Aspectizing Design Patterns: Rewards and Pitfalls

Alessandro Fabricio Garcia Cláudio Nogueira Sant'Anna Eduardo Figueiredo Uirá Kulesza
Carlos José Pereira de Lucena Arndt von Staa

Computer Science Department – SoCAgents/TecComm Group
Pontifical Catholic University of Rio de Janeiro – PUC-Rio
Rua Marquês de São Vicente, 225 – Ed. Pe. Leonel Franca, 10º Andar
Rio de Janeiro – Brazil

{afgarcia,claudios,emagno,uira,lucena,arndt}@inf.puc-rio.br

PUC-RioInf.MCC43/04 November, 2004

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 decrease 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 object-oriented and aspect-oriented implementations of classical patterns with respect to important software engineering attributes for reusability and maintainability, such as coupling, cohesion. This paper presents a quantitative study that compares aspect-based and OO solutions for the 23 Gang-of-Four patterns. We have used stringent software engineering attributes as the assessment criteria. We have found that most aspect-oriented solutions improve separation of pattern-related concerns, although some aspect-oriented implementations of specific patterns result in higher coupling or lower cohesion.

Keywords: Separation of concerns, design patterns, aspect-oriented programming, metrics.

Resumo: Padrões de projeto oferecem soluções flexíveis para problemas comuns de desenvolvimento de software. Estudos recentes mostraram que vários padrões de projeto envolvem interesses transversais. Infelizmente, as abstrações orientadas a objetos (OO) muitas vezes não são capazes de modularizar bem tais interesses transversais, piorando a manutenibilidade e o reúso dos sistemas de software. Portanto, é importante verificar se abordagens orientadas a aspectos permitem uma melhor modularização dos interesses transversais relativos a padrões de projetos. Idealmente, estudos quantitativos devem ser realizados para comparar implementações OO com implementações orientadas a aspectos de padrões clássicos com respeito a atributos importantes para reutilização e manutenibilidade, tais como acoplamento e coesão. Este artigo apresenta um estudo quantitativo que compara soluções baseadas em aspectos com soluções OO para os 23 padrões da *Gang of Four*. Neste estudo, foram usados atributos rigorosos de engenharia de software como critérios de avaliação. O estudo mostrou que a maioria das soluções orientadas a aspectos melhorou a separação dos interesses relativos aos padrões, no entanto algumas implementações orientadas a aspectos de padrões específicos apresentaram maior acoplamento e mais linhas de código.

Palavras-chave: Separação de interesses, padrões de projeto, programação orientada a aspectos, métricas.

1 Introduction

Since the introduction of the first software pattern catalog containing the 23 Gang-of-Four (GoF) patterns [5], 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 [9]. The pattern roles often crosscut several classes in a software system. Moreover, recent studies [7, 8, 9] have shown that object-oriented abstractions are not able to modularize 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 [13, 19] support improved modularization of the crosscutting concerns relative to the patterns.

To the best of our knowledge, Hannemann and Kiczales [9] have developed the only systematic study that explicitly investigated the use of aspects to implement classical design patterns. They performed a preliminary study in which they develop and compare Java [11] 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. 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 complements Hannemann and Kiczales' work [9] by performing quantitative assessments of Java and AspectJ implementations for the 23 GoF patterns. Our study was 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 separation of pattern-related concerns. In addition, we have found that:

(i) the use of aspects 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 4 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; and
(v) the use of coupling, cohesion and size measures was helpful to assist the detection of opportunities for aspect-oriented refactoring of design patterns.

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 Section 4 presents the study results in terms of coupling, cohesion and size attributes. These results are interpreted and discussed in Section 5. 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. Our study supplements the Hannemann and Kiczales work that is presented in Section 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 assessment process. Section 2.4 describes our assessment procedures.

2.1 Hannemann & Kiczales' Study

Several design patterns exhibit crosscutting concerns [9]. In this context, Hannemann and Kiczales (HK) have undertaken a study in which they have developed and compared Java [11] and AspectJ [2] implementations of the 23 GoF design patterns [5]. They claim that programming languages affect pattern implementation. Hence it is natural to explore the effect of aspect-oriented programming 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 [9]. 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 operation in each "Colleague" in the pattern.

Two kinds of pattern roles are identified in the HK study. They are called "defining" and "superimposed" roles. These kinds of roles and respective interactions with participant classes are used by the authors to analyze the crosscutting structure of design patterns. 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 and responsibility outside of the pattern. An example of superimposed role is the role Colleague of the Mediator pattern, since a participant class playing this role usually has functionality not related to the pattern.

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. The degree of improvement has varied. They found out that patterns whose crosscutting structures involve roles and participant classes yield the largest improvement in the AspectJ implementation. These improvements were manifested in terms of four modularity properties: locality, reusability, composition transparency and (un)pluggability. The next subsection discusses these improvements as well as the 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 [5]. 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. Hanneman and Kiczales [9] 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 Media-

tor 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.

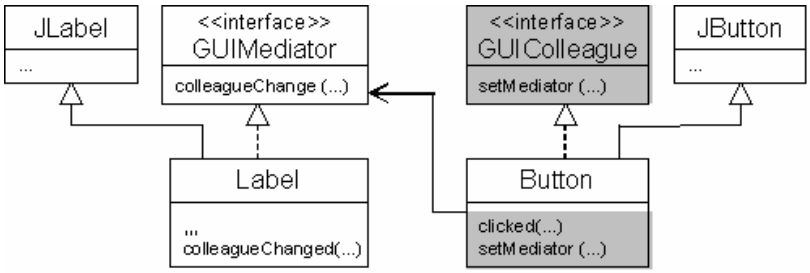


Figure 1. The OO Design of the Mediator Pattern

Figure 2 illustrates the source code of the `Button` class. The necessary changes to implement the Colleague role are shadowed. The `Button` class implements the `GUIColleague` interface by defining an attribute to reference a mediator (line 4) and the respective `setMediator()` method (line 6-8). Moreover, the `clicked()` method of the `Button` class defines the functionality to communicate with the mediator (line 21).

```
01 public class Button extends JButton
02                      implements GUIColleague {
03
04   private GUIMediator mediator;
05
06   public void setMediator(GUIMediator mediator){
07    this.mediator = mediator;
08   }
09   public Button(String name) {
10    super(name);
11    this.setActionCommand(name);
12    this.addActionListener(
13           new ActionListener() {
14          public void actionPerformed(
15                                ActionEvent e)
16              clicked();
17          }
18    });
19    }
20    public void clicked() {
21    mediator.colleagueChanged(this);
22    }
23 }
```

Figure 2. The Button Class of the OO Implementation

In their study, Hanneman and Kiczales identified the common 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. For example, in the AspectJ solution of the Mediator pattern, the code for implementing the pattern is textually localized in two aspects: (i) `MediatorProtocol` abstract aspect that encapsulates the common part to all po-

tential instantiations of the pattern; and (ii) a concrete extension of the abstract aspect that instantiates the pattern for specific contexts.

```
01 public abstract aspect MediatorProtocol {
02
03  protected interface Mediator { }
04
05  protected abstract void notifyMediator
06                   (Colleague c, Mediator m);
07
08  protected interface Colleague  { }
09
10  private WeakHashMap mappingColleagueToMediator =
11                          new WeakHashMap();
12
13  private Mediator getMediator(Colleague c){
14    Mediator mediator = (Mediator)
15    mappingColleagueToMediator.get(c);
16    return mediator;
17  }
18
19  public void setMediator(Colleague c, Mediator m){
20    mappingColleagueToMediator.put(c, m);
21  }
22
23  protected abstract pointcut change(Colleague c);
24
25  after(Colleague c): change(c) {
26    notifyMediator(c, getMediator(c));
27  }
28 }
```

Figure 3. The MediatorProtocol Aspect

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 8, 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 10-11). Changes to the Colleague-Mediator mapping can be realized via the public `setMediator()` method (line 19-21). 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 (line 25-27) in terms of the `change` pointcut and the `notifyMediator()` method.

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.

2.3 The Metrics

In our study, a suite of metrics for separation of concerns, coupling, cohesion and size [17] was selected to evaluate Hannemann and Kiczales' pattern implementations. These metrics have already been used in a significant number of other studies [6, 7]. Some of them have been automated in the context of a query-based tool for aspect understanding measurement and analysis [1]. This metric suite was defined based on the reuse and refinement of some classical and object-oriented metrics [3, 4]. The original definitions of the object-oriented metrics [3] were extended to be applied in a paradigm-independent way, supporting the generation of comparable results.

The metrics suite also encompasses new metrics for measuring separation of concerns. The separation of concerns metrics measure the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advices), and lines of code. Table 2 presents a brief definition of each metric, and associates them with the attributes measured by each one. Refer to [6, 17] for further details about the metrics.

	Metrics	Definition
Separation of Concerns	Concern Diffusion over Components (CDC)	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.
	Concern Diffusion over Operations (CDO)	Counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.
	Concern Diffusions over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is "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.

Table 1. The Metrics Suite

In order to better understand the separation of concerns metrics, consider the object-oriented example of the Mediator pattern, shown in Figure 1 (Section 2.2). In that example, there is code related to the Colleague role in the `GUIColleague` interface and in the shadowed methods of `Button` class, i.e., this concern is implemented by one interface and one class. Therefore, the value of the Concern Diffusion over Components metric (CDC) for this concern is two. Similarly, the value of the Concern Diffusion over Operations metric (CDO) 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 metric Concern Diffusion over Lines of Code (CDLOC) allows to measure the number of transition points for each concern through the lines of code. A transition point is the point in the code where there is "concern switch". CDLOC is measured by shadowing lines of code in the application classes related to the specific concern that you are interested to investigate. 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 Figure 2, the `Button` class was shadowed in order to make it

possible to measure the value of CDLOC for the Colleague role 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.

2.4 Assessment Procedures

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 code of the patterns. Examples of such kinds of changes were:

(i) to add or remove a functionality – a method, a class or an aspect – in the aspect-oriented (or object-oriented) implementation of the 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; and

(ii) to ensure that both versions were using the same coding styles.

Afterwards, we changed both Java and AspectJ implementation 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 Colleague role were added, as the `Button` class in Figure 1 (Section 2.2); furthermore, four classes playing the Mediator role were added, as the `Label` class in Figure 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. Table 1 presents the 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).

Design Pattern	Introduced Changes
Abstract Factory	4 Factories
Adapter	4 methods adaptees
Bridge	2 Abstractions and 2 Implementors
Builder	4 Builders
Chain Of Responsibility (CoR)	4 Handlers
Command	4 Commands and 2 Invokers
Composite	2 Composites and 2 Leafs
Decorator	4 Decorators
Façade	No Change
Factory Method	4 Creators
Flyweight	4 Flyweights
Interpreter	4 Expressions
Iterator	2 Iterators and 2 Aggregates
Mediator	4 Mediators and 4 Colleagues
Memento	2 Mementos and 2 Originators
Observer	4 Observers and 4 Subjects
Prototype	4 Prototypes
Proxy	4 Proxies and 2 Real Subjects
Singleton	4 Singletons and 4 subclasses
State	4 States
Strategy	4 Strategies and 4 Contexts
Template Method	4 Concrete Classes
Visitor	4 Elements and 2 Visitors

Table 2. The Design Patterns and Respective Changes

In the measurement process, the data was partially gathered by the CASE tool Together 6.0 [20]. 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, 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. Likewise 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, CDLOC) was manually collected. Due to space limitation, this paper focuses on the description of the more relevant results. The complete description of the data gathered is reported elsewhere [16].

3 Separation of Concerns

This Section and Section 4 present the results of the measurement process. The data have been collected based on the set of defined metrics (Section 2.4). The goal is to describe the results through the application of the metrics before and after the selected changes (Section 2.3). The analysis is broken into two parts. This section focuses on the analysis of to what extent the aspect-oriented (AO) and object-oriented (OO) solutions 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 Section 5. Section 5 also discusses 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 (Section 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 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 (Section 2.3). For example, the isolation of the roles Mediator and Colleague was analyzed in the implementations of the Mediator pattern, while the modularization of the roles Context and State was investigated in the implementations of the State pattern. The pattern roles crosscut participant classes. 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 (Section 3.1). Group 2 represents the patterns in which the OO solutions have shown as superior (Section 3.2). Group 3 involves the patterns in which the use of aspects did not impact the results (Section 3.3).

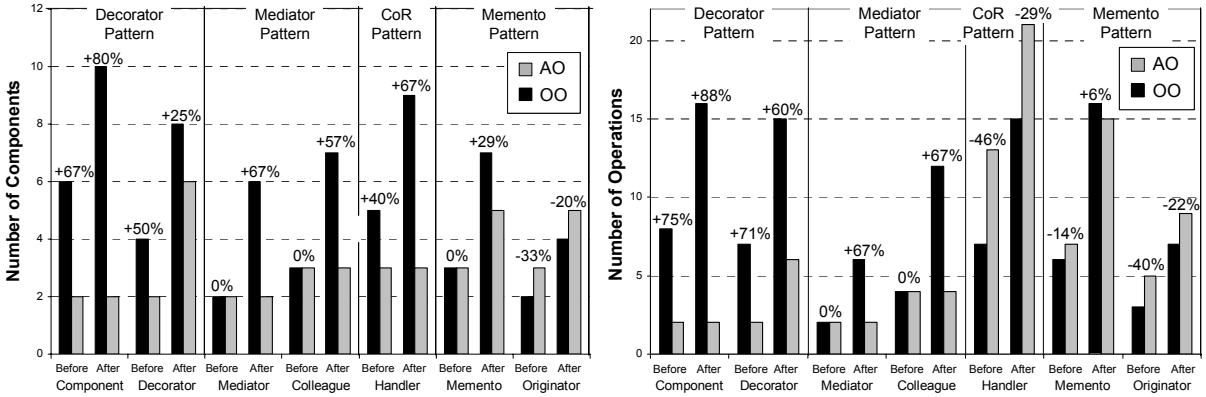
3.1 Group 1: Increased Separation

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 5 and 6 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 5a presents the CDC results, i.e. to what extent the pattern roles are isolated through the system components in both solutions. Figure 5b presents the CDO results, the degree of separation of the pattern roles through the system operations. Figure 2 illustrates the CDLOC measures – the tally of concern switches (transition points) through the lines of code.

All 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. In fact, these patterns were ranked with good "locality" in the Hannemann's analysis [9]. An analysis of Figures 5 and 6 shows that the best improvements come primarily from isolating the superimposed roles of the patterns (Section 2.1) in the aspects. For example, the definition of the Component role required 18 classes, while only 4 aspects were able to encapsulate this concern before the changes. It is equivalent to 78% in favor of the aspect-oriented design for the Mediator pattern. In fact, most superimposed roles were better modularized in the AO solution, such as Mediator (12 against 4), Colleague (8 against 6), and Handler (26 against 4). The results were similar to the separation of concerns over operations (Figure 5b) and lines of code (Figure 6). In addition, we can also observe that good results are achieved on the modularization of some defining roles, such as Decorator and Colleague.



(a) Concern Diffusion over Components **(b)** Concern Diffusion over Operations

Figure 5. Separation of Concerns over Components and Operations (Group 1)

After a careful analysis of Figures 5 and 6, we come to the conclusion that after the changes most AO implementations isolated the roles 25% or higher than the OO implementations. There are some cases where the difference is even bigger - the superiority of aspects exceeds 70%. For the Component and Colleague roles, the aspect-oriented solutions are even better before of 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 inter-object interactions (Mediator pattern – Section 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 aspect-oriented solutions gradually improves as new components are introduced into the system. It means that as more components are

included into an object-oriented system, more role-related code is replicated through the system components. Thus a gradual improvement takes place in the aspect-oriented solutions of the patterns. The series of small introduced changes (Section 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 aspect-oriented abstractions for segregating crosscutting structures.

Among the list of 14 patterns mentioned above, the 6 first ones 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. It 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 [9], requiring fewer classes and operations to address the isolation of the roles. All these 6 patterns have another common characteristic: they either involve no reusable aspect (Decorator and Adapter) or involve very simple reusable aspects (Prototype, Visitor, Proxy, Singleton).

The Decorator pattern is the representative of this kind of patterns in Figures 5 and 6. 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 do not change as the scenarios are applied to the aspect-oriented implementation. For example, the number of operations and components for specifying the Component role is the same before and after the scenarios in the AO implementation. The changes do not affect the measures. It demonstrates how well the aspect-oriented 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.

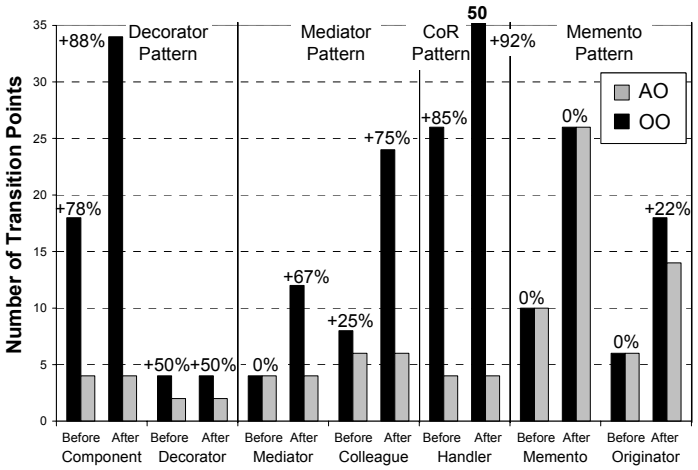


Figure 6. Concern Diffusion over LOC (Group 1)

The following 5 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 implemented changes. It

happens because the aspect-oriented implementations of these patterns involve generic aspects that are richer; they encapsulate more operations and code than the reusable aspects defined for the other 6 patterns mentioned before. 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 Mediator pattern represents these 5 patterns in Figures 5 and 6. 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 4 aspects were able to encapsulate this concern. This result was similar in the other 4 patterns, i.e. absolute number of components (CDC) did not vary after the modifications in the aspect-oriented solutions. This reflects the suitability of aspects for the complete separation of the roles associated with the 5 patterns. When new classes are introduced, they do not need to implement pattern-related code.

Finally, there were aspect-oriented solutions of three design patterns in this group (CoR, Strategy, and Memento), which although provided overall improved isolation of the roles, presented some negative results in terms of some measures. Figures 5 and 6 illustrate two examples: CoR and Memento. The AO implementation of the CoR pattern has fewer components (Figure 5a) and transition points (Figure 6) both before and after the changes. However, it has more operations involved in the implementation of the pattern role (Figure 5b). The AO solution of the Memento pattern 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 (Figure 6), it does not happen with respect to number of operations and components (Figure 5).

3.2 Group 2: Decreased Separation

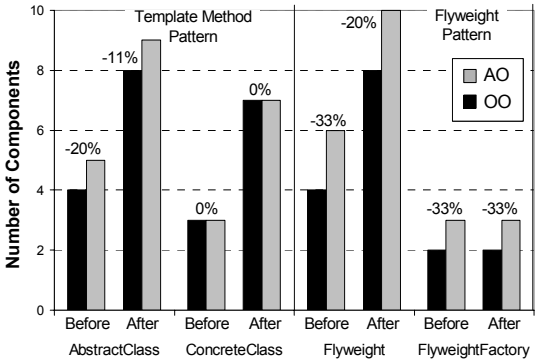
This second group includes design patterns in which AO implementations exhibited decreased separation of concerns. This group includes 6 patterns, namely Template Method, Abstract Factory, Factory Method, Bridge, Builder, and Flyweight. Figure 7 depicts the CDC, CDO and CDLOC measures of separation of concerns for the patterns 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. In general, the OO implementation provided better results, mainly with respect to the CDC measures (Figure 7a).

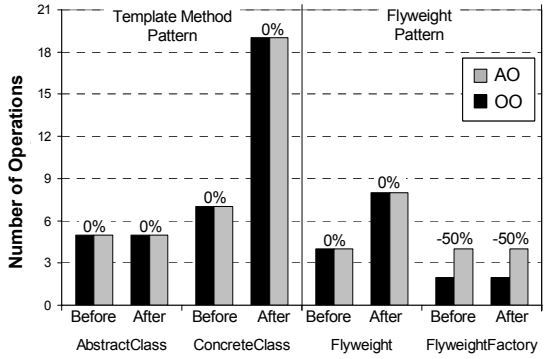
Another reason for this result is that all the patterns in this group, except the Flyweight, are structurally similar: they have an additional aspect to replace the abstract class mentioned in the GoF solution by interfaces without losing the ability to associated (default) implementations to their methods [9]. 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 implementation provided better results than the AO implementation 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 had no pointcuts and advices, and the generic aspect `FlyweightProtocol` could be implemented as a simpler class. As a result, the additional components

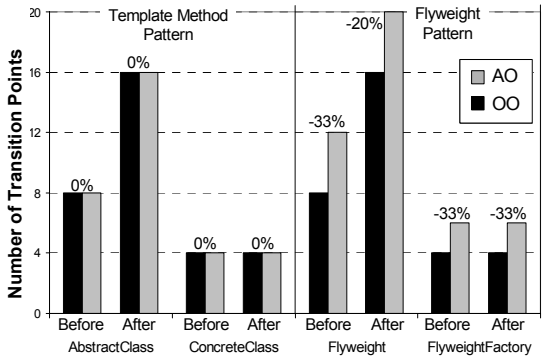
and operations introduced by the AO solution decreases the separation of concerns since the roles implementation are scattered over more code elements.



(a) Concern Diffusion over Components



(b) Concern Diffusion over Operations



(c) Concern Diffusion over LoC

Figure 7. Separation of Concerns (Group 2)

3.3 Group 3: No Effect

This group includes 3 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. 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 (Section 3.1) because the OO implementation of the patterns here do not imply in a significant crosscutting structure. The role-related code in these patterns affects a very small number of methods.

4 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 (Section 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 for all the implementation. The patterns were classified into 5 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 8 shows the graphics with results for the Mediator and Visitor patterns, which represent this group.

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

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 (Section 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 Figure 1). The aspect-oriented 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 classes. 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.

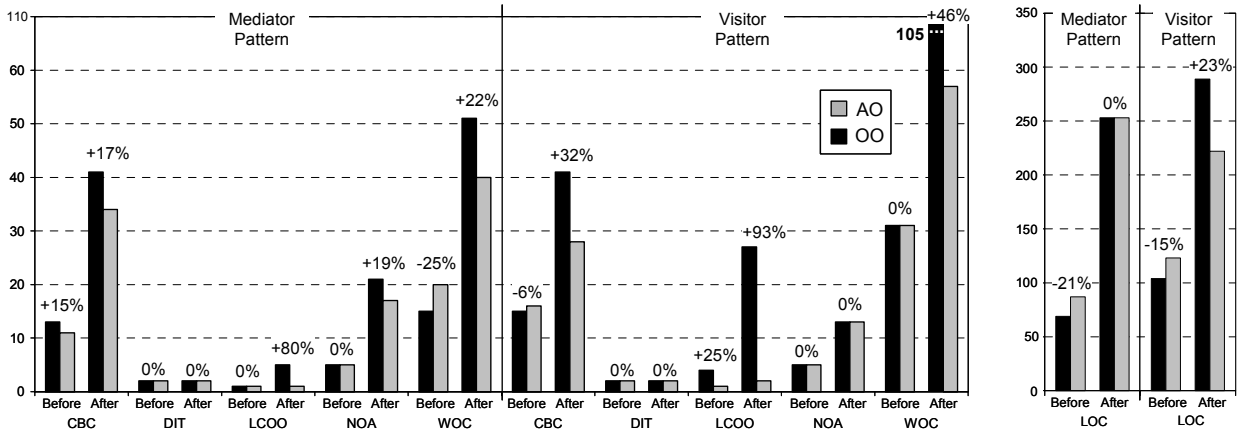


Figure 8. The Mediator and Visitor Patterns: Coupling, Cohesion and Size

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 (Section 3.1). As explained above, 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 4 patterns.

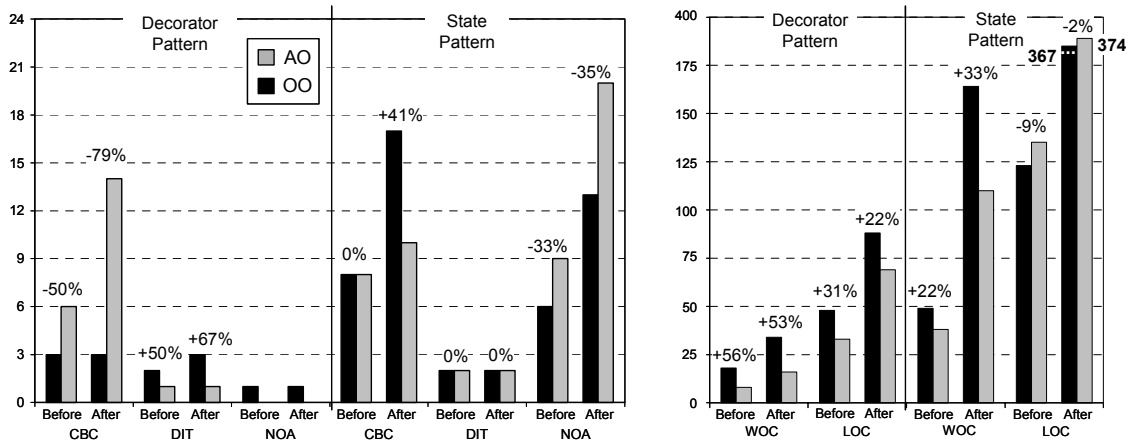


Figure 9. The Decorator and State Patterns: Coupling, Cohesion and Size

4.2 Group 2: Better Results for AO with Exceptions

This group encompasses the patterns in which aspect-oriented solutions produced better results in most of the measures except one. 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 expect 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 9 presents the results of the Decorator and State patterns as representative of this group.

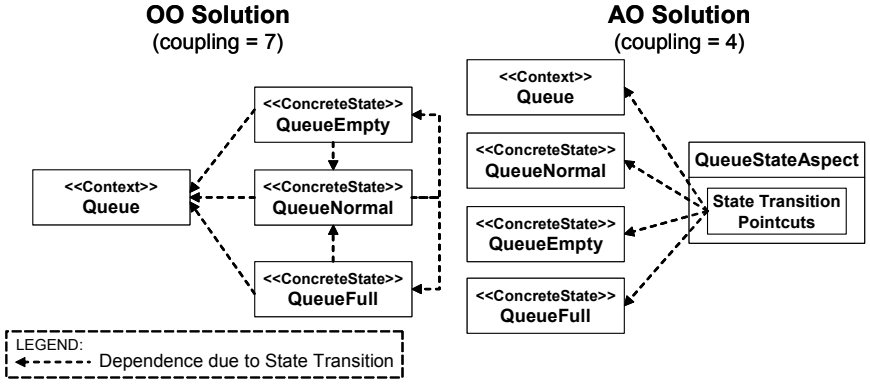


Figure 10. Coupling in the State Pattern: OO vs. AO.

The aspect-oriented implementations of the Decorator, Singleton and Proxy patterns manifest similar benefits to the patterns of Group 1 (Section 4.1). That is, the improvement in the separation of the pattern-specific code (Section 3.1) conducted to improvements in other attributes, such as, cohesion and size. However, as shown in Figure 9 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 particular results. Despite showing no improvements related to the separation of concerns metrics (Section 3.3), the AO implementation of the State pattern was superior in coupling, cohesion and weight of operations (Figure 9). 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 aspect-oriented solution because the aspects modularize the state transitions (Figure 10), minimizing the coupling between the pattern participants. Figure 10 shows that the coupling in the OO solution is seven because each State class needs to have references to the other State classes. From the NOA point of view, the OO implementation was superior because the aspect-oriented implementation has additional attributes in the aspects to hold references to the State elements.

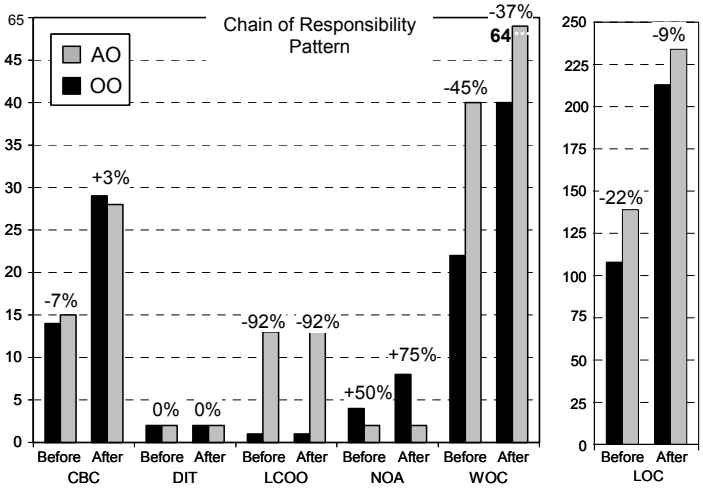


Figure 11. The Chain of Responsibility Pattern: Coupling, Cohesion and Size

4.3 Group 3: Better Results for OO with Exceptions

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 their AO solutions improved the results one size metric. In general, the OO implementations provided better or similar results with respect to the other metrics. 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 11 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 Section 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 affected negatively the LOC and WOC measures of the AO implementation. Furthermore, due to the coupling between the aspect and all 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 Prototype classes. However, this was only sufficient to reduce the weight of operations (WOC metric)

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 Figure 12.

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 Section 3.2 that these patterns are already nicely realized in OO, thus could not be given more modularized aspect-oriented 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 Figure 12). 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 Memento pattern showed the worst results. Removing the pattern-related code from the Originator classes and placing it in an aspect makes the aspect more complex. This is shown by the results of the CBC, DIT, WOC and LOC metric (see Figure 12).

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 that creates the Iterator class 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 (Section 3.1), the AO implementation provided insignificant improvements in terms of coupling, cohesion and size.

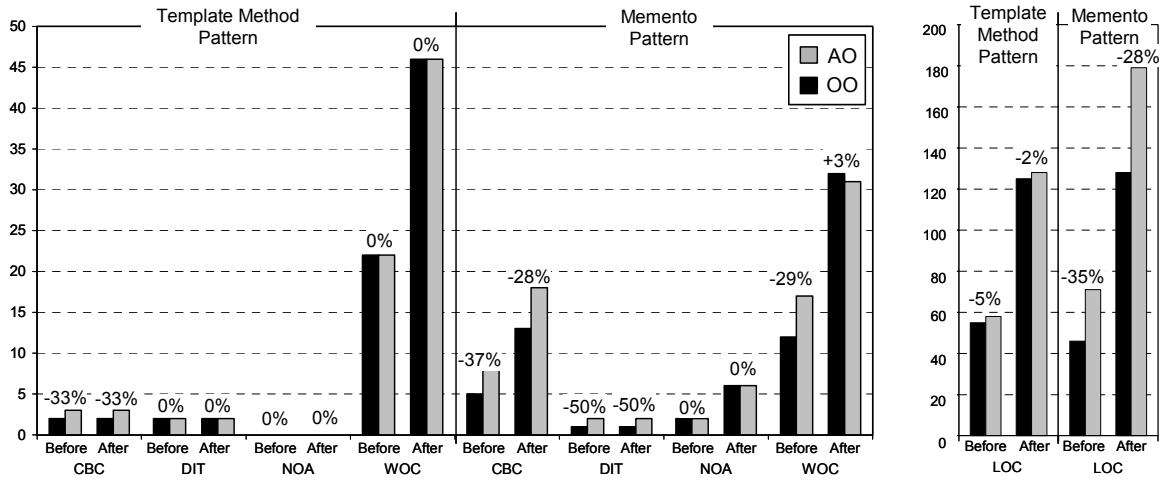


Figure 12. The Template Method and Memento Patterns: Coupling, Cohesion and Size

5 Discussions

Empirical studies [4] are the most effective way to supply evidence that may improve our understanding about software engineering phenomena. Although quantitative studies have some disadvantages [6], 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 [6]. Quantitative studies investigating the implementation of design patterns as aspects are rare [9]. This section provides a more general analysis of the previously observed results in Sections 3 and 4, and discussions about the constraints on the validity of our empirical evaluation.

5.1 General Analysis

This section presents an overall analysis of the previously observed results on the application of metrics for separation of concerns, coupling, cohesion and size. The following subsections also discuss the interplay between these different software attributes on the "aspectization" of design patterns.

5.1.1 Separable and Inseparable Concerns

As presented in Section 3.1, the AspectJ implementation of 14 patterns has shown better results in terms of the metrics of separation of concerns. In addition, the Java implementation of 6 patterns presented superior separation of roles (Section 3.2), and 3 patterns presented similar results in both implementations (Section 3.3). This observation provides evidence of the effectiveness of aspect-oriented abstractions for segregating crosscutting structures. 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 3 additional patterns offered locality improvements in the respective AO implementations: Template Method, Flyweight, and State. Our study's results somewhat contradicts these claims. The solution of patterns in Group 2 (Section 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 [9] (Section 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 (Section 3.2). The separation of concerns in the aspect-oriented version of the State pattern helps to separate state transitions, but the differences in the measures are not significant (Section 3.3).

An additional interesting observation in our study is that sometimes the pattern roles are expressed separately as aspects, but it remains non-trivial to specify how these separate aspects should be recombined 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 implementation of the Memento pattern provided better separation of the pattern-related concerns (Section 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 Section 4.4. The same observation can be made for the Strategy and CoR patterns (Section 4.3). In this context, there are some cases where the separation of the pattern-related concerns leads to more complex solutions.

5.1.2 Reducing Coupling and Increasing Cohesion

Based on the interplay of the results in Sections 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. The Mediator, Observer, State, Composite, Visitor patterns are examples of this kind of patterns. 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 aspects was useful to reduce the coupling between the participants in the pattern and increase their cohesion, since the aspect code modularizes the collaboration protocol between the pattern roles. Figure 10 illustrates how the aspect was used to reduce the coupling of the OO solution of the State pattern. This finding is similar to the conclusion presented in [9], in which the authors claim that aspects are useful to break cyclic dependencies. In fact, 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 Section 4.4.

5.1.3 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. In our study, expressive reusability was observed only in 4 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 (Section 3.1), (iii) low coupling – CBC – and high cohesion – LCOO (Section 4.1), and (vi) decreased values for the LOC and WOC measures as the changes are applied.

However, note that in our investigation the presence of generic abstract aspects did not conduct necessarily to improved reusability in several cases. The Flyweight, Command, CoR, Memento, Prototype, Singleton, Strategy patterns have abstract aspects and were ranked as "reusable" patterns in the HK study. In contrast, an analysis of the results presented in Sections 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.

5.1.4. Aspects and Size Attributes

We have found that the use of aspects has a considerable impact on the size attributes of the pattern implementations in addition to lines of code. For 10 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 7 patterns with respect to the WOC metric.

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 aspect-oriented paradigm itself, but to some design or implementation decisions taken in the HK implementations (Section 3.1). In this sense, quantitative assessments based on well-known software attributes, as performed in this study, are also useful to capture opportunities for refactoring in aspect-oriented software. This section presents some examples of how the metrics used in this quantitative study can be useful to support the refactoring 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 inter-type declaration in order to improve the separation of concerns, overcoming the crosscutting problem present in the original version of the AspectJ implementation [9].

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 implementation of some client classes, such as in the CoR and Memento patterns, has explicit reference to the aspects implementing the pattern roles, which increases the system coupling. This reference is used in the client classes to trigger some aspect initializations. This kind of coupling is unnecessary and could be avoided. The aspects associated with these patterns could incorporate the definition of simple pointcuts to capture the join points where the initializations should be made. This finding was also supported by the metrics for separation of concerns.

5.2.3 Flyweight

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

5.3 Study Constraints

Concerning our experimental assessment, there is one general type of criticism that could be applied to the used software metrics (Section 2.4). 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 [23]. In fact, these measures are sometimes difficult to evaluate with respect 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 [10]) have criticized the cohesion metric used in this study as being without solid theoretical bases and lacking empirical validation. However, we have used the LCOO metric because this is the most used cohesion metric and the other existing proposals have not presented convincing improvements; each of them present different limitations [24]. However, we understand this issue as a general research problem in terms of cohesion metrics. In the future, we intend to use another emerging cohesion metrics based on program dynamics.

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 professional developers and real-world systems, 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 a qualitative work (e.g. [9]) performed previously. 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.

6 Related Work

There is little related work focusing either on the quantitative assessment of aspect-oriented 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 [18], we have analyzed only 6 patterns. The present paper presents a complete study involving all the 23 design patterns.

One of the first case studies was conducted by Kersten and Murphy [12]. They have built a web-based learning system using AspectJ. In this study, they have discussed the effect of aspects on their object-oriented practices and described some rules and policies 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 have considered which of them should be expressed as classes and which should be expressed as aspects. They have found that Builder, Composite, Façade, and Strategy patterns [5] were more easily expressed as classes, once these patterns were had little or no crosscutting properties. We have found here similar results for the Strategy, Builder and Façade patterns (Section 5.2). On the other hand, the AO implementation of the Composite pattern achieved better separation of concerns in our study.

AOSD introduces new abstractions and composition mechanisms to support the separation of concerns in software development. Zhao and Xu [21, 22] have proposed new suites of measures that consider the characteristics and peculiarities of the AO abstractions and mechanisms. Their metrics are based on a dependence model for aspect-oriented 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 aspect-oriented program. The cohesion measures [22] proposed by the authors are formally defined. Also, the authors show that their cohesion measures satisfy some properties that a good measure should have. However, the new metrics proposed have not still been applied to the assessment of AO systems.

7 Conclusion and Future Work

This paper presented a quantitative study comparing the aspect-oriented and object-oriented implementations of the GoF patterns. The results have shown that most aspect-oriented 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 can not 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. However, 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 non-rigorous 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 [5] and HK study [9], (c) the Java and AspectJ programming language, 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 aspect-oriented abstractions might create, as well as what and when features of the aspect-oriented paradigm might be useful for the modularization of classical design patterns. Finally, it should also be noted that properties such as reliability and understandability must be also examined before one could establish preference recommendations of one approach relative to the other. We are planning now to perform a quantitative assessment of the combined use of design patterns in the development of different application contexts; this paper focused on the separate assessment of each design pattern.

Acknowledgements

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.

References

1. Alencar, P. et al. *A Query-Based Approach for Aspect Measurement and Analysis*. TR CS-2004-13, School of Computer Science, Univ. of Waterloo, Canada, Feb 2004.
2. AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.
3. Chidamber, S. and Kemerer, C. A Metrics Suite for Object Oriented Design. *IEEE Trans. on Software Eng., 20, 6* (June 1994), 476-493.
4. Fenton, N. and Pfleeger, S. *Software Metrics: A Rigorous Practical Approach*. London: PWS, 1997.
5. Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.
6. Garcia, A. *From Objects to Agents: An Aspect-Oriented Approach.* Doctoral Thesis, PUC-Rio, Rio de Janeiro, Brazil, April 2004.
7. Garcia, A. et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. *In Software Engineering for Multi-Agent Systems II, Springer, LNCS 2940*, January 2004.
8. Garcia, A., Silva, V., Chavez, and C., Lucena, C. Engineering Multi-Agent Systems with Aspects and Patterns. *J. of the Brazilian Computer Society, 1, 8* (July 2002), 57-72.
9. Hannemann, J. and Kiczales, G. Design Pattern Implementation in Java and AspectJ. *Proc. of OOPSLA'02* (November 2002), 161-173.
10. Henderson-Sellers, B. Object-Oriented Metrics: Measures of Complexity. Prentice Hall, 1996.
11. Java Reference Documentation. http://java.sun.com/reference/docs/index.html.

12. Kersten, A. and Murphy, G. Atlas: A Case Study in Building a Web-based learning environment using aspect-oriented programming. *Proceedings of OOPSLA'99*, November 1999.
13. Kiczales, G. et al. Aspect-Oriented Programming. *Proceedings of ECOOP'97, LNCS (1241)*, Springer, Finland, (June 1997), 220-242.
14. Lippert, M. and Lopes, C. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *Proc.of ICSE'00,* Limerick, Ireland, (May 2000), 418 - 427.
15. Lopes, C. D: *A Language Framework for Distributed Programming.* PhD Thesis, Northeastern University, 1997.
16. Modularizing Design Patterns whith Aspects: A Quantitative Study. http://www.teccomm.les.inf.puc-rio.br/alessandro/GoFpatterns/empiricalresults.htm
17. Sant'Anna, C. et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. *Proc. of Brazilian Symp. on Software Engineering (SBES'03)*, Brazil, Oct 2003, 19-34.
18. Sant'Anna, C. et al. Design Patterns as Aspects: A Quantitative Assessment. *Proceedings of Brazilian Symposium on Software Engineering (SBES'04)*, Brasília, Brazil, Oct 2004 (to appear).
19. Tarr, P. et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proc. ICSE'99,* USA, (May 1999), 107-119.
20. Together Technologies. http://www.borland.com/together/.
21. Zhao, J. *Towards a Metrics Suite for Aspect-Oriented Software* TR SE-2002-136-25, Inf. Processing Society of Japan, 2002.
22. Zhao, J. and Xu, B. Measuring Aspect Cohesion, *Proc. Intl. Conf. on Fundamental Approaches to Software Engineering (FASE'2004),* LNCS 2984, Springer, Barcelona, Spain, March 29-31, 2004), 54-68.
23. Zuse, H. History of Software Measurement. Disponível on-line em: http://irb.cs.tu-berlin.de/~zuse/metrics/History_00.html
24. Briand, L., Daly, J., Wüst, J. A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, 25(1): 91-121 (1999)