion should be also considered when deciding for the aspectization of the design patterns since the more successful AspectJ implementations were the ones where there was a higher inter-role interaction (Sect. 5.1.3). The coordinated analysis of these factors would certainly result in a more consistent prediction mechanism according to our findings.

5.1.8 Comparison with the HK Study

Through the replication of case studies with similar goals, the AOSD community can build an experience factory of empirical findings. In this context, when performing systematic case studies it is important to compare the new results with those of previous studies so that we can effectively build a body of knowledge about the theme under assessment. This information is also important to researchers and practitioners who intend to replicate this experiment. This section summarizes the outcomes of our study that confirms, contradicts or refines the claims in the HK study [15]. We have focused only on three issues where there was a direct intersection in the findings:

(i) While the HK study has found improved separation of concerns in 17 AspectJ pattern implementations, our study detected only 14 improvements (Sect. 5.1.1).
(ii) The first study ranked 12 AspectJ solutions as reusable against 4 of this study (Sect. 5.1.6).
(iii) The findings in this study suggest that the original prediction model, presented by the HK study, should be refined to also consider coupling and cohesion (Sect. 5.1.7).

The differences in the two studies are mainly because the HK study has used only simple pattern instances, which did not allow a clear understanding of the benefits and drawbacks of the aspect-oriented implementations. In addition, the authors took a narrow view of reusability, and the definition of the proposed predictive model was naturally biased by the role-oriented strategy that they have used to "aspectize" the design patterns.

5.1.9 Need for Multidimensional Analysis

As discussed in Sect. 5.1.7, it seems imperative to analyze other software attributes when assessing AO solutions. The HK study has centered the comparative analysis only on separation of concerns, and how the achieved separation helps to improve directly associated high-level qualities, such as (un)pluggability and composability. Lopes [24] has also carried out a case study that rests only on separation of concerns as assessment criteria. However, based on the results of this study (Sects. 3 and 4) and the discussion above, it seems clear that the analysis of other software dimensions or attributes, such as coupling and internal complexity of operations, are extremely important to compare AO and OO designs. In fact, the interaction between the aspects and the classes is sometimes so intense that the separation of aspects in the source code seems to be a more complex solution with respect to other software attributes.

5.2 Analysis of Specific Patterns

The measurements in this study were also important to assess the AO implementation of each design pattern in particular. We have found that some problems in the AO solutions are not related to the AO paradigm itself, but to some design or implementation decisions taken in the HK implementations. In this sense, quantitative assessments are also useful to capture opportunities for refactoring in AO software, for discarding a specific solution or for just clarifying important limitations of the

solution. This section presents some examples of how the metrics used in this quantitative study were useful to support either the refactoring (Sects. 5.2.1 and 5.2.2) or the discarding (Sects. 5.2.3–5.2.6) of some AO solutions of the GoF patterns.

5.2.1 Prototype

The use of the selected metrics for separation of concerns was important to detect remaining crosscutting concerns relative to the design patterns. For example, the original AspectJ implementation of the Prototype pattern left the declaration of the `Cloneable` interface, which is a pattern-specific responsibility, in the description of the application-specific classes. This solution was refactored based on the use of an intertype declaration in order to improve the separation of concerns, overcoming the crosscutting problem present in the original version of the AspectJ implementation [15].

5.2.2 Chain of Responsibility and Memento

The coupling measures were also important to detect opportunities for improvements in the AO implementations. For example, the implementations of some client classes, such as in the CoR and Memento patterns, have explicit references to the aspects implementing the pattern roles that increase the system coupling. These references are used in the client classes to trigger aspect initializations. This kind of coupling is unnecessary and could be avoided. The aspects associated with these patterns could incorporate, in addition to the initialization methods in the aspects, the definition of simple pointcuts to capture the joinpoints where the initializations should be triggered. This finding was also supported by the metrics for separation of concerns.

5.2.3 Flyweight and Interpreter

The presence of several negative results can also serve as warnings of unhelpful designs. As mentioned before, the AspectJ implementation of the Flyweight pattern did not provide evident benefits. All the metrics for separation of concerns (Sect. 3.2) and almost all the metrics for coupling, cohesion and size (Sect. 4.4) supported this finding.

In the same way, the metrics did not show advantages for the AO solution of the Interpreter pattern. In fact, there is no difference between the AO and OO implementations in terms of the structure of this pattern. This claim is supported by similar results for all the metrics. There are minor differences in favor of the OO version in terms of coupling and size. This difference is caused by the use of an aspect to attach methods to the participant classes by means of the intertype declaration mechanism. However, this aspect does not change the OO structure of the pattern. It is only used to add methods in the participant classes without changing them. Therefore, the AO solution is not useful for removing pattern code from the participant classes. Actually, in this aspect code there is a comment where Hannemann and Kiczales claim that, due the very nature of the Interpreter pattern, using aspect to remove the pattern code from the participants does not work nicely [15].

5.2.4 Strategy

As stated earlier, for some patterns, the AO solution was more complex than the OO solution in terms of coupling and size. This problem occurred for the Strategy pattern and was detected with the help of the coupling between components (CBC) and lines of code (LOC) metrics. The results of these metrics showed high values for the concrete aspect used to assign the roles to the Strategy and Context classes and trigger the execution of the strategy algorithm. In order to choose what is the strategy to be executed for a given Context class, this aspect uses a sequence of "if" statements and references to all Strategy classes. This design is less flexible than the OO design since this aspect has to be changed whenever a new Strategy class is created.

5.2.5 Command

The problem of the aspect-oriented solution of the Command pattern is similar to the problem described for the Strategy pattern (Sect. 5.2.4). The aspects, which modularize pattern roles, are highly coupled to the other elements in the design. In the case of the Command pattern, a concrete aspect is coupled to all Invoker, Receiver and Command classes. As a consequence, adding new participants to an instance of the AspectJ version of this pattern requires more effort than to an instance of the Java version. This occurs because the aspect needs to be inevitably changed.

Another deficiency of the AO version of this pattern concerns to the use of parameters on the `execute()` method of the Command classes. In the AspectJ implementation, the Invoker classes are not aware of the command execution as they are in the OO implementation. Instead, the execution of the commands is triggered by the aspects. This design decision does not allow the Invokers to pass information of their context to the commands as parameters of the `execute()` method. Thus, if the Command classes need information from the context of the Invokers, this AO solution of the Command pattern should not be used.

5.2.6 Decorator

The AO implementation of the Decorator pattern showed better results for most metrics. However the inferior results obtained for the coupling between components (CBC) metric highlight an important limitation of this design. One of the Decorator aspects is coupled to all other aspects, since it determines the order in which the decorators are applied to the component by means of the `declare precedence` construct. Therefore, this aspect has to be changed whenever a new decorator is created. Besides, this design is very rigid in the sense that the decorators must be applied in the same order for every component. Hence, if it is necessary to apply decorators in different orders, this AO solution should be discarded.

5.3 Study Constraints and Lessons Learned

Concerning our experimental assessment, there is one general type of criticism that could be applied to the used software metrics (Sect. 2.3). This refers to theoretical arguments leveled at the use of conventional size metrics (e.g., LOC), as they are applied to traditional (non-AO software) development. Despite, or possibly even because of, simplicity of these metrics, it has been subjected to severe criticism [37]. In fact, these measures are sometimes difficult to evaluate with respect to a software

quality attribute. For example, the LOC measures are difficult to interpret since sometimes a high LOC value means improved modularization, but sometimes it means code replication.

However, in spite of the well-known limitations of these metrics, we have learned that their application cannot be analyzed in isolation, and they have shown themselves to be extremely useful when analyzed in conjunction with the other used metrics. In addition, some researchers (such as Henderson-Sellers [16]) have criticized the cohesion metric as being without solid theoretical bases and lacking empirical validation. However, we understand this issue as a general research problem in terms of cohesion metrics. In the future, we intend to use other emerging cohesion metrics based on program dynamics.

We have also learned some lessons when using the separation of concerns metrics. We have observed that these three metrics complement each other. CDC and CDO respectively measure the number of components and operations that implement a concern. However, a concern may be spread through many classes, but may not be tangled with other concerns, since these components and operations may only implement a single concern. The isolate use of CDC and CDO are not enough to capture the noncrosscutting nature of such a concern; even worse, they will likely provide false warnings to the AO designers. In this way, CDLOC metric complements CDC and CDO metrics by measuring if the concern is tangled with other concerns. Therefore, these metrics are complementary since we need to measure both degrees of scattering and tangling in order to verify whether a concern is well modularized. In addition, CDC and CDO also complement each other because a concern may be scattered over few components but may affect many operations in those components. This situation was observed in the AO solution for the Chain of Responsibility pattern, where Handler role was implemented by few aspects but scattered over many operations, indeed, more operations than the OO solution.

The limited size and complexity of the examples used in the implementations may restrict the extrapolation of our results. In addition, our assessment is restricted to the specific pattern instances at hand. However, while the results may not be directly generalized to the context of real-world systems and professional developers, these representative examples allow us to make useful initial assessments of whether the use of aspects for the modularization of classical design patterns would be worth studying further. In spite of its limitations, the study constitutes an important initial empirical work and is complementary to qualitative work (e.g., [15]) previously performed. In addition, although the replication is often desirable in experimental studies, it is not a major problem in the context of our study due to the nature of our investigation. Design patterns are generic solutions and, as a consequence, exhibit similar structures across the different kinds of applications where they are used.

Finally, we have also learned that some problems may be directly related to the programming language used in this study. There is a pressing need to perform similar studies applying other AO programming languages, such as Hyper/J [18] and Caesar [26]. Each of these languages has different features that certainly impact on the pattern implementations with respect to the quality software attributes investigated in this quantitative study. In fact, other quantitative studies on the aspectization of design patterns are needed; for example, it would be important to investigate whether and how the AO solutions scale in real large-scale systems. In this sense, it would be

possible to quantify the effects of modularizing pattern-related crosscutting concerns with aspects in systems where the pattern implementations are not simple pattern instances and are inserted in richer application contexts. In addition, it would be important to explore and assess the use of aspects when combining the use of two or more design patterns, as was done in [21] where an OO version of the Builder pattern and an AO version of the Decorator pattern were composed.

6 Related Work

There is little related work focusing either on the quantitative assessment of AO solutions in general, or on the empirical investigation of using aspects to modularize crosscutting concerns of classical design patterns. Up to now, most empirical studies involving aspects rest on subjective criteria and qualitative investigation. In a previous work [30], we have quantitatively analyzed only six patterns. The present paper presents a complete study involving all the 23 design patterns. There are some other works [13, 14, 17, 27] that investigate the interplay between aspects and design patterns. However, they focus on specific patterns and do not provide systematic quantitative assessments.

One of the first case studies was conducted by Kersten and Murphy [21]. They built a Web-based learning system using AspectJ. In this study, they discussed the effect of aspects on their OO practices and described some rules they employed to achieve their goals of modifiability and maintainability using aspects. Since several design patterns were used in the design of the system, they considered which of them should be expressed as classes and which should be expressed as aspects. They found that Builder, Composite, Façade and Strategy patterns [9] were more easily expressed as classes, once these patterns had little or no crosscutting behaviors. We have found here similar results for the Strategy, Builder and Façade patterns (Sects. 3 and 4). However, the AO implementation of the Composite pattern achieved better separation of concerns in our study.

Soares et al. [32] reported their experience using AspectJ to implement distribution and persistence aspects in a Web-based information system. They implemented the system in Java using specific design patterns and restructured it with AspectJ. They argued that the AspectJ implementation of the system bring significant advantages with the corresponding pure Java implementation.

Garcia et al. [11] have presented a quantitative study designed to compare the maintenance and reuse support of a pattern-oriented approach and an AO approach for a multiagent system. The subjects in the study used both approaches to try to modularize agent-related concerns, including autonomy, interaction, mobility, learning, adaptation and collaboration. They used an assessment framework that includes the same metrics suite used in our study. The results showed that the AO approach allowed the construction of the investigated system with improved modularization of the crosscutting agent-specific concerns. The use of aspects resulted in superior separation of the agent-related concerns, lower coupling (although less cohesive) and fewer lines of code. However, their study was also not focused on the use of aspects to isolate the crosscutting concerns relative to classical design patterns.

Zhao and Xu [35, 36] have proposed new cohesion measures that consider the peculiarities of the AO abstractions and mechanisms. Their metrics are based on a dependence model for AO software that consists of a group of dependence graphs; each of them can be used to explicitly represent various dependence relations at different levels of an AO program. Also, the cohesion measures [36] proposed by the authors are formally defined. The authors have shown that their measures satisfy some properties that good measures should have. However, these metrics have not yet been validated or applied to the assessment of realistic AO systems.

7 Conclusion

This paper presented a quantitative study comparing the AO and OO implementations of the GoF patterns. The results have shown that most AO implementations provided improved separation of concerns. However, some patterns resulted in higher coupled components, more complex operations and more LOCs in the AO solutions. Another important conclusion of this study is that separation of concerns cannot be taken as the only factor to conclude for the use of aspects. It must be analyzed in conjunction with other important factors, including coupling, cohesion and size. Sometimes, the separation achieved with aspects can generate more complicated designs. Hence, based on our analysis, many AO implementations present implementation alternatives with different tradeoffs from their OO equivalents. Also, since this is a first exploratory study, to further confirm the findings, other rigorous and controlled experiments are needed.

It is important to notice that from this experience, especially in a nonrigorous area such as software engineering, general conclusions cannot be drawn. The scope of our experience is indeed limited to (a) the patterns selected for this comparative study, (b) the specific implementations from the GoF book [9] and the HK study [15], (c) the Java and AspectJ programming languages, and (d) a given subset of application scenarios that were taken from our development background. However, the goal was to provide some evidence for a more general discussion of what benefits and dangers the use of AO abstractions might create, as well as what and when features of the AO paradigm might be useful for the modularization of classical design patterns. Finally, it should also be noted that properties such as reliability must be also examined before one could establish preference recommendations of one approach relative to the other.

Acknowledgments. We would like to thank Jan Hannemann and Gregor Kiczales for making the pattern implementations available, and Brian Henderson-Sellers and Barbara Kitchenham for the discussions on the selection of the software metrics. This work has been partially supported by CNPq-Brazil under grant No. 381724/04-2 for Alessandro, grant No. 140214/04-6 for Cláudio and under grant No. 140252/03-7 for Uirá. The authors are also supported by the ESSMA Project under grant 552068/02-0.

References

[1] Aspect-Oriented Design Pattern Implementations. http://www.cs.ubc.ca/~jan/AODPs/. Cited May 2005

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

[3] Basili V., Briand, L., Melo W. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996

[4] Basili V., Selby R., Hutchins D. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12, 733–743, 1986

[5] Chidamber S., Kemerer C. A metrics suite for OO design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994

[6] Fenton N., Pfleeger S. Software metrics: A rigorous practical approach. PWS, London 1997

[7] Figueiredo E., Garcia A., Sant'Anna C., Kulesza U., and Lucena C. Assessing aspect-oriented artifacts: Towards a tool-supported quantitative method. In: *QAOOSE.05: Proceedings of the 9th ECOOP Workshop on Quantitative Approaches in OO Software Engineering*, Glasgow, 2005

[8] Filho F., Rubira C., and Garcia A. A quantitative study on the aspectization of exception handling. In: *Proceedings of the ECOOP Workshop on Exception Handling in Object-Oriented Systems*, 2005

[9] Gamma E. et al. Design patterns: Elements of reusable object-oriented software. Addison-Wesley, Reading, 1995

[10] Garcia A. From objects to agents: An aspect-oriented approach. Doctoral Thesis, PUC-Rio, Rio de Janeiro, Brazil, 2004

[11] Garcia A. et al. Separation of concerns in multi-agent systems: An empirical study. In: *Software Engineering for Multi-Agent Systems II*, *LNCS vol. 2940*, Springer, 2004

[12] Garcia A., Silva V., Chavez C., Lucena C. Engineering multi-agent systems with aspects and patterns. *Journal of the Brazilian Computer Society*, 8(1):57–72, 2002

[13] Hachani Q., and Bardou D. On Aspect-oriented technology and object-oriented design patterns. In: *ECOOP*: *Workshop on Analysis of Aspect-Oriented Software*, Springer, Germany, 2003

[14] Hachani Q., and Bardou D. Using aspect-oriented programming for design patterns implementation. In: *OOIS*: *Workshop on Reuse in Object-Oriented Information Systems Design*, 2002

[15] Hannemann J., and Kiczales G. Design pattern implementation in java and AspectJ. In: *OOPSLA'02: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 161–173, 2002

[16] Henderson-Sellers B. Object-oriented metrics: Measures of complexity. Prentice Hall, New Jersey, USA, 1996

[17] Hirschfeld R et al. Design patterns and aspects - Modular designs with seamless run-time integration. *3rd German GI Workshop on Aspect-Oriented Software Development*, German Informatics Association, University of Essen, Germany, 2003

[18] Hyper/J Web page. http://www.research.ibm.com/ hyperspace/HyperJ/HyperJ.htm, 2001

[19] Godil I., Jacobsen H. Horizontal decomposition of prevayler. In: *Proceedings of CASCON 2005*, Richmond Hill, Canada, 2005

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

[21] Kersten M., and Murphy G. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In: *OOPSLA'99: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1999

[22] Kiczales G. et al. Aspect-oriented programming. In: *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, *LNCS vol. 1241*, Springer, pp. 220–242, 1997

[23] Kitchenham B. Evaluating software engineering methods and tools, part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–15, 1996

[24] Lopes C. D: A language framework for distributed programming. *PhD Thesis*, Northeastern University, Boston, USA, 1997

[25] Malaiya Y., and Denton J. Module size distribution and defect density. In: *ISSRE'00: Proceedings of the 11th International Symposium on Software Reliability Engineering*, 2000

[26] Mezini M., and Ostermann K. Conquering aspects with caesar. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, 2003

[27] Miles R. AspectJ cookbook. *O'Reilly*, UK, 2004

[28] Modularizing Patterns with Aspects: A Quantitative Study. http://www.teccomm.les.inf.puc-rio.br/alessandro/GoFpatterns/empiricalresults.htm

[29] Sant'Anna C. et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In: *SBES'03: Proceedings of the Brazilian Symposium on Software Engineering*, Manaus, Brazil, pp. 19–34, 2003

[30] Sant'Anna C. et al. Design patterns as aspects: A quantitative assessment. *Journal of the Brazilian Computer Society (SBES'04 Best Paper Award)*, 10(2), Porto Alegre, Brazil, 2004

[31] Soares S. An aspect-oriented implementation method. *Doctoral Thesis*, Federal University of Pernambuco, Recife, Brazil, 2004

[32] Soares S., Laureano E., and Borba P. Implementing distribution and persistence aspects with AspectJ. In: *OOPSLA'02: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 174–190, 2002

[33] Tarr P. et al. N degrees of separation: Multi-dimensional separation of concerns. In: *ICSE'99: Proceedings of the International Conference on Software Engineering*, pp. 107–119, 1999

[34] Together Technologies. http://www.borland.com/together/

[35] Zhao J. Towards a metrics suite for aspect-oriented software. Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), 2002

[36] Zhao J., and Xu B. Measuring aspect cohesion. In: *FASE'04: Proceedings Conference on Fundamental Approaches to Software Engineering*, *LNCS vol. 2984*, Springer, pp. 54–68, 2004

[37] Zuse H. History of software measurement. http://irb.cs.tu-berlin.de/~zuse/metrics/History_00.html