Towards a Catalog of Aspect-Oriented Refactorings

Miguel P. Monteiro
Escola Superior de Tecnologia
Instit. Politécnico de Castelo Branco
Avenida do Empresário
6000-767 Castelo Branco PORTUGAL
mmonteiro@di.uminho.pt

João M. Fernandes
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga PORTUGAL
jmf@di.uminho.pt

ABSTRACT
In this paper, we present a collection of aspect-oriented refactorings covering both the extraction of aspects from object-oriented legacy code and the subsequent tidying up of the resulting aspects. In some cases, this tidying up entails the replacement of the original implementation with a different, centralized design, made possible by modularization. The collection of refactorings includes the extraction of common code in various aspects into abstract superaspects. We review the traditional object-oriented code smells in the light of aspect-orientation and propose some new smells for the detection of crosscutting concerns. In addition, we propose a new code smell that is specific to aspects.

Keywords
Aspect-oriented programming, object-oriented programming, refactoring, code smells, programming style.

1. INTRODUCTION
Refactoring [11][24] and Aspect-oriented programming (AOP) [17] are two techniques that contribute to deal with the problems of permanent evolution of software. Refactoring facilitates the continuous change of source code, enabling it to evolve in line with changes in environments and requirements. AOP provides stronger modularization and software composition mechanisms than those provided by previous technologies, thus diminishing the potential impact that changes to the code related to a given concern have on code unrelated to that concern.

AOP's steady progress from "bleeding edge" research field to mainstream technology [25] brings forward the problem of how to deal with large number of object-oriented (OO) legacy code bases. Experience with refactoring of OO software in the last half-decade suggests that refactoring techniques have the potential to bring the concepts and mechanisms of aspect-orientation to existing OO frameworks and applications.

In this paper, we review some of the traditional OO *code smells* [11] in the light of AOP and we propose several new AOP-specific smells. We present a collection of AOP refactorings aiming to remove those smells from legacy code (including updated versions of 4 of the refactorings presented in [23]). The subject language we use is AspectJ [19], whose backward compatibility with Java opens the way for refactoring existing Java applications by introducing AOP constructs. Space constraints prevent us from providing the complete descriptions of the refactorings in this paper: these can be found in [21].

We do not claim these collections are complete or comprehensive, but we believe they extend the existing proposals [18][14][15], thus contributing to further mature AOP. Though 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.

The rest of this paper is structured as follows. In section 2, we provide the motivation for our work. In section 3, we describe the approach we took to develop the collection of refactorings. In section 4, we review some of the traditional smells in the light of AOP, provide the motivation for new code smells, specific to AOP, and propose several such smells. These smells require new refactorings targeting AOP specific constructs, which we present in section 5. In section 6, we survey related work and in section 7 we consider future directions. In section 8, we summarize the paper.

2. MOTIVATION
We believe there are three main hurdles in need to be addressed so that refactoring techniques can be used in AOP software in an effective and widespread way.

The first hurdle is the present lack of a fully developed idea of "good" AOP style. This is an important issue, for a clear notion of style is a fundamental prerequisite for the use of refactoring, enabling programmers to see where they are heading when refactoring their code. Fowler *et al.* [11] advocated a specific notion of style for OO code through a catalog of 22 *code smells*, compounded by a catalog of 72 refactorings through which those smells can be removed from existing code. These catalogs proved very useful in bringing the concepts of refactoring and good OO style to a wider audience and in providing programmers with guidelines on when to refactor and how best to refactor. Refactoring and a notion of good style are key concepts of *Extreme Programming* [1], which regards a system's source code as primarily a communication mechanism between people, rather than computers.

A second hurdle – both a cause and a consequence of the first – is the present lack of an AOP equivalent of such catalogs. Our work is based on the assumption that AOP would equally benefit from AOP specific catalogs of smells and refactorings, helping programmers to detect situations in the source code that could be improved with aspects and guiding them through the corresponding transformation processes.

A third hurdle is the absence of tool support for AOP constructs in current integrated development environments. The catalogs presented by Fowler *et al.* [11] provided a basis on which developers could rely to build tool support for OO refactoring: similar catalogs for AOP are likely to bring similar benefits to tool developers. Tool developers will not be able to provide adequate support to refactoring operations unless they first have a clear idea of AOP style, and consequently of exactly which refactorings are worthy of their development efforts. Though we developed the refactorings presented in this paper to be performed manually, we believe they can be helpful to developers of tool support in identifying useful material on which they can focus their work.

3. THE APPROACH

We took the approach of using refactoring experiments based on case studies, as a vehicle for gaining the necessary insights. The case studies we used are code bases in Java and AspectJ with the appropriate structural characteristics. We approached the Java code bases as bad-style or "smelly" AspectJ code, and searched for the kinds of refactorings that would be effective in removing those smells.

The first case study comprised the extraction of one crosscutting concern from a workflow framework, whose results are presented in [23]. Our second case study was the collection of implementations (version 1.1) in both Java and AspectJ of the 23 Gang-of-Four (GoF) design patterns [12], presented by Hannemann and Kiczales [13].

The 23 GoF patterns illustrate a variety of design and structural issues that would be hard to find in a single code base (except in very large and complex systems). The GoF patterns effectively comprise a microcosm of many possible systems. They provided us with a rich source of insights, without the need to analyze large code bases or learn domain-specific concepts.

The implementations presented by Hannemann and Kiczales [13] are currently one of the nearest things to examples of good AOP style and design, presenting a clear notion of the desirable internal structure for aspects. Many of the findings presented in this paper stem from our study of these examples, compounded with studies of Java implementations of the same patterns by various authors [9][6], which further enriched the patterns' potential as providers of insights.

Our approach was to pinpoint the refactorings that would be needed to transform the Java implementations into the AspectJ implementations. We then tested and refined the refactorings thus obtained on other Java implementations of the same patterns [9][6]. The refactoring process described in [22] derived from one of our test sessions.

4. CODE SMELLS

Code smells are the way proposed by Beck and Fowler (chap.3 of [11]) to diagnose problems in existing code that could be removed through refactorings. Code smells do not aim to provide precise criteria for when refactorings are overdue. Instead, code smells suggest symptoms that *may* be indicative of something wrong in the code. Programmers are required to develop their own sense of when a symptom indeed warrants a change. Decisions also depend on the specific aims of the programmer and the specific state and structure of the code on which he is working.

4.1 On the Need for AOP Specific Smells

The notion of style in a programming language expresses the coding practices that yield code easier to maintain and evolve. Whenever a programming language provides alternative ways to achieve some result, the way that causes the least problems to present and future programmers is the one considered in the best style. Throughout the various stages of development of programming languages, many ideas of style appeared due to the advent of new, superior mechanisms. We briefly mention three examples:

1. Dijkstra's famous dictum that the "Go-to statement [should be] considered harmful" [8] stemmed from the availability of control structures, namely loops.
2. Fowler *et al.* [11] considered the use of the switch statement to be a code smell, due to the availability of polymorphism and dynamic binding.
3. Orleans suggested in [20] that the 'if' statement be considered harmful in the context of languages using elaborate forms of predicate dispatch.

All these considerations suggest that the appropriate notion of style for a given language strongly depends on what can be achieved with that language. In this light, the suitable style of AspectJ can not be the same as for Java. AspectJ enables programmers to perform compositions that are impossible with Java and avoid negative qualities such as code scattering and code tangling. This suggests that many of traditional OO solutions resulting in those negative qualities should now be considered bad style, including the OO implementations of some design patterns [13].

The very compositional power of AspectJ can be cause for problems. AspectJ offers multiple ways to achieve various effects and compositions. For instance, the implementation of mixins [2] can be achieved both through marker interfaces and through inner static aspects placed within interfaces. Likewise, non-singleton aspect associations provide alternatives to solutions obtained with the default singleton aspects. AspectJ programmers are sometimes faced with so many choices that it becomes hard to decide on the design most appropriate to a particular situation. There is a need to further study the consequences and implications of each solution in order to make choices clear. We believe that catalogs of code smells and refactorings are an effective way to present this knowledge to programmers.

4.2 OO Smells in the Light of AOP

We analyzed the code smells presented in [11], [30] and [16], and believe some can be used by AOP programmers as symptoms of the presence of crosscutting concerns. This particularly applies to *Divergent Change* ([11], p.79) and *Shotgun Surgery* ([11], p.80). According to Fowler *et al.*, "*Shotgun Surgery* is one change that alters many classes" (i.e. a symptom of code scattering) and "*Divergent Change* is one class that suffers many kinds of changes" (i.e. a symptom of code tangling). We think it is useful to extend these definitions to cover methods as well as classes. Wake [30] mentions configuration information, logging and persistence as possible causes to the *Shotgun Surgery* smell, all of which can be counted among the favorite examples for the use of AOP.

Kerievsky [16] proposes a variant of *Shotgun Surgery* that he calls *Solution Sprawl*. Kerievsky states ([16], p.43) that "you become aware of this smell when adding or updating a system feature causes you to make changes to many different pieces of code". The difference between the two smells is the way they are sensed – "we become aware of *Solution Sprawl* by observing it, while we detect *Shogun Surgery* by doing it". Both variants are equally promising as indicators of crosscutting concerns.

We propose the *Extract Feature into Aspect* refactoring ([21], p.5; see also Table 1 and section 5.2) as a general framework for the modularization of concerns detected through these smells.

4.3 The *Double Personality* Code Smell

The *Double Personality* smell can be found in classes that play multiple roles. Ideally, each class should play a single role, meaning that it contains only one, coherent, set of responsibilities. This often is not possible in OO frameworks and applications.

Examples of *Double Personality* can be found in the OO implementations of design patterns [12] that include what Hannemann and Kiczales call *superimposed roles* – roles assigned by the pattern to classes that have functionality and responsibility outside the pattern [13]. Examples are the Chain of Responsibility ([12], p.223) pattern, which superimposes the Handler role to some of the participant classes, and the Observer pattern ([12], p.293), which superimposes the Subject and Observer roles.

One symptom that can help to detect *Double Personality* in Java source code is implementation of interfaces. Interfaces are a popular way to model roles in Java – e.g. the motivation for *Extract Interface* ([11], p.341). When a class implements an interface modeling a role that does not relate to the class' primary concern, the class smells of *Double Personality*.

When *Double Personality* is detected in one class, we suggest that developers analyze the code base to see if it applies to just that class. Again, looking to the interfaces may help: if multiple classes implement the interface, this means the secondary concern is crosscutting (it cuts across multiple classes).

If only one class is affected, or if the code of the secondary role is restricted to the implementation of the interface, the solution is to extract the secondary role to a mixin [2]. There are several ways to do this. Laddad's *Extract Interface Implementation* [18] suggests placing the secondary concern inside an inner aspect enclosed within the interface modeling the superimposed role. If the programmer strives for total obliviousness [10] of the secondary role, she can use *Replace Implements with Declare Parents* ([21], p.21; Table 1). As an alternative to *Extract Interface Implementation* [18], we propose *Split Abstract Class into Aspect and Interface* ([21], p.21; Table 1), which completely encapsulates the secondary concern into an aspect, which introduces the extra state and behavior to the interface.

When the related code is more complex than a simple implementation of an interface, we suggest using *Extract Feature into Aspect* ([21], p.5; Table 1) to move all the related code to an aspect (see section 5.2).

4.4 Abstract Classes as a Code Smell

The AspectJ composition mechanisms enabling the emulation of mixins [2] also enable the separation of definitions (i.e. implementation code) from declarations in abstract classes, so that these can be turned into interfaces. Hannemann and Kiczales take this approach in implementing five of the GoF design patterns in AspectJ [13]. This separation has the advantage that classes become free to inherit from some other class and interfaces can still be provided with a default implementation. This suggests that abstract classes should now be considered a code smell. Two of the refactorings presented here (see Table 1) remove that smell by moving implementation code to an aspect and turning abstract classes into interfaces. We use *Split Abstract Class into Aspect and Interface* ([21], p.21) to extract the concrete members of an abstract class into an aspect, and we turn the resulting pure abstract class into an interface using *Change Abstract Class to Interface* ([21], p.4).

4.5 The *Aspect Laziness* Code Smell

The *Aspect Laziness* smell applies to aspects that do not carry the full weight of their responsibilities and instead pass the burden to classes, in the form of inter-type declarations. We detect this smell in aspects that resort to the mechanism of inter-type declarations to add state and behavior to a class when something more dynamic and/of flexible would be desirable. AspectJ inter-type declarations are a static mechanism, applying to all instances of the target class, throughout their entire life cycles. We detect the *Aspect Laziness* smell in uses of inter-type declarations for solving problems whose requirements have one or several of the following characteristics:

- The additional state and/or behavior are needed by only a subset of the instances of the target classes.
- The additional state and/or behavior are needed only during certain specific phases in the execution of the program.
- Instances of the target classes (may) require multiple instances of that state and behavior simultaneously.

In such cases, the mechanism of inter-type declarations is not dynamic or flexible enough. It is preferable for the aspect itself to hold the additional state and behavior and manage a map between the additional state and the specific target instances.

We propose *Replace Inter-type Field with Aspect Map* ([21], p.28) and *Replace Inter-type Method with Aspect Method* ([21], p.33) to replace the existing design with a mapping logic that provides the same functionality more flexibly.

4.6 Evil Demons

Fowler *et al.* [11] also make brief references to *evil demons*, symbolizing wrong ways of thinking that negatively impact on the code. One is *Procedural Thinking*, representing approaches to OO programming stemming from procedural programming. We detect a similar problem with AOP – *Object-Oriented Thinking*, or *Decentralized Thinking* – an OO-style approach to the use of aspect-specific constructs. This thinking translates in not appreciating that aspects can hold state of their own, and in designs that excessively rely on inter-type declarations, in fact recreating within an aspect the decentralized designs typical of OO. Such designs lead to the *Aspect Laziness* smell.

4.7 Illustrative Example

In this section, we present a code example that is used in various sections of this paper to illustrate several smells and effects of some of the refactorings. The example is based on Eckel's implementation [9] of the Observer pattern [12]. We describe a refactoring process targeting this example in [22].

The intent of Observer is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [12]. The example includes two observers, one of which is shown in Figure 1, with the primary concern shaded (the other observer, class Hummingbird, is similar). Figure 2 shows the class playing the role of Subject: Flower (shaded code relates to the primary concern). Each of Flower's two operations, open and close the petals, gives rise to one observing relationship.

```
01 public class Bee {
02    private String name;
03    private OObserver oObserver = new OObserver();
04    private CObserver cObserver = new CObserver();
05
06    public Bee(String nm) { name = nm; }
07    private class OObserver implements Observer {
08       public void update(Observable o, Object a){
09          System.out.println(
10             "Bee " + name + "'s breakfast time!");
11       }
12    }
13    private class CObserver implements Observer {
14       public void update(Observable o, Object a){
15          System.out.println(
16             "Bee " + name + "'s bed time!");
17       }
18    }
19    public Observer openObs() {
20       return oObserver;
21    }
22    public Observer closeObs() {
23       return cObserver;
24    }
25 }
```

Figure 1. Bee class as Observer in the implementation of the Observer pattern from [9].

Eckel's implementation uses the Observer/Observable protocol from Java's standard java.util API, which requires the Subject participant to inherit from java.util.Observable. Eckel's design manages to separate the two observing relationships by defining inside each participant an inner class for each relationship. Thus, Flower defines 2 inner classes (Figure 2, lines 27-39 and 40-51 respectively) that inherit from java.util.Observable. Flower uses 2 inherited methods: (1) setChanged (lines 31 and 44), used to mark a subject as having been changed, and (2) notifyObservers, which notifies all its observers if the subject was changed. Though notifyObservers is overridden (lines 29-35 and 42-48), its functionality is reused (lines 32 and 45).

```
01 public class Flower {
02    private boolean isOpen;
03    private ONotifier oNotify = new ONotifier();
04    private ClNotifier cNotify = new ClNotifier();
05
06    public Flower() {
07       isOpen = false;
08    }
09    public void open() { // Opens its petals
10       System.out.println("Flower open.");
11       isOpen = true;
12       oNotify.notifyObservers();
13       cNotify.open();
14    }
15    public void close() { // Closes its petals
16       System.out.println("Flower close.");
17       isOpen = false;
18       cNotify.notifyObservers();
19       oNotify.close();
20    }
21    public Observable opening() {
22       return oNotify;
23    }
24    public Observable closing() {
25       return cNotify;
26    }
27    private class ONotifier extends Observable {
28       private boolean alreadyOpen = false;
29       public void notifyObservers() {
30          if(isOpen && !alreadyOpen) {
31             setChanged();
32             super.notifyObservers();
33             alreadyOpen = true;
34          }
35       }
36       public void close() {
37          alreadyOpen = false;
38       }
39    }
40    private class ClNotifier extends Observable {
41       private boolean alreadyOpen = false;
42       public void notifyObservers() {
43          if(isOpen && !alreadyOpen) {
44             setChanged();
45             super.notifyObservers();
46             alreadyOpen = true;
47          }
48       }
49       public void close() {
50          alreadyOpen = false;
51       }
52 }
```

Figure 2. Flower class as Subject in the implementation of the Observer pattern from [9].

Each observer likewise encloses one inner class implementing java.util.Observer for each observing relationship (Figure 1, lines 07-12 and 13-18 respectively). As prescribed by the interface, each inner class defines an update method (lines 08-11 and 14-17). Because of this design, all participants betray strong doses of *Double Personality*.

As is plain from the example, OO does not cope well with concerns affecting multiple objects and classes, forcing programmers to produce decentralized designs for crosscutting concerns, when they would like to centralize the concern's implementation within some module. Such designs lead to duplicated code in every class playing some role in the concern.

Programmers trying to cope with code scattering and tangling often resort to interfaces and/or inner classes to ameliorate the effects. These constructs improve both the interface and internal

structure of classes: interface types help to better organize the interactions of a class with other classes, and inner classes help to better structure the internals of a class, namely to separate the code related to the class' primary concern from unrelated code. We believe the limitations in the compositions achievable with OO provide one of the motivations to use inner classes and interfaces. Independent authors reached the same conclusion regarding interfaces [28].

5. THE REFACTORINGS

This section presents an overview of our refactorings. These are fully documented in [21], using a format and level of detail similar to the one used by Fowler *et al.* [11] (Kerievsky took the same approach in [16]). The format includes (1) name, (2) typical situation, (3) recommended action, (4) motivation stating the situations when applying the refactoring is desirable, (5) a detailed Mechanics section, and (6) code examples. Tables 1-3 present the refactorings, mentioning the first three elements of the format.

The refactorings do not attempt to cover all possible situations that can potentially arise in source code. For instance, they do not cover uses of reflection. Likewise, they do not deal with what we call the *fragile base code problem* [23][22], also known as the *fragile pointcut problem* [26] – caused by the fact that almost all refactorings can potentially break existing aspects, particularly pointcuts. We believe human programmers will only be able to deal thoroughly with this problem when provided with a new generation of tools, specifically designed to account for the presence of aspects. However, we also believe it is possible to keep this problem under control, provided adequate practices are followed, including programming AspectJ's constructs with a prudent and appropriate style, such as that proposed by Laddad [18]. This is particularly important with pointcuts, which should be made in a style stressing intent rather than a specific case (e.g. expressions using wildcards). This way pointcuts can express a general policy and may be robust enough not to be affected by minor modifications in the target code, such as the removal or addition of a new class or method. Another good practice is to place the aspects close to the code they affect whenever possible, to increase the likelihood that all team members are aware of the aspects potentially affected by refactorings. This often entails placing the aspect in the same package, or even within the same source file as the target class (as inner or peer aspects).

The traditional OO refactorings can be used in AspectJ code as well. We did not detect any refactoring from [11] targeting an OO construct that could not be applied to that construct within aspects. For instance, in the mechanics of *Extend Marker Interface with Signature* ([21], p.24) we prescribe the use of *Extract Method* ([11], p.110) inside aspects.

5.1 Grouping the Refactorings

The collection is structured in groups of refactorings with similar purposes, as is done in [11]. The adopted grouping also reflects a strategy likely to be followed in many refactoring processes. This establishes that prior to anything else, all elements related to a crosscutting concern should be moved to a single module (following *Extract Feature into Aspect* [21], p.5). Only afterwards should we start improving the underlying structure of the resulting aspects (following *Tidy Up Internal Aspect Structure* [21], p.36), because such tasks are considerably easier to perform after the associated implementation is modularized. In case duplication is detected among different but related aspects, we extract the commonalities to a (possibly reusable) superaspect (using *Extract Superaspect* [21], p.37). This strategy leads to the following grouping: (1) extraction of crosscutting concerns, (2) improving of the internal structure of an aspect, and (3) generalization of aspects. The sequence of code transformations described in [22] also fits naturally with this grouping.

The three refactorings mentioned above are composite refactorings. Rather than prescribe specific actions on the source code, as is the case of those documented in [11], they provide a framework for the other refactorings from the same group, specifying the situations when they should be used and when they should not. For this reason they also provide suitable entry points to someone approaching the catalog.

The use of a composite refactoring is useful to provide a broader view of a refactoring process. For instance, most extraction processes as prescribed by *Extract Feature into Aspect* ([21], p.5) entail a relaxation of the access qualifier of fields (usually private), for a period spanning several refactorings (while there is code accessing the field both inside and outside the aspect). The above composite refactoring enables us to specify exactly when the access to the field should be relaxed, and when it can again be turned private to the aspect.

5.2 Refactorings for Extracting Features to Aspects

We expect the refactorings from this group will comprise the starting point for the majority of the refactoring processes targeting OO legacy code.

Extract Feature into Aspect ([21], p.5) pinpoints the procedures for extracting the scattered elements of a crosscutting concern into a single module. Four of the refactorings from this group are updated versions of those presented in [23]. *Extract Fragment into Advice* ([21], p.9) is an updated version of *Extract Advice* from [23][22]).

We suggest using *Move Field From Class to Inter-type* ([21], p.17) to move state to the aspect. Behavior can be moved using *Move Method From Class to Inter-type* ([21], p.19) and *Extract Fragment into Advice* ([21], p.9).

Moving an inner class to an aspect is done in two stages: first using *Extract Inner Class to Standalone* ([21], p.13), to obtain a standalone class from the inner class, and next using *Inline Class within Aspect* ([21], p.15) to turn the resulting class into an inner class within the aspect. We did not see a justification for defining a refactoring equivalent to *Extract Inner Class to Standalone* ([21], p.13) for interfaces, as interfaces are not generally used within classes. Interfaces are inlined into aspects using *Inline Interface within Aspect* ([21], p.16), after which they can be turned into marker interfaces.

We propose *Replace Implements with Declare Parents* ([21], p.21) for inlining the implements clause of classes implementing the interfaces.

Table 1 – Refactorings for Extraction of Crosscutting Concerns

Name of the refactoring	Typical situation	Recommended action
Change Abstract Class to Interface	An abstract class prevents their subclasses from inheriting from another class	Turn the abstract class into an interface and change its relationship with the subclasses from inheritance to implementation
Extract Feature into Aspect	Code related to a feature is scattered across several methods and classes, tangled with unrelated code	Extract all the implementation elements related to the feature to an aspect
Extract Fragment into Advice	Part of a method is related to a concern whose code is being moved to an aspect	Create a pointcut capturing the required joinpoint and context and move the code fragment to an appropriate advice based on the pointcut
Extract Inner Class to Standalone	An inner class relates to a concern being extracted into an aspect	Eliminate dependencies from the enclosing class and turn the inner class into a standalone class
Inline Class within Aspect	A small standalone class is used only by code within an aspect	Move the class to within the aspect
Inline Interface within Aspect	One or several interfaces are used only by an aspect	Move the interfaces to inside the aspect
Move Field from Class to Inter-type	A field relates to a concern other than the primary concern of its enclosing class	Move the field from the class to the aspect as an inter-type declaration
Move Method from Class to Inter-type	A method belongs to a concern other than the primary concern of its owner class	Move the method into the aspect encapsulating the secondary concern as an inter-type declaration
Replace Implements with Declare Parents	Classes implement interface related to a secondary concern. Implementation of the interface is used only when the related concern is present in the system	Replace the implements in the class with a declare parents in the aspect
Split Abstract Class into Aspect and Interface	Classes are prevented from using inheritance because they already inherit from an abstract class defining some concrete members	Move all concrete members from the abstract class to an aspect. You can then turn the abstract class into an interface

Table 2 – Refactorings for Restructuring the Internals of Aspects

Name of the refactoring	Typical situation	Recommended action
Extend Marker Interface with Signature	An inner interface represents a role used only within the aspect. You would like the aspect to call a method specific to one implementing type, not declared by the interface	Add an inter-type abstract declaration of the specific signature to the interface
Generalize Target Type with Marker Interface	An aspect refers to specific concrete types, preventing it from being reused	Replace the references to specific types with a marker interface and make the specific types implement the marker interface
Introduce Aspect Protection	You would like a inter-type member to be visible in an aspect al all its subaspects, but not outside the aspect inheritance chain	Declare the inter-type member as public and place a declare error preventing its use outside the aspect inheritance chain
Replace Inter-type Field with Aspect Map	An aspect statically introduces additional state to a set of classes, when a more dynamic or flexible link between state and targets would be desirable	Replace the inter-type declarations with a structure owned by the aspect performing a map between the additional state and target objects
Replace Inter-type Method with Aspect Method	An aspect introduces additional methods to a class or interface, when a more dynamic and flexible composition would be desirable	Replace the inter-type method with a aspect method getting the target object as parameter
Tidy Up Internal Aspect Structure	The internal structure of an aspect resulting from the extraction of a crosscutting concern is sub-optimal	Tidy up the internal structure of the aspect by removing duplication and dependencies on case specific target types

Table 3 – Refactorings to deal with Generalization

Name of the refactoring	Typical situation	Recommended action
Extract Superaspect	Two or more aspects contain similar code and functionality	Move the common features to a superaspect
Pull Up Advice	All subaspects use the same advice acting on a pointcut declared in the superaspect	Move the advice to the superaspect
Pull Up Declare Parents	All subaspects use the same declare parents	Move the declare parents to the superaspect
Pull Up Inter-type Declaration	An inter-type declaration would be best placed in the superaspect	Move the inter-type declaration to the superaspect
Pull Up Marker Interface	All subaspects use a marker interface to model the same role	Move the marker interfaces to the superaspect
Pull Up Pointcut	All subaspects declare identical pointcuts	Move the pointcuts to the superaspect
Push Down Advice	A piece of advice is used by only some subaspects, or each subaspect requires a different advice	Move the advice to the subaspects that use it
Push Down Declare Parents	A declare parents in a superaspect is not relevant for all the subaspects	Move the declare parents to the subaspects where it is relevant
Push Down Inter-type Declaration	An inter-type declaration would be best placed in a subaspect	Move the inter-type declaration to the subaspect where it is relevant
Push Down Marker Interface	A marker interface declared within a superaspect models a role used only in some subaspects	Move the marker interface to those subaspects
Push Down Pointcut	A pointcut in the superaspect is not used by some subaspects inheriting it	Move the pointcut to those subaspects that use it

Figure 3 shows the participants from Figures 1-2, after each of the two observing relationships was extracted to its own aspect, using the refactorings presented in this section. During the extraction of both observing relationships [22] the isOpen field (line 4) was encapsulated, yielding two new methods for the Flower class: isOpen (lines 6-8) and setIsOpen (lines 9-11). The code for the reaction of the observers when they are notified of open and close events was likewise extracted to methods breakfastTime (lines 26-29) and bedtimeSleep (lines 30-33) respectively. Figure 4 shows part of the aspect related to observing the open operation. The other aspect (not shown), related to the observation of close, is similar.

5.3 Restructuring the Internals of Aspects

We can see from Figures 3 and 4 that the code for implementing the Observer pattern is no longer spread across the participant classes, but the structure of the aspect resulting from the extraction still hardly resembles the one presented in [13], as it ideally would be the case (Figure 5 shows a refactored structure closer to that presented in [13]). The aspect's internal structure still relates to the original, decentralized, design. The aspect betrays *Duplicated Code* ([11], p.76), as it introduces identical fields (Figure 4, lines 10-11 and 12-13) and methods (lines 18-20 and 21-23) to the two observer participants. The duplication has always been present, but now that the code is modularized, it is clearly exposed. After modularization, the original design is no longer justified and the inner classes comprise a needlessly complicated structure.

The code also betrays *Aspect Laziness*, because in this example it is desirable to select the individual objects participating in the observing relationships and the moments when these become effective, but the present structure does not enable this.

```
01 public class Flower {
02    private boolean isOpen;
03    public Flower() {
04       isOpen = false;
05    }
06    boolean isOpen() {
07       return isOpen;
08    }
09    private void setIsOpen(boolean newValue) {
10       isOpen = newValue;
11    }
12    public void open() { // Opens its petals
13       System.out.println("Flower open.");
14       setIsOpen(true);
15    }
16    public void close() { // Closes its petals
17       System.out.println("Flower close.");
18       setIsOpen(false);
19    }
20 }
```

```
21 public class Bee {
22    private String name;
23    public Bee(String nm) {
24       name = nm;
25    }
26    public void breakfastTime() {
27       System.out.println(
28          "Bee " + name + "'s breakfast time!");
29    }
30    public void bedtimeSleep() {
31       System.out.println(
32          "Bee " + name + "'s bed time!");
33    }
34 }
```

Figure 3. Code of Flower and Bee after extracting the observing relationships to an aspect.

Hannemann and Kiczales [13] mention four modularity properties for their implementation of the Observer pattern: locality, reusability, composition transparency and (un)pluggability. Just after the extraction, the aspect (Figure 4) has only the first and last of these properties.

```
01 public aspect ObservingOpen {
02    static class ONotifier extends Observable {
03       //...
04    }
05    static class OObserver implements Observer {
06       //...
07    }
08    private ONotifier Flower.oNotify =
09       new ONotifier(this);
10    private OObserver Hummingbird.oObserver =
11       new OObserver(this);
12    private OObserver Bee.oObserver =
13       new OObserver(this);
14
15    public Observable Flower.opening() {
16       return oNotify;
17    }
18    public Observer Bee.openObs() {
19       return oObserver;
20    }
21    public Observer Hummingbird.openObs(){
22       return oObserver;
23    }
24
25    pointcut flowerOpen(Flower flower):
26       execution(void open()) && this(flower);
27    after(Flower flower) returning :
28          flowerOpen(flower) {
29       flower.oNotify.notifyObservers();
30    }
31    pointcut flowerClose(Flower flower):
32       execution(void close()) && this(flower);
33    after(Flower flower): flowerClose(flower) {
34       flower.oNotify.close();
35    }
36 }
```

Figure 4. Part of the extracted aspect ObservingOpen modularizing observations of Flower's open operation.

Inter-type declarations are one of the reasons why the structure of aspects resulting from extraction processes is often unsuitable. Inter-type declarations are usually transparent to client code (to our knowledge, only code using AspectJ's within pointcut designator can be affected by extraction refactorings based on inter-type declarations) and therefore make it simple to move members from classes to aspects. However, only the source code is modularized: the inter-type members still belong to their respective target classes at the binary and runtime levels. Their static nature can lead to the *Aspect Laziness* smell. At the very least, the extracted aspect will need a tidying up. In some cases, including this one, it will require a complete redesign.

The *Tidy Up Internal Aspect Structure* ([21], p.36) refactoring provides the general framework for improving the internal structure of extracted aspects. The refactorings it prescribes can transform the ObservingOpen aspect from Figure 4 to the one shown in Figure 5. The mechanics prescribe at the start the use of *Generalise Target Type with Marker Interface* ([21], p.25). Using the refactoring we replace references to concrete types (Flower, Bee and Hummingbird in the example) with marker interfaces representing the roles played by the participants (Subject and Observer in the example). This refactoring removes the duplication caused by multiple inter-type declarations of the same member. In simpler cases, it is enough to attain (un)pluggability.

When using *Generalise Target Type with Marker Interface* ([21], p.25) we may sometimes find that a single call to a case specific method prevents a code fragment to be reusable. For such cases, we propose *Extend Marker Interface with Signature* ([21], p.24), which separates the generically applicable code from case-specific code, by extending marker interface with the method's signature. This way we avoid the use of downcasts and eliminate dependencies to specific types (i.e. the module no longer needs to import those types). The abstract declarations of methods isOpen and breakfastTime (Figure 5) result from using this refactoring.

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   public abstract boolean Subject.isOpen();
   public abstract void Observer.breakfastTime();
   private boolean Subject.alreadyOpen = false;

   private WeakHashMap subject2ObserversMap =
      new WeakHashMap();
   private List getObservers(Subject subject) {
      List observers =
         (List)subject2ObserversMap.get(subject);
      if(observers == null) {
         observers = new ArrayList();
         subject2ObserversMap.put(subject, observers);
      }
      return observers;
   }
   public void addObserver(Subject subject,
                           Observer observer) {
      List observers = getObservers(subject);
      if(!observers.contains(observer))
         observers.add(observer);
      subject2ObserversMap.put(subject, observers);
   }
   public void removeObserver
   (Subject subject, Observer observer) {
      getObservers(subject).remove(observer);
   }
   public void clearObservers(Subject subject) {
      getObservers(subject).clear();
   }
   private void notifyObservers(Subject subject) {
      if(subject.isOpen()&& !subject.alreadyOpen) {
         subject.alreadyOpen = true;
         List observers = getObservers(subject);
         for(ListIterator it =
             observers.listIterator();
             it.hasNext();) {
            ((Observer)it.next()).breakfastTime();
         }
      }
   }
   pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning:
         flowerOpen(subject) {
      notifyObservers(subject);
   }
   pointcut flowerClose(Subject subject):
      execution(void close()) && this(subject);
   after(Subject subject): flowerClose(subject) {
      subject.alreadyOpen = false;
   }
   declare parents: Flower implements Subject;
   declare parents:
      (Bee || Hummingbird) implements Observer;
}
```

Figure 5. Aspect ObservingOpen after being tidied up.

The motivation to both *Replace Inter-type Field with Aspect Map* ([21], p.28) and *Replace Inter-type Method with Aspect Method* ([21], p.33) is twofold. One is to remove the *Aspect Laziness* smell. Another is to deal with hurdles arising with the movement

of duplicated inter-type declarations along aspect hierarchies (see section 5.4). These two refactorings prescribe how to replace inter-type state and behavior with a mapping structure providing the same functionality in a more dynamic way, and amenable to be controlled by client objects. In this example we used the same implementation, based on a weak hash map, as in the reusable aspect for the Observer pattern, presented in [13].

The motivation for *Introduce Aspect Protection* ([21], p.27) stems from the impossibility of using the protected access in inter-type members. This refactoring prescribes how to preserve this access through declare error clauses.

5.4 Dealing with Generalization

The refactorings from this group deal with the extraction of commonalities to superaspects, with *Extract Superaspect* ([21], p.37) providing the general framework. All the other refactorings in this group deal with moving members up and down the inheritance hierarchies of aspects. Refactorings for moving traditional OO members such as fields and methods are not included, as the issues and mechanics are similar to those documented in [11]. In [22] we show how the reusable aspect presented in [13] can be extracted from the one illustrated in Figure 5.

Pull Up Inter-type Declaration ([21], p.39) and *Push Down Inter-type Declaration* ([21], p.42) have a very restricted scope of applicability, only to simple cases not involving duplication. They are almost anti-refactorings: one motivation for including them in the collection is to better document the related problems and warn against attempts to treat inter-type declarations as if they were like the other kinds of members. The hurdles arise because duplicated inter-type declarations of fields can not generally be moved between superaspects and subaspects. Such movements change the number of instances of inter-type fields and their relation to aspect instances. It is important to keep in mind that (1) the visibility scopes of multiple inter-types declarations of the same member can not overlap and that (2) target objects (i.e. instances of classes affected by the inter-type declaration) have one separate instance of the inter-type member for *each* subaspect. If the various inter-type declarations are factored out to a single declaration in a superaspect, target objects will have just *one* instance of the introduced member. This situation is somewhat similar to changing a class member from instance to static. In most cases, dealing with duplicated inter-type declarations entails the prior replacement of the introduced fields with some mapping logic, establishing the links between target objects and the additional state and behavior. Such replacements happen to be exactly what is accomplished by *Replace Inter-type Field with Aspect Map* ([21], p.28) and *Replace Inter-type Method with Aspect Method* ([21], p.33).

The remaining refactorings from this group deal with pulling up and pushing down aspect-specific constructs, including pointcuts, advice and declare parents clauses. Inner interfaces are also included due to their widespread use as marker interfaces.

6. RELATED WORK

Deursen *et al.* [7] give a brief overview of the state of art in the area of aspect mining and refactoring. Though their main concern seems to be tools for the automatic detection of aspects, they also mention several open questions about refactoring to aspects, including "how can existing code smells be used to identify candidate refactorings?" and "how can the introduction of aspects be described in terms of a catalog of new refactorings?". In this paper, we contribute to answering these two questions.

Iwamoto and Zhao announced in [15] their intention to build a catalog of AOP refactorings. They present a catalog of 24 refactorings, but the information provided about them is limited to the names of the refactorings. The refactorings we present in this paper and document in [21] include a description of the situations where the refactoring applies, mention of preconditions, detailed mechanics and code examples.

Several authors [15][14][29][31] call into attention the *fragile base code problem* (though they do not use this name), in some cases illustrating it with some code examples. These authors conclude that existing OO refactorings [11] can not be applied to code bases with aspects. We believe these problems can be ameliorated if adequate procedures are followed [18], including adoption of an appropriate style for programming and evolving aspect constructs, particularly pointcuts. Hanenberg *et al.* [14] propose *aspect-aware* refactorings – refactorings that take into account the presence of aspects and preserve behavior by updating any pointcuts that may be affected by the transformation – and propose a set of enabling conditions to preserve the observable behavior. By the author's admission, these conditions must be automatically verified by an aspect-aware tool, as the manual verification is an exhausting task, even in small systems. Hanenberg *et al.* announce a tool providing a subset of the functionality they deem desirable.

Hanenberg *et al.* [14] also propose three AOP refactorings – *Extract Advice*, *Extract Introduction* and *Separate Pointcut*. Their *Extract Advice* corresponds to our *Extract Fragment into Advice* refactoring ([21], p.9). Our collection of refactorings [21] goes deeper in exploring the refactoring space, providing more detail and tackling issues such as the tidying up of the internal structure of aspects resulting from extraction processes. We do not subscribe the recommendation, in their *Extract Advice* refactoring, to use 'around' advice in the general case. We think that in cases where either 'before' or 'after' advice can be used, these should be used in preference to 'around', because it makes the scope of the advice easier to perceive at a first look at the code. In addition, the 'around' advice is also more powerful than is often needed. In the case of code using it without a strict need for it, we envision refactorings such as *Change Around Advice to Before* and *Change Around Advice to After Returning*. Their proposed *Extract Introduction* refactoring corresponds to our *Move Field from Class to Inter-type* ([21], p.17) and *Move Method from Class to Inter-type* ([21], p.19) refactorings, which provide more detail. *Separate Pointcut* relates to evolution of pointcuts and has no correspondence in our collection. This refactoring argues that, just as it is beneficial to organize our systems using small methods with meaningful names, we should do the same with pointcuts. Hanenberg *et al.* do not elaborate on code smells, but we can infer from *Separate Pointcut* that anonymous pointcuts could be a code smell.

In [18] Laddad prescribes several guidelines to ensure AOP refactorings for concern extraction are applied in a safe way. These involve the creation of a first version of the pointcut, based on a case-by-case enumeration of the interesting joinpoints, followed by its replacement with a semantically more meaningful

pointcut, based on wildcards. Laddad also proposes a mechanism based on AspectJ's declare error mechanism to verify whether two different pointcut expressions capture exactly the same set of joinpoints. In addition, Laddad recommends that aspects start being developed with a restricted scope, often affecting the methods of a single class, in order to make it simpler to test their impact on the base code. Only afterwards should the scope of the aspect widen, when its functionality is already tested with the restricted case. Considering that at present there is no adequate tool support for AOP refactorings, and that aspects can potentially impact a large number of joinpoints across an entire system, procedures such as these are essential to any refactoring process targeting non-trivial systems.

In addition, Laddad presents a collection of refactorings [18] with a significant utility value, particularly to developers of J2EE applications. The refactorings vary widely in both level and scope of applicability, including generally applicable refactorings like *Extract Interface Implementation, Extract Method Calls*, and *Replace Override with Advice*, but also concern-specific refactorings such as *Extract Concurrency Control* and *Extract Contract Enforcemement*. In addition, some refactorings belong to the category of "refactoring to patterns" as presented by Kerievsky [16] – *Extract Worker Object Creation* and *Replace Argument Trickle by Wormhole*. These two refactorings are based on two of the design patterns presented by Laddad in [19] – *Worker Object Creation* ([19], p.247) and *Wormhole* ([19], p.256) respectively. The *Extract Exception Handling* refactoring as presented in [18] goes towards a variant implementation of the *Exception Introduction* pattern ([19], p.260).

We believe programmers would benefit if Laddad's refactorings were presented in the same format as used by Fowler *et al.* [11] and Kerievsky [16], and which we use as well [21][23] (some refactorings are presented with only a mention of its name and a brief motivating paragraph). A mechanics section would be particularly beneficial, having proved very useful as a checklist and to lead developers through the safest sequences of steps, in preference to riskier or less convenient ones. The important step-by-step guidelines proposed by Laddad for creating a new aspect and subsequently evolving it are included in the code example illustrating the use of *Extract Method Calls*, but not in several other refactorings to which they also apply (Laddad places some reminders). A mechanics section would make that part process clearer, and would clarify the relations between refactorings.

We noticed that several of Laddad's refactorings, namely the problem-specific ones, can be decomposed into simpler, lower level steps – always an important thing with refactoring. During our work on the mechanics of the refactorings documented in [21], we focused on the minute details of the refactoring process, enabling us to improve their characterization. In some cases, this led us to decompose the refactoring under study into several smaller steps. For instance, *Split Abstract Class into Aspect and Interface* ([21], p.21) and *Change Abstract Class to Interface* ([21], p.4) were initially conceived as a single refactoring. We believe similar benefits can be obtained by similarly approaching Laddad's refactorings – some of the resulting lower level steps would correspond to existing steps, while others would possibly yield new refactorings.

Laddad does not pinpoint the code smells that his refactorings are supposed to remove. We think that the material presented by Laddad has the potential to throw new light on existing OO code smells or to yield new ones. For instance, his *Extract Method Calls* and *Replace Argument Trickle by Wormhole* refactorings respectively suggest the *Scattered Method Calls* and *Argument Trickle* smells. Further research is required to discover latent smells and assess their feasibility and applicability.

Tonella and Ceccato [28] base their work on the assumption that interfaces are often (not always) related to concerns other than the one pertaining to the system's main decomposition. This is an *Interface Implementation* smell, though the authors do not name it this way. They provide specific guidelines for when an interface implementation is a symptom of a latent aspect and present a tool for mining and extracting aspects based on these criteria, and report on experimental results. These extractions are also covered by the refactorings we present here and document in [21]. The authors also point out various issues that can arise in a typical extraction of an interface implementation into an aspect. Our refactorings prescribe procedures to deal with all these issues.

In [3] and [4] Cole and Borba propose programming laws from which refactorings for AspectJ can be derived. The authors focus on the use of their laws to derive existing refactorings such as those proposed in [18], [14] and [15], and describe two case studies in which the laws were tested, comprising the extraction of concurrency control and distribution respectively. Many, though not all, of the laws relate to the extraction of crosscutting concerns to aspects, and therefore there is some overlap between the refactorings they derive and our own extraction refactorings (section 5.2). However, their main emphasis is to provide proofs that the transformations are behavior preserving, while we focus on covering new ground in the refactoring space. Nevertheless, the authors remark that extraction procedure for the second case study is generalizable, because its implementation of distribution is commonly used, and claim that it is possible to derive a concern-specific *Extract Distribution* refactoring.

To our knowledge, no work besides ours deals with the potentially bad internal structure of aspects resulting from extraction processes. With the exception of the work by Tonella and Ceccato [28], we do not have knowledge of any other work covering the issue of AOP code smells.

7. FUTURE WORK

7.1 Other Code Smells

In addition to the traditional OO smells we mentioned in section 4.2, there are a few others we believe can be useful in detecting crosscutting concerns, but which we did not sufficiently explore to pinpoint suitable refactorings to remove them. We're considering the possibility that *Parallel Inheritance Hierarchies* ([11], p.83) and *Combinatorial Explosion* ([30], p.109) may be indicative of the presence of crosscutting concerns in some cases (the latter is a variant of the former, proposed by Wake [30]). We're presently considering whether these smells could be considered symptoms of the "tyranny of the dominant decomposition" [27] in some cases.

Besides existing OO smells, there are many latent AOP specific smells waiting to be discovered. For instance, privileged aspects: the rationale for avoiding them is the same as for avoiding the use of public data. As Colyer and Clement remark in [5], aspect privilege confers the general privilege to see any private state

anywhere, while one often wishes to express privilege with respect to a single class or a restricted set of classes. Presently, this is not possible with AspectJ. Unfortunately, privileged aspect may be unavoidable in cases affecting multiple packages and in which the aspect needs access to non-public (e.g. protected and package-protected) data. Refactoring the affected code bases to expose the non-public data is one alternative. We need to study use cases of privileged aspects to assess whether common patterns can be found, and pinpoint refactorings that tackle this issue.

7.2 Maturing the Refactorings

There is scope for maturing the refactorings presented here. It is important to test the refactorings with more case studies, particularly larger and more complex ones. In addition, we consider the possibility that the composite refactorings (section 5.1) will evolve to give origin to various refactorings in the conventional sense, or to be turned into an introductory text to a group of related refactorings.

7.3 Other Refactoring Ideas

We detected more latent refactorings in the material from our case studies. Next, we present some promising ideas for refactorings that were not yet fully explored:

- *Replace Throws with Declare Error* – many existing instances of Java code throw an exception upon detection of illegal situations. Some of these situations can be specified by statically determinable pointcuts, in which case it is more effective to replace them with a declare error clause.
- *Remove Signatures from Inner Interface* – As a rule, marker interfaces do not declare operations, so it is worth exploring a refactoring to remove the operations declared by an inlined interface.
- *Replace Downcast with Interface Extension* – We proposed *Extend Marker Interface with Signature* ([21], p.24) to resolve dependencies to concrete types caused by calls to type-specific methods. This idea can be taken further by completely removing dependencies on a type, namely type casts, to the point of removing the import of the type.

In addition, there are many possible variants to the refactorings documented in [21]. One example is to extract common code from multiple, similar aspects through an *Extract Subaspect* refactoring instead of an *Extract Superaspect* ([21], p.37).

7.4 Covering Other Language Characteristics

The refactorings we present here are the result from the two specific case studies, and do not use every possible aspect construct. New research should cover the remaining aspect constructs, as well as the interactions between them and with existing Java constructs. We next mention two subjects.

7.4.1 Non Singleton Aspect Association

Our work so far concentrated on singleton aspects. In future, we expect to cover other kinds of aspect association in order to obtain a clearer idea of the advantages and disadvantages of non-singleton aspects, e.g., when should they be preferred and what refactorings should be used to transform singleton aspects.

7.4.2 Pointcuts

At present, refactorings and code smells specifically targeting pointcuts are still a largely unexplored area. AspectJ's pointcut protocol comprises a rich language for quantification [10] and is likely to yield an equally rich pattern language for refactoring pointcut expressions, as well as their interaction with advice. Further research is needed on the adequate use of pointcut designators (e.g. pointcut smells), and how best to evolve pointcut expressions.

7.5 Restructuring the Remaining Base Code

In this paper, we cover the restructuring of aspect code resulting from the extraction of crosscutting concerns, taking advantage of the newfound modularization. It is also worth to study the impact of such extractions on the remaining code base and what actions would be desirable (e.g. post-extraction refactorings).

7.6 Dealing with Published Interfaces

Refactoring legacy code entails dealing with published interfaces, i.e. interfaces used by clients that developers could not change. Occasionally the tangling resulting from the presence of crosscutting concerns is present in the signatures, in which case it can not be readily removed. In such cases, developers have the option to refactor *towards* rather than *to* a goal, while a deprecation policy is pursued. We partially dealt with that issue in [23], having devised the *Partition Constructor Signature* refactoring ([21], p.44). We did not continue our work in that direction, but deem it worthy of further research.

7.7 Opposite Refactorings

We do not provide opposites for the presented refactorings, preferring to focus on extending the reach of the existing collection of refactorings. However, opposites are important to enable developers to backtrack, whenever they find out they took a wrong turn, and because opposites are often useful in their own right (e.g. pull up vs. push down refactorings).

8. SUMMARY

In this paper, we review existing OO code smells in the light of AOP. *Divergent Change* can be a sign of code tangling and both *Shotgun Surgery* and *Solution Sprawl* can be signs of code scattering. We propose AOP specific code smells, both for detecting crosscutting concerns in existing OO code and for improving the structure of extracted aspects – *Double Personality*, *Abstract Classes* and *Aspect Laziness*.

Simply moving the members relating to a crosscutting concern does not yield a well-formed aspect. Extracted aspects expose problems caused by crosscutting, including *Duplicated Code* ([11], p.76). *Aspect Laziness* relates to the static nature of inter-type declarations. We can take advantage of the new-found modularity to tidy up the aspect's internal structure with further refactorings. We present a collection of 27 AOP refactorings, documented in [21], which can remove these smells from source code, comprising the following groups:

- 10 refactorings to remove the smells related to crosscutting concerns from existing OO code. Besides covering common members such as fields and methods, these refactorings also deal with inner classes and interfaces.

- 6 refactorings to remove problems found in extracted aspects, including *Duplicated Code* and *Aspect Laziness*.
- 11 refactorings to deal with the generalization of aspects (i.e. the extraction of common code to superaspects).

We discuss some of the many future directions in the hunt for new AOP refactorings and code smells, taking the contributions of this paper and related work as the starting point.

Acknowledgements

Miguel Monteiro is supported by *PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior)* and by project *PPC-VM (POSI/CHS/47158/2002)*. João M. Fernandes is supported by project *METHODES* (POPI/CHS/37334/2001). Four anonymous referees made valuable suggestions which helped to improve this paper.

9. REFERENCES

[1] Beck, K., Extreme Programming Explained: Embrace Change, Addison-Wesley 2000.

[2] Bracha, G., Cook, W., *Mixin-Based Inheritance*, ECOOP/ OOPSLA 1990.

[3] Cole, L., Borba, P., *Deriving Refactorings for AspectJ*, presented in AOSD'2005, Chicago, USA, March 2005.

[4] Cole, L., Borba, P., *Using Programming Laws to Modularize Concurrency in a Replicated Database Application*, 1st Brasilian Workshop on Aspect-Oriented Software Development - WBSOA'04 - SBES'04, Brazil, October 2004.

[5] Colyer, A., Clement, A., *Large-scale AOSD for Middleware*, AOSD 2004, Lancaster, UK, March 2004.

[6] Cooper, J., *Java Design Patterns: A Tutorial*, Addison-Wesley 2000. Also availabe at www.patterndepot.com/put/8/ DesignJava.PDF.

[7] Deursen, A., Marin, M., Moonen, L., *Aspect Mining and Refactoring*, workshop on REFactoring: Achievements, Challenges, Effects (REFACE03), Waterloo, Canada, November 2003.

[8] Dijkstra, E., *Go-to Statement Considered Harmful*, Communications of the ACM, 11 (3), March 1968.

[9] Eckel, B., *Thinking in Patterns*, revision 0.9. book in progress, May 20, 2003. Available at http://64.78.49.204/ IPatterns-0.9.zip

[10] Filman, R. E., Friedman, D. P., *Aspect-Oriented Programming is Quantification and Obliviousness*, workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, October 2000.

[11] Fowler, M. (with contributions by K. Beck, W. Opdyke and D. Roberts), *Refactoring – Improving the Design of Existing Code*, Addison Wesley 2000.

[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[13] Hannemann, J., Kiczales, G., *Design Pattern Implementation in Java and AspectJ*, OOPSLA 2002, November 2002.

[14] Hanenberg, S., Oberschulte, C., Unland, R., *Refactoring of Aspect-Oriented Software*, Net.ObjectDays 2003, Erfurt, Germany, September 2003.

[15] Iwamoto, M., Zhao, J., *Refactoring Aspect-Oriented Programs*, 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, USA, October 2003.

[16] Kerievsky, J., *Refactoring to Patterns*, Addison-Wesley, 2004.

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., *Aspect-Oriented Programming*, ECOOP'97, Finland, June 1997.

[18] Laddad, R., *Aspect-Oriented Refactoring*, parts 1 and 2, The Server Side, 2003. www.theserverside.com/

[19] Laddad, R., AspectJ in Action – Practical Aspect-Oriented Programming, Manning 2003.

[20] Orleans, D., *Separating behavioral concerns with predicate dispatch, or, if statement considered harmful*, workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01, Tampa Bay, USA, October 2001.

[21] Monteiro, M. P., *Catalogue of Refactorings for AspectJ*, Technical Report UM-DI-GECSD-200402, Universidade do Minho, December 2004. Available at www.di.uminho.pt/~jmf/ PUBLI/papers/2004-TR-02.pdf

[22] Monteiro, M. P., *Refactoring a Java Code Base to AspectJ – An Ilustrative Example*, Technical Report UM-DI-GECSD-200403, Universidade do Minho, December 2004. Available at www.di.uminho.pt/~jmf/PUBLI/papers/2004-TR-03.pdf

[23] Monteiro, M. P., Fernandes, J. M., *Object-to-Aspect Refactorings for Feature Extraction*, industry paper presented at AOSD'2004, UK, Lancaster, March 2004. Available at http://aosd.net/ 2004/archive/Monteiro.pdf

[24] Opdyke, W., *Refactoring Object-Oriented Frameworks*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[25] Sabbah, D., *Aspects – from Promise to Reality*, AOSD 2004, Lancaster, UK, March 2004.

[26] Störzer, M., Koppen, C., *PCDiff: Attacking the Fragile Pointcut Problem*, Interactive Workshop on Aspects in Software (EIWAS) 2004, Berlin, Germany, September 2004.

[27] Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, ICSE'99, May, 1999.

[28] Tonella, P., Ceccato, M., *Migrating Interface Implementation to Aspects*, ICSM'04, Chicago, USA, September 2004.

[29] Tourwé, T., Brichau, J., Gybels, K., *On the Existence of the AOSD-Evolution Paradox*, AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA, 2003.

[30] Wake, W., *Refactoring Workbook*, Addison Wesley, 2004.

[31] Wloka, J., *Refactoring in the Presence of Aspects*, ECOOP2003 PhD workshop, July 2003.