Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 233 (2009) 105–125
www.elsevier.com/locate/entcs

# Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics

Bruno Carreiro da Silva[a,1] Eduardo Figueiredo[b,2]
Alessandro Garcia[b,3] Daltro Nunes[a,4]

[a] *Institute of Informatics*
*Federal University of Rio Grande do Sul*
*Porto Alegre, Brazil*

[b] *Computing Department*
*Lancaster University*
*Lancaster, United Kingdom*

**Abstract**

It has been advocated that Aspect-Oriented Programming (AOP) is an effective technique to improve software maintainability through explicit support for modularising crosscutting concerns. However, in order to take the advantages of AOP, there is a need for supporting the systematic refactoring of crosscutting concerns to aspects. Existing techniques for aspect-oriented refactoring are too fine-grained and do not take the concern structure into consideration. This paper presents two categories towards a metaphor-based classification of crosscutting concerns driven by their manifested shapes through a system's modular structure. The proposed categories provide an intuitive and fundamental terminology for detecting concern-oriented design flaws and identifying refactorings in terms of recurring crosscutting structures. On top of this classification, we define a suite of metaphor-based refactorings to guide the "aspectisation" of each concern category. We evaluate our technique by classifying concerns of 23 design patterns and by proposing refactorings to aspectise them according to observations made in previous empirical studies. Based on our experience, we also determine a catalogue of potential additional categories and heuristics for refactoring of crosscutting concerns.

*Keywords:* Refactoring, Aspect-oriented programming, Crosscutting concerns, Metaphor-based classification, Design heuristics.

## 1 Introduction

Aspect-Oriented Programming (AOP) [20] provides new programming languages constructs for improving the separation of concerns with the goal of enhancing design modularity. Aspects are new units of modularisation for encapsulating crosscutting concerns, i.e., system features or properties that naturally affect many system

[1] Email: bruno.carreiro@inf.ufrgs.br
[2] Email: e.figueiredo@lancaster.ac.uk
[3] Email: garciaa@lancaster.ac.uk
[4] Email: daltro@inf.ufrgs.br


doi:10.1016/j.entcs.2009.02.064

modules [20]. AOP is therefore expected to be an effective technique to enhance software maintainability through the modularisation of crosscutting concerns. However, there is a need for a systematic migration of existing object-oriented (OO) systems towards the aspect-oriented decompositions in order to reap the benefits of AOP.

Migration of legacy code to aspects requires proper mechanisms for identifying [25] and refactoring crosscutting concerns [16] [22]. Those mechanisms have to be aware of all fragments that are contributing to the implementation of a specific crosscutting concern, and whether and what aspect solutions should be applied to them. In fact, refactoring of crosscutting concerns [18] [22] requires a holistic treatment of the concern under consideration; it consists of many small, complementary transformations aiming at modularising elements of a crosscutting concern. Those pieces of concern have different relationships with the target modules making the refactoring even harder. Therefore, the final decision if the crosscutting concern should be entirely or partially refactored also depends on a broad analysis of the concern under consideration.

The recognition that concern identification and refactoring are important issues through the software maintenance activities is not new. Actually, with the emergence of AOP, there is a growing body of relevant work in the software engineering literature focusing on concern analysis techniques [8] [25] and refactoring of crosscutting concerns [1] [18]. However, there is a lack of work integrating those techniques to enhance refactoring of crosscutting concerns based on concern analyses. In addition, only initial works have classified common crosscutting structures using metaphors [4] and explored how taming these structures help in software maintenance activities, such as refactoring.

In this context, the main contributions of this paper are threefold. First, an initial set of metaphors is defined in order to characterise concerns regarding their crosscutting structures. Heuristics are used to identify occurrences of two concern metaphors, namely Octopus and Black Sheep [4]. Second, taking advantages from the crosscutting structure defined by concern metaphors, refactorings are presented to modularise Octopus and Black Sheep crosscutting concerns. Third, an empirical study involving concerns of 23 design patterns is used to evaluate (i) the accuracy of our heuristic classification and (ii) the applicability of the metaphor-based refactorings. Based on our experience, we also determine a catolog of additional categories and heuristics for refactoring crosscutting concerns.

The rest of this paper is organised as follows. Section 2 motivates this work by presenting some limitations of existing aspect-oriented refactorings and illustrating those problems in an example. Section 3 presents our heuristic-based refactoring technique in the light of two metaphors: Octopus and Black Sheep. Section 4 demonstrates the steps of a bi-dimensional evaluation targeting the heuristics' accuracy and refactorings' applicability. Section 5 presents some discussions and ongoing work while Section 6 concludes this paper with the final remarks.

## 2 Aspect-Oriented Refactoring

This section presents a review of existing object-oriented (OO) refactorings that have been extended to become aspect-aware (Section 2.1). It also discusses refactorings tailored for supporting the extraction and modularisation of crosscutting concerns (Section 2.2). Section 2.3 summarises the shortcomings of existing refactoring approaches for aspect-oriented (AO) systems and Section 2.4 demonstrates some of those shortcomings in an illustrative example.

### *2.1 Aspect-aware Refactoring*

Some authors [15] [19] [24] have recently identified limitations when applying well-known OO refactorings [11] in the presence of aspects. For example, one problem arises from the fact that OO refactoring usually changes the structure of join points of the program and, thus, potentially changes how the aspects affect classes. To address this problem, Hanenberg, Oberschulte, and Unland [15] proposed preconditions which have to be respected when applying a conventional refactoring in the presence of aspects. They use *Extract Class* [11] in order to exemplify the preconditions which make the refactoring aspect-aware. In addition, they propose some new refactorings to extract concerns from an existing design to aspects.

Iwamoto and Zhao [19] also proposed modifications to existing OO refactorings in order to make them aspect-aware. They show examples and give guidelines on how to avoid ripple effects when applying OO refactoring to an aspectual code. Similarly, Monteiro and Fernandes [24] proposed a catalogue of AO refactorings and described them in a similar way to Fowler's object-oriented ones [11]. However, all the aforementioned refactoring approaches are too fine grained and they do not allow the designer to holistically reason about the elements involved in a crosscutting concern. They also often require a huge list of disconnected transformations to modularise a typical crosscutting concern, such as the Observer design pattern. The designer is thus in charge of figuring out from the scratch how unrelated transformations need to be combined to refactor conventional crosscutting concerns. Therefore, coarse-grained refactorings for modularising crosscutting concerns have emerged as outlined in the next section.

### *2.2 Refactoring of Crosscutting Concerns*

Several approaches have been proposed to deal with the problem of fine-grain refactorings for modularising crosscutting concerns [1] [18] [22]. For example, some recent cookbooks have been documented to guide the "aspectisation" of specific crosscutting concerns, such as exception handling [9] [10]. Using a different strategy, Binkley *et al.* [1] present a tool to support the composition of refactorings which extract Java code fragments into aspects. Their tool follows an extraction workflow consisting of discovery, transformation, selection, and refactoring. The first two steps determine applicable refactorings for the selected code fragments and apply the admissible OO refactorings in order to prepare the design for aspectisation. Then, the

third step allows developers to select appropriate AO refactorings. Finally, the last step refactors that code and (partially) modularises the crosscutting concern.

The previously discussed refactoring approaches in their own nature look individually at each code fragment and do not consider the crosscutting sturcture of the target concern. Therefore, Hannemann and his colleagues proposed the notion of role-based refactoring [18] to describe transformations based on an abstract model of the target crosscutting concern. The underlying idea is that refactoring instructions are the same for all concrete program elements playing a given role in the concern realisation. The strategy for role-based refactorings is composed of three main stages. First, the developer chooses an appropriate refactoring. Then, it is necessary to map the roles defined by the chosen refactoring to elements of the design realising those roles. Finally, a tool opens dialogue boxes in order to confirm the user choices and automatically performs the refactoring steps.

Similarly to Hannemann's approach, Marin, Moonen, and van Deursen [22] presented refactorings based on crosscutting concern types. A concern type consists of a general intent, an implementation idiom, and one aspect language mechanism to address it. A concrete concern is an instance of its type. Both Hannemann and Marin approaches try to group concerns with similar structures. However, while the former rely on abstract roles, the latter restricts the concern classification to implementation idioms or specific AOP mechanisms.

### *2.3 Liabilities of Conventional Aspect-Oriented Refactoring*

An analysis of existing AO refactorings points out fundamental deficiencies in the manner how concern refactoring is addressed. We discuss below three significant differences observed among the various AO refactorings and the liabilities associated.

**Lack of holistic treatment of scattered changes**. Refactorings for concern modularisation (Section 2.2) are composed of a long list of smaller aspect-aware refactorings [18] [22]. As this category of refactoring changes many seemly unrelated classes of the system, it is hard to recover to a consistent status when one of the smaller refactorings does not finish successfully. Therefore, the system is sometimes left in an inconsistent state between two transformation steps. Such observation suggests that the entire list of transformations has to be treated as a single refactoring. In other words, when a sub-refactoring does not hold, we have to be able to roll back all transformations and recover the previous consistent design. Furthermore, a holist treatment of the target concern is required in order to decide (i) whether the concern should be entirely or partially refactored, and (ii) where in the concern structure it might be harder to aspectise due to, for example, intricate dependencies between the crosscutting concern and the base code [2] [8] [9].

**Inconsistent terminology of the concern structure**. Role-based refactoring [18] and refactoring based on concern type [22] try to address the previous problem by assigning abstract names to a set of similar crosscutting concerns. Nonetheless, they fail due to the lack of a unified terminology of concerns. For example, in role-based refactoring, names for roles are intended to be abstract and not

tied to any specific concern implementation. However, apart from design patterns, Hannemann [16] uses application-specific names, such as *CurrencyControl*, when presenting the role-based refactorings. Furthermore, although Marin *et al.* [22] compiled an initial list of concern classifications, their concern types are usually tied to implementation-specific constructs of programming languages. For example, the *Declare Throws Clause* and *Exception Propagation* refactorings [22] are related to exception handling mechanisms.

**Lack of support to detect concern-related bad smells**. Recent empirical studies have highlighted that crosscutting concerns are key factors to the observance of bad smells in the system design [6] [8] [9]. A bad smell is any symptom that indicates something may be wrong and it generally indicates that the overall design should be refactored [11]. Metrics-based analyses are traditionally the fundamental mechanisms for assessing design modularity and detecting design flaws. In fact, a number of papers [21] [23] have associated bad smells with modularity metrics [3], such as coupling, cohesion, and conciseness. For example, Marinescu [23] proposed a mechanism called *design heuristic rule* (or detection strategy) for formulating metrics-based rules that capture bad smells. However, to the best of our knowledge, there is no work which integrates those three prominent techniques: metrics-based analysis for assessing design modularity; heuristic rules for detection of concern-related bad smells; and AO refactorings driven by holistic-detected anomalies.

### *2.4 A Motivating Example*

To better illustrate some problems with existing refactorings of crosscutting concerns, let's consider an OO instance of the Mediator design pattern [12] developed by Hannemann and Kiczales [17] and presented in Figure 1. Refactoring this pattern to an AO solution requires many small changes in different design elements and relationships involved in the implementation of the Mediator concern (e.g., elements shadowed in Figure 1). Since it is difficult to consider those transformations separately, refactoring of crosscutting concerns, such as role-based refactoring, plans and executes all changes together. In fact, role-based refactoring would associate this refactoring to the Colleague and Mediator roles of the target pattern and call it "Mediator Refactoring". Similarly, a refactoring to aspectise the Observer design pattern (Figure 3 in Section 3.2) would be called "Observer Refactoring" in this approach.

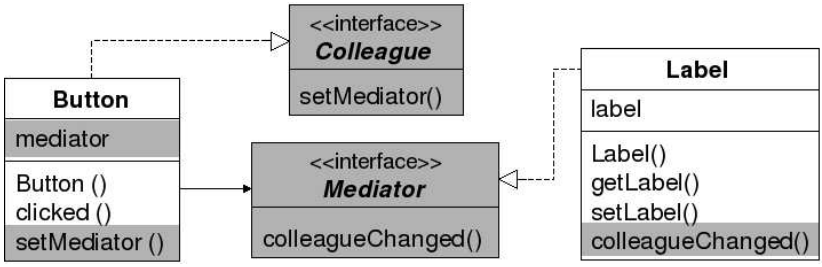


Fig. 1. An instance of the Mediator design pattern [17]

However, existing refactoring approaches, such as role-based refactoring, have at least three drawbacks. First, not all instances of Mediator take advantages of the aspectisation process. For example, Cacho *et al.* [2] claimed that the Mediator pattern cannot be aspectised in their middleware system case study due to many factors (e.g., performance). Second, when the aspectisation is possible, it depends on specific characteristics of the pattern instance, such as level of scattering [6] [8] and coupling between the pattern concern and the other concerns of the system [7]. Hence, an analysis of each particular pattern instance is essential to decide which parts of the concern should be aspectised. Last, but not least, the aspectisation of two distinct concerns might be very similar depending on the involved concerns and their specific implementations. For example, the instances of Mediator and Observer presented in Figures 1 and 3, respectively, have essentially the same basic refactoring steps (Section 3.3).

# 3 Heuristic-driven Refactoring

This section presents refactorings based on concern metaphors to support the aspectisation of crosscutting concerns. These refactorings aim at addressing the shortcomings of existing AO refactorings (Section 2.3). Section 3.1 presents the workflow of each heuristic-driven refactoring. They are applied to crosscutting concerns which possess certain characteristics identified by a set of metrics-based heuristics (Section 3.2). In our case, a heuristic is a composed logical condition, based on metrics, which detects design fragments with bad smells [23]. Although we are working on a catalogue of metaphor-driven heuristics and associated refactorings, this paper presents in more details only two of them. Examples of refactorings for two concern metaphors, named Octopus and Black Sheep, are presented in Sections 3.3 and 3.4, respectively.

## *3.1 Workflow*

The heuristic-driven refactorings support an interactive extraction of concerns into aspects and are composed of four main steps. The first two steps depend on our proposed metaphors and heuristics; the last two are basically the same steps used by other approaches to refactor crosscutting concerns [1] [22].

**(1) Heuristic classification.** The target concern is classified according to its structural pattern by the use of metrics-based heuristic rules (Section 3.2). A terminology of concerns, based on metaphors, is used to provide an unified and intuitive vocabulary of crosscutting concerns. We believe that this vocabulary allows developers to use meaningful terms when referring to a concern.

**(2) Selection of refactoring steps.** A set of fine-grain AO refactorings, such as those ones discussed in Sections 2.1 and 2.2, are selected to modularise the specific concern structure identified in Step 1. In some cases, it might be possible to select one refactoring from two or more alternatives. This flexibility allows developers to choose the best transformations in that application-specific context.

**(3) Human calibration.** Once the fine-grained refactorings have been chosen, the developer makes all necessary configurations and adjustments before changing the code. For example, some fine-grained refactoring steps might require adequate parameters.

**(4) Code transformation.** The code transformations which implement the refactoring can be automated by appropriate tools [1] [19]. Since each heuristic-based refactoring of crosscutting concern is also composed of smaller aspect-aware refactorings, existing tools for the latter [1] [19] can be also used to support the former.

### *3.2 Heuristic-Based Concern Metaphors*

Recent research work has classified concerns in different ways [4] [18] [22]. Generally, if a concern is not well-encapsulated it assumes a crosscutting structure over the system. This crosscutting structure might have different shapes depending on each concern. In our approach we use concern metaphors as instances of concern-related bad smells, i.e., concerns with a particular harmful crosscutting structure for the system modularity.

This section focuses on an initial classification proposed by Ducasse, Girba and Kuhn [4] which is based on how a given concern is distributed over the system. These authors identified two metaphors, namely Black Sheep and Octopus, representing patterns of concern manifestation over a system. We extend this classification in Section 5 with new concern metaphors. Black Sheep is described as a specialised category of crosscutting concerns that touch only few points of the system [4]. On the other hand, Octopus is a crosscutting concern which is partially well modularised by one or more classes, but it is also spread across a number of other classes.

Table 1
Heuristics for concern metaphors

| **Classification** | **Heuristic Description** |
|---|---|
| Black Sheep | The given concern is only implemented by few attributes and methods in all classes where it is. |
| Octopus | The given concern is well-modularised in at least one class and touches few attributes and methods in others. |

Table 1 presents the aforementioned concern metaphors and the heuristic descriptions we use to find them. Our heuristics rely on metrics for the number of attributes and methods implementing a given concern [8]. For instance, the first heuristic in Table 1 classifies a concern as *Black Sheep* if all classes which have this concern dedicate only few percentage points of attributes and methods to that concern (no more than 33 %). We choose the value of 33 % for the Black Sheep heuristic based on a meaningful ratio that represents the definition *"touches very few elements"* [4]. In addition, Lanza and Marinescu [21] suggest the use of meaningful threshold values in metrics-based heuristics, such as 0.33 (1/3), 0.5 (1/2), and 0.67 (2/3).

Figure 2 illustrates a concrete example of Black Sheep using an implementation of the Singleton design pattern [12] proposed by Hannemann and Kiczales (H&K)

```
public class PrinterSingleton {
  protected static int objects = 0;
  protected static PrinterSingleton single;
  protected int id;

  protected PrinterSingleton() {
     id = ++objects;
  }

  public static PrinterSingleton instance() {
   if(single==null) single=new PrinterSingleton();
   return single;
  }
  public void print() {
   System.out.println("My ID is "+id);
  }
}
```

Fig. 2. Singleton pattern design: a Black Sheep instance

[17]. This figure presents the code of the `PrinterSingleton` class which is part of the H&K solution. Elements of this class realising the Singleton concern are shadowed in Figure 2. This specific instance of Singleton is classified as Black Sheep by the corresponding heuristic (Table 1) because it requires only one method, `instance()`, and one attribute, `single`, in the pattern implementation.

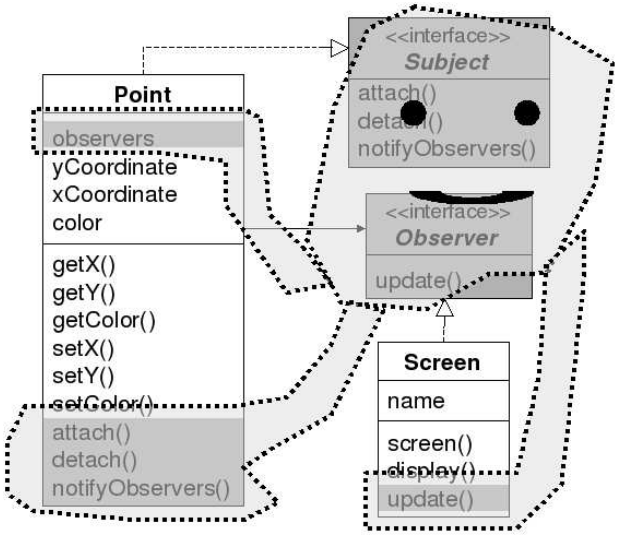


Fig. 3. Observer design pattern: an Octopus instance

The second heuristic in Table 1 is tailored to verify if a crosscutting concern, not classified as Black Sheep, is an *Octopus*. According to this heuristic, a concern is classified as Octopus if every class realising parts of this concern dedicates either (i) many attributes and methods (body of the Octopus), or (ii) few attributes and methods (tentacles of the Octopus) to it. We define that a class belongs to the

body of an Octopus when the percentage of members (i.e., attributes and methods) is higher than 67 %. Similarly, a class is touched by a tentacle when the percentage of members is lower than 33 %. Again, the thresholds values try to be a meaningful approximation of the Octopus' definition [4].

Figure 3 presents an UML class diagram of an instance to the Observer design pattern [12]. According to our heuristics, this specific implementation of the Observer pattern is classified as Octopus. The two interfaces, `Subject` and `Observer`, are completely dedicated to the pattern implementation and, therefore, they represent the Octopus' body. Besides, the two classes in the design have only few methods realising the Observer concern and, so, they represent tentacles of the Octopus. Figure 1 (Section 2.4) presents another instance of Octopus in the context of the Mediator design pattern.

### *3.3 Refactoring of Octopus*

The goal of this refactoring is to better modularise concerns classified as Octopus by our metaphor-based heuristics (Section 3.2). The following refactoring steps aim at moving to aspects parts of code composing the body or touched by tentacles of an Octopus. The body and tentacles are the main characteristics of this kind of crosscutting concerns.

**1) Identify the Octopus members.** The refactoring starts with the identification of classes which compose the Octopus parts (body and tentacles) and their internal structures used to implement this concern. A metrics-based heuristic is used in this step to determine which attributes, methods, interface implementations, and class extensions realise the target concern. Detailed discussions about the metrics used in the concern identification are out of the scope of this paper, but they appear documented elsewhere [8] [13]. We use the example of Octopus presented in Figure 3 to illustrate the refactoring steps. As discussed before, the `Observer` and `Subject` interfaces compose the Octopus body and the `Point` and `Screen` classes are touched by tentacles. For the sake of simplicity, unless otherwise clearly stated we use the term "class" when referring to class, interface, or both.

**2) Refactoring steps to the Octopus body.** The following steps target at separating the Octopus body structure. There are two possibilities of classes composing the body: (a) those entirely dedicated to the Octopus concern, and (b) those mainly dedicated to this concern, but they mixed it with other concerns.

**2.a)** If one class in the Octopus body is 100 % dedicated to this concern, that class can be optionally moved to a new aspect. In other words, the aspectisation of classes in the Octopus body that are not mixed with any other concern is optional. This decision depends on developers' judgement, but measures may support them in this choice.

**2.b)** If one class in the Octopus body is not totally dedicated to this concern, we have to separate the Octopus concern in its own component. After that, this class

can be moved into an aspect.

In our example, the two interfaces are completely dedicated to the Octopus concern, but we choose to aspectise them anyway. Hence, we apply the step 2.a and obtain a new aspect (see Figure 4).

**3) Refactoring steps to modularise Octopus tentacles.** Once the Octopus body is encapsulated into aspects, we have to restructure the Octopus tentacles. For each tentacle it is required to apply the following steps. If no aspect exists yet, these steps may create a new aspect when appropriate.

**3.a)** If the tentacle implements one or more interfaces related to the Octopus, move each interface implementation to an aspect as an introduction.

**3.b)** If the tentacle extends any class only for the concern implementation, move the class extension to the aspect as an introduction.

**3.c)** If there are methods and attributes completely assigned to the target concern, move them to the aspect as introductions.

**3.d)** If a class has any constructor related to the Octopus concern, this constructor has to be modified and used independently of this concern. Optionally, the constructor may be removed if it becomes unnecessary by previous aspectisation steps.

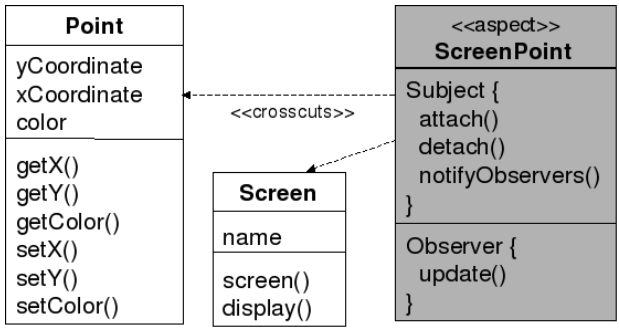


Fig. 4. Separation of the Octopus code

According to our running example, the steps 3.a and 3.c should be applied to the Observer pattern. Hence, attributes, methods, and interfaces implementations in classes touched by tentacles are moved to an aspect as shown in Figure 4. The *crosscuts* stereotype in this figure is used to express relationships where an aspect affects classes by introducing some structural or behavioural features. For example, the `ScreenPointObserver` aspect introduces the attributes, methods and interface realisations moved in steps 3.a and 3.c to those classes.

Designers should be aware of remaining attribute initialisations or calls to the moved methods in other parts of the design. For example, Figure 5 presents two calls to `notifyObservers()` in the `setX()` and `setY()` methods. In this case, the following two steps are applied in order to add pointcuts and advices to the aspect

```
public class Point{
   ...
  public void setX(int x) {
    this.x = x;
    notifyObservers();
  }

  public void setY(int y) {
    this.y = y;
    notifyObservers();
  }
}
```

Fig. 5. Extracting pointcut and advice in the Point tentacle

and separate such pieces of code.

**3.e)** Create pointcuts to pick up necessary join points related to the concern during the program execution. If necessary, use the *Extract Method* refactoring [11] in order to expose joinpoints.

**3.f)** Create advice member(s) to introduce necessary behaviour related to the concern during the program execution. In Figure 5, the dotted lines (`setX()` and `setY()` method signatures) become a call pointcut, while the `notifyObservers()` behaviour are moved to an advice in the `ScreenPointObserver` aspect.

After Steps 3.e and 3.f, classes cannot be aware of any behaviour related to the Octopus concern implementation. In our running example, these last two steps remove the remaining code of the concern from classes, e.g., the `Point` class does not know anymore about the notifying behaviours when coordinates change. The created advice is responsible to call the moved method `notifyObservers()` when the created pointcut pickes up the appropriate joint points (coordinates setting). Additional refactoring steps (e.g., [19] [15] [24]) can also be applied to improve the internal structure of aspects and prepared them for reuse. We discuss this issue in Section 5.

### *3.4 Refactoring of Black Sheep*

The goal of this refactoring is to better encapsulate concerns classified as Black Sheep by our heuristics. First, it identifies classes implementing parts of the Black Sheep concern and, then, it modularises those parts into aspects. Using metaphors, each class is called *sheep* (or black sheep) and the whole system is a *herd*. Alternatively, each part of the class realising the target concern is called a *sheep slice*.

**1) Identify black sheep in the herd.** This refactoring starts with the identification of classes with the Black Sheep concern. The heuristic of Black Sheep

presented in Table 1 (Section 3.2) supports this refactoring step. For example, Figure 2 shows how the corresponding heuristic identified the Singleton design pattern as Black Sheep. The `Printer` class is a sheep while the `single` attribute and the `instance()` method are sheep slices dedicated to the Singleton concern.

**2) Refactoring steps to separate Black Sheep.** The following six steps (labelled 2.a to 2.f) aim to separate sheep slices into aspects. Some of those steps might not be applied to specific instances of Black Sheep. Furthermore, those steps assume an pre-existing aspect (empty or not) assigned to this concern. In case this aspect does not exist, a preliminary step is its creation.

**2.a)** If a class implements one or more interfaces related to the Black Sheep concern, move each interface implementation to an aspect by adding an introduction statement to it.

**2.b)** If a class extends any class only for the Black Sheep concern implementation, move the class extension to an aspect as an introduction.

**2.c)** If a class has methods and attributes exclusively assigned to the Black Sheep concern, move them to an aspect as introductions.

**2.d)** If a class has any constructor related to the Black Sheep concern, either this constructor has to be modified and used independently of the concern or it may be removed.

```
public class PrinterSingleton {
  protected static int objects = 0;
  protected static Printer single;
  protected int id;

  protected Printer() {
    id = ++ objects;
  }

  public static Printer instance() {
    if(single == null) single = new Printer();
    return single;
  }

  public void print() {
    System.out.println("My ID is "+id);
  }
}
```

Fig. 6. Extracting Black Sheep slices

Figure 6 illustrates the modifications inside the `Printer` class after applying the step 2.c. The `single` attribute and the `instance()` method are removed from this class. In addition, the `SingletonInstance` aspect presented in Figure 7 is created

with the two corresponding introduction declarations. Steps 2.a, 2.b, and 2.d do not match our example of Black Sheep, i.e, the Singleton pattern in Figure 2. The first two steps (2.a and 2.b) are applied, for example, when one class extends another or implements interfaces from some third party libraries in order to realise the Black Sheep concern. The Prototype pattern in Section 4.1 is an example where the first step is required.

```
public aspect SingletonInstance {

 static Printer Printer.single;

  public static Printer Printer.instance() {
    if(single == null) single = new Printer();
    return single;
  }
}
```

Fig. 7. Separation of Black Sheep in the SingletonInstance aspect

Consistently, during the program execution it might be required the execution of some behaviour related to the Black Sheep concern that should be aspectised. The *Extract Fragment into Advice* refactoring [24] can be used to capture the appropriate join points by creating pointcuts and to move the code fragments to advices. The following two steps are defined for this purpose.

**2.e)** Create pointcuts to pick up necessary join points related to the Black Sheep concern during the program execution. If necessary, use the *Extract Method* refactoring [11] in order to expose join points.

**2.f)** Create advice(s) to simulate the necessary concern behaviour during the program execution.

# 4 Evaluation

Our evaluation is divided in two parts according to our goals. The first part (Section 4.1) aims to evaluate the metaphor-based heuristics by applying them to a set of concerns and analysing the results. In the second part (Section 4.2), we applied the proposed refactorings to the subset of concerns classified as Octopus or Black Sheep in the first part. Our case study involves concerns of relevance to the 23 Gang-of-Four (GoF) design patterns [12].

## *4.1 On Effectiveness of the Heuristic Concern Classification*

As explained in Section 3, a heuristic rule is a composed logical condition based on metrics [23]. Hence, a set of metrics have to be applied first in order to classify a concern using metaphor-based heuristics. We used concern-sensitive metrics [6] [8] in order to obtain the required measures. We evaluated instances of the 23 GoF design patterns proposed by Hannemann and Kiczales (H&K) [17]. Each

design pattern was classified in a concern metaphor by analyzing its roles. Hence, the overall pattern classification depends on the roles' evaluation. As expected, some pattern roles do not hold in either two metaphor-based heuristics. In this case, the pattern was not classified since its respective roles present neither the Octopus nor the Black Sheep structure. In fact, this situation indicates that other metaphor-based heuristics might have to be applied in order to verify different kinds of crosscutting structure. The row measurement for all pattern roles can be found in the study website [26].

Tables 2 and 3 present the design patterns identified as Octopus and Black Sheep, respectively. These tables also show the classifications of the respective pattern roles. Patterns in Table 2 are classified as Octopus if at least one of their roles holds to this metaphor. Six design patterns were classified as Octopus (Table 2): Decorator, Iterator, Observer, Template Method, Visitor, and Mediator. Other two patterns were classified as Black Sheep (Table 3): Prototype and Singleton. Both Prototype and Singleton patterns have a single role with the Black Sheep crosscutting shape.

Table 2
Octopus design patterns

| **Patterns** | **Roles** | **Role Classification** |
|---|---|---|
| Decorator | Decorator | - |
| | Component | Octopus |
| Iterator | Aggregate | - |
| | Iterator | Octopus |
| Observer | Subject | Octopus |
| | Observer | Octopus |
| Template Method | Abstract Class | Octopus |
| | Concrete Class | - |
| Visitor | Element | Octopus |
| | Visitor | Octopus |
| Mediator | Colleague | Octopus |
| | Mediator | Octopus |

Table 3
Black Sheep design patterns

| **Patterns** | **Roles** | **Role Classification** |
|---|---|---|
| Prototype | Prototype | Black Sheep |
| Singleton | Singleton | Black Sheep |

The crosscutting structure of the GoF design patterns has already been detected and explored in previous studies [2] [13] [17]. These studies enable us to compare our heuristic classification to their findings. Our comparison focuses mainly on the H&K study [17] and on our previous experience on design patterns assessment [2] [13].

All patterns classified as Octopus or Black Sheep by our heuristics have also been classified as crosscutting in H&K study. In other words, these authors pointed out that the AspectJ version is better modularised than its Java counterpart because those patterns have a crosscutting nature (called *super-imposed* roles by them).

Garcia *et al.* [13] confirmed these findings for all eight aforementioned patterns, except for the Template Method pattern. Regarding Template Method, Garcial *et al.* refuted H&K claims and verified that the AspectJ solution brings no improvement in relation to the OO one. To support this observation, Garcial *et al.* performed a quantitative study using a plethora of modularity metrics.

This comparison between our heuristic classification and previous knowledge also allowed us to find at least one false positive of our technique. In other words, we confirmed that Template Method should not be classified as Octopus since the OO solution cannot be improved by the use of aspects. Despite this false positive, we believe all other classifications are correct. If so, our metaphor-based heuristics would present an accuracy of about 85 %. Of course, further empirical investigation is needed to confirm or refute this percentage.

### *4.2 On the Refactoring Evaluation*

The results of the first part of our evaluation (Section 4.1) brings to us a set of concerns classified as Octopus or Black Sheep. This set is used as input to the application of the proposed refactorings (Sections 3.3 and 3.4). In this second part of our evaluation we applied the proposed refactorings to those design patterns classified as Octopus and Black Sheep.

In this part, we decided to take two sets of design pattern instances in order to assess the scalability of refactorings. The first set contains the original versions from H&K [17] and the second one includes extended pattern instances used in a more complex empirical study [13]. The difference between them is that Garcia *et al.* [13] created new participant members playing each pattern role, i.e, they scaled up the original pattern instances by adding new classes, although following the H&K original structure. As a catalogue of fine-grained refactorings, we focus manly on the Monteiro's work [24], but other refactorings [1] [19] have also been used.

Table 4
Octopus refactoring steps

| **Pattern** | **2.a** | **2.b** | **3.a** | **3.b** | **3.c** | **3.d** | **3.e** | **3.f** |
|---|---|---|---|---|---|---|---|---|
| Decorator | | | | | | | x | x |
| Iterator | x | | | | x | | | |
| Observer | x | | x | | x | | x | x |
| Visitor | x | | x | | x | | | |
| Mediator | x | | x | | x | | x | x |

Tables 4 and 5 show the refactoring steps (Sections 3.3 and 3.4) used in the aspectisation of each Octopus and Black Sheep pattern, respectively. The refactoring steps applied to both the original and extended pattern instances were essentially the same. They differed only in the number of fine-grained refactorings that were applied. For example, both Singleton instances require steps 2.c, 2.e, and 2.f in their aspectisation (Table 5). However, 4 fine-grained refactorings were applied to the original version while 16 were applied to the extended Singleton instance (Table 6).

Table 6 presents the number of fine-grained refactorings for each pattern instance (original and extended). These data suggest that Octopus concerns have more complex crosscutting structures than Black Sheep ones. For example, the aspectisation of Octopus' body and tentacles usually requires a higher number of refactorings than Octopus slices. In average, the aspectisation of Octopus requires 21 refactorings against 14 of Black Sheep; considering the extended instances in both cases.

We can also note that in some patterns the number of refactorings increases in a linear rate according to the number of components. For instance, the Prototype pattern needed 2 refactorings for each Prototype class. While the H&K version has only two Prototype classes (4 refactorings), the Garcia's version has six classes (12 refactorings). On the other hand, there are patterns which do not follow this linear rate. Particularly, the Observer instance from Garcia *et al.* requires 44 refactorings while the H&K version requires 16. The number of refactorings for the Observer pattern varies according to (i) the number of concrete Subject and Observer classes and (ii) the number of observation points (e.g., coordinates setting or button clicking).

Although different numbers of refactorings were used, the refactored version of each design pattern instance was successfully obtained by following the refactorings steps and by selecting the appropriate fine-grained refactorings. The Template Method pattern was the only exception as described in Section 4.1. Although the Template Method structure matches the Octopus structure, we decided to not aspectise it due to our previous knowledge that the OO solution cannot be improved with the use of aspects.

# 5 Discussions and Ongoing Work

In this section we aim to provide some discussions about our study relating to the liabilites of previous refactoring approaches. A briefly explanation is given regarding additional metaphors besides those ones explored in Section 3. We also present the study constraints and the ongoing work at the end of this section.

## *5.1 Addressing the Shortcomings of Existing Refactorings*

In section 2.3 we pointed out three liabilities found in previous refactoring approaches: inconsistent terminology of the concern structure, the lack of support to detect concern-related bad smells, and lack of holistic treatment of scattered changes. The metaphor-based refactoring approach helps to address these shortcomings in a number of ways.

First, our concern classification based on metaphors gives an intuitive, consis-

Table 5
Black Sheep refactoring steps

| **Pattern** | **2.a** | **2.b** | **2.c** | **2.d** | **2.e** | **2.f** |
|---|---|---|---|---|---|---|
| Prototype | x | | x | | | |
| Singleton | | | x | | x | x |

Table 6
Summary of refactoring steps per pattern

| Pattern | Classification | Pattern Instance | Number of Refactorings |
|---|---|---|---|
| Decorator | Octopus | Original [15] | 7 |
| | | Extended [13] | 19 |
| Decorator | Octopus | Original [15] | 2 |
| | | Extended [13] | 6 |
| Decorator | Octopus | Original [15] | 16 |
| | | Extended [13] | 44 |
| Decorator | Octopus | Original [15] | 5 |
| | | Extended [13] | 13 |
| Decorator | Octopus | Original [15] | 8 |
| | | Extended [13] | 23 |
| Singleton | Black Sheep | Original [15] | 4 |
| | | Extended [13] | 16 |
| Prototype | Black Sheep | Original [15] | 4 |
| | | Extended [13] | 12 |

tent and unified terminology to classify crosscutting concerns manifested as design flaws. Hence, it addresses this problem in existing refactorings of crosscutting concerns. For example, the role-based approach [16] uses application-specific names, such as CurrencyControl, while the refactoring based on concern types [22] is usually tied to implementation-specific constructs of programming languages. On the other hand, our metaphor-based refactorings are based on a terminology that is abstract enough to be applied in different application domains, technologies and programming languages.

Second, we claim that our approach of concern classification based on metaphors is an efficient way to express concern-related bad smells. As far as we are concerned, existing refactoring approaches which deals with crosscutting concerns modularisation [16] [22] do not use metrics-based analysis for design modularity assessment. Although a number of papers [21] [23] have associated bad smells with modularity metrics and heuristics, existing refactorings do not explore heuristic rules for detection of concern-related bad smells. On the contrary, our refactoring technique copes with this limitation by applying concern-oriented metrics and, then, using the collected data as input to design heuristic rules.

Finally, the main goal of our refactorings is to provide holistic treatment to the modularisation of crosscutting concerns. Conventional refactoring approaches rely mainly on the designers' opinion to decide when and what concerns to aspectise. For example, role-based refactoring requires a designer to identify the crosscutting roles. On the other hand, our heuristic classification provides concrete means to identify which concerns of the system need to be refactored. In other words, Octopus and Black Sheep concerns are, in fact, strong indicatives of bad smells that need to be addressed by refactoring of crosscutting concerns. Furthermore, since our metaphor-based refactorings are composed of low-level refactorings [15] [19] [14] [24], they can be easily automated by existing refactoring tools. Particularly, an metaphor-based refactoring is atomic, i.e., it should be able to roll back and recover a consistent state in case one of its refactoring steps could not execute successfully. We are implementing a tool to support atomic metaphor-based refactorings (Section 5.3).

### 5.2 Additional Metaphor-based Concern Classification

Besides the Octopus and Black Sheep metaphors presented in section 3, we have already defined an initial set of additional concern metaphors (Figure 8): *King Snake*, *Climbing Plant*, *Hereditary Disease*, *Copy Cat*, and *Dolly Sheep*. A King Snake concern has a large noncyclic chain of inter-connected pieces of code. Each element in the chain is connected to one or two other elements in such a way that when one element of the concern is executed, it propagates the call to others. The final called element (in the end of the chain) is optionally called head of the snake.

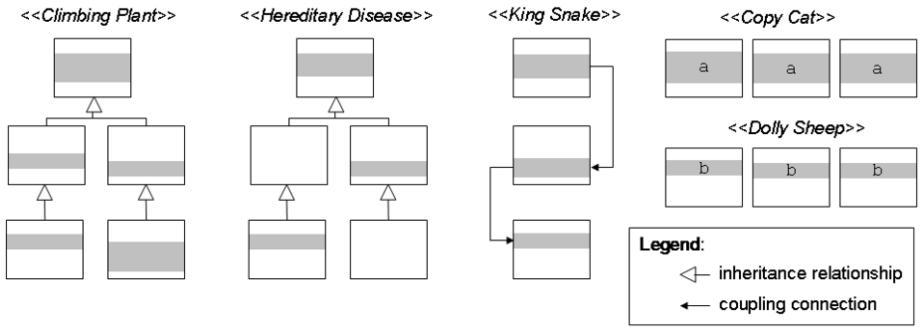


Fig. 8. Abstract representation of five concern metaphors.

Table 7
Heuristics for additional concern metaphors

| Concern Metaphor | Heuristic Description |
|---|---|
| King Snake | The given concern is implemented by inter-connected methods and attributes in different classes. Metrics for concern coupling [7] are used to detect connected pieces of code. |
| Climbing Plant | The given concern is implemented by methods and attributes in all classes in the same inheritance tree. It commonly happens in abstract methods which are implemented by all subclasses. |
| Hereditary Disease | The given concern is implemented by methods and attributes in some classes in the same inheritance tree. It commonly happens in methods which are overridden. |
| Copy Cat | The given concern is implemented by similar pieces of code in different classes. Clone detection techniques [5] help to identify the Copy Cat concerns. |
| Dolly Sheep | The Dolly Sheep concern is detected by composing two previously described heuristics: Copy Cat (above) and Black Sheep (Section 4.1). |

A Climbing Plant is a concern which affects the root of an inheritance tree and, then, propagates its structure to all children of this root. The concern can be totally or partially propagated to the root descendents. However, all descendents of the concern root are somehow affected. As Climbing Plant, a Hereditary Disease concern affects the root of an inheritance tree and, then, propagates its crosscutting structure to some root descendents. However, a Hereditary Disease does not manifest in all nodes of a tree, i.e., some nodes are disease-free. Copy Cat is a concern with replicated code in many places. A special type of Copy Cat occurs when the

concern is also a Black Sheep. In this case, the concern is called Dolly Sheep since it is a replicated Black Sheep.

Figure 8 shows an abstract representation for each category of those five concern metaphors. In the used notation, each box refers to a class and gray areas indicate elements implementing the respective concern. Note that, two shadow areas are labelled with the same letter ('a' or 'b') when those areas have similar pieces of code. We have already defined a set of preliminary heuristic rules to detect those concern metaphors. Table 7 presents a heuristic description to detect each concern metaphor (heuristics for Octopus and Black Sheep were described in Section 3.2). Those heuristics use combined information from concern-oriented metrics, conventional metrics, and other concern analysis techniques, such as clone detection [5]. For example, heuristics aiming at detecting Climbing Plant and Hereditary Disease use metrics for concern scattering and inheritance-based metrics.

### *5.3 Study Constraints and Ongoing Work*

Aspect-oriented refactorings have been proposed to deal with the modularisation of crosscutting concerns [19] [1] [14] [15] [24]. We recommend the use of those low-level refactorings after the proposed Octopus and Black Sheep aspectisation in order to modify the internal structure of aspects and prepared them for reuse. We should highlight that the aspectised versions of concerns following our technique are not expected to directly be considered as optimal solutions. The proposed refactorings (Sections 3.3 and 3.4) are aimed to be generic and applied into any Octopus or Black Sheep concern. Thus, they are not specific enough to obtain the best AO solutions of those design patterns. Although we have not evaluated whether the refactored concerns represent optional solutions, the patterns in our evaluation (Section 4) had their respective crosscutting structure eliminated by the aspectisation which is our main goal.

Our ongoing work encompasses a catalogue of metaphor-driven heuristics and their associated refactorings including the additional metaphor-based classification presented in Table 7. Besides design patterns, we are also applying the metaphor-based refactorings to heterogeneous sets of crosscutting concerns, such as non-functional requirements [9] and features of software product lines [7]. This last step includes the selection of real software systems which allow us to verify the suitability and scalability of our approach.

We are planning as future work to develop a refactoring tool that supports our approach. This tool would help the developer to follow the workflow presented on section 3.1. Basically, it encompasses the identification of design flaws through the metaphor-driven heuristics and the application of the metaphor-based refactorings for each detected concern metaphor. Automation of the refactoring application involves three steps: selection of refactoring steps, human calibration as input parameter, and code transformation.

# 6 Final Remarks

This paper proposes aspect-oriented refactorings to better modularise crosscutting concerns classified by metaphor-based heuristic rules. Our refactoring technique complements previous research work, such as role-based refactoring [18] and refactoring of crosscutting concerns types [22], which has similar goals. In fact, all these approaches target at aspectising crosscutting concerns by using a higher level representation of concerns. However, we explicitly provide a set of heuristics (Section 3.3) to classify concerns before refactoring them. In addition, the concern metaphors used in this paper are more abstract and intuitive than previous categories, such as roles [18] and crosscutting concern types [22].

In our evaluation, we used the GoF design patterns [12] and organised them according to our heuristic-based concern classification (Section 4.1). The heuristic classification presented an accuracy of about 85 % in our preliminary evaluation (Section 4.1). In addition, we applied our refactoring technique to design patterns which match the Octopus and Black Sheep metaphors (Section 4.2). Although the refactorings presented in this paper derive from studies of design patterns [12], they aim to be general-purpose, rather than case specific or pattern specific. Finally, we proposed five additional metaphors (Section 5.2) to describe recurring structures of crosscutting concerns.

# Acknowledgement

This work is supported in part by Brazilian Research Agencies: National Counsel of Technological and Scientific Development (CNPq); Coordination of Higher Education Staff and Graduate Studies (CAPES); and by the European Commission, grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

# References

[1] Binkley, D., M. Ceccato, M. Harman, F. Ricca and P. Tonella, *Automated refactoring of object oriented code into aspects*, in: *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance* (2005), pp. 27–36.

[2] Cacho, N., C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista and C. Lucena, *Composing design patterns: a scalability study of aspect-oriented programming*, in: *AOSD '06: Proceedings of the 5th International Conference on Aspect-oriented Software Development* (2006), pp. 109–121.

[3] Chidamber, S. and C. Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering **20** (1994), pp. 476–493.

[4] Ducasse, S., T. Girba and A. Kuhn, *Distribution map*, in: *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance* (2006), pp. 203–212.

[5] Ducasse, S., O. Nierstrasz and M. Rieger, *On the effectiveness of clone detection by string matching*, Journal on Software Maintenance and Evolution: Research and Practice **18** (2005), pp. 37 – 58.

[6] Eaddy, M., A. Aho and G. C. Murphy, *Identifying, assigning, and quantifying crosscutting concerns*, in: *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques* (2007), p. 2.

[7] Figueiredo, E. and N. Cacho, *Evolving software product lines with aspects: an empirical study on design stability*, in: *ICSE '08: Proceedings of the 13th International Conference on Software Engineering* (2008), pp. 261–270.

[8] Figueiredo, E., C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola and A. Marchetto, *On the maintainability of aspect-oriented software: A concern-oriented measurement framework*, in: *CSMR'08: Proceedings of the 12th European Conference of Software Maintenance and Reengineering*, 2008, pp. 183–192.

[9] Filho, F. C., N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia and C. M. F. Rubira, *Exceptions and aspects: the devil is in the details*, in: *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2006), pp. 152–162.

[10] Filho, F. C., A. Garcia and C. Rubira, *Extracting error handling to aspects: A cookbook*, in: *ICSM '07: Proceedings of the 23nd IEEE International Conference on Software Maintenance* (2007), pp. 134–143.

[11] Fowler, M., K. Beck, J. Brant, W. Opdyke and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison-Wesley Professional, 1999.

[12] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] Garcia, A., C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena and A. von Staa, *Modularizing design patterns with aspects: a quantitative study*, in: *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development* (2005), pp. 3–14.

[14] Garcia, V., E. Piveta, D. Lucrdio, E. Almeida, E., A. Prado and L. Zancanella, *Manipulating crosscutting concerns*, in: *SugarLoafPlop'04: Proceedings of 4th Latin American Conference on Patterns Languages of Programming*, 2004.

[15] Hanenberg, S., C. Oberschulte and R. Unland, *Refactoring of aspect-oriented software*, in: *Proceedings of Net.ObjectDays Conference (NODe'03)*, 2003.

[16] Hannemann, J., *Aspect-oriented refactoring: Classification and challenges*, in: *LATE'06: Proceedings of the 5th International Workshop on Linking Aspect Technology and Evolution at AOSD'06*, 2006.

[17] Hannemann, J. and G. Kiczales, *Design pattern implementation in java and aspectj*, SIGPLAN Not. **37** (2002), pp. 161–173.

[18] Hannemann, J., G. C. Murphy and G. Kiczales, *Role-based refactoring of crosscutting concerns*, in: *AOSD'05: Proceedings of the 4th International Conference on Aspect-oriented Software Development* (2005), pp. 135–146.

[19] Iwamoto, M. and J. Zhao, *Refactoring aspect-oriented programs*, in: *UML'03: Proceedings of the 4th International Workshop on Aspect-oriented Modeling*, 2003.

[20] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Aksit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming* (1997), pp. 220–242.

[21] Lanza, M., R. Marinescu and S. Ducasse, "Object-Oriented Metrics in Practice," Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[22] Marin, M., L. Moonen and A. van Deursen, *An approach to aspect refactoring based on crosscutting concern types*, SIGSOFT Software Engineering Notes **30** (2005), pp. 1–5.

[23] Marinescu, R., *Detection strategies: Metrics-based rules for detecting design flaws*, in: *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance* (2004), pp. 350–359.

[24] Monteiro, M. P. and J. M. L. Fernandes, *Towards a catalogue of refactorings and code smells for aspectj*, Transactions on Aspect Oriented Software Development (TAOSD) - Lecture Notes in Computer Science (2006), pp. 214–258.

[25] Robillard, M. P. and G. C. Murphy, *Representing concerns in source code*, ACM Transactions on Software Engineering Methodologies **16** (2007), p. 3.

[26] Silva, B., *Refactoring of crosscutting concerns with metaphor-based heuristics: Measurement results*, http://www.inf.ufrgs.br/∼bcsilva/sqm_evaluation.html (2008).