Design Patterns as Aspects: A Quantitative Assessment

Cláudio Sant'Anna, Alessandro Garcia, Uirá Kulesza,
Carlos Lucena, Arndt von Staa
PUC-Rio, Computer Science Department,
Software Engineering Laboratory, SoC+Agents Group
[claudios,afgarcia,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 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, such as coupling and cohesion. This paper presents a quantitative study that compares aspect-based and OO solutions for a representative set of design 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 resulted in higher coupling and more lines of code.

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 emerging development paradigms support improved modularization of the crosscutting concerns relative to the patterns.

Aspect-oriented software development (AOSD) [13, 19] is a promising paradigm to promote improved separation of concerns, leading to the production of software systems that are easier to maintain and reuse. AOSD is centered on the aspect notion as an abstraction aimed to modularize crosscutting concerns. Hence, aspect-oriented approaches are candidates to address the crosscutting property of design patterns. However, up to now there is only consensus that classical and obvious crosscutting concerns should be modularized as aspects, such as logging [2] and exception handling [14].

To the best of our knowledge, Hannemann and Kiczales [9] have developed the only systematic study that investigates 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 a representative set of the 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, although some aspect-oriented implementations of specific patterns resulted in higher coupling, more complex operations and more lines of code than object-oriented implementations.

The remainder of this paper is organized as follows. Section 2 introduces basic concepts in aspect-oriented programming. Section 3 presents our study setting, while giving a brief description of Hannemann and Kiczales' study. Section 4 presents the study results with respect to separation of concerns, and Section 5 presents the study results in terms of coupling, cohesion and size attributes. These results are interpreted and discussed in Section 6. Section 7 introduces some related work. Section 8 includes some concluding remarks and directions for future work.

2. Aspect-Oriented Software Development

Separation of concerns is a well-established principle in software engineering. A *concern* is some part of the problem that we want to treat as a single conceptual unit [19]. Concerns are modularized throughout software development using different abstractions provided by languages, methods and tools. However, these abstractions may not be sufficient for separating some special concerns found in most complex systems. These concerns have been called *crosscutting concerns* since they naturally cut across the modularity of other concerns. Aspect-oriented software development (AOSD) [13, 19] has been proposed as a technique for improving separation of concerns in the construction of OO software and supporting improved reusability and maintainability. AOSD supports the modularization of crosscutting concerns by providing the aspect abstraction that makes it possible to separate and compose them to produce the overall system.

AspectJ [2] is an aspect-oriented extension to the Java programming language. *Aspect* is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts classes in a program. An aspect is defined by an aspect declaration, which has a similar form of class declaration in Java. Similar to a class, an aspect can be instantiated and can contain attributes and methods, and it can be specialized in *subaspects*. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can introduce methods, attributes, and interface implementation declarations into types by using the *inter-type declaration* construct.

The essential mechanism provided for composing an aspect with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, exception handler, etc. Sets of join points may be represented by *pointcuts*. AspectJ provides various pointcut designators that may be combined through logical operators to build up complete descriptions of pointcuts of interest. An aspect can specify *advices* that are used to define some code that should be executed when a pointcut is reached. An advice is a method-like mechanism that consists of a

piece of code to be executed before, after, or around a pointcut. An AspectJ program can be divided into two parts: a base code part which includes classes, interfaces, and other language constructs for implementing the basic functionality of the program, and an aspect code part which includes aspects for modeling crosscutting concerns in the program. For further information about AspectJ, one can refer to [2].

3. Study Setting

This section describes the configuration of our empirical study. Our study supplements the Hannemann and Kiczales work that is presented in Section 3.1. Section 3.2 uses the Observer pattern to illustrate the crosscutting property of some design patterns. Section 3.3 describes the design patterns selected for our study as well as our assessment procedures. Section 3.4 introduces the metrics used in the assessment process.

3.1. Hannemann & Kiczales' Study

Several design patterns exhibit crosscutting concerns [9]. In this context, Hannemann and Kiczales have undertaken a study in which they have developed and compared Java [11] 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 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 "Subject" and "Observer" roles are defined in the Observer 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 Observer pattern, an operation that changes any "Subject" must trigger notifications to the corresponding "Observers"; in other words the act of notification crosscuts one or more operation in each "Subject" in the pattern.

In Hannemann and Kiczales' study, AspectJ implementations of the GoF patterns were generated to modularize the pattern roles. They have demonstrated modularity improvements 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 Observer pattern.

3.2. Example: The Observer Pattern

The Observer pattern [5] is one of the most popular design patterns. Object-oriented implementations of the Observer pattern usually add an attribute to all potential Subjects that stores a list of Observers interested in that particular Subject. When a Subject wants to report a state change to its Observers, it calls its own `notify` method, which in turn calls an `update` method on all Observers in the list. Figure 1 shows a concrete example of the Observer pattern in the context of a simple figure handling package. In such a program the Observer pattern would be used to update the screen whenever a figure element is changed. The shadowed methods contain code necessary to implement this instance of the Observer pattern. This shows that code for implementing this pattern is spread across the classes. All

participants (i.e. `Point` and `Line`) have to know about their role in the pattern and consequently have pattern code within them.

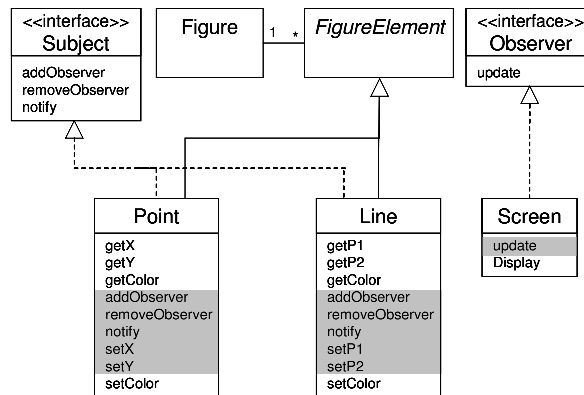


Figure 1. A simple graphical element that uses the Observer pattern in Java.

In this context, Hannemann and Kiczales have developed an AspectJ solution in which the code for implementing the Observer pattern is textually localized in two aspects: an abstract aspect, and one concrete extension of this aspect for each instance of the pattern. The abstract `Observer Protocol` aspect implemented by Hannemann and Kiczales is shown in Figure 2. The roles are realized as protected inner interfaces named Subject and Observer (line 3-4). Concrete extensions of the `ObserverProtocol` aspect assign the roles to particular classes. Implementation of the mapping from Subjects to Observers is realized using a weak hash map of linked lists to store the Observers for each Subject (line 6). Changes to the Subject-Observer mapping can be realized via the public `addObserver` and `removeObserver` methods (line 20-25) that concrete subaspects inherit. An abstract pointcut named `subjectChange` (line 27) and an abstract update method `updateObserver` (line 29) are defined. They are to be reified by instance-specific subaspects. Finally, the abstract aspect implements the update logic in terms of the pointcut `subjectChange` and the method `updateObserver`. This logic is contained in the after advice (line 31-36).

```
01 public abstract aspect ObserverProtocol {
02
03   protected interface Subject { }
04   protected interface Observer { }
05
06   private WeakHashMap perSubjectObservers;
07
08   protected List getObservers(Subject s) {
09     if (perSubjectObservers == null) {
10       perSubjectObservers = new WeakHashMap();
11     }
12     List observers =
             (List)perSubjectObservers.get(s);
13     if ( observers == null ) {
14       observers = new LinkedList();
15       perSubjectObservers.put(s, observers);
16     }
17     return observers;
18   }
19
20   public void addObserver(Subject s,
                                     Observer o) {
21     getObservers(s).add(o);
22   }
23   public void removeObserver(Subject s,
                                     Observer o) {
24     getObservers(s).remove(o);
25   }
26
27   abstract protected pointcut
              subjectChange(Subject s);
28
29   abstract protected void
              updateObserver(Subject s, Observer o);
30
31   after(Subject s): subjectChange(s) {
32     Iterator iter = getObservers(s).iterator();
33     while (iter.hasNext()) {
34       updateObserver(s,((Observer)iter.next()));
35     }
36   }
37 }
```

Figure 2. The ObserverProtocol Aspect.

Each concrete subaspect of `ObserverProtocol` defines one particular kind of observing relationship, in other words a single pattern instance. Figure 3 shows an instance of the Observer pattern involving the classes `Point`, `Line` and `Screen` implemented by the aspect `ColorObserver`. This subaspect defines that the classes `Point` and `Line` play the role of

Subject, and `Screen` plays the role of Observer. This is done using the `declare parents` inter-type declaration construct, which adds interfaces to the classes, to assign the roles defined in the abstract aspect (line 3-5). The subaspect also concretizes the `subjectChange` pointcut to define the operations on the subject that require updating the Observers (line 7-10). Furthermore, it defines how to update the observers by concretizing the `updateObserver` method (line 12-14). As we can see, in the AspectJ version of the Observer pattern, all code pertaining to the relationship between Observers and Subjects 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 aspect ColorObserver
                 extends ObserverProtocol {
02
03   declare parents: Point implements Subject;
04   declare parents: Line implements Subject;
05   declare parents: Screen implements Observer;
06
07   protected pointcut subjectChange(Subject s):
08     (call(void Point.setColor(Color)) ||
09      call(void Line.setColor(Color)) ) &&
10      target(s);
11
12   protected void updateObserver(Subject s,
                                   Observer o) {
13     ((Screen)o).display("Color change.");
14   }
15 }
```

Figure 3. An Observer instance.

3.3. The Assessed Patterns and Introduced Changes

Hannemann and Kiczales grouped the 23 GoF patterns by common features, either of the pattern structures or their AspectJ implementations. They have identified six groups based on their structural similarities. The groups are: (1) Observer, Mediator, Chain of Responsibility, Composite and Command, (2) Singleton, Prototype, Memento, Iterator and Flyweight, (3) Adapter, Decorator, Strategy, Visitor and Proxy, (4) Abstract Factory, Factory Method, Template Method, Builder, Bridge, (5) State and Interpreter, and (6) Façade.
In our study, we have decided to assess the implementation of six of the GoF patterns in order to have representative examples of each group. However, we have excluded the Façade pattern, since there is no difference between Java and AspectJ implementations of this pattern. Thus, we have chosen the following patterns: Observer, Mediator, Prototype, Strategy, State and Abstract Factory.
We have applied a metrics suite [16, 17] (Section 3.4) to both Java and AspectJ code of these six design patterns. First, we applied the metrics in Hannemann and Kiczales original code. Afterwards, we changed their implementation to add new participant classes to play pattern roles. For instance, in the Observer pattern implementation, four classes playing the "Subject" role were added, as the `Point` class in Figure 1 (Section 3.2); furthermore, four classes playing the "Observer" role were added, as the `Line` class in Figure 1 (Section 3.2). These changes were introduced because Hannemann and Kiczales' implementation encompasses 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. 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). Table 1 presents the roles of each studied pattern and the participant classes introduced to each pattern implementation example.

Table 1. The Selected Design Patterns and Respective Changes.

Selected Patterns	Pattern Roles	Introduced Participant Classes
Observer	Subject and Observer	4 Subject classes and 4 Observer classes
Mediator	Colleague and Mediator	4 Colleague classes
Prototype	Prototype	4 Prototype classes
Strategy	Context and Strategy	4 Context classes
State	Context and State	2 State classes
Abstract Factory	Product and Factory	2 Factory classes

3.4. The Metrics

In our study, a suite of metrics for *separation of concerns*, *coupling*, *cohesion* and *size* [16, 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]. Some of the object-oriented metrics [3] were tailored to be also applicable to aspect-oriented software. The original definition of each metric was 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. Table 2 also presents the sources for the metrics which the aspect-oriented metrics are based on. Refer to [6, 16, 17] for further details about the metrics.

In order to better understand the separation of concerns metrics, consider the example of the Observer pattern, shown in Figure 1 (Section 3.2). In that example, there is code related to the "Subject" role in the `Subject` interface and in the shadowed methods of `Point` class and `Line` class, i.e., this concern is implemented by one interface and two classes. Therefore, the value of the Concern Diffusion over Components metric (CDC) for this concern is three. Similarly, the value of the Concern Diffusion over Operations metric (CDO) for the "Subject" role is 13, since this concern is implemented by the three methods of the `Subject` interface, the five shadowed methods of the `Point` class, and the five shadowed methods of the `Line` class.

4. Results: Separation of Concerns

This section and Section 5 present the results of the measurement process. The data have been collected based on the set of defined metrics (Section 3.4). The goal is to describe the results through the application of the metrics before and after the selected changes (Section 3.3). The data was partially gathered by the CASE tool Together 6.0. It supports some metrics: LOC, NOA, WOC (WMPC2 in Together), CBC (CBO in Together), LCOO (LOCOM1 in Together) and DIT (DOIH in Together). 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].

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 5 presents the results regarding to coupling, cohesion, and size. The discussion about the interplay among all the results is concentrated in

Table 2. The Metrics Suite.

	Metrics	Definition	Based on
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.	Chidamber[3]
	Depth of Inheritance Tree (DIT)	Counts how far down in the inheritance hierarchy a class or aspect is declared.	Chidamber[3]
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.	Chidamber[3]
Size	Lines of Code (LOC)	Counts the lines of code.	Fenton [4]
	Number of Attributes (NOA)	Counts the number of attributes of each class or aspect.	Fenton [4]
	Weighted Operations per Component (WOC)	Counts the number of methods and advices of each class or aspect and the number of its parameters.	Chidamber[3]

Section 6. 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 3.3). 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 separation of concerns metrics (Section 3.4). 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. Likewise Hannemann and Kiczales, we treated each pattern role as a concern, because the roles are the primary sources of crosscutting structures. The pattern roles crosscut participant classes. The investigated patterns are classified into two groups: Group 1 and Group 2. Group 1 represents the patterns whose aspect-oriented solution provided better results (Section 4.1). Group 2 represents the patterns whose either the use of aspects did not impact the results or the OO solutions have shown as superior (Section 4.2).

4.1. Group 1: Observer, Mediator, Strategy and Prototype

The first group includes the Observer, Mediator, Strategy and Prototype patterns. All the aspect-oriented implementations of these patterns exhibited improved separation of concerns. Figures 4 and 5 depict the overall results of the AO and OO solutions for all the separation of

concerns metrics. Note that the graphics present the measures before and after the execution of the changes (Section 3.3). Figure 4 presents the CDC results, i.e. to what extent the pattern roles are isolated through the system components in both solutions. Figure 5a presents the CDO results – the separation degree of the pattern roles through the system operations – and Figure 5b illustrates the CDLOC results – the tally of concern switches (transition points) through the lines of code. In fact, all these graphics show significant differences in favor of the aspect-based solution. The improvement comes primarily from isolating the roles of the patterns in the aspects.

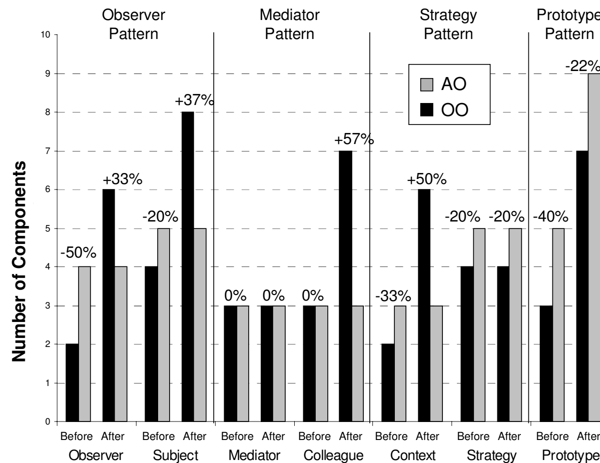


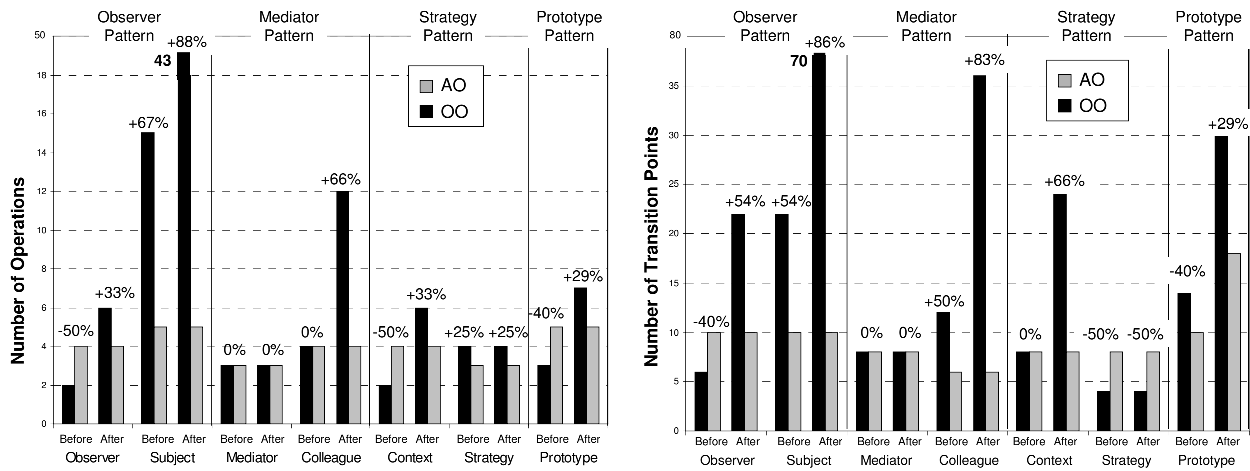
Figure 4. Concern Diffusion over Components (Group 1).

In general, the use of aspects led to inferior results before the application of the changes, but led to substantially superior outcomes after the implemented changes. After a careful analysis of Figures 4 and 5, we come to the conclusion that after the changes most aspect-oriented implementations isolated the roles 30% more than the object-oriented implementations. There are some cases where the difference is even bigger - the superiority of aspects exceeds 60%, an interesting fact given that in these cases the values were equivalent in both object-oriented and aspect-oriented solutions before the implementation of the changes. For the "Subject" and "Colleague" roles (Figure 5), 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 event handling mechanism (Observer pattern) 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.

After the changes, the majority of the pattern roles required more components in the definition of the OO solution than in the AO solution (Figure 4). For example, the definition of the "Colleague" role required 7 classes, while only 3 aspects were able to encapsulate this concern. It is equivalent to 57% in favor of the aspect-oriented design for the Mediator pattern. In fact, most roles were better modularized in the AO solution: Observer (2 against 4), Subject (5 against 8), Context (3 against 6), and Colleague (3 against 7). The results were similar to the separation of concerns over operations (Figure 5a) and lines of code (Figure 5b).

An additional interesting observation is that the absolute number of components (CDC), operations (CDO) and transition points (CDLOC) in the aspect-oriented solutions did not vary after the modifications, except for the Prototype role. For example, the Context role required three components before the changes and the same three components after the changes (Figure 4). The same behavior is observed in the measures of operations and transition points (Figure 5). For the Context role, 3 operations and 8 transition points were used both before

and after the modifications. This reflects the suitability of aspects for the complete separation of the roles associated with the four patterns in this category. When new classes are introduced, they do not need to implement pattern-related code. The problem with the Prototype role is that the declaration of the implementation of the `Cloneable` interface (that is a pattern-specific concern) is amalgamated to the implementation of business classes in the AO solution. However, this problem is not implicit to the use of aspects, but the specific implementation of Hannemmann and Kiczales [9] (Section 3.1).



(a) Concern Diffusion over Operations **(b)** Concern Diffusion over Lines of Code

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

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 the 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 3.3) 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.

4.2. Group 2: State and Abstract Factory

This group includes the State and Abstract Factory patterns. Figures 6 and 7 depict the overall results of separation of concerns in the AO and OO solutions for the patterns in this group. Figure 6 presents the CDC results, Figure 7a presents the CDO results, and Figure 7b illustrates the CDLOC results. Overall, no significant difference was detected in favor of a specific solution; the results were mostly similar for the aspect-oriented and OO implementations of these patterns. This observation is mainly supported by CDO (Figure 7a) and CDLOC (Figure 7b). As those 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.

The outcomes of this group were highly different from the ones obtained in group 1 (Section 4.1) 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 and classes.

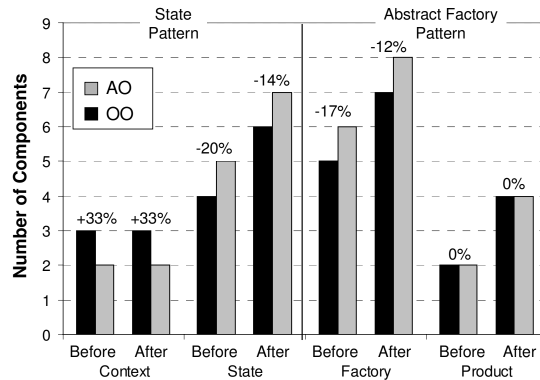
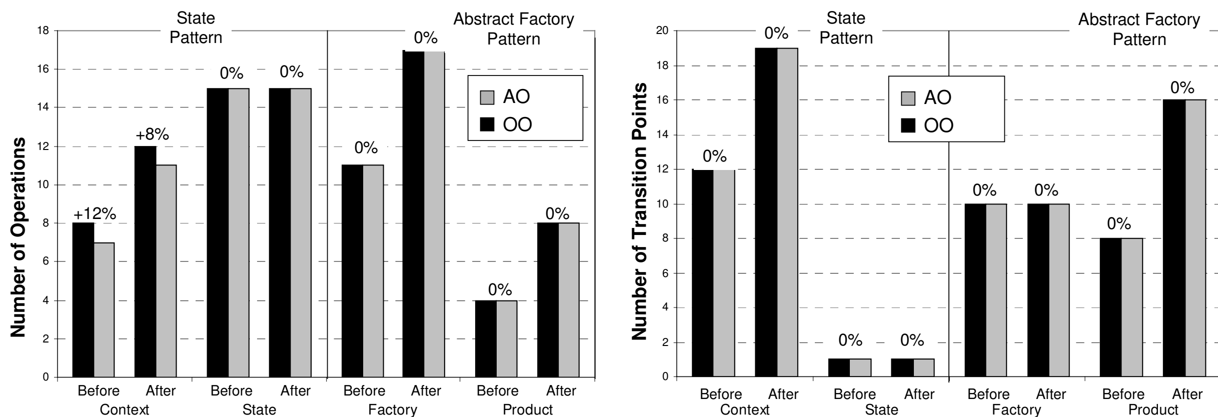


Figure 6. Concern Diffusion over Components (Group 2).

There were some differences detected in the evaluation of the solutions, such as in the State pattern. The aspect-oriented design of the State role presented inferior results in the CDC measures; it required 20% more components than the OO solution before the changes, and 14% more components after the changes (Figure 6). The reason for this difference is that the AO solution has an additional aspect for modularizing the transition of states. On the other hand, the OO design of the Context role involved 33% more components than the AO design before as well as after the changes. The object-oriented solution has an interface, which defines a method to support the state transition; the aspect-oriented implementation does not require this interface.

The sole difference observed in the Abstract Factory pattern was related to the number of components used to modularize the Factory role. This role was more localized in the OO design, although the difference consists of only one component when compared with the AO design (Figure 6). The aspect-based design has one additional aspect that provides a default implementation of the factory methods defined in the AbstractFactory interface, which is attached to this interface on the basis of inter-type declarations [9].



(a) Concern Diffusion over Operations **(b)** Concern Diffusion over Lines of Code

Figure 7. Separation of Concerns over Operations and Lines of Code (Group 2).

5. 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 (Section 3.3), similarly to the previous section. The results represent the tally of metric values associated with all the classes

and aspects for each pattern implementation. The patterns were classified into 4 groups according to the similarity in their measures.

5.1. Group 1: Observer and Mediator

For the Observer and Mediator patterns, the aspect-oriented design and implementation manifest several closely related benefits. As the changes were accomplished, the aspect-oriented solution exhibited superior results with respect to operation complexity (WOC), lines of code (LOC), number of attributes (NOA), cohesion (LCOO) and inter-component coupling (CBC). The differences were mostly more than 10% in favor of the aspect-oriented solution for both design patterns.

Figure 8 shows the graphic with results for the Observer pattern. In the aspect-oriented implementation of this pattern, the major improvements were detected in the LOC, LCOO and NOA measures. The use of aspects led to a 27% reduction of LOC in relation to the OO code. Thus aspects solve the problem of code replication (Section 3.2) related to the implementations of the method `notifyObservers()`. The cohesion in the AO implementation was 62% higher than the OO implementation because the latter incorporates a number of classes that play the Subject and Observer roles and, as a consequence, implement role-specific methods that in turn do not access the attributes of the classes. In the aspect-oriented design, these methods are localized in the aspects that implement the roles, increasing the cohesion of both classes and aspects. The tally of attributes in the OO implementation was respectively 17% and 19% higher than in the AO code before and after the introduction of new components into the implementations. In the OO solution, the "subject" classes need attributes to hold references to their "observer" classes; these attributes are not required in the aspect-oriented design.

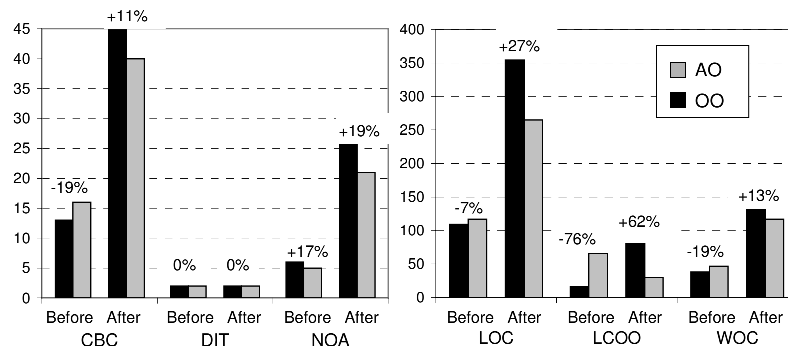


Figure 8. The Observer Pattern: Coupling, Cohesion and Size.

5.2. Group 2: Prototype and Strategy

The measures gathered from implementations of the Prototype and Strategy patterns were mostly similar. In general, the OO implementation provided better results, mainly with respect to coupling between components (CBC), complexity of operations (WOC), and lines of code (LOC). Figure 9 shows the results for the Strategy pattern. Note that inheritance was the only factor that was not affected by the use of aspects. In the OO solution, class inheritance is used to implement the variability of the strategies [5]; in the AO solution, aspect inheritance is used to define a specific strategy protocol [9]. As a result, the maximum DIT was two for both solutions.

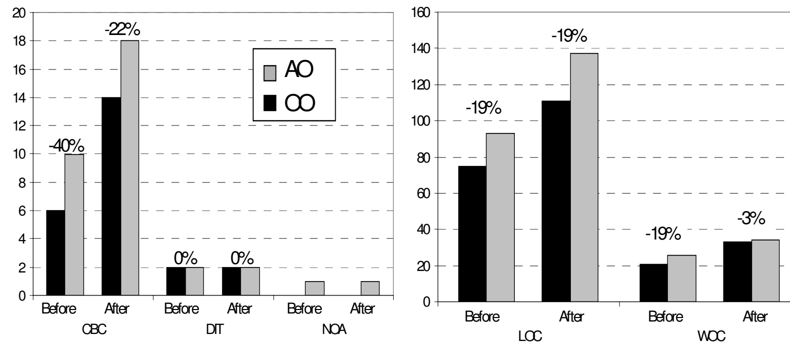


Figure 9. The Strategy Pattern: Coupling, Cohesion and Size.

The system coupling was substantially higher in the aspect-oriented solution. For example, the difference between the solutions was 4 units in favor of the OO design of the Strategy pattern. As new components were added to both designs, the difference remained constant (Figure 9). It happens because the aspects, which implement the pattern roles, are coupled to the business classes. The coupling of the business classes introduced into the AO implementation is zero since they are not aware of the presence of aspects. However, the coupling of the aspect, which implements the strategy protocol, increases linearly. Table 3 illustrates this problem: the coupling of the `SortingStrategy` aspect is 7, while the coupling of the `SortingStrategy` class is 0. This table also shows that LOC was higher in the aspect-oriented solution. The aspects require more lines of code as the changes are carried out. For example, the `SortingStrategy` aspect has 17 LOC, while the `SortingStrategy` class has 4 LOC. Cohesion is not a valid metric for both patterns because most classes and aspects do not have internal attributes. The differences in the NOA measures are not significant (Figure 9). In both patterns, WOC measures decreases as the changes are implemented. However, the OO implementation remains superior.

Table 3. The Strategy Pattern: Measures Per Component.

Object-Oriented Solution						**Aspect-Oriented Solution**					
Class	**CBC**	**DIT**	**LOC**	**NOA**	**WOC**	**Class(C)-Aspect(A)**	**CBC**	**DIT**	**LOC**	**NOA**	**WOC**
BubbleSort	1	2	18	0	6	BubbleSort (C)	1	1	18	0	6
LinearSort	1	2	21	0	6	LinearSort (C)	1	1	21	0	6
Main	6	1	27	0	2	Main (C)	8	1	42	0	4
Sorter	2	1	17	0	5	Sorter (C)	0	1	6	0	2
Sorter1	1	1	6	0	3	Sorter1 (C)	0	1	6	0	2
Sorter2	1	1	6	0	3	Sorter2 (C)	0	1	6	0	2
Sorter3	1	1	6	0	3	Sorter3 (C)	0	1	6	0	2
Sorter4	1	1	6	0	3	Sorter4 (C)	0	1	6	0	2
SortingStrategy	0	1	4	0	2	SortingStrategy (A)	7	2	17	0	3
						StrategyProtocol (A)	1	1	9	1	5
TOTAL	**14**	**2**	**111**	**0**	**33**	**TOTAL**	**18**	**2**	**137**	**1**	**34**

5.3. Group 3: State

The aspect-oriented implementation of the State pattern was superior in three measures: coupling, cohesion and complexity of operations (Figure 10). On the other hand, the OO implementation provided better results in two measures: number of attributes and lines of code. 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 overcame in the aspect-oriented solution because the aspects modularize the state transitions (Figure 11), minimizing the system-level coupling. Figure 11 shows that the coupling in the OO solution is seven because each “state” class needs to have references to the other “state” classes.

The OO solution produced more complex operations (WOC measures) 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 aspect-oriented design because a unique aspect is responsible for managing the transitions between states.

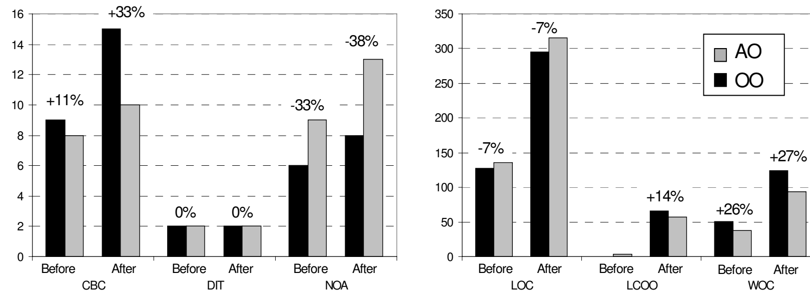


Figure 10. The State Pattern: Coupling, Cohesion and Size.

From the NOA metric 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. This difference increases as new "state" elements are added to the system (Figure 11). The OO implementation provided fewer LOCs in spite of the "state" classes have fewer lines of code. However, 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) one additional advice associated with methods of "state" classes.

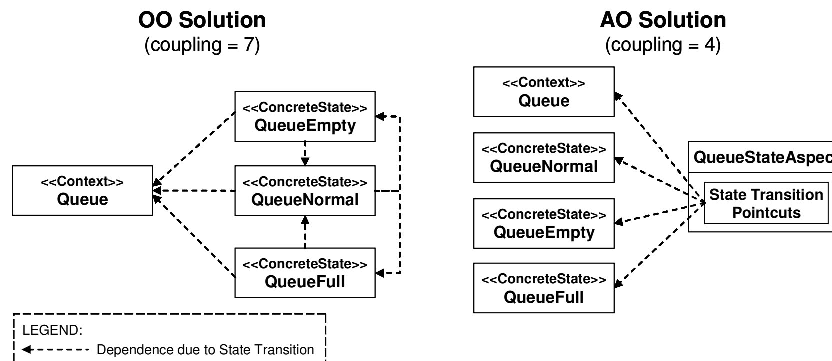


Figure 11. Coupling in the State Pattern: OO vs. AO.

5.4. Group 4: Abstract Factory

No significant difference was detected in the AO and OO implementations of the Abstract Factory pattern. As illustrated in Figure 12, the measures were similar with respect to cohesion (LCOO), inheritance (DIT), number of attributes (NOA), and complexity of operations (WOC). The differences detected in LOC and coupling measures are not significant. The reason for such results is that the OO and AO designs are very similar. The difference relies on an aspect that introduces default behavior to the methods of the interface that plays the Abstract Factory role [9].

6. Discussion

This section provides a more general analysis (Section 6.1) of the previously observed results in Sections 4 and 5, and discussions about the constraints on the validity of our empirical evaluation (Section 6.2).

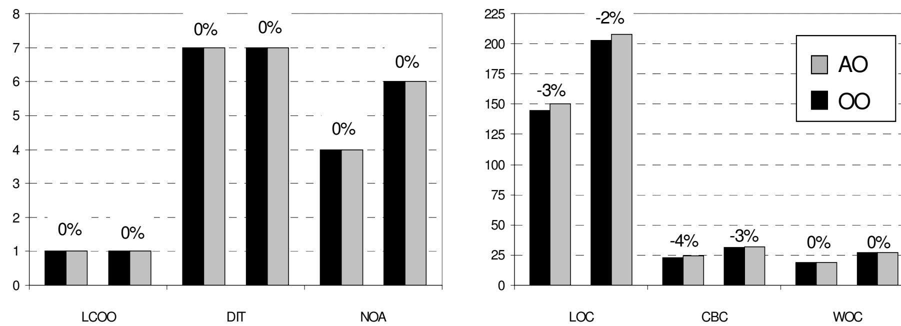


Figure 12. The Abstract Factory Pattern: Coupling, Cohesion and Size.

6.1. General Analysis

Separation of Concerns. As presented in Section 4.1, the AspectJ implementations of the Observer, Mediator, Prototype and Strategy patterns have shown better results in terms of the metrics of separation of concerns. Indeed, these results have confirmed that their AspectJ implementations manifest modularity improvements, which in turn was also observed by Hannemann and Kiczales in terms of locality, transparency composability and (un)pluggability. In addition, the results about the Abstract Factory pattern (Section 4.2) support Hannemann and Kiczales' claims that this pattern did not benefit from AspectJ implementation. However, AspectJ implementation of the State pattern has not shown relevant separation of concern improvements, which contradicts some Hannemann and Kiczales' claims about this pattern.

Inseparable Concerns. Sometimes the pattern is expressed separately as an aspect, but it remains non-trivial to specify how this separate aspect 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 aspect-oriented implementation of the Strategy pattern provided better separation of the pattern-related concerns (Section 4.1). However, although the aspect-oriented solution isolates the pattern roles in the aspects, it resulted in higher complexity in terms of coupling (CBC), operation complexity (WOC), and lines of code (LOC), as described in Section 5.2. In this context, there are some cases where the separation of the pattern-related concerns lead to more complex solutions.

Reducing Coupling. Based on the interplay of the results in Section 4 and 5, we can conclude that the use of aspects provided better results for the patterns with high interaction between the roles in their original definition. The Mediator, Observer, State 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, since the aspect code modularizes the collaboration protocol between the pattern roles. Figure 11 illustrates how the aspect was used to reduce the coupling of the OO solution of the State pattern. In fact, the use of aspects did not succeed in the patterns whose roles are not highly interactive. This is the case for the Abstract Factory, Prototype and Strategy patterns (Sections 5.2 and 5.4).

Multi-Dimensional Analysis. Hannemann and Kiczales [9] have centered their analysis only on separation of concerns, and how the achieved separation helps to improve high-level qualities of the pattern and the application, such as *(un)pluggability* and *composability*. Lopes [15] has also carried out a case study that rests only on separation of concerns as assessment criteria. However, based on the discussion above, we found that the analysis of other software dimensions or attributes, such as coupling and internal complexity of operations, are

extremely important to compare aspect-oriented and object-oriented 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 in terms of other software attributes.

Refactoring Aspect-Oriented Solutions. Based on the measurements, we have found that some problems in the aspect-oriented solutions are not related to the aspect paradigm itself, but to some design or implementation decisions taken in the Hannemmann and Kiczales implementation (Section 3.1). For example, the problem related to the aspect-oriented solution for the Prototype pattern occurred because the developers have left the declaration of the Cloneable interface in the description of the classes (Section 4.1). However, this solution can be refactored in order to improve the separation of concerns, overcoming the problem detected in Section 4.1. In this sense, we can conclude that 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.

6.2. Study Constraints

Concerning our experimental assessment, there is one general type of criticism that could be applied to the used software metrics (Section 3.4). 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. 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 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 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. 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.

7. Related Work

There is a few 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. 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 a similar result for the Strategy pattern (Section 5.2).
Soares et al [18] have reported their experience using AspectJ to implement distribution and persistence aspects in a web-based information system. They have implemented the system in Java and restructured it with AspectJ. They have argued that the AspectJ implementation of the system bring significant advantages with the corresponding pure Java implementation. Walker et al. [20] have conducted two exploratory experiments to study the increased modularization provided by AspectJ. In these experiments, they have compared the performance of a small number of participants working on two common programming tasks: debugging and changing. However, these studies are qualitative assessments, which are not focused on the use of aspects for modularizing pattern-related concerns.
Garcia et al. [7] have presented a quantitative study designed to compare the maintenance and reuse support of a pattern-oriented approach and an aspect-oriented approach for a multi-agent system development. They used an assessment framework that includes the same metrics suite used in our study. The results have shown that, for the system at hand, the aspect-oriented approach allowed the construction of this system with improved structuring for reuse and maintenance of the multi-agent system concerns. The use of aspects resulted in better separation of concerns, lower coupling between its components (although less cohesive), and fewer lines of code. However, their study is also not focused on the use of aspects to isolate the crosscutting concerns relative to classical design patterns.

8. Final Remarks and Future Work

This paper presented a comparative study comparing the aspect-oriented and object-oriented implementations of a representative set of 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 comperations and more LOCs in the aspect-oriented solutions. Another important conclusion of this study is that SoC 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 Hannemann and Kiczales' 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. This work has been partially supported by CNPq under grant No. 140214/2004-6 for Cláudio, and under grant No. 140252/2003-7 for Uirá. Alessandro was supported by FAPERJ under grant No. E-26/150.699/2002. The authors are also supported by ESSMA Project under grant 552068/2002-0 and by the art. 1st of Decree number 3.800, of 04.20.2001.

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, February 2004
2. AspectJ Team. "The AspectJ Programming Guide". http://eclipse.org/aspectj/.
3. Chidamber, S., Kemerer, C. "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, 20 (6), June 1994, pp. 476-493.
4. Fenton, N., 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, Computer Science Department, 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, C., Lucena, C. "Engineering Multi-Agent Systems with Aspects and Patterns". J. of the Brazilian Computer Society, 1 (8), July 2002, pp 57-72.
9. Hannemann, J., Kiczales, G. "Design Pattern Implementation in Java and AspectJ". Proceedings of OOPSLA'02, November 2002, pp. 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., Murphy, G. "Atlas: A Case Study in Building a Web-based learning environment using aspect-oriented programming". Proc. of OOPSLA'99, November 1999.
13. Kiczales, G. et al. "Aspect-Oriented Programming". Proceedings of ECOOP'97, LNCS (1241), Springer-Verlag, Finland, June 1997, pp. 220-242.
14. Lippert, M., Lopes, C. "A Study on Exception Detection and Handling Using Aspect-Oriented Programming." Proc. ICSE'00, Limerick, Ireland, May 2000, pp. 418 - 427.
15. Lopes, C. "D: A Language Framework for Distributed Programming". PhD Thesis, Northeastern University, 1997.
16. Sant'Anna, C. "Maintainability and Reusability of Aspect-Oriented Software: An Assessment Framework". Masters Thesis, PUC-Rio, March 2004 (in Portuguese).
17. Sant'Anna, C. et al. "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework". Proc. of SBES'03, Manaus, Brazil, October 2003, pp. 19-34.
18. Soares, S., Laureano, E., Borba, P. "Implementing Distribution and Persistence Aspects with AspectJ". Proceedings of the OOPSLA'02, pp. 174-190.
19. Tarr, P. et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". Proceedings ICSE'99, Los Angeles, USA, May 1999, pp. 107-119.
20. Walker, R., Baniassad, E., Murphy, G. "An Initial Assessment of Aspect-oriented Programming". Proceedings of ICSE'99, Los Angeles, USA, May 1999, pp. 120-130.