Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics

Bruno Silva[1], Eduardo Figueiredo[2], Alessandro Garcia[2], and Daltro Nunes[1]

[1]*Institute of Informatics, Federal University of Rio Grande do Sul, Brazil*
[2]*Computing Department, Lancaster University, United Kingdom*

{bruno.carreiro, daltro}@inf.ufrgs.br, {e.figueiredo, a.garcia}@lancaster.ac.uk

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 a metaphor-based classification of crosscutting concerns, which is driven by their manifested shapes through a system's modular structure. Our categories provide an intuitive vocabulary for concern-oriented design flaws and identify refactorings in terms of recurring crosscutting structures. On top of this classification, we define a suite of heuristic-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 the appropriate ones.

1. Introduction

Aspect-Oriented Programming (AOP) [19] provides explicit constructs for improving the separation of concerns with the goal of enhancing design modularisation. Aspects are new units of modularity for encapsulating crosscutting concerns, i.e., system features or properties that naturally affect many system modules [19]. AOP is 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 systems to the aspect-oriented decompositions in order to reap the benefits of AOP.

Migration of legacy code to aspects requires proper mechanisms for identifying [26] and refactoring crosscutting concerns [17, 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 [16, 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 [6, 26] and refactoring of crosscutting concerns [1, 16]. 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 structure. 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, a case 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.

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 two-dimension 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 [14, 18, 23, 24] have recently identified limitations when applying well-known OO refactorings [10] 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 [14] proposed preconditions which have to be respected when applying a conventional refactoring in the presence of aspects. They use *Extract Class* [10] 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 [18] 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 [10]. 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 charging of figuring out from the scratch how unrelated transformations need to be combined to refactor conventional crosscutting concerns. Therefore, coarse-grained refactorings 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, 16, 22]. For example, some recent cookbooks have been documented to guide the "aspectisation" of specific crosscutting concerns, such as exception handling [8, 9]. 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.

Hannemann and colleagues proposed the notion of role-based refactoring [16] 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 dialog 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 structure. 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 AO Refactoring

An analysis of existing AO refactorings points out fundamental deficiencies in the manner concern refactorings is addressed. We discuss 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 [16, 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, 6, 9].

Inconsistent terminology of the concern structure. Role-based refactoring [16] 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 [17] 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 design [5, 6, 9]. A bad smell is any symptom that indicates something may be wrong and it generally indicates that the overall design should be refactored [10]. Metrics-based analyses are traditionally the fundamental mechanisms for assessing design modularity and detecting design flaws. In fact, a number of papers [20, 21] have associated bad smells with modularity metrics [3], such as coupling, cohesion, and conciseness. For example, Marinescu [21] 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 concern-related bad smells detection; and AO refactorings driven by holistic-detected anomalies.

2.4. A Motivating Example

Consider an OO instance of the Mediator design pattern [11] developed by Hannemann and Kiczales [15] 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 roles of the target pattern and call it "Mediator Refactoring". Similarly, a refactoring to aspectise the Observer design pattern (Figure 3) would be called "Observer Refactoring" in this approach.

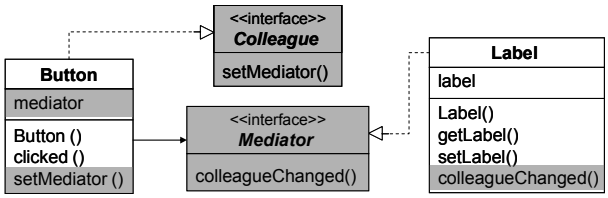

Figure 1. An instance of the Mediator pattern [15]

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 [5, 6] 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). They are applied to crosscutting concerns which possess certain characteristics identified by a set of metrics-based heuristics (Section 3.2). A heuristic is a composed logical condition, based on metrics, which detects design fragments with bad smells [21]. Although we are working on a catalogue of metaphor-driven heuristics and associated refactorings, this paper presents only two of them. Section 3.1 presents the workflow of each heuristic-driven refactoring. Sections 3.3 and 3.4 present examples of refactorings for two concern metaphors.

3.1. Workflow

The heuristic-driven refactorings support an interactive extraction of concerns into aspects and is 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 a vocabulary of crosscutting concerns. We believe that this vocabulary allows developers to use meaningful terms when defining 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, 18]. Since each heuristic-based refactoring of crosscutting concern is also composed of smaller aspect-aware refactorings, existing tools for the latter [1, 18] can be also used to support the former.

3.2. Heuristic-Based Concern Metaphors

Recent research work has classified concerns in different ways [4, 16, 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.

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 presents the aforementioned concern metaphors and the heuristic descriptions used to find them. 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 [20] 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).

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.

Figure 2 illustrates a concrete example of Black Sheep using an implementation of the Singleton design pattern [11] proposed by Hannemann and Kiczales (H&K) [15]. This figure presents the code of the `Printer` 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 its implementation.

```
public class Printer {
  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);
  }
}
```

Figure 2. Singleton pattern as Black Sheep

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 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].

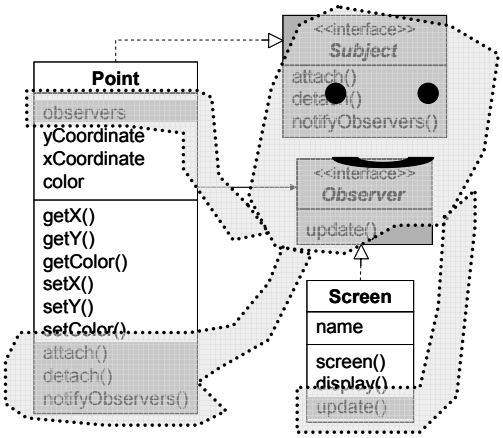

Figure 3. Observer design pattern as Octopus

Figure 3 presents an UML class diagram of an instance to the Observer design pattern [11]. 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.

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 [6, 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.

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.

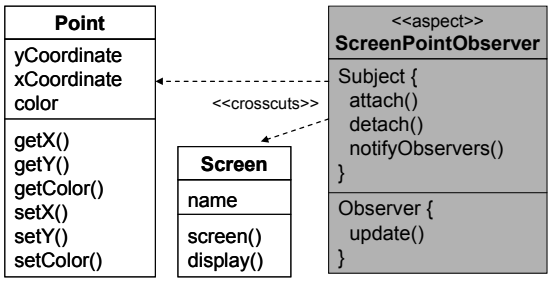

Figure 4. Separation of the Octopus code

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 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 [10] 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.

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

Figure 5. Extracting pointcut and advice in the `Point` tentacle

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 picks up the appropriate joint points (coordinates setting). Additional refactoring steps (e.g., [18, 23, 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 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.

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.

```
public class Printer {
  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);
  }
}
```

Figure 6. Extracting Black Sheep slices

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

Figure 7. Separation of Black Sheep in the `SingletonInstance` aspect

Consistently, during 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 [10] 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 the 23 Gang-of-Four (GoF) design patterns [11].

4.1. On the Heuristic Classification

As explained in Section 3, a heuristic is a composed logical condition based on metrics. Hence, first we have to apply a set of metrics in order to classify a concern using metaphor-based heuristics. We used concern-oriented metrics [5, 6] to get the necessary measurement. We evaluated instances of the 23 GoF design patterns proposed by Hannemann and Kiczales (H&K) [15]. Each design pattern was classified in a concern metaphor by analyzing its roles. Hence, the overall pattern classification depends on the roles' evaluation. 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 at [25].

Tables 2 and 3 present the 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 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.

The crosscutting structure of the GoF design patterns has already been detected and explored in previous studies [2, 13, 15]. These studies enable us to compare our heuristic classification to their findings. Our comparison focuses mainly on the H&K study [15] and on our previous experience on design patterns assessment [2, 13].

Table 2. Octopus design patterns

Pattern	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

Pattern	Roles	Role Classification
Prototype	Prototype	Black Sheep
Singleton	Singleton	Black Sheep

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.

Although a more systematic evaluation is required, this simple comparison between our heuristic classification and previous knowledge allow us to find at least one false positive of our technique. In other works, 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 %. Further 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 refactoring (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 [15] and the second one includes extended instances proposed by Garcia et al. [13]. The difference between them is that Garcia et al. 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, 18] 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

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

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 the original pattern instances and extended ones are essentially the same. They differ 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. Summary of refactoring steps per pattern

Pattern	Classification	Pattern Instance	Number of Refactorings
Decorator	Octopus	Original [15]	7
		Extended [13]	19
Iterator	Octopus	Original [15]	2
		Extended [13]	6
Observer	Octopus	Original [15]	16
		Extended [13]	44
Visitor	Octopus	Original [15]	5
		Extended [13]	13
Mediator	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

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, button clicking, etc.).

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

Aspect-oriented refactorings have been proposed to deal with the modularisation of crosscutting concern [1, 12, 14, 23, 24]. We recommend the use of those low-level refactorings after the Octopus and Black Sheep aspectisation in order to modify the internal structure of aspects and prepared them for reuse.

Of course, the aspectised versions of concerns following our technique do not represent an optimal solution. 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 solution of those design patterns. Even though we have these considerations, the patterns in our evaluation 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. In fact, we have already defined an initial set of concern metaphors including King Snake, Climbing Plants, Hereditary Disease, Copy Cat, and Dolly Sheep. A *King Snake* concern has a large non-cyclic 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 one end of the chain) is optionally called head of the snake.

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.

In addition to design patterns, we are also planning the application of metaphor-based refactorings to heterogeneous set of crosscutting concerns, such as non-functional requirements [9] and features of software product lines [7]. This next step includes the selection of real software systems which allow us to verify the suitability and scalability of our approach.

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 [16] 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 3.2) 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 [16] and crosscutting concern types [22].

In our evaluation, we used the GoF design patterns [11] 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 [11], they aim to be general-purpose, rather than case specific or pattern specific.

7. Acknowledgements

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).

7. References

[1] Binkley, D. et al. "Automated Refactoring of Object-Oriented Code into Aspects". In Proc. of the 21st IEEE Int'l. Conference on Software Maintenance (ICSM'05), pp. 27-36, Washington, DC, USA, 2005.

[2] Cacho, N. et al. "Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming". Proc. of 5th Int'l. Conf. on Aspect Oriented Software Development (AOSD'06), pp. 109-121, Bonn, Germany, 2006.

[3] Chidamber, S. and Kemerer, C. "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, pp. 476-493, 1994.

[4] Ducasse, S., Girba, T., and Kuhn, A. "Distribution Map". In Proc. of the 22nd IEEE International Conference on Software Maintenance (ICSM'06), pp. 203-212, Philadelphia, USA, 2006.

[5] Eaddy, M.; Aho, A.; and Murphy, G. "Identifying, Assigning, and Quantifying Crosscutting Concerns". Proc. of the Workshop on Assessment of Contemporary Modularization Techniques (ACoM), 2007.

[6] Figueiredo, E. et al. "On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework". In Proc. of 12th European Conference on Software Maintenance and Reengineering (CSMR'08). Athens, Greece, 2008.

[7] Figueiredo, E. et al. "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability". To appear in Proc. of Int'l. Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008.

[8] Filho, F., Garcia, A., Rubira, C. "Extracting Error Handling to Aspects: A Cookbook". In Proc. of the 23rd Int'l. Conference on Software Maintenance (ICSM'07), pp. 134-143, Paris, France, 2007.

[9] Filho, F. et al. "Exceptions and Aspects: The Devil is in the Details". In Proc. of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'06), pp. 152-162, Portland, USA, 2006.

[10] Fowler, M. "Refactoring: Improving the Design of Existing Code". Addison-Wesley, Reading, MA, USA, 1999.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

[12] Garcia, V. et al. "Manipulating Crosscutting Concerns". In Proc. of 4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop), Brazil, 2004.

[13] Garcia, A. et al. "Modularizing Design Patterns with Aspects: A Quantitative Study". In Proc. of 4th Int'l. Conference of Aspect-Oriented Software Development, (AOSD'05), pp. 3-14, Chicago, USA, 2005.

[14] Hanenberg, S., Oberschulte, C., and Unland, R. "Refactoring of Aspect-Oriented Software". In Proc. of Net.ObjectDays Conference (NODe'03), 2003.

[15] Hannemann, J., and Kiczales, G. "Design Pattern Implementation in Java and AspectJ". In Proc. of the 17th ACM Conference on Programming, Systems, Languages, and Applications (OOPSLA'02), pp. 161-173. Seattle, 2002.

[16] Hannemann, J., Murphy, G., and Kiczales, G. "Role-Based Refactoring of Crosscutting Concerns". In Proc. of the 4th Int'l. Conference of Aspect-Oriented Software Development (AOSD'05), pp. 135-146. Chicago, 2005.

[17] Hannemann, J. "Aspect-Oriented Refactoring: Classification and Challenges". In Proc. of the 2nd AOSD Workshop on Linking Aspect Technology and Evolution (LATE'06), Chicago, 2006.

[18] Iwamoto, M. and Zhao, J. "Refactoring Aspect-Oriented Programs". In Proc. of the 4th Int'l. Workshop on Aspect-Oriented Modeling at UML Conference (AOM'03), San Francisco, USA, 2003.

[19] Kiczales, G. et al. "Aspect-Oriented Programming". In Proc. of the 21st European Conference on Object-Oriented Programming (ECOOP'97), pp. 220-242, Berlin, 1997.

[20] Lanza, M., and Marinescu, R. "Object-Oriented Metrics in Practice". Springer, 2006.

[21] Marinescu, R. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws". In Proc. of the 20th Int'l. Conference on Software Maintanance (ICSM'04), pp. 350-359, Chicago, USA, 2004.

[22] Marin, M., Moonen, L., and van Deursen, A. "An Approach to Aspect Refactoring Based on Crosscutting Concern Types". ACM SIGSOFT Software Engineering Notes, 30(4), pp. 1-5, 2005.

[23] Miller, G. "Refactoring with Aspects". In Proc. of 4th International Conference on Extreme Programming, pp. 343-346, Genova, Italy, 2003.

[24] Monteiro, M., Fernandes, J, "Towards a Catalogue of Refactorings and Code Smells for AspectJ". Transactions on AOSD, Springer LNCS 3880, pp. 214-258, 2006.

[25] Refactoring of Crosscutting Concerns with Metaphors. http://www.inf.ufrgs.br/~bcsilva/sqm_evaluation.html

[26] Robillard, M., and Murphy, G. "Representing Concerns in Source Code". ACM Transactions on Software Engineering and Methodology (TOSEM), 16(1), 2007.