A Multiparadigm Study of Crosscutting Modularity in Design Patterns

Martin Kuhlemann[1], Sven Apel[2], Marko Rosenmüller[1], and Roberto Lopez-Herrejon[3]

[1] School of Computer Science, University of Magdeburg
{mkuhlema,rosenmue}@ovgu.de
[2] Department of Informatics and Mathematics, University of Passau
apel@uni-passau.de
[3] Computing Laboratory, University of Oxford
rlopez@comlab.ox.ac.uk

Abstract. Design patterns provide solutions to recurring design problems in object-oriented programming. Design patterns typically crosscut class boundaries so previous work aimed at improving modularity of their implementations. A series of recent studies has focused on aspect-oriented programming while other crosscutting techniques such as collaboration-based designs have remained unexplored. In this paper, we address this limitation by presenting a qualitative case study based on the Gang-of-Four design patterns comparing and contrasting mechanisms of representative languages of collaboration-based designs (Jak) and aspect-oriented programming (AspectJ). Our work yields guidelines for using both paradigms when implementing design patterns exploiting their relative strengths.

Keywords: Aspect-oriented programming, collaboration-based design, design patterns, crosscutting modularity.

1 Introduction

Design patterns are customizable and reusable solutions for recurring problems in object-oriented applications [15]. The implementation of patterns commonly involves or *crosscuts* multiple classes and interfaces that play different roles in a pattern. The crosscutting nature of design patterns has attracted the attention of *aspect-oriented programming (AOP)* advocates who conduct research on techniques and tools to modularize *crosscutting concerns* [25] for the development of customizable software [43, 31, 26]. A core tenet of AOP is that crosscutting concerns, when poorly modularized, lead to code *tangling* (a module contains code of multiple concerns) and *scattering* (the implementation of a concern is spread across multiple modules) [25].

Several studies highlight the relative advantages of AOP over traditional *object-oriented programming (OOP)* for implementing design patterns [22,13,16, 17]. However, AOP is not the only technology capable of modularizing crosscutting concerns. Research on *collaboration-based designs (CBD)* predates AOP [36,

R.F. Paige and B. Meyer (Eds.): TOOLS EUROPE 2008, LNBIP 11, pp. 121–140, 2008.

```
1  public class Point {
2   public int pos;
3   public void resetPos(){
4    this.pos = 0;
5   }
6   public void click(){
7    Locker.lock();
8    this.resetPos();
9    Locker.unLock();
10  }
11 }
```

```
12 public aspect ObserverAspect {
13  protected pointcut observedCalls():
      call(* Point.*(..));
14  before():observedCalls(){
15   System.out.println();
16  }
17  public HashMap Point.obs;
18  public HashMap Point.getObs(){
19   return obs;
20  }
21  declare parents: Point implements
      SubjectInterface;
22 }
```

Fig. 1. AspectJ concepts used in the case study

8,41] and embraces multiple technologies that extend OOP to attain goals similar to those of AOP [6]. In CBD, a class is divided into code units that modularize the roles played by that class. Collaborations in CBD modularize roles of different classes that collaborate to perform a task.

In this paper, we compare and contrast AspectJ[1] and Jak[2] implementations of the 23 *Gang-of-Four (GoF)* design patterns [15]; AspectJ is an AOP language extension for Java [25] and Jak is a CBD language extension for Java [7]. Both languages are representatives of their paradigms and have been used in several studies, e.g., [6, 22, 16, 17, 13, 26]. For our qualitative comparison, we devise two basic modularity criteria: cohesion and reusability. We measure their relative support in the AspectJ and Jak pattern implementations. Subsequently, we optimize the implementations with respect to each of the two criteria and repeat this evaluation. Based on *all* implementations (initial and optimized), we analyze AspectJ's and Jak's crosscutting mechanisms and offer guidelines for choosing aspects or collaborations in concrete contexts. We show that both criteria cannot fully be met simultaneously using one of either paradigms; our study reveals individual strengths of both approaches and outlines ways for their combination.

2 Background

In the following, we describe the basics of AspectJ and Jak, the categories used for our evaluation, and the criteria analyzed in the AspectJ and Jak implementations.

2.1 Language Mechanisms Used in the Case Study

AspectJ. The key abstraction mechanism of AspectJ is an aspect [24, 28]. An aspect is a class-like entity that includes pointcuts and pieces of advice. Pointcuts select join points (well-defined events during program execution) from a join point model of a base application. Advice is code executed at these join points.

[1] http://www.eclipse.org/aspectj/
[2] http://www.cs.utexas.edu/users/schwartz/

Advice may be bound to *abstract* pointcuts that do not select join points directly but get overridden in inheriting aspects to select join points.

Figure 1 depicts a pointcut (Line 13) and a piece of advice (Lines 14-16) that extend all calls to methods of class *Point*. Inter-type declarations introduce new methods or fields into existing classes and declare errors statically if user-defined constraints are violated. In Lines 17-20 of Figure 1, *ObserverAspect* introduces the field *obs* and the method *getObs* into class *Point*. Aspects can declare a class to implement interfaces or to extend a superclass with *declare parents* clauses. In Line 21, the aspect declares the class *Point* to implement the interface *SubjectInterface*.

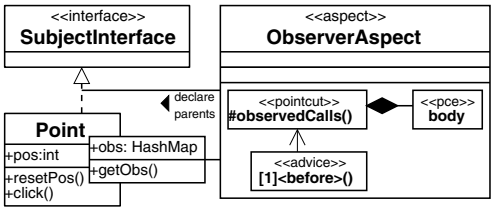


Fig. 2. Graphical notation of an aspect

```
1  public class Point {
2    public int pos;
3    public void resetPos(){
4      this.pos = 0;
5    }
6    public void click(){
7      Locker.lock();
8      this.resetPos();
9      Locker.unlock();
10   }
11 }
```

(a) Collaboration: *Base*

```
12 refines class Point implements
     SubjectInterface {
13   public void resetPos(){
14     System.out.println();
15     Super.resetPos();
16   }
17   public HashMap obs;
18   public HashMap getObs(){
19     return obs;
20   }
21 }
```

(b) Collaboration: *Observer*

Fig. 3. Jak concepts used in the case study

Figure 2 depicts *ObserverAspect* of Figure 1 in an extended UML notation.[3]

Jak. A *collaboration* is a set of objects (hence the crosscutting) and a protocol that determines how the objects interact. The part of an object that enforces or implements the protocol in a collaboration is called a *role* [37,41]. Collaborations can be implemented using several techniques, like inheritance, encapsulation, or polymorphism. Layers are abstractions for collaborations in Jak and superimpose standard classes with *mixin*

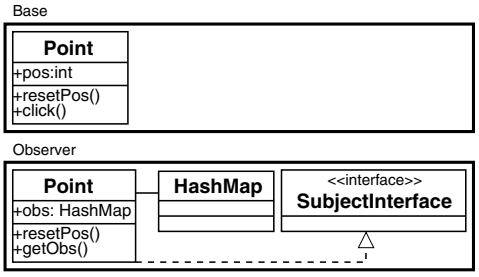


Fig. 4. Graphical notation of collaborations

[3] To the best of our knowledge, there is no commonly agreed UML notation for aspects [21, 18, 38]. For this paper, we use the one of Han et al. [21].

Table 1. Pattern categories

Category Name	Design Patterns
No General Accessible Interfaces for Classes	Chain of Responsibility, Command, Composite, Mediator, Observer
Aspects as Object Factories	Flyweight, Iterator, Memento, Prototype, Singleton
Language Constructs	Adapter, Decorator, Proxy, Strategy, Visitor
Multiple Inheritance	Abstract Factory, Bridge, Builder, Factory Method, Template Method
Scattered Code Modularized	Interpreter, State
No Benefit From AspectJ Implementation	Facade

classes [9] that can add new fields and methods to the superimposed class. In Figure 3, the Jak collaboration *Observer* (Fig. 3b) contains a mixin class *Point* which adds a field *obs* (Line 17) and a method *getObs* (Lines 18-20) to the class *Point* of the collaboration *Base* (Fig. 3a).

In Jak, *method refinements* extend methods via overriding, similar to method overriding in Java subclasses. In Figure 3, the collaboration *Base* (Fig. 3a) contains a class *Point*. Method *resetPos* of *Point* of the collaboration *Observer* (Fig. 3b) refines method *resetPos* of class *Point* of the collaboration *Base* by overriding. This overriding method calls the overridden refined method using *Super* (Fig. 3b, Line 15) and adds further statements (Line 14). Figure 4 depicts the collaborations *Base* and *Observer*.[4]

2.2 Categories of Design Patterns

Hannemann et al. [22] defined six categories of design patterns based on the benefits of AspectJ implementations compared to their Java counterparts (see Tab. 1). We use their categories to focus our analysis. The categories consider crosscutting support and allow us to highlight differences between AspectJ and Jak.

No General Accessible Interfaces for Classes. The patterns of this category do not have an interface accessible from clients. Clients that use a class or a set of classes are neither affected nor aware of whether a design pattern of this category is applied or not. Consequently, the interfaces of these patterns mainly structure the code instead of providing reusability of code. That is, these patterns can be implemented entirely with aspects, in which roles are bound to classes through pointcuts. The patterns in this category are Chain Of Responsibility, Command, Composite, Mediator, and Observer.

Aspects as Object Factories. The patterns of this category control access to objects. Factory methods provide access for clients on an aspect instance (using *aspectOf*) or on a class whose methods are advised or introduced by an aspect. That is, an aspect may advise an object-creating method and provide object instances based on the parameters passed to the extended method. Note that

[4] We are not aware of a commonly agreed UML notation for collaborations. In this paper, we use the notation of [6, 4, 5].

patterns in this category define only one role. The patterns in this category are Flyweight, Iterator, Memento, Prototype, and Singleton.

Language Constructs. The patterns of this category replace OO constructs by AO language constructs, e.g., methods by inter-type declarations or advice. Thereby, AO implementations sometimes do not completely provide the same capabilities as the OO counterparts. For example, the AO implementation of Decorator cannot be applied dynamically to the decorated object like the OO counterpart [22]. However, implementing patterns with AO language constructs sometimes simplifies the design [11,12]. The patterns in this category are Adapter, Decorator, Proxy, Strategy, and Visitor.

Multiple Inheritance. In the pattern implementations of this category, aspects detach implementations from abstract classes which become interfaces. The pattern-related interfaces are assigned to classes and are extended by the aspects. This results in a limited form of multiple inheritance [22]. The developer can assign different role implementations to a single class by replacing the abstract classes by interfaces that can be extended by aspects. The patterns in this category are Abstract Factory, Bridge, Builder, Factory Method, and Template Method.

Scattered Code Modularized. This category includes patterns that scatter code across different classes in their OO implementation. This code can be modularized using AspectJ, effectively decoupling the set of classes from the communication protocol of the patterns. The patterns in this category are Interpreter and State.

No Benefit from AspectJ Implementation. Patterns in this category do not differ in their OO and AO implementations. The only pattern in this category is Facade.

2.3 Evaluation Criteria

For our case study, we use two criteria that are common in software-engineering, reusability and cohesion, and adapt their definition to our analysis of crosscutting modularity.

Reusability. Reusability allows to use a piece of software in a context different than that it has been developed for [10, 1]. A reusable piece of software must provide benefits when reused but also must include as less design decisions as possible, because the decisions can conflict with decisions made in the new context [10, 40, 33].

Design patterns define reusable designs but do not provide reusable implementations [15,34,11]. To attain code reuse, each pattern ideally is implemented by (1) an abstract piece of code that is common to all implementations of the pattern across different applications and (2) application-specific binding code [22, 20] – this way, even concerns different from design patterns should be implemented [23]. Both conceptual parts of a pattern implementation offer possibilities for reuse, as we explain next.

(a) Free-standing units (b) Nested units

Fig. 5. Different modularizations of one application

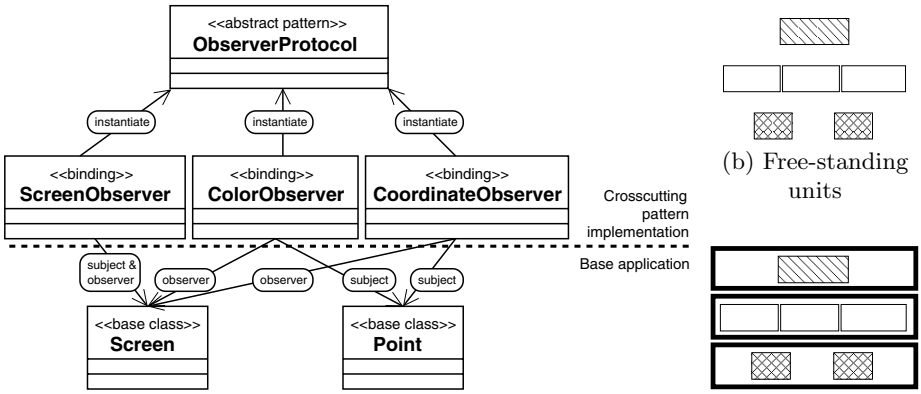


Fig. 6. Principle Observer implementation and two modularizations

(1) Abstract-Pattern-Reusability is the capability of reusing an abstract design pattern implementation across different applications (Inter-Application-Reusability), that do not share code otherwise. A pattern implementation can only be reused across applications if its implementation is decoupled from other parts of a software. That is, the pattern implementation cannot be reused if it includes application-specific code or if it depends on the application by reference.

(2) Binding-Reusability is the capability of reusing application-specific binding code across different variants of a single application (Intra-Application-Reusability), i.e., across software products that do share a common base. Binding code cannot be reused if a role of a pattern shall be bound to different classes in different variants but all roles of that pattern are bound within one closed code unit[5] [33, 23]. Open code units can be adapted and allow independent reuse of their nested parts.

[5] A closed code unit exhibits a well-defined stable description while open units are adaptable [33]. An open code unit is available for extensions or manipulations; a closed unit solely can be used as it is without any adaptation. Please note that this notion of open code units should not be confused with open modules of Aldrich [3] that expose join points explicitly.

In Figure 5, we show two designs, in which code units are depicted with shapes (circle, square, or rounded quare). The *free-standing* units of Figure 5a are reusable because every unit may be reused independently of other units (e.g., other circles). The *nested* units of Figure 5b (e.g., circles) can only be reused independently if the surrounding composite units grouping them are not closed. That is, nesting code units may effect their reusability.

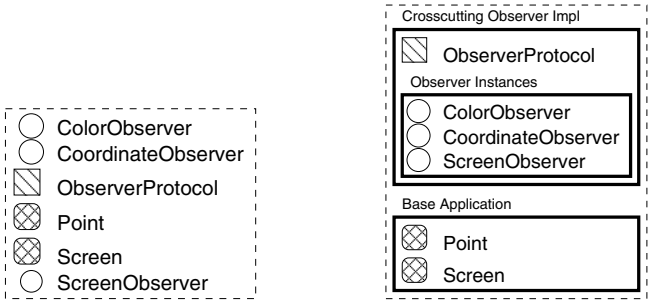
In Figure 6a, we depict the Observer design pattern in UML notation that consists of an abstract pattern implementation (top), binding units (middle), and base application units (bottom); in Figures 6b and 6c, we show different modularizations of this implementation with free-standing code units (Fig. 6b) and nested code units (Fig. 6c). Striped boxes represent abstract code units, white boxes represent binding units, and cross hatch boxes represent base application units. For optimal reuse, every binding code unit, e.g., *ScreenObserver*, and every abstract pattern code unit, e.g., *ObserverProtocol*, should be reusable independently. This is possible if these units are free-standing (cf. Fig. 6b). In case the units are nested (cf. Fig. 6c), they can only be reused if the surrounding composite code unit is open. If the composite units of Figure 6c are closed, their nested code units cannot be reused independently.

Cohesion. *Cohesion* is the grouping of a set of programming constructs that implement a specific task into a single modular unit and referring to the collective group using a name [31].[6] A cohesive code unit implements a specific task completely and, thus, is valuable when being reused [10, 40]. Low cohesive structures of code units increase the complexity of an application [39, 30, 29] because they contain an unstructured set of unrelated units. Hence, semantically related code units cannot be referenced by name in a low-cohesive structure because they are not grouped.

The structure of free-standing code units of Figure 5a is not cohesive because units of different kinds and concerns (depicted by different shapes) are intermixed and sets of semantically related code units cannot be referenced by name. The composite units of Figure 5b are cohesive because every one of these units in turn composes different code units of one concern (here: different atomic code units, e.g., all circles) and, thus, implements this concern completely – every composite unit can be referred to by name.

Figures 6b and 6c show different modularization approaches (with free-standing code units and with nested code units) for the Observer pattern of Figure 6a. Based on our definition of cohesion, nested modularization (as in Fig. 6c) is more cohesive than free-standing modularization (as in Fig. 6b) because it separates unrelated code units (binding, abstract, and base units) and groups semantically related units (e.g., different binding units). Moreover, through nesting, sets of related code units (e.g., all binding units of that pattern or all crosscutting units) can be referenced by name.

[6] We use this extended definition of cohesion to highlight the need for module names. This is important especially for modularizing crosscutting concerns because these are designed and coded separately from the classes they crosscut [26].



(a) AspectJ impl. (b) Jak impl.

Fig. 7. AspectJ and Jak implementation of Observer

3 Case Study

In our case study, we reimplemented the AspectJ pattern implementations of Hannemann et al. in Jak.[7] To transform AspectJ to Jak implementations, we used the guidelines presented in [27]. We now discuss the implementations of all 23 GoF design patterns with respect to our two criteria (reusability and cohesion) and our six categories. Firstly, we analyze the initial AspectJ and Jak implementations and then we discuss the implementations optimized towards reuse and cohesion. Due to space limitations, we focus our discussion on representative patterns in each category.

3.1 Initial Implementations

No General Accessible Interfaces for Classes

Cohesion: AspectJ and Jak implementations differ in cohesion.
Binding-reusability: Jak binding implementations are more reusable than AspectJ binding implementations.
Abstract-pattern-reusability: Abstract pattern implementations using AspectJ are more reusable than their Jak counterparts.

Patterns in this category do not need an interface for clients, i.e., they can be implemented abstract and bound to an application in AspectJ. Jak does not provide a mechanism to bind implementations to arbitrary methods directly like abstract pointcuts, but the Jak pattern implementations have been bound to the base code using method refinements. Method refinements are very application-specific in contrast to abstract pointcut bindings, which reduces support for abstract-pattern-reusability for all Jak implementations. However, Jak refinements of bindings can be reused across variants of the application because the composite collaborations that surround the bindings are open code units.

[7] See: http://wwwiti.cs.uni-magdeburg.de/∼mkuhlema/patternsCBD/

Typical for this category is the Observer implementation. In AspectJ, Observer is implemented in one abstract pattern aspect[8] and three binding aspects[9] that crosscut the classes and bind all pattern roles. The abstract pattern aspect and bindings are not grouped but located among classes they crosscut, that is, the aspects lack cohesion. Each binding aspect binds all pattern roles together which does not allow to vary individual bindings of a class to a pattern role, e.g., *Subject*.

Observer implemented with Jak (Fig. 7b) exhibits high cohesion because the set of base classes is grouped in one composite and named collaboration and the crosscutting and extending concern of the pattern is implemented cohesively in one other collaboration. In that composite collaboration, the whole pattern implementation is aggregated and can be referred to by name (here: *CrosscuttingObserverImpl*). Moreover, the nested collaborations can be reused independently of each other. Binding collaborations like *ColorObserver* are further decomposed (not depicted for brevity) where the nested bindings allow to configure the binding of single pattern roles, e.g., *Subject*.

Using abstract pointcuts in the AspectJ implementation, an abstract pattern implementation can be bound to different applications. Jak extension mechanisms depend on names and types of the base application; thus, they do not allow to detach an abstract implementation of the pattern as easy as AspectJ.

Aspects as Object Factories

Cohesion: AspectJ implementations are less cohesive than their Jak counterparts.
Binding-reusability: Binding code of the AspectJ and Jak implementations in this category is largely equally reusable.
Abstract-pattern-reusability: The AspectJ pattern implementations promote reuse of abstract pattern implementations compared to the Jak implementations.

In this category, the abstract aspects act as hashmap containers to reference objects that they instantiate and control based on intercepted runtime events. Intercepting runtime events can be performed more flexibly with pointcuts and advice in AspectJ than with method refinements in Jak. That is, advice in AspectJ can wrap methods and method calls (and other) which is more fine-grained than the events extensible with Jak, where Jak only allows to wrap methods. Due to missing abstract binding mechanisms in Jak, abstract pattern implementations (here: the hashmap containers) are moved into the binding collaborations – this increases cohesion because only one named pattern module is left but decreases abstract-pattern-reusability in Jak implementations.

The representative pattern of this category is Singleton. Its AspectJ implementation includes an abstract implementation that creates and controls (using a hashmap) singleton objects. A free-standing aspect binds the Singleton role to a specific class and wraps methods and calls that instantiate this class using pointcut and advice; the wrapper code manipulates the abstract implementation's hashmap. The extended classes are not separated from the implementation

[8] The abstract aspect is a hashmap container with nested role interfaces.
[9] Each binding aspect extends the abstract aspect and binds both role interfaces *Subject* and *Observer* to the classes of the base implementation.

of the Singleton pattern and the pattern implementing aspects are not aggregated. In the Jak counterpart, the whole pattern implementation of Singleton is aggregated in a single composite collaboration and separated from the set of pattern-unrelated and extended classes, i.e., collaborations are cohesive. However, using the Jak binding mechanism, i.e., method refinements, we can solely extend whole object-creating methods but nothing else, such as constructors or method calls. Method refinements can be reused independently (good binding-reusability) but the role they implement is closely bound to the extended base class (bad abstract-pattern-reusability). The abstract pointcut mechanism of AspectJ is applied to implement the Singleton role abstractly – this implementation is bound afterward to base classes, that is, implementing Singleton abstract is more difficult with Jak than with AspectJ concepts.

Language Constructs

Cohesion, binding-reusability, abstract-pattern-reusability: The AspectJ implementations in this category differ in all criteria from their Jak counterparts.

Patterns in this category deal with redirecting method calls by wrapping existing methods and introducing new ones. Wrapping and introducing methods can be performed in AspectJ as well as in Jak. However, the code that wraps a method has to be assigned one-to-one in Jak while AspectJ allows to wrap different methods with one piece of advice. Due to this weak dependency, AspectJ implementations outperform their Jak counterparts with regard to abstract-pattern-reusability. Due to the close relationship between base classes and collaborations, both were structured cohesively without loosing more reusability; the free-standing (and mostly unrelated) code units of the AspectJ counterparts are less cohesive.

The Proxy implementation with AspectJ uses pointcuts and advice to shield an object by redirecting methods called on it. To shield different methods of an object independently, different free-standing binding aspects shield that object. These binding aspects are ungrouped, which decreases cohesion. The method calls to the shielded object are redirected within an abstract implementation and each binding aspect in essence declares which methods to redirect. In the Jak implementation, abstract pattern implementation and bindings also exist but the overall pattern implementation, i.e., different binding collaborations (each shields a method) together with the abstract role implementation, is aggregated within one collaboration; the extended base implementation of classes is aggregated in another collaboration. Thus, in Jak, both sets, the crosscutting pattern collaborations and the extended base classes, can be referred to by name – this is typical for high cohesion. In Jak as in AspectJ, the nested binding collaborations and the free-standing binding aspects can be reused in different variants of the software. However, due to completely grouping the pattern implementation in composite collaborations in Jak, there is no reusable abstract pattern implementation like in the AspectJ counterpart.

Multiple Inheritance

Cohesion: Jak implementations are more cohesive than the AspectJ counterparts.
Binding- and abstract-pattern-reusability: AspectJ and Jak implementations are equivalent.

Patterns in this category deal with method insertions into classes – for the sake of reuse often realized with multiple inheritance. Both languages, AspectJ and Jak, allow to introduce methods into classes. This way, both languages obey strong bindings of extensions toward the extended class; thus, abstract pattern implementations are tangled with bindings for all patterns in this category. Consequently, no bindings or abstract pattern implementations can be reused independently in the AspectJ or Jak implementations – AspectJ and Jak are equivalent with regard to reuse.

The aspect of Template Method introduces an algorithm method into a class whose subclasses implement the algorithm steps. The aspect that implements the crosscutting pattern Template Method is not separated from the crosscut classes, i.e., different extended pattern-unrelated classes are not grouped but free-standing among crosscutting pattern aspects, this structure of free-standing code units lacks cohesion. In Jak, the set of basic classes and the set of exchangeable Jak extensions are grouped; however, members of these groups can be reused as easily across variants of the software as free-standing aspects in AspectJ. Abstract pattern implementations cannot be separated and reused because introducing the algorithm method into a class binds respective aspects and collaborations closely to this class.

Scattered Code Modularized

Cohesion and binding-reusability: The AspectJ implementations differ in cohesion and binding-reusability from the Jak implementations.
Abstract-pattern-reusability: All AspectJ and Jak counterparts are equivalent for abstract-pattern-reuse because no abstract pattern implementation is separated.

Patterns in this category include different roles, in this case different methods, that have to be introduced into classes. These introductions result in a strong binding of the pattern code in AspectJ and Jak, i.e., code that is not abstract and cannot be reused across applications.

In contrast to the Jak implementation of Interpreter, the AspectJ version does not group and separate crosscutting pattern aspects from the set of pattern-unrelated classes thus are not cohesive. All roles of the Interpreter pattern are bound to classes by one closed aspect or by one open and adaptable collaboration, i.e., bindings for different Interpreter roles can be adapted and thus reused across variants more easily in Jak than in AspectJ. Roles are assigned to classes by extending the classes with methods – this prevents abstract pattern implementations in AspectJ and Jak.

No Benefit from AspectJ Implementation

Cohesion, binding-reusability, abstract-pattern-reusability: The AspectJ and Jak implementations of Facade, the only pattern in this category, are equivalent with regard to the criteria.

Table 2. Results of evaluating initial and optimized implementations

Comparison \ Patterns by Categories	Chain of Responsibility	Command	Composite	Mediator	Observer	Flyweight	Iterator	Memento	Prototype	Singleton	Adapter	Decorator	Proxy	Strategy	Visitor	Abstract Factory	Bridge	Builder	Factory Method	Template Method	Interpreter	State	Facade
a) initial AspectJ/initial Jak																							
Cohesion	●	◐	◐	●	●	●	●	●	●	●	◐	●	●	●	◐	●	●	●	●	●	●	●	◐
Binding-Reusability	●	●	●	●	●	●	◐	◐	●	◐	◐	◐	◐	●	●	◐	◐	◐	◐	◐	●	◐	◐
Abstract-Pattern-Reusability	○	○	○	○	○	○	◐	○	○	○	◐	◐	○	○	○	◐	◐	◐	◐	◐	◐	◐	◐
b) cohesive AspectJ/initial Jak																							
Cohesion	◐'	○'	○'	◐'	◐'	◐'	●	◐'	◐'	◐'	◐	◐'	◐'	◐'	◐	●	●	●	◐'	◐'	●	●	◐
Binding-Reusability	●	●	●	●	●	●	◐	●	●	◐	◐	●'	●'	●	●	◐	◐	◐	●'	◐	●	◐	◐
Abstract-Pattern-Reusability	◐'	◐'	◐'	◐'	◐'	◐'	◐	◐'	◐'	◐'	◐	◐	◐'	◐'	◐'	◐	◐	◐	◐	◐	◐	◐	◐
c) initial AspectJ/reusable Jak																							
Cohesion	◐'	◐	◐	◐'	●	◐'	●	◐'	◐'	●	◐	●	●	◐'	◐	●	●	●	●	●	●	●	◐
Binding-Reusability	●	●	●	●	●	●	◐	◐	●	◐	◐	◐	◐	●	●	◐	◐	◐	◐	◐	●	◐	◐
Abstract-Pattern-Reusability	◐'	◐'	◐'	◐'	◐'	◐'	◐	◐'	◐'	○	◐	◐	○	◐'	◐'	◐	◐	◐	◐	◐	◐	◐	◐

○: AspectJ impl. outperforms Jak impl.; ●: Jak impl. outperforms AspectJ impl.;
◐: both impl. are equal; primed result: changed through optimization

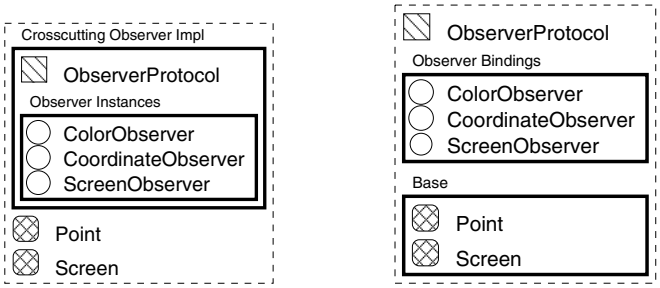
The AspectJ version of Facade uses error-declaration mechanisms that do not affect reuse or cohesion. Both implementations, AspectJ and Jak, do not differ in their structure.

Summary

In Table 2a, we summarize our results. The initial implementations exhibit high cohesion for Jak collaborations and low cohesion for AspectJ aspects. This is because aspects should be reused across applications and therefore must be free-standing. Extensions in Jak are typically bound to an application and thus abstract pattern implementations are often integrated and aggregated with bindings which hampers abstract-pattern-reuse but improves cohesion.

3.2 Optimized Implementations

Interestingly, reusability and cohesion are not satisfied simultaneously in any implementation. Nearly all AspectJ implementations lack cohesion and the majority of Jak implementations lack abstract-pattern-reusability compared to their respective counterparts (cf. Tab. 2a). Trying to eliminate the possibility of this result being a consequence of the legacy initial implementations, we decided to develop a new version of the patterns aiming to improve the lacking property identified. That is, we improved the AspectJ implementations with regard to cohesion and improved the Jak implementations with regard to abstract-pattern-reusability.



(a) cohesive AspectJ impl. (b) reusable Jak impl.

Fig. 8. Optimized modularizations of Observer

Cohesion-Optimal AspectJ Implementations. We improved the AspectJ implementations toward cohesion by nesting aspects. For example, to maximize cohesion for Observer, we aggregated the abstract pattern aspect and all binding aspects inside a single composite aspect.[10] The composite aspect encloses related classes and aspects and can be referred to by name. By nesting abstract pattern and binding aspects within composite aspects we improved cohesion in the AspectJ implementations of Chain Of Responsibility, Command, Composite, Decorator, Factory Method, Flyweight, Mediator, Memento, Observer, Prototype, Proxy, Singleton, Strategy, and Template Method. For these patterns, the new implementations became at least equivalent with respect to cohesion compared to their Jak counterparts. In the case of Command and Composite, the new AspectJ implementations even outperform their Jak counterparts because both patterns need redefinitions of existing class declarations (only additions to classes but no changes are possible in Jak).

However, we have observed that all but three cohesion-optimized AspectJ implementations exhibit reduced abstract-pattern-reusability and become equivalent to their Jak counterparts in this respect.[11] The reason for reduced reusability is that composing aspects results in a composite aspect that is closed, i.e., the abstract pattern implementation gets tangled with its bindings and cannot be reused independently. AspectJ implementations cannot be optimized for cohesion without reducing their abstract-pattern-reusability and vice versa. Some of the AspectJ solutions (patterns Decorator, Flyweight, Memento, and Proxy) additionally led to a reduced support for binding-reusability after being optimized for cohesion.[12]

For example, we improved Observer by composing the free-standing aspects (cf. Fig. 7a); thus, we created the structure of Figure 8a with nested aspects.

[10] Different nesting techniques of AspectJ, like packages, are possible but are considered inappropriate for aspects [6] because packages are already used as abstractions of (non-crosscutting) classes and cannot be composed.

[11] Decorator, Factory Method, and Template Method already were equivalent here before.

[12] Many of them already lacked binding-reusability before (cf. Tab. 2a).

Now, the AspectJ code structure exhibits high cohesion and is as cohesive as the Jak counterpart. Unfortunately, in this code structure nested aspects cannot be reused independently because their surrounding composite aspects are closed. Consequently, reusability decreases.

Reuse-Optimal Jak Implementations. We have improved abstract-pattern-reusability of Jak implementations by separating abstract pattern implementations from the binding code and from the pattern-unrelated classes into free-standing collaborations. We detached abstract implementations for Command, Chain Of Responsibility, Composite, Flyweight, Mediator, Memento, Observer, Prototype, Strategy, and Visitor. Of these patterns, Chain Of Responsibility, Flyweight, Mediator, Memento, Prototype, and Strategy now exhibit less cohesion and are equivalent to their AspectJ counterparts in this respect. The cohesion of Command, Composite, and Visitor remain equivalent as before, while the Jak Observer remains more cohesive than the AspectJ counterpart. The reuse-optimized Jak Observer implementation remains more cohesive because it still groups (in an open collaboration) binding collaborations, that are free-standing aspects in the AspectJ code. Notably, patterns Proxy and Singleton could not be improved to become as reusable as their AspectJ counterpart. Abstract implementations of these patterns require advanced binding mechanisms such as call-advice or abstract pointcuts unavailable in Jak.

For example, consider Observer where in the initial Jak implementation (cf. Fig. 7b) all pattern collaborations (abstract implementation and binding collaborations) are grouped in a composite collaboration and the extended base classes are grouped within another composite collaboration. After optimizing for reuse, the abstract pattern implementation is extracted from the composite pattern collaboration into a free-standing collaboration and can be reused (see Fig. 8b).

We summarize both comparisons in Table 2 (parts b and c). Firstly, we compare the cohesion-optimized AspectJ versions with the initial Jak implementations (Tab. 2b). Secondly, we compare the reuse-optimized Jak versions with the initial AspectJ implementations (Tab. 2c). We highlight the evaluation results of those patterns with a prime whose evaluation changed due to optimizing.

4 Consequences, Insights, and Guidelines

Consequences and Insights. Using Jak, we were able to build cohesive pattern implementations that obey binding-reusability similar to AspectJ counterparts but not abstract-pattern-reusability. In neither technique it is possible to implement cohesive software that separated abstract pattern implementations.

We have optimized AspectJ and Jak implementations with respect to reusability and cohesion. We observed that when AspectJ implementations are optimized to allow reuse they loose cohesion and vice versa. Thus, an AspectJ developer has to decide between reusability and cohesion uncompromisingly. AspectJ implementations are scattered across free-standing aspects if abstract pattern implementations or binding aspects are to be reused independently. Large composite aspects obey cohesion but are closed and thus their nested aspects cannot be

reused independently. For Jak, we observed that with a proper cohesion we could also optimize reusability to some extend (namely binding-reusability). However, neither Jak nor AspectJ implementations are cohesive when they are to be reused across applications (abstract-pattern-reusability). A reason for that might be that AspectJ mechanisms are designed to maximize reuse of abstract (pattern) and application-specific (binding) aspects while scattering and free-standing aspects are accepted for adaptability. Jak provides two options, (1) software can be implemented cohesively including (binding) collaborations that are reusable within an application or (2) the software's collaborations can be implemented in a more reusable manner without cohesion even (everything is scattered in free-standing collaborations).

Guidelines. In summary, we suggest to implement patterns of the categories *Multiple Inheritance* and *Scattered Code Modularized* with Jak instead of AspectJ because all (even all optimized) according Jak pattern implementations outperform their AspectJ counterparts. We do not recommend any particular technique for the pattern of category *No Benefit From AspectJ Implementation* (pattern Facade) because all AspectJ and Jak implementations are pairwise equivalent for both criteria. Pattern implementations in the remaining categories are very diverse with regard to the criteria such that no general guidelines are discernable. To implement each design pattern in a software with the paradigm we proposed, we suggest multi-paradigm approaches, like Caesar or Aspectual Feature Modules [35,6]. In order to choose the best implementation technique for the classes of patterns, the developer has to decide on a per pattern basis or has to balance certain desired criteria of the software to build. For example, if the developer mainly aims at cohesion he/she should implement design patterns with Jak instead of AspectJ for all categories, except *No General Accessible Interface* and *No Benefit From AspectJ Implementation*.

In the case of reusability within applications, i.e., across variants of one application, we favor Jak to implement patterns of the categories *No Accessible Interfaces for Classes* and *Aspects as Object Factories* because all Jak implementations in these categories allow better reuse of bindings (except patterns Iterator and Singleton whose implementations are equivalent). Pattern implementations of the remaining categories are equivalent for binding-reusability.

To build libraries and abstract components that should be reused across applications, AspectJ should be preferred over Jak. The reason is that only AspectJ's abstract pointcut declarations allow to bind abstract pattern implementations to arbitrary base programs. Furthermore, some patterns require sophisticated AspectJ mechanisms not available in Jak to implement reusable abstract implementations, e.g., method call extensions. Jak extensions rely on names in base code more than AspectJ extensions – a fact that hampers abstract pattern implementation. In the categories *No General Accessible Interfaces for Classes* and *Aspects as Object Factories*, all AspectJ implementations but one (pattern Iterator) are suited better for abstract-pattern-reuse than their Jak counterparts; the AspectJ Iterator implementation is as reusable as its Jak counterpart.

Since AspectJ and Jak provide common mechanisms of either AOP and CBD, we argue that our results hold for most AOP and CBD languages.

Threats to Validity. In this study, we made some assumptions. We took the pattern implementations of Hannemann et al. as a base because they are well documented and commonly referred. Our insights and guidelines might differ if other languages of AOP and CBD are evaluated; but, we observed that AspectJ and Jak are most widely used as representatives for AOP and CBD. For criteria different than cohesion and reusability our guidelines cannot say anything – however, cohesion and reusability strongly impact other criteria (maintainability, extensibility, etc.). We did not weight our criteria because we assume each to be equivalently important for evolving and reusing software. Other initial implementations of GoF patterns may fit our criteria better, but therefore we optimized the implementations toward the evaluated criteria.

5 Related Work

Different studies compared OOP and AOP by means of the GoF design patterns. Hannemann et al. defined criteria of well-formed software designs (locality, reusability, composition transparency, and (un)pluggability) to evaluate different design pattern implementations using AspectJ [22]. Garcia et al. evaluated AOP concepts quantitatively on the basis of cohesion and coupling criteria using a case study of design patterns [16]. Gélinas et al. compared AO and OO design patterns with respect to cohesion [17]. Our qualitative study focused on comparing AOP and CBD languages and found that both paradigms provide crosscutting support but exhibit specific strengths with respect to the criteria reusability and cohesion.

We used a definition of cohesion similar to the definition of locality of Hannemann et al. Their work is primarily focused on detaching the pattern code into aspects to enhance cohesion of the pattern. We additionally analyze the cohesion of bindings. In contrast to the notion of cohesion of Garcia et al., we do not limit our focus on pairs of method and advice that access the same field but also attend dependencies between code units based on inheritance or method calls. We have used different criteria than Gélinas et al. because their cohesion criterion is hardly usable to contrast refinements with (abstract) pointcut mechanisms and nested with free-standing code units.

Our definition of reusability roughly corresponds to composition transparency of Hannemann et al. In contrast, we do not only focus on the reusability of the classes that are assigned to a pattern but also on the reusability of binding units across different variants of one software.

Apel et al. performed a qualitative case study on AOP and CBD [6]. They used AOP and CBD to cope with *different* kinds of problems, e.g., the problem of homogeneous crosscutting was tackled using AOP and heterogeneous crosscutting was tackled using CBD concepts. This study neither uses design patterns nor is focused on the duality of AOP and CBD concepts, i.e., they used both,

AOP and CBD concepts, within the same application and evaluated when they used which technique within that application. We evaluate AOP and CBD languages using related programs implemented in mature AOP and CBD languages twice, once in AspectJ and once in Jak. Consequently, we are able to evaluate different implementations based on common criteria.

Meyer et al. focused on the transformation of the GoF design patterns into reusable components [34]. They used language concepts that are specific to the Eiffel programming language, e.g., genericity, contracts, tuples, and agents to implement pattern components. We focus on AOP and CBD to improve design pattern implementations. AOP and CBD are programming techniques not bound to a single programming language as Eiffel but can be applied to different programming languages, like Java and C++, and – in theory – to arbitrary software units [9, 36].

We also have considered category systems for design patterns by Gamma et al. [15], Garcia et al. [16], and others [19, 44, 14, 2] but found them inappropriate for our study. Gamma et al. defined categories of design patterns based on the purpose a design pattern serves and on the scope, i.e., whether the pattern deals with objects or classes. Garcia et al. gave a new categorization system based on every criterion they analyzed in their case study, i.e., every category is associated with one single criterion. Other researchers categorized patterns based on the pattern's adequacy to be a language construct, on the predominant relationship (relationship: one pattern *uses* another pattern), or categorized different patterns than the GoF. We have also applied these categories to analyze our evaluation results but find them too diverse and consequently not meaningful to derive commonalities and significant guidelines for using AspectJ and Jak. For example, Gamma et al. assigned eleven patterns to a category (behavioral patterns) but according pattern implementations in our study were too diverse for cohesion and reuse and did not allow significant results. Furthermore, other researcher's categories overlap, which hampers reasoning about language properties and programming guidelines.

Several researchers introduced further design patterns than the GoF patterns, also for other programming paradigms [14, 42, 44, 28, 32]. All these researchers (including Gamma et al.) aimed at increased flexility and reuse. We focused on patterns of Gamma et al. because they comprise the best-known patterns for OOP and are domain independent [2].

6 Conclusions

Several researchers observed a lack of modularity in object-oriented design pattern implementations and improved the pattern implementations using AspectJ. We followed this line of research and reimplemented their aspect-oriented design pattern implementations with collaboration-based concepts using the Jak language because Jak also provides crosscutting support needed for modularizing design patterns but both approaches show different mechanisms to support crosscutting.

We have used cohesion and reusability as the qualitative criteria to compare and contrast the AspectJ and Jak design pattern implementations. Based on this evaluation, we have inferred guidelines for implementing design patterns. They apply for initial AspectJ and Jak implementations as well as for implementations that are optimized for cohesion or reuse. We further have shown that AspectJ and Jak are complementary and the developer has to balance desired aims of cohesion and reuse. Concepts of AspectJ and Jak on their own do not suffice in structuring software appropriately. But, we propose to use AspectJ and Jak concepts adequately depending on categories of design patterns, possibly resulting in mixed implementations. This can be achieved by using existing multi-paradigm approaches that combine AspectJ and Jak or similar languages. Finally, our results of comparing AspectJ and Jak can be easily mapped to other AOP and CBD languages.

As further work we target on analyzing and evaluating overlapping design pattern implementations. For that, we modularize design patterns in a large-sized framework. This also addresses the limitation of this work that the evaluated case studies of design patterns are rather small.

Acknowledgments

We thank Jan Hannemann and Gregor Kiczales for their AspectJ pattern implementations which have been the basis of this case study. We thank Christian Kästner for helpful discussions and comments on earlier drafts of this paper. Martin Kuhlemann is supported and partially funded by the *DAAD Doktorandenstipendium* (No. D/07/45661).

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, New York (1996)
2. Agerbo, E., Cornils, A.: How to Preserve the Benefits of Design Patterns. SIGPLAN Not. 33(10), 134–143 (1998)
3. Aldrich, J.: Open Modules: Modular Reasoning About Advice. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 144–168. Springer, Heidelberg (2005)
4. Apel, S., Kästner, C., Trujillo, S.: On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In: Workshop on Assessment of Contemporary Modularization Techniques, p. 161 (2007)
5. Apel, S., Kästner, C., Leich, T., Saake, G.: Aspect Refinement - Unifying AOP and Stepwise Refinement. JOT 6(9), 13–33 (2007)
6. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE TSE 34(2), 162–180 (2008)
7. Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multi-Dimensional Separation of Concerns. In: FSE, pp. 48–57 (2003)
8. Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM TOSEM 1(4), 355–398 (1992)
9. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE TSE 30(6), 355–371 (2004)

10. Biggerstaff, T.J.: A Perspective of Generative Reuse. Annals of Software Engineering 5, 169–226 (1998)
11. Bosch, J.: Design Patterns as Language Constructs. JOOP 11(2), 18–32 (1998)
12. Bryant, A., Catton, A., De Volder, K., Murphy, G.C.: Explicit Programming. In: AOSD, pp. 10–18 (2002)
13. Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C.: Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In: AOSD, pp. 109–121 (2006)
14. Coplien, J.O., Schmidt, D.C. (eds.): PLoPD. ACM Press/Addison-Wesley Publishing Co. (1995)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
16. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., A.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: AOSD, pp. 3–14 (2005)
17. Gélinas, J.-F., Badri, M., Badri, L.: A Cohesion Measure for Aspects. JOT 5(7), 75–95 (2006)
18. Georg, G., France, R.B.: UML Aspect Specification Using Role Models. In: OOIS, pp. 186–191 (2002)
19. Gil, J., Lorenz, D.H.: Design Patterns vs. Language Design. In: Workshop on Object-Oriented Technology, pp. 108–111 (1998)
20. Hachani, O., Bardou, D.: On Aspect-Oriented Technology and Object-Oriented Design Patterns. In: Workshop on Analysis of Aspect-Oriented Software (2003)
21. Han, Y., Kniesel, G., Cremers, A.B.: Towards Visual AspectJ by a Meta Model and Modeling Notation. In: Workshop on Aspect-Oriented Modeling (2005)
22. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: OOPSLA, pp. 161–173 (2002)
23. Hölzle, U.: Integrating Independently-Developed Components in Object-Oriented Languages. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 36–56. Springer, Heidelberg (1993)
24. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
25. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
26. Klaeren, H., Pulvermueller, E., Rashid, A., Speck, A.: Aspect Composition Applying the Design by Contract Principle. In: GCSE, pp. 57–69 (2001)
27. Kuhlemann, M., Rosenmüller, M., Apel, S., Leich, T.: On the Duality of Aspect-Oriented and Feature-Oriented Design Patterns. In: Workshop on Aspects, Components, and Patterns for Infrastructure Software, p. 5 (2007)
28. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co. (2003)
29. Lieberherr, K.: Controlling the Complexity of Software Designs. In: ICSE, pp. 2–11 (2004)
30. Liskov, B.: Data Abstraction and Hierarchy. In: OOPSLA, pp. 17–34 (1987)
31. Lopez-Herrejon, R., Batory, D., Cook, W.R.: Evaluating Support for Features in Advanced Modularization Technologies. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 169–194. Springer, Heidelberg (2005)
32. Lorenz, D.H.: Visitor Beans: An Aspect-Oriented Pattern. In: Workshop on Object-Oriented Technology, pp. 431–432 (1998)

33. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall PTR (1997)
34. Meyer, B., Arnout, K.: Componentization: The Visitor Example. IEEE Computer 39(7), 23–30 (2006)
35. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: AOSD, pp. 90–99 (2003)
36. Reenskaug, T., Anderson, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., Stenslet, P.: OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. JOOP 5(6), 27–41 (1992)
37. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM 11(2), 215–255 (2002)
38. Stein, D., Hanenberg, S., Unland, R.: A UML-based Aspect-Oriented Design Notation for AspectJ. In: AOSD, pp. 106–112 (2002)
39. Stevens, W.P., Myers, G.J., Constantine, L.L.: Structured Design. IBM Syst. J. 13(2), 115–139 (1974)
40. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: ICSE, pp. 107–119 (1999)
41. VanHilst, M., Notkin, D.: Using Role Components in Implement Collaboration-based Designs. In: OOPSLA, pp. 359–369 (1996)
42. Woolf, B.: Null Object. In: PLoPD, pp. 5–18 (1997)
43. Zhang, C., Jacobsen, H.-A.: Quantifying Aspects in Middleware Platforms. In: AOSD, pp. 130–139 (2003)
44. Zimmer, W.: Relationships Between Design Patterns. In: PLoPD, pp. 345–364 (1995)