Towards a Catalogue of Refactorings and Code Smells for AspectJ

Miguel P. Monteiro[1] and João M. Fernandes[2]

[1] Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Avenida do Empresário, 6000-767, Castelo Branco, Portugal
mmonteiro@di.uminho.pt
[2] Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal
jmf@di.uminho.pt

Abstract. In this paper, we contribute to the characterisation of a programming style specific to aspect-oriented programming. For this purpose, we present a collection of refactorings for aspect-oriented source code, comprising refactorings to enable extraction to aspects of crosscutting concerns from object-oriented legacy code, the subsequent tidying up of the extracted aspects and factoring out of common code from similar aspects to superaspects. The second group of refactorings is documented in detail. In addition, we propose some new aspect-oriented code smells, including one smell that is specific to aspect modules. We also propose a reinterpretation of some of the traditional object-oriented code smells in the light of aspect-orientation, to detect the presence of crosscutting concerns.

1 Introduction

Refactoring [10, 13, 31] and aspect-oriented programming (AOP) [23] are two techniques that contribute to dealing with the problems of continuous evolution of software. Refactoring processes enable the improvement of the internal structure of source code without changing a system's external behaviour, thus facilitating its evolution in line with changes in environments and requirements. AOP enables the modularisation of *crosscutting concerns* (CCCs), thus diminishing the potential impact of changes to the code related to a given concern on code not related to that concern.

AOP's steady progress from a "bleeding edge" research field to mainstream technology [33] 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 latest half-decade suggests that refactoring techniques have the potential to bring the concepts and mechanisms of aspect-orientation to existing OO frameworks and applications.

1.1 Some Challenges of Refactoring Aspect-Oriented Systems

We believe there are three main hurdles that should be addressed so that refactoring techniques can be effectively used in AOP software. The first hurdle is the present

A. Rashid and M. Aksit (Eds.): Transactions on AOSD I, LNCS 3880, pp. 214 – 258, 2006.

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. Notions of good style enable programmers to see where they are heading when refactoring their code. For instance, Fowler et al. [10] advocated a specific notion of style for OO code through a catalogue of 22 *code smells*, compounded by a catalogue of 72 refactorings through which those smells can be removed from existing code. These catalogues 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 notions 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 catalogues. Our work is based on the assumption that AOP would equally benefit from AOP-specific catalogues of smells and refactorings, helping programmers to detect situations in the source code that could be improved with aspects, as well as guiding them through the transformation processes.

A third hurdle is the absence of tool support for AOP constructs and mechanisms in integrated development environments (IDEs). The catalogues presented by Fowler et al. [10] provided a basis on which developers could rely to build tool support for OO refactoring; similar catalogues 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 which specific refactorings are worthy of their development efforts.

1.2 On the Need for an AOP-Specific Notion of Style

The notion of style in a programming language expresses the coding practices that yield code that is 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 should be considered the one 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 mention three examples:

1. Dijkstra's famous dictum that the "Go-to statement [should be] considered harmful" [7] stemmed from the availability of control structures, namely loops.
2. Fowler et al. [10] considered the use of the "switch" statement a code smell, due to the availability of polymorphism and dynamic binding.
3. Orleans suggested in [32] 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 [22, 26] cannot be the same as for Java. AspectJ enables programmers to perform compositions that are impossible with Java and to 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. This includes OO implementations of many design patterns [16].

The compositional power itself of AspectJ can be cause for problems. AspectJ offers multiple ways to achieve various effects and compositions. For instance, implementation of mixins [2] can be achieved both through marker interfaces and through inner static aspects placed within interfaces. Likewise, nonsingleton aspect associations provide alternatives to solutions obtained with 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 catalogues of code smells and refactorings [10] are an effective way to present this knowledge to programmers.

1.3 Contributions

In this paper, we expand the existing refactoring space for AOP and thus contribute to the characterisation of an AOP style. We present a collection of refactorings for AOP source code. The refactorings were developed to be performed manually, and for this reason we describe them with a style similar to that of [10]. We complement the refactorings with descriptions of AOP code smells [10], which the refactorings are supposed to remove. In addition, we review the traditional OO code smells in the light of AOP and propose a reinterpretation of a few traditional OO smells as indicators of the presence of CCCs.

The subject language we use is AspectJ [22, 26] whose backward compatibility with Java opens the way for refactoring existing Java applications by introducing AOP constructs. The task of assessing the extent to which our results can be applied to different aspect-oriented languages is left to future work.

This paper is a revised and extended version of a paper presented at AOSD 2005 [30]. The main additional contribution relative to the other paper is the detailed documentation of a group of refactorings. This paper also provides more information on how refactorings and code smells were derived and updated and revised sections on related and future work.

1.4 Issues Not Addressed

Our focus in this paper is on creating a catalogue of refactorings that can enable the development of tool support rather than on the implementation of the support. We analyze the effect of our refactorings qualitatively because our focus is on understanding the breadth of refactorings needed to transform OO code into well-styled AO code, rather than on a formal description of each refactoring, which is left as future work [3]. To further clarify the context of this paper, we next mention several related subjects that we do not cover.

No tool support. Developing tools that automate standard transformations of source code is related to the subject covered in this paper, but it is not the same. Even when provided with appropriate refactoring tools, developers still need to have a proper notion of style to decide when code should be refactored and to be able to choose the

specific refactoring appropriate for each situation. It is this knowledge that we aim to expand. Nevertheless, we believe this paper can be helpful to developers of tool support for aspect-oriented refactorings by suggesting refactorings that may be worthy of their development efforts. Therefore, this paper indirectly contributes to the development of future tools.

No metrics. We do not attempt to formally measure and quantify the benefits in code of the proposed refactorings. Work on metrics for the complexity of aspect-oriented source code can be found in [12, 38, 39].

No formalism. We do not attempt to provide a formal, mathematical basis for the refactorings. Cole and Borba worked in this field, and in [3] they state their interest in extending their work to cover our refactorings.

1.5 Outline

The rest of this paper is structured as follows. In Sect. 2, we describe the approach we took to develop the collection of refactorings. In Sect. 3, we present an overview of the refactorings, which are documented in Sect. 4. In Sect. 5, we review some of the traditional smells in the light of AOP and propose three novel such smells. In Sect. 6, we present a code example illustrating the presence of some smells and results of applying the refactorings that remove those smells. In Sect. 7, we survey related work, and in Sect. 8 we consider future directions. In Sect. 9, we summarise the paper.

2 The Approach

We took the approach of performing refactoring experiments on code bases, as a vehicle for gaining the necessary insights. The selected case studies were code bases in Java and/or AspectJ with the appropriate structural characteristics. We approached Java code as bad-style or "smelly" AspectJ code, and looked for the kinds of refactorings that would be effective in removing the smells. The selected case studies were systems likely to include CCCs or code bases that promised to yield interesting insights.

The first experiment comprised the extraction of a CCC from a workflow framework to an aspect, yielding refactorings *extract feature into aspect*, *extract fragment into advice*, *move field from class to intertype*, *move method from class to intertype* (Table 1), as well as experience that was invaluable for the subsequent case study. Despite yielding some positive results, we do not consider the extraction we undertook to be a good example of the use of AspectJ. The extraction we undertook was really an attempt to decompose the system according to *use cases* [19] or *features* [20] (for the purposes of this paper we regard the two concepts as equivalent). The extracted aspect is a monolithic module that uses the mechanisms of AspectJ to compose its internal elements to the appropriate points of the primary system. Though the extracted aspect as a whole is crosscutting, each intertype declaration has a single target type and each pointcut captures a single joinpoint. We concluded that the feature we extracted does not comprise a good example of the sort of CCC that AspectJ can

advantageously modularise. AspectJ is appropriate for cases with many duplicated fragments that can be replaced by one or a few pointcuts plus advice acting on the captured joinpoints, thus yielding significant savings in lines of code. Since the extracted CCC is an instance of Interpreter, we compared its code with the AspectJ implementation of Interpreter proposed by Hannemann and Kiczales [16]. Unlike several of the examples from the collection from [16], the implementation of Interpreter comprises a single *concrete* aspect (i.e., it does not extend a reusable abstract aspect). Hannemann and Kiczales placed a few comments at the beginning of the aspect source file, remarking in the end that Interpreter "does not lend itself nicely to aspectification". The aspect we extracted is simply a more complex instance with similar problems. For more information regarding the relevant characteristics of the framework, the extraction experiment and the results derived from it, the reader is referred to [28] and [27].

The 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 [11], presented by Hannemann and Kiczales [16]. 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 analyse large code bases or learn domain-specific concepts. The implementations presented by Hannemann and Kiczales [16] can be counted among the currently available examples of good AOP style and design, presenting a clear picture of the desirable internal structure of 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 other authors [5, 8] which further enriched the patterns' potential as providers of insights.

Our approach to the GoF implementations was to pinpoint the refactorings that would be needed to transform the Java implementations into the AspectJ implementations. This comprised an iterative process, in which each Java code example was subject to multiple refactoring sessions aiming to yield the corresponding AspectJ version. The experience gained from each session was used to refine and enrich the descriptions of the code transformations being used. The descriptions of the refactorings presented in this paper emerged gradually through this process. Care was taken to only develop descriptions of generally applicable transformations, i.e., refactorings that can be applied to multiple, unrelated cases. During this process, various refactoring candidates were discarded because they turned out to be too case-specific.

In the subsequent phase, we tested and refined the refactorings thus obtained on the implementations of other, structurally similar patterns, or in different Java implementations of the same patterns [5, 8]. The code examples presented at the end of each description found in this paper originate from those test sessions, as well as the refactoring process described in detail in [29].[1] The latter also serves as a first validation effort.

[1] [29] is complemented with an eclipse project containing 33 complete code snapshots, available at www.di.uminho.pt/~jmf/PUBLI/papers/ObserverExample.zip.

Throughout our work on the mechanics of the refactorings, we took care to choose the safest path. As the refactorings are intended to be performed manually, it is important that each refactoring step be small, in order to ensure an easy backtracking and to maximise safety. In a few cases, this led us to decompose the refactoring under study into several smaller steps.

After the experiments were carried out and the refactoring descriptions were stable, we analysed the results in order to characterise the smells that the refactorings were supposed to remove. The novel smells presented in Sects. 5.2 – 5.4 are distillations of these ideas. In addition, we analysed existing, traditional OO smells [10, 21, 37,] to assess whether some of these smells could also be used as indicative of the presence of CCCs (see Sect. 5.1).

The refactorings described in this paper are to some extent specific to the characteristics of the languages used – Java and AspectJ. Our approach has the limitation that insights obtained to derive refactorings and code smells directly depend on the characteristics of the code bases used as case studies, and are only as good as the insights obtained from them. If a given characteristic or mechanism is not used in the subject code base, the experiments are not likely to yield insights related to that characteristic or mechanism. For instance, none of the code bases we used includes elaborate uses of exceptions. For this reason, our work did not yield any refactorings related to exceptions or exception handling. Further work on more case studies is needed to overcome these limitations. We elaborate on this subject in Sect. 8.

All refactorings presented in this paper were applied in at least one code example, with the exception of most of the simple *push down* refactorings from Table 3, which were derived for completeness. *push down advice* is used in the refactoring process described in [29].

3 Overview of the Refactorings

This section presents an overview of the refactorings. All descriptions use a format and level of detail similar to the one used by Fowler et al. [10] (Kerievsky took the same approach in [21]). 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. Section 4 presents complete descriptions of the refactorings from Table 2. Complete descriptions of refactorings from Tables 1–3 can also be found in [27].

The mechanics do not attempt to cover all possible situations that can potentially arise in source code. For instance, they do not account for uses of reflection. Likewise, they do not deal with the *fragile pointcut problem* [24], which is caused by the fact that almost all refactorings can potentially break existing aspects, particularly pointcuts (in [28, 29] we call it the *fragile base code problem*). We believe human programmers will be able to thoroughly deal with this problem only when provided with a new generation of tools, specifically designed to account for the presence of aspects.

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 subclasses from inheriting from another class	Turn the abstract class into an interface and change its relationship with subclasses from inheritance to implementation
Extract feature into aspect	Code related to a feature is scattered across multiple methods and classes, tangled with unrelated code	Extract to an aspect all implementation elements related to the feature
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 appropriate joinpoint and context and move the code fragment to an advice based on the pointcut
Extract inner class to stand-alone	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 stand-alone class
Inline class within aspect	A small stand-alone class is used only 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 within the aspect
Move field from class to intertype	A field relates to a concern other than the primary concern of its owner class	Move the field from the class to the aspect as an intertype declaration
Move method from class to intertype	A method belongs to a concern other than the primary concern of its owner class	Move the method into the aspect that encapsulates the secondary concern as an intertype declaration
Replace implements with declare parents	Classes implement an interface related to a secondary concern. Class code implementing the interface is used only when the secondary concern is included in the system build	Replace the implements in the class with a equivalent declare parents in the aspect
Split abstract class into aspect and interface	Classes are prevented from using inheritance because they inherit from an abstract class defining several 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 models a role used within the aspect. You would like the aspect to call a method specific to a type that implements the interface but that is not declared by it	Add an intertype abstract declaration of the case-specific method signature to the interface
Generalise target type with marker interface	An aspect refers to case-specific concrete types, preventing it from being reusable	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 an intertype member to be visible within the declaring aspect and all its subaspects, but not outside the aspect inheritance chain	Declare the intertype member as public and place a declare error preventing its use outside the aspect inheritance chain
Replace intertype 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 intertype declarations with a structure owned by the aspect that performs a map between the target objects and the additional state
Replace intertype 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 intertype method with an aspect method that gets the target object as an extra parameter
Tidy up internal aspect structure	The internal structure of an aspect resulting from the extraction of a crosscutting concern is suboptimal	Tidy up the internal structure of the aspect by removing duplicated intertype declarations and dependencies on case-specific target types

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 [25]. 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 to not 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

Table 3. Refactorings to deal with generalisation

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 intertype declaration	An intertype declaration would be best placed in the superaspect	Move the intertype 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 different advice code	Move the advice to the subaspects that use it
Push down declare parents	A declare parents in a superaspect is not relevant for all subaspects	Move the declare parents to the subaspects where it is relevant
Push down intertype declaration	An intertype declaration would be best placed in a subaspect	Move the intertype 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	Move the pointcut to the subaspects that use the pointcut

possible, to increase the likelihood that all team members be 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).

All refactorings from Tables 1–3 assume AspectJ as the subject language. However, the refactorings from Table 1 are a special case in that the starting points of all refactorings from that group are in plain Java. This is not a specifically intended

restriction, it just follows that all refactorings deal with extractions of the various elements of a CCC. CCCs are expected to reside in plain Java bases but not in AspectJ code, and therefore the existence of aspects in the code base is not taken into account in the mechanics. Actually, the code bases targeted by the refactorings from Table 1 can include aspects (namely a Java base that is undergoing the extraction of multiple aspects). However, we assume that already existing aspects do not interfere with the constructs manipulated during the extraction process.

The traditional OO refactorings can be used in AspectJ code as well. We did not detect any refactoring from [10] targeting an OO construct that could not be applied to the same construct within aspects. For instance, in the mechanics of *Extend Marker Interface with Signature* we prescribe the use of *Extract Method* ([10], p. 110) inside aspects.

3.1 Grouping the Refactorings

The collection is structured in groups of refactorings with similar purposes, as is done in [10]. 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 CCC should be moved to a single module (following *extract feature into aspect*[2]). Only afterwards should we start improving the underlying structure of the resulting aspects (following *tidy up internal aspect structure*), because such tasks are considerably easier to perform after the associated implementation is modularised. In case duplication is detected among different but related aspects, we extract the commonalities to a (possibly reusable) superaspect (using *extract superaspect*). This strategy leads to the following grouping: (1) extraction of CCCs, (2) improvement of the internal structure of an aspect and (3) generalisation of aspects. The sequence of code transformations described in [29] also fits naturally with this grouping.

The three refactorings mentioned above are composite refactorings that provide the entry points to someone approaching the catalogue. Rather than prescribe specific actions on the source code, as is the case of those documented in [10], 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. They are also useful in providing a broader view of a refactoring process.

3.2 Refactorings for Extracting Features to Aspects

We expect the refactorings from this group (Table 1) to comprise the starting point for the majority of refactoring processes targeting OO legacy code. *Extract feature into aspect* pinpoints procedures for extracting scattered elements of a CCC into a single module [28]. We suggest using *move field from class to intertype* to move state to the aspect. Behaviour can be moved using *move method from class to intertype* and *extract fragment into advice*. Moving an inner class to an aspect is done in two stages: first using *extract inner class to stand-alone*, to obtain a stand-alone class from the inner class, and next using *inline class within aspect* to turn the resulting class into an

[2] In the context of these refactorings, we use the term "feature" to mean a CCC of the kind that can be effectively modularised by an AOP language such as AspectJ.

inner class within the aspect. We did not see a justification for defining a refactoring equivalent to *extract inner class to stand-alone* for interfaces, as interfaces are not generally used within classes. Interfaces are inlined into aspects using *inline interface within aspect*, after which they can be turned into marker interfaces. To complete the modularisation of the code related to the interface, we propose *replace implements with declare parents* for inlining the "implements" clause of implementing classes.

Split abstract class into aspect and interface enables the extraction of definitions from an abstract class to an aspect, opening the way to using *change abstract class to interface* to turn the abstract class into an interface. This way, subclasses of the abstract class become free to inherit from some other class. Together, the pair effectively extracts a mixin [2] from the original abstract class. The pair was derived from the analysis on the group of the GoF patterns that Hannemann and Kiczales related to multiple inheritance (Sect. 4.2.4 of [16]) and can be used to transform the Java implementations of those patterns into the corresponding AspectJ implementations.

3.3 Restructuring the Internals of Aspects

The refactorings from this group (Table 2) deal with the task of improving the internal structure of an aspect after all elements from a CCC were moved into it, using the refactorings presented in Sect. 3.2 (Table 1). *Tidy up internal aspect structure* provides the general framework for improving the internal structure of extracted aspects. The mechanics prescribe at the start the use of *generalise target type with marker interface*, which entails replacing references to case-specific types with marker interfaces representing the roles played by the participants. *Generalise target type with marker interface* removes the duplication caused by multiple intertype declarations of the same member. In straightforward cases, it is enough to attain (un)pluggability.

When using *generalise target type with marker interface* we may sometimes find that a single call to a case-specific method prevents a code fragment from being reusable. For such cases, *extend marker interface with signature* separates the generically applicable code from case-specific code, by extending marker interface with the method's signature.

Replace intertype field with aspect map and *replace intertype method with aspect method* prescribe how to replace intertype state and behaviour with a mapping structure providing the same functionality in a more dynamic way, and amenable to being controlled by client objects. These two refactorings can also deal with hurdles that arise when we try to move duplicated intertype declarations along aspect hierarchies (Sect. 3.4).

The motivation for *introduce aspect protection* stems from the impossibility of using the protected access in intertype members. This refactoring prescribes how to preserve this access through declare error clauses.

Split abstract class into aspect and interface and *change abstract class to interface* deal with the extraction of inner classes to aspects. The former removes dependencies of the inner class on the enclosing class and turns into a stand-alone class. The latter inlines the class within the aspect.

3.4 Dealing with Generalisation

The refactorings from this group (Table 3) deal with the extraction of common code to superaspects, with *extract superaspect* providing the general framework. All the remaining refactorings in this group deal with moving members up and down the inheritance hierarchies of aspects. New refactorings for moving traditional OO members such as fields and methods were not created, as the issues and mechanics are similar to those documented in [10]. In [29] we show how the reusable aspect presented in [16] can be extracted from similar aspects.

Pull up intertype declaration and *push down intertype declaration* have a very restricted scope of applicability, only to simple cases not involving duplication. They are almost antirefactorings – one motivation for including them in the collection is to better document some issues and warn against attempts to treat intertype declarations as if they were like other members. The hurdles arise because duplicated intertype declarations of fields cannot generally be moved between superaspects and subaspects: such movements change the number of instances of intertype fields and their relation to aspect instances. It is important to keep in mind that (1) the visibility scopes of multiple intertype declarations of the same member cannot overlap and that (2) target objects (i.e., instances of classes affected by the intertype declaration) have one separate instance of the intertype member for *each* subaspect. If duplicated intertype declarations are factored out to a single declaration in a superaspect, target objects will have just *one* instance of the introduced member. In most cases, dealing with duplicated intertype declarations entails the prior replacement of the introduced fields with some mapping logic that maintains the association between target objects and the additional state and behaviour (using *replace intertype field with aspect map* and *replace intertype method with aspect method*).

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 covered due to their widespread use as marker interfaces.

3.5 Refactorings for Plain Java

Two pairs of refactorings presented in the previous sections were initially conceived as single refactorings but were later split into the present pairs because this way seemed to have a more appropriate granularity:

- *extract inner class to stand-alone* and *inline interface within aspect* (Sect. 3.2)
- *split abstract class into aspect and interface* and *change abstract class to interface* (Sect. 3.3)

In both cases, one of the resulting refactorings deals only with plain Java constructs: *extract inner class to stand-alone* and *change abstract class to interface*, though this was not specifically intended. We believe the motivation for these particular plain Java refactorings arises only or mostly in the context of aspects. For these reasons they are included in their respective groups.

4 Refactorings for Tidying Up Extracted Aspects

This section documents the refactorings from Table 2. Complete descriptions of all refactorings from Tables 1–3 can be found in [27].

4.1 Extend Marker Interface with Signature

Typical situation. An inner interface models a role used within the aspect. You would like the aspect to call a method specific to a type that implements the interface but that is not declared by it.

Recommended action. Add an intertype abstract declaration of the case-specific method signature to the interface.

Motivation. Sometimes you would like to temporarily resolve a dependence on a case-specific part because that would enable you to do some tidying up of the aspect's internals, after which you would be in a better position to deal with the dependence. *Extend marker interface with signature* can be used as a stopgap in such situations to temporarily resolve dependences to a type-specific method. One case in which this situation arises often is during the use of *generalise target type with marker interface*.

An alternative solution to these problems would be to resort to downcasts. However, downcasts create dependencies to the target type of the cast: the specific type will need to be included in the aspect's "import" section, the type's binary file will have to be available when performing a build, etc. *Extend marker interface with signature* can be preferable in some situations because it avoids such dependencies. The dependence it creates is restricted to a method signature only, not to specific types. For these reasons, this refactoring is worth using in simple cases.

Preconditions. The signature must be public in order to be acceptable to the compiler. In addition, this solution is feasible only if all the types made to implement the marker interface export the signature.

Mechanics.

- If the method is not public, change it to public.
- Create in the aspect an intertype abstract declaration of the method's signature targeting the marker interface that will be used in place of the specific type.
- Compile and test.

Example. The ExampleAspect aspect uses the Role marker interface. Some instructions using Role resort to a downcast to specific type SpecificType, to resolve the call to the doSomething method, which is specific to this type. By using *extend marker interface with signature,* we eliminate this dependence to SpecificType. Provided this is the only use of SpecificType within ExampleAspect, the import clause itself can be removed, as shown below.

```
import ...SpecificType;

public aspect ExampleAspect {
    private interface Role { }
    ... action(Role obj) {
       //...
       ((SpecificType)obj).doSomething()
```



```
~~import ...SpecificType;~~

public aspect ExampleAspect {
    private interface Role { }
    public abstract void Role.doSomething();
    //...
    obj.doSomething()
```

4.2 Generalise Target Type with Marker Interface

Typical situation. An aspect refers to case-specific concrete types, preventing it from being reusable.

Recommended action. Replace the references to specific types with a marker interface and make the specific types implement the marker interface.

Motivation. This refactoring contributes to reduce the coupling between an aspect and its target code bases. It can also be used to expose and eliminate much duplication that could not be eliminated if the code kept referring to specific types. It can also be useful when we want to apply *extract superaspect* to aspects providing similar functionality, because it contributes to rationalise its internal structures.

Several situations can prevent *extract superaspect* from being applied to a set of similar aspects. The aspects can contain code specific to concrete classes in the midst of generally applicable code. If a general marker interface could be used instead of the specific types, use *generalise target type with marker interface*. The resulting marker interfaces may be candidates for pulling up to a superaspect.

Mechanics.

- Create a marker interface representing the role played by the target classes. Create the "declare parents" to associate the concrete classes to the role.
- Replace the references to the class with references to the marker interface. In cases when the aspect introduces the same field or method to more than one class, remove the duplication by replacing the various introductions with a single introduction to the interface.
- Sometimes the replacement cannot be made in method bodies because parts of the code depend on elements specific to a concrete class. In such cases, consider using *extract method* ([10], p. 110) to separate the parts covered by the role interface from the parts specific to particular classes. This may be an indication that in the future the aspect should be split into a generally applicable abstract superaspect and one or several specific concrete subaspects, using *extract superaspect*.

- Compile and test.
- When all method introductions refer to the interface, it is possible to remove the declarations of operations (methods) within the interface (if the interface is a inner interface, nested within the aspect, the related operations are defined within the aspect anyway, so removing the declarations from the interface will result in simpler code). If, however, the interface is kept stand-alone, leave the declarations in place. This way the code will be easier to understand.

Example: Simple Replacements. In the following example, GUIColleague is an interface representing a role. The aspect Mediator assigns the GUIColleague role to the Button class, but some parts of the code still specifically refer to Button instead of GUIColleague. We want to make all code to depend only on the interface (see below).

```
public aspect Mediator {
    declare parents: Button implements GUIColleague;
    declare parents: Label implements GUIMediator;
    GUIMediator Button._mediator;
    public void Button.setMediator(GUIMediator mediator) {
       this._mediator = mediator;
    }
    pointcut buttonClicked(Button button):
       execution(public void clicked()) && this(button);
    after(Button button): buttonClicked(button) {
       button._mediator.colleagueChanged(button);
    }
    //...
}
```



```
public aspect Mediator {
    declare parents: Button implements GUIColleague;
    declare parents: Label implements GUIMediator;
    GUIMediator GUIColleague._mediator;
    public void GUIColleague.setMediator(GUIMediator mediator) {
       this._mediator = mediator;
    }
    pointcut buttonClicked(GUIColleague button):
       execution(public void clicked()) && this(button);
    after(GUIColleague button): buttonClicked(button) {
       button._mediator.colleagueChanged(button);
    }
    //...
}
```

Naturally, the names of some variables (such as button) should now be renamed to reflect their more general context.

Example: Eliminating Duplication. This example is based on the Observer pattern ([11], p. 293). The ObservingOpen aspect encapsulates an observing relationship that was extracted from the participant classes. ObservingOpen introduces some fields and methods into several classes playing the Observer role (in this case Bee and Hummingbird). The classes are the only differing things among the introductions. By applying *generalise target type with marker interface*, we create the Subject marker interface and remove the duplication.

```
public aspect ObservingOpen ... {
    //...
    private OpenObserver Hummingbird.openObsrv =
       new OpenObserver(this);
    private OpenObserver Bee.openObsrv = new OpenObserver(this);

    public java.util.Observer Bee.openObserver() {
       return openObsrv;
    }
    public java.util.Observer Hummingbird.openObserver() {
       return openObsrv;
    }
}
```



```
public aspect ObservingOpen ... {
    //...
    private interface Subject { }
    declare parents: (Bee || Hummingbird) implements Subject;
    private OpenObserver Subject.openObsrv = new OpenOb-
server(this);

    public java.util.Observer Subject.openObserver() {
       return openObsrv;
    }
}
```

4.3 Introduce Aspect Protection

Typical situation. You would like an intertype member to be visible in an aspect and all its subaspects, but not outside the aspect inheritance chain.

Recommended action. Declare the intertype member as public and place a "declare error" preventing its use outside the aspect inheritance chain.

Motivation. AspectJ does not allow the protected access on intertype members, so whenever we would like to extend its access to subaspects we must classify the member as public. In some cases, it is desirable to have some form of access protection preventing the use of the member outside aspect code. The "declare error" mechanism enables us to emulate that protection.

Mechanics.

- Add a "declare warning" in the aspect enclosing the intertype member, specifying the intended restriction on its use.
- Compile and test.
- For each warning generated by the compiler, perform the refactorings necessary to move the use of the member to the authorised modules of the system.
- When there are no more warnings, change the "declare warning" to "declare error".

Example: Protecting an Intertype Field. Consider an abstract superaspect GeneralPolicy declaring intertype the field _sensitiveData. We want to restrict use of the field to aspect and its subaspects.

```
abstract aspect GeneralPolicy {
    protected interface Participant {}
    public Data Participant._sensitiveData;
    //...
}
```

```
aspect ConcretePolicy extends GeneralPolicy {
    //code using Participant._sensitiveData
}
```

We can add in the superaspect the following “declare warning”:

```
abstract aspect GeneralPolicy {
    protected interface Participant {}
    public Data Participant._sensitiveData;
    declare warning:
       (set(public Data Participant+._sensitiveData) ||
       get(public Data Participant+._sensitiveData))
       && !within(GeneralPolicy+):
       "field _sensitiveData is aspect protected. Not visible
here.";
    //...
}
```

Next, we deal with all points in the system, giving rise to warnings. After all warnings are gone, we change the “declare warning” to “declare error”.

Example: Protecting an Intertype Method. Suppose the same abstract aspect as in the previous example also includes method processSensitiveData, which we also would like to protect:

```
abstract aspect GeneralPolicy {
    protected interface Participant {}
    public Data Participant._sensitiveData;
    public void processSensitiveData() {
       //code using Participant._sensitiveData
    }
    //...
}
```

We create the following “declare warning”:

```
abstract aspect GeneralPolicy {
    protected interface Participant {}
    public Data Participant._sensitiveData;
    public void processSensitiveData() {
       //code using caspule._sensitiveData
    }
    declare warning:
       call(void processSensitiveData())
       && !within(GeneralPolicy+):
       "method processSensitiveData is aspect protected. Not visible
here.";
    //...
}
```

Likewise, the "declare warning" should be changed to "declare error" when all the warnings are gone.

Example: Protecting Intertype Method from Access Outside Inheritance Class and Aspect Inheritance Chains. What if we want to allow the access to a member in the host class, in addition to the aspect and their descendents? In the above example all that is needed is one more within to the above "declare error":

```
declare error:
    call(void processSensitiveData())
    && !within(Participant+)
    && !within(GeneralPolicy+):
    "Call to processSensitiveData() outside Participant and General
Policy chains.";
```

4.4 Replace Intertype Field with Aspect Map

Typical situation. 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.

Recommended action. Replace the intertype declarations with a structure owned by the aspect that performs a map between the target objects and the additional state.

Motivation. An intertype declaration is a static mechanism. It affects all instances of the target class, throughout their entire life cycles. For some problems, this is exactly right, but for others something more flexible would be preferable. In some cases only a subset of all instances of a class needs the extra state and behaviour, or they need it only in a specific phase of their life cycles. Sometimes the same instance simultaneously needs multiple instances of the extra state and behaviour. Sometimes the application only knows at run time which instances need the extra state and behaviour. Intertype declarations do not provide the necessary flexibility in these cases.

An intertype declaration is itself a kind of mapping, usually from a class to a field or method. However, we cannot control the moments when it applies, when it ceases to apply, and the precise set of objects to which it applies. Whenever this kind of flexibility is required and the existing solution relies on introductions, use *replace intertype field with aspect map* to replace the introductions with a suitable mapping.

This refactoring is also useful in a different situation. Sometimes we have several aspects performing similar actions on similar data, and these include intertype declarations. Such duplication should be removed by pulling the common parts to a superaspect. Here arises another problem. Target objects have separate instances of the additional state for each subaspect, but if the code is pulled up to the superaspect, there will be a single instance of the introduced state common to all subaspects. A similar problem would arise if we tried to replace an instance field with a static field. Such pulls will almost certainly not be behaviour-preserving. In most cases, an intertype declaration cannot be pulled up to a superaspect as is. The pulls usually require the prior replacement of intertype state with aspect state.

As it happens, the kind of replacements that solve the first problem can solve the second problem as well. Unlike with intertype declarations, there is a separate instance of the state declared in the superaspect in each active subaspect. In most cases, solving

the problem merely entails selecting a suitable structure to replace the intertype fields, and update the associated logic accordingly.

To ease the replacement of the original intertype state with the new mapping structure, you should first isolate it behind a small layer within the aspect, to protect the rest of the aspect code from being exposed to it. In the simplest case, all that has to be done is to ensure that the aspect is provided with accessor methods encapsulating the intertype fields. Only those methods will need to be changed when the structure is replaced. In the case of preparing intertype declarations to be pulled up, *replace intertype field with aspect map* must be applied to each of subaspects in turn. Next, use *pull up field* ([10], p. 320) and *pull up method* ([10], p. 322) to pull the state and its associated logic to the common superaspect.

Preconditions. Ensure that the fields in the various aspects do indeed provide equivalent interfaces and functionality.

Mechanics.

- Use *encapsulate field* ([10], p. 206) on the introduced field. Unlike traditional accessor methods, create aspect methods, receiving the target object as argument.
- Add to the aspect a mapping structure capable of supporting the equivalent mapping functionality. Add accessors similar to the ones created in the previous step, retrieving the introduced fields from the mapping structure. Ideally, these map-based accessors should have the same signatures and names as those created in the previous step. Add any additional management methods (i.e., for insertion, removal, etc.) that may also be required.
- If the aspect has intertype methods using the intertype field, use *replace intertype method with aspect method* to create aspect versions of those methods, based on the new mapping structure.
- Compile and test.
- Replace each call to the accessors created in the first step with the map-based accessors. Compile and test when all replacements are done.
- Remove the accessor methods created in the first step. Compile and test.
- Remove the intertype field and related code. Compile and test.

Example: Replacing an Intertype Field with an Aspect Map. The following example presents fragments of an aspect implementing an instance of the Mediator pattern ([11], p. 273), adapted from a Java implementation by Cooper [5]. In this example, there is a mediator object (of type Mediator) acting as the hub of communication between various colleagues. The colleagues are instances of ClearButton and Move Button, both subclasses of javax.swing.JButton, and KidList, which is a subclass of javax.swing.JScrollPane, implementing a listener interface from the javax.swing.event API. This example declares the Colleague role as a marker interface and assigns it to the three colleague participant types. The aspect indirectly introduces in each colleague a reference to the mediator, by way of the marker interface.

This implementation is unsuitable because it introduces the additional state and behaviour to all instances of the participant classes, independently of whether all of them

need it or not. By replacing this implementation with one based on a map, we eliminate this inflexibility.

```
public aspect Mediating ...
    private interface Colleague {}
    private Mediator Colleague.mediator;

    declare parents:
       (ClearButton || MoveButton || KidList) implements Colleague;

    pointcut clearButtonExecute(ClearButton clearButton): ...
    after(ClearButton clearButton):clearButtonExecute(clearButton){
       clearButton.mediator.clear();
    }

    pointcut moveButtonExecute(MoveButton moveButton): ...
    after(MoveButton moveButton): moveButtonExecute(moveButton) {
       moveButton.mediator.move();
    }

    pointcut kidListChanged(KidList kidList): ...
    after(KidList kidList) returning: kidListChanged(kidList) {
       kidList.mediator.select();
    }
```

As a first step, we perform a refactoring similar to *encapsulate field* ([10], p. 206) to produce a temporary setter method for the intertype field. The same setter can be used in all different target types. It cannot be given exactly the same name as the map-based setter, so we add a zero to avoid compiler errors.

```
public aspect Mediating ...
    private Mediator getMediator0(Colleague colleague) {
       return colleague.mediator;
    }
    pointcut ...
    after(ClearButton clearButton):clearButtonExecute(clearButton) {
       getMediator0(clearButton).clear();
    }
    pointcut ...
    after(MoveButton moveButton): moveButtonExecute(moveButton) {
       getMediator0(moveButton).move();
    }
    pointcut ...
    after(KidList kidList) returning: kidListChanged(kidList) {
       getMediator0(kidList).select();
    }
```

Now that all accesses to the intertype field are done through this temporary setter, the intertype nature of the mediator field is effectively encapsulated. Next, we add a suitable data structure to map the target objects to the mediator field. A hash table is a good choice for these cases. The introduced field was private to the aspect, so the setters are private as well. The access mode of the map-based setter can be more problematic. Note that the map-based setter is responsible for associating the target object with the mediator field, using the newly added mapping structure. It does not

have a correspondent statement in the original version of the code, but we must find an appropriate point of the program to place it. The access mode of the map-based setter depends on where the field is used in the system: private if it is used only within the aspect, nonprivate otherwise. In this example, we assume a public access.

```
import java.util.WeakHashMap;

public aspect Mediating ...
   WeakHashMap colleague2mediatorMap = new WeakHashMap();

   private Mediator getMediator(Colleague colleague) {
       return (Mediator)colleague2mediatorMap.get(colleague);
   }
   public void setMediator(Colleague colleague, Mediator mediator) {
       colleague2mediatorMap.put(colleague, mediator);
   }
   private Mediator getMediator0(Colleague colleague) {
       return colleague.mediator;
   }
```

We must now decide on the places where to put the calls to the map-based setters. The places where the objects containing the field are created could be used as a basis, though in some cases it may be preferable to place the calls elsewhere. After all, that is precisely one of the advantages of replacing a static mapping with a dynamic one: we have more choices. Outside the aspect, the calls to the final setter should be something like this:

```
Mediating.aspectOf().setMediator(clearButton, mediator);
```

Inside advice within the aspect, the same call can be expressed in a simpler way:

```
setMediator(clearButton, mediator);
```

We insert the calls to the map-based setter and make the calls to the temporary setter refer to the map-based setter. After compiling and testing again, we can delete the original declaration and the temporary setter. Now the aspect's code looks like this:

```
public aspect Mediating ...
   private Mediator Colleague.mediator;
   declare parents: (ClearButton || MoveButton || KidList)
       implements Colleague;

   WeakHashMap colleague2mediatorMap = new WeakHashMap();

   private Mediator getMediator(Colleague colleague) {
       return (Mediator)colleague2mediatorMap.get(colleague);
   }
   public void setMediator(Colleague colleague, Mediator mediator) {
       colleague2mediatorMap.put(colleague, mediator);
   }

~~   private Mediator getMediator0(Colleague colleague) {~~
~~       return colleague.mediator;~~
```

```
}
pointcut clearButtonExecute(ClearButton clearButton): ...
after(ClearButton clearButton): clearButtonExecute(clearButton) {
    getMediator(clearButton).clear();
}
pointcut moveButtonExecute(MoveButton moveButton): ...
after(MoveButton moveButton): moveButtonExecute(moveButton) {
    getMediator(moveButton).move();
}
pointcut kidListChanged(KidList kidList): ...
after(KidList kidList) returning: kidListChanged(kidList) {
    getMediator(kidList).select();
}
```

Example: Making an Implementation of Observer Amenable for the Extraction of a Superaspect. This second example is an implementation of Observer ([11], p. 293). This implementation was extracted into an aspect from the example by Cooper [5], using *extract feature into aspect*. This example is a bit more complex than the previous one, because it includes intertype methods that use the intertype field. These intertype methods must be replaced using *replace intertype method with aspect method*. We assume the scenario in which the system has other, similar implementations of the pattern and we would like to factor out the common elements by pulling them up to a superaspect. These implementations rely on the introduction of a java.util.Vector field to the subject participant, which is among the elements we would like to pull up, along with its associated logic.

The present implementation does not lend itself to be pulled up to the superaspect, for the same reasons as in the previous example: It was designed assuming there would be only one instance of the pattern for each subject. That is, the vector cannot support multiple observing relationships for the same object. To solve this problem, we will replace the intertype vector with a more suitable hash table owned by the aspect, which will manage the mappings between subjects and the list (i.e., a java.util.Vector object) of its observers. We will use *replace intertype method with aspect method* to replace the original logic using the vector with aspect logic using the hash table.

Cooper's example includes a Watch2LSubject object as subject and two types of observers, which are instances of ListFrameObserver and ColorFrameObserver (both subclasses of javax.swing.JFrame). The Watch2LSubject object includes three radio buttons, one for each of the colours red, green and blue. Whenever a different radio button is selected, the ColorFrameObserver instances change their background colour accordingly, and the ListFrameObserver adds the name of the selected colour to its list.

The refactored aspect uses two inner interfaces (they were inlined to within the Observing aspect during the refactoring process) to represent the roles of subject and observer. It introduces the java.util.Vector field to the objects playing the role of subject, which holds the subject's registered observers. The aspect also introduces two methods to the subjects: addObserver(Observer), which is used to register a new observer for the subject, and notifyObservers(JRadioButton), through which subjects notify all their registered observers of a change in the selected colour. That notification is carried out through the sendNotify method, which is declared in the Observer

inner interface. The sendNotify method receives a string representing the new colour as parameter. The aspect also introduces the implementation of sendNotify for each concrete observer type.

```
public aspect Observing ...
    private interface Subject {}
    interface Observer {
      /** notify the Observers that a change has taken place */
      public void sendNotify(String s);
    }
    declare parents: Watch2LSubject implements Subject;
    declare parents: (ListFrameObserver || ColorFrameObserver)
      implements Observer;

    private Vector Subject._observingFramesList = new Vector();

    public void Subject.addObserver(Observer obs) {
      //   adds observer to list in Vector
      _observingFramesList.addElement(obs);
    }
    /* sends text of selected button to all observers */
    private void Subject.notifyObservers(JRadioButton rad) {
      String sColor = rad.getText();
      for (int i = 0; i < _observingFramesList.size(); i++ ) {
        ((Observer) (_observingFramesList.elementAt(i))).
          sendNotify(sColor);
      }
    }

    public void ListFrameObserver.sendNotify(String s) {
      _listData.addElement(s);
    }
    public void ColorFrameObserver.sendNotify(String str) {
      changeColor(str);
    }
```

The aspect also includes a pointcut and corresponding advice to trigger the adequate behaviour when the subject changes the selected colour:

```
pointcut watchStateChange(Watch2LSubject watch,ItemEvent event) :...
after(Watch2LSubject watch, ItemEvent event):
      watchStateChange(watch, event) {
    if(event.getStateChange() == ItemEvent.SELECTED)
      watch.notifyObservers((JRadioButton) event.getSource());
}
```

The mechanics prescribe the use of *Encapsulate Field* ([10], p. 206) on the existing field. In this particular case, we must instead create a new field as the mapping structure (we will create the accessor methods for the structure as soon as there is a need to do so).

```
import java.util.WeakHashMap;
...
public aspect Observing ...
    //...
    WeakHashMap _subject2Observers = new WeakHashMap();
```

Next, we use *replace intertype method with aspect method* to replace the addObserver and notifyObservers intertype methods with aspect versions using the new mapping structure. See the example section of *replace intertype method with aspect method* (Sect. 4.5) for more details of this step.

The new implementation is now in place and working. There was no need to add accessors to the mapping structure, as it is already encapsulated by addObserver and notifyObservers. These two aspect methods comprise a small layer hiding the structure. We can now delete the old implementation, after which the aspect looks like this:

```
public aspect Observing ...
    private interface Subject {}
    interface Observer {
      /** notify the Observers that a change has taken place */
      public void sendNotify(String s);
    }
    declare parents: Watch2LSubject implements Subject;
    declare parents: (ListFrameObserver || ColorFrameObserver)
      implements Observer;

    private Vector Subject._observingFramesList = new Vector();

    public void Subject.addObserver(Observer obs) {
       //   adds observer to list in Vector
       _observingFramesList.addElement(obs);
    }
    /* sends text of selected button to all observers */
    private void Subject.notifyObservers(JRadioButton rad) {
       String sColor = rad.getText();
       for (int i = 0; i < _observingFramesList.size(); i++ ) {
          ((Observer) (_observingFramesList.elementAt(i))).
             sendNotify(sColor);
       }
    }

    WeakHashMap _subject2Observers = new WeakHashMap();

    public void addObserver(Subject subject, Observer observer) {
      Vector observers;
      Object obj = _subject2Observers.get(subject);
      if(obj == null)
        observers = new Vector();
      else observers = (Vector) obj;
      observers.add(observer);
      _subject2Observers.put(subject, observers);
    }
    public void
    notifyObservers(Subject subject, JRadioButton radioButton) {
      String sColor = radioButton.getText();
      Vector observersList =
        (Vector)_subject2Observers.get(subject);
      for (int i = 0; i < observersList.size(); i++ ) {
        ((Observer) (observesList.elementAt(i))).
          sendNotify(sColor);
      }
    }
```

```
    public void ListFrameObserver.sendNotify(String s) {
       _listData.addElement(s);
    }
    /* Observer is notified of change here */
    public void ColorFrameObserver.sendNotify(String str) {
       changeColor(str);
    }

    pointcut watchStateChange(Watch2LSubject watch,ItemEvent event):
       ...
    after(Watch2LSubject watch, ItemEvent event):
         watchStateChange(watch, event) {
       if(event.getStateChange() == ItemEvent.SELECTED)
         notifyObservers(watch, (JRadioButton) event.getSource());
    }
}
```

4.5 Replace Intertype Method with Aspect Method

Typical situation. An aspect introduces additional methods to a class or interface, when a more dynamic and flexible composition would be desirable.

Recommended action. Replace the intertype method with an aspect method that gets the target object as an extra parameter.

Motivation. This refactoring was designed to be a follow-up to *replace intertype field with aspect map*. That refactoring deals with intertype fields and the present refactoring deals with the (intertype) methods that use those fields.

The present refactoring is made possible by the fact that a method introduced to a class can always be replaced by a similar aspect method receiving an instance of the target class as an additional argument, which will use the target object as a key.

```
public class Capsule {
    private int _value;
    public Capsule(int value) {
       _value = value;
    }
```

```
public aspect Additional {
    public void Capsule.doSomethingMore() {
       System.out.println("Doing something more with " + this);
    }
```

```
       Capsule capsule = new Capsule(7);
       capsule.doSomethingMore();
```



```
public class Capsule {
    private int _value;
    public Capsule(int value) {
       _value = value;
    }
```

```
public aspect Additional {
    public void doSomethingMore(Capsule capsule) {
       System.out.println("Doing something more with " + capsule);
    }
```

```
       Capsule capsule = new Capsule(7);
       Additional.aspectOf().doSomethingMore(capsule);
```

Replacements of this kind should not be made in the general case, and that is why we prescribe using this refactoring only in the context of *replace intertype field with aspect map*. This refactoring is equally useful to deal with both situations covered by the other refactoring: (1) replacing intertype declarations with a dynamic mechanism and (2) preparing intertype state duplicated in various aspects to be factored out to a common superaspect. This refactoring transforms existing intertype methods into aspect methods based on the map that was created when applying *replace intertype field with aspect map*.

Mechanics.

- Create in the aspect a copy of the intertype method, with the same name and signature. Insert, in the beginning of the aspect method's parameter list, an additional parameter whose type is the original target of the intertype declaration.
- Replace each reference to "this" with the new parameter. Change all self-calls and references to fields to refer to the new first parameter.
- Compile and test.
- Change the body of the intertype method so that it calls the aspect method, if there are no further dependences preventing you.
- Add a "declare warning" exposing all calls to the intertype method:

```
declare warning:
    (call(<type> <host class>.someMethod(<arguments>)):
    "method <host class>.someMethod() is called here.";
```

- Following the warnings, replace each call to the intertype method with a call to the aspect method. Compile and test after each change.
- When there are no more warnings, delete the "declare warning" and the intertype method (when covering the mechanics of several refactorings from [10], Fowler considers the situation when the existing method is part of the interface and cannot be changed; Fowler recommends that in such cases the old method be left in place and marked as deprecated).
- Compile and test.

Example. This example is part of the second example for *replace intertype field with aspect map*. In it, an aspect introduces the following methods to the Subject marker interface:

```
public void Subject.addObserver(Observer obs) {
   _observingFramesList.addElement(obs);
}
private void Subject.notifyObservers(JRadioButton rad) {
   String sColor = rad.getText();
   for (int i = 0; i < _observingFramesList.size(); i++ ) {
      ((Observer) (_observingFramesList.elementAt(i))).
         sendNotify(sColor);
   }
}
```

As an example of client code, the following subject and observers are created and registered, through calls to the Subject.addObserver method:

```
Watch2LSubject subject = new Watch2LSubject();
//Observing.aspectOf().setSubject(subject);

ColorFrameObserver cframeObs1 = new ColorFrameObserver();
ColorFrameObserver cframeObs2 = new ColorFrameObserver();
ColorFrameObserver cframeObs3 = new ColorFrameObserver();
ListFrameObserver lframeObs = new ListFrameObserver();

subject.addObserver(cframeObs1);
subject.addObserver(cframeObs2);
subject.addObserver(cframeObs3);
subject.addObserver(lframeObs);
```

The aspect itself also includes an advice calling the other method, Subject.notifyObservers:

```
after(Watch2LSubject watch, ItemEvent event):
    watchStateChange(watch, event) {
  if(event.getStateChange() == ItemEvent.SELECTED)
    watch.notifyObservers((JRadioButton) event.getSource());
}
```

This functionality should be replaced by aspect methods based on a hash table owned by the aspect: the aspect field _subject2Observers, which uses subject objects as keys, and vectors of observers as values:

```
WeakHashMap _subject2Observers = new WeakHashMap();
```

As a first step, we create the following two aspect methods, with the same names:

```
public void addObserver(Subject subject, Observer observer) {
  Vector observers;
  Object obj = _subject2Observers.get(subject);
  if(obj == null) observers = new Vector();
  else observers = (Vector) obj;
  observers.add(observer);
  _subject2Observers.put(subject, observers);
}
public void
notifyObservers(Subject subject, JRadioButton radioButton) {
  String sColor = radioButton.getText();
  Vector observersList =
    (Vector)_subject2Observers.get(subject);
  for (int i = 0; i < observersList.size(); i++ ) {
    ((Observer) (observersList.elementAt(i))).
      sendNotify(sColor);
  }
}
```

We cannot replace the body of the intertype methods with calls to the new ones at this point. We must first replace the calls to the addObserver method, which register the observers to their subjects. Otherwise, the tests would fail. We therefore perform the next step as prescribed, adding "declare warning" clauses that will expose all calls to these methods:

```
declare warning: call(void Subject.addObserver(Observer)):
    "Method Subject.addObserver(Observer) is called here.";
declare warning: call(void Subject.notifyObservers(JRadioButton)):
    "Method Subject.notifyObservers(JRadioButton) is called here.";
```

We compile, resulting in a series of warnings locating the calls to the old methods. After replacing each of them with calls to the aspect methods, we compile again. All warnings disappeared, and we test. We remove the "declare warning" clauses. Now the client code calling addObservers looks like this:

```
      Watch2LSubject watch2LFrame = new Watch2LSubject();

      ColorFrameObserver cframeObs1 = new ColorFrameObserver();
      ColorFrameObserver cframeObs2 = new ColorFrameObserver();
      ColorFrameObserver cframeObs3 = new ColorFrameObserver();
      ListFrameObserver lframeObs = new ListFrameObserver();

      subject.addObserver(cframeObs1);
      subject.addObserver(cframeObs2);
      subject.addObserver(cframeObs3);
      subject.addObserver(lframeObs);

      Observing.aspectOf().addObserver(watch2LFrame, cframeObs1);
      Observing.aspectOf().addObserver(watch2LFrame, cframeObs2);
      Observing.aspectOf().addObserver(watch2LFrame, cframeObs3);
      Observing.aspectOf().addObserver(watch2LFrame, lframeObs);
```

The call to notifyObservers now takes the form:

```
    after(Watch2LSubject watch, ItemEvent event):
        watchStateChange(watch, event) {
      if(event.getStateChange() == ItemEvent.SELECTED)
        notifyObservers(watch, (JRadioButton) event.getSource());
    }
```

4.6 Tidy Up Internal Aspect Structure

Typical situation. The internal structure of an aspect resulting from the extraction of a CCC is suboptimal, being based on static compositions and betraying duplication.

Recommended action. Tidy up the internal structure of the aspect by removing duplicated intertype declarations and dependencies on case-specific target types.

Motivation. This refactoring serves as the general framework indicating when to use the remaining refactorings from the same group,[3] and in what situations.

[3] Each refactoring from the group is not necessarily referred to *directly*.

AOP adds a new type of situation in which code duplication can arise (i.e., is exposed). Refactoring an object-oriented (OO) code base to aspects entails extracting concerns and features whose very crosscutting nature gives rise to duplication that is hard or impossible to avoid when using traditional OO mechanisms. A typical situation is a system containing repeated implementations of the same functionality scattered in multiple classes. Simply extracting those code snippets into an aspect does not guarantee, by itself, removal of this duplication. It merely moves the duplicated code into aspects. In some cases, the duplication becomes obvious only when it is placed in a single module. Therefore, extracting the crosscutting code is only the first part of the job. Next, duplication within the aspect must be removed and its internal structure improved.

Intertype declarations make it very easy to move members from classes to aspects without impact on client code, and aspects resulting from extractions are likely to use them. However, in some cases, we would like the aspect to introduce the additional state and behaviour on an object-by-object basis, and intertype declarations are not flexible enough to achieve that. This entails the replacement of these introductions with different logic.

Mechanics.

- If the code assigns roles to participant classes, see if the aspect code uses marker interfaces to represent those roles instead of referring directly to case-specific classes. If it is not the case, use *generalise target type with marker interface*.
- If parts of the code make explicit references to specific classes that cannot be generalised, separate the specific parts from the generally applicable ones by using *extract method* ([10], p. 110). You should do this if the aspect contains enough generally applicable logic to be worth extracting to a reusable abstract superaspect.
- Inspect the intertype declarations looking for cases in which the extra state and behaviour is needed only at specific times, or is needed by only a subset of the instances of the target classes, or may be needed in multiple instances simultaneously. In such cases, consider using *replace intertype field with aspect map* to deal with the introduced state, and *replace intertype method with aspect method* to deal with the behaviour based on that state.

Example. The refactoring process described in [29] includes a thorough example of this composite refactoring.

5 Code Smells

Code smells are the way proposed by Beck and Fowler (Chap. 3 of [10]) 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 style and to decide when a symp-

tom 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 she is working.

5.1 OO Smells in Light of AOP

We analysed the code smells presented in [10, 21, 37] and propose that some be used as symptoms of the presence of CCCs. This particularly applies to *divergent change* ([10], p. 79) and *shotgun surgery* ([10], 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). Wake [37] mentions configuration information, logging and persistence as possible causes to the *shotgun surgery* smell, all of which can be counted among the favourite examples for the use of AOP.

Kerievsky [21] proposes a variant of *shotgun surgery* that he calls *solution sprawl*. Kerievsky states ([21], 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 CCCs.

We think it is useful to extend the above definitions to cover methods as well as classes, to account for class-wide aspects that cut across the methods of a single class. We propose the *extract feature into aspect* refactoring (Table 1 and Sect. 3.2) as a general framework for the modularisation of concerns detected through these smells.

5.2 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 [11] that include what Hannemann and Kiczales call *superimposed roles* – roles assigned by the pattern to classes that have functionality and responsibility outside the pattern [16]. Examples are *chain of responsibility* ([11], p. 223), which superimposes the Handler role to some of the participant classes, and *observer* ([11], 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* ([10], p. 341). When a class implements an interface modelling a role that does not relate to the class's primary concern, the class smells of *double personality*.

When *double personality* is detected in one class, we suggest that developers analyse 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 a single 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* [25] suggests placing the secondary concern inside an inner aspect enclosed within the interface modelling the superimposed role. If the programmer strives for total obliviousness [9] of the secondary role, she can use *replace implements with declare parents* (Table 1). As an alternative to *extract interface implementation* [25], we propose *split abstract class into aspect and interface* (Table 1), which completely encapsulates the secondary concern into an aspect, including the "implements" clause. When the related code is more complex than a simple implementation of an interface, we suggest using *extract feature into aspect* (Table 1) to move all the related code to an aspect (see also Sect. 3.2).

5.3 Abstract Classes as a Code Smell

The AspectJ composition mechanisms that enable the emulation of mixins [2] also enable the separation of definitions (i.e., implementation code) from declarations in abstract classes, opening the way to turn the classes into interfaces. Hannemann and Kiczales take this approach in implementing five of the GoF design patterns in AspectJ [16]. 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 be considered a code smell in some situations – e.g., whenever we would like a class to inherit from some other class, but the class already inherits from an abstract class that contains implementation elements. Two of the refactorings presented here (Table 1) remove that smell. *Split abstract class into aspect and interface* can be used to extract the concrete members of an abstract class into an aspect, and resulting pure abstract class can be turned into an interface using *change abstract class to interface*.

5.4 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 intertype declarations. We detect this smell in aspects that resort to the mechanism of intertype declarations to add state and behaviour to a class when something more dynamic and/or flexible would be desirable. Intertype declarations are static mechanisms that apply to all instances of the target class, throughout their entire life cycle. Its use should be considered a smell in some situations. We detect *aspect laziness* in uses of intertype declarations for solving problems whose requirements have one or several of the following characteristics:

- The additional state and/or behaviour are needed by only a subset of the instances of the target classes.
- The additional state and/or behaviour 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 behaviour simultaneously.

In such cases, intertype declarations are not dynamic or flexible enough. It is preferable for the aspect itself to hold the additional state and behaviour and programmatically associate the additional state to the individual target objects. We propose *replace intertype field with aspect map* and *replace intertype method with aspect method* (Table 2) to replace the existing design with a mapping logic that provides the same functionality more flexibly.

6 Illustrative Example

In this section, we present a code example to illustrate some of the smells and the results of many of the refactorings. The example is based on an implementation of the *Observer* pattern ([11], p. 293) by Eckel [8]. In [29], we describe in detail a refactoring process that starts with Eckel's implementation and ends with the AspectJ implementation proposed by Hannemann and Kiczales [16]. The process uses 17 of the refactorings presented in this paper, shown in Table 4.

Table 4. Refactorings used in the illustrating example

Encapsulate implements with declare parents	*Move field from class to intertype*
Extend marker interface with signature	*Move method from class to intertype*
Extract feature into aspect	*Push down advice*
Extract inner class to stand-alone	*Pull up marker interface*
Extract fragment into advice	*Pull up pointcut*
Extract superaspect	*Replace intertype field with aspect map*
Generalize target type with marker interface	*Replace intertype method with aspect method*
Inline class within aspect	*Tidy up internal aspect structure*
Inline interface within aspect	

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" [11]. The example includes two observers, one of which is class Bee, shown in Fig. 1 with the primary concern shaded (the other observer class, Hummingbird, is similar). Figure 2 shows the class Flower, which plays the role of Subject (shaded code relates to the primary concern). Each of Flower's two operations, open and close the petals, originates one observing relationship.

Eckel's implementation uses the Observer/Observable protocol from Java's standard java.util API, which requires Subject participant to inherit from java.util.Observable. Eckel's design manages to partially isolate the two observing relationships

```
01 public class Bee {
02   private String name;
03   private OpenObserver openObsrv = new OpenObserver();
04   private CloseObserver closeObsrv = new CloseObserver();
05
06   public Bee(String nm) { name = nm; }
07   private class OpenObserver implements Observer {
08     public void update(Observable ob, Object a) {
09       System.out.println("Bee "+name +"'s breakfast time!");
10     }
11   }
12   private class CloseObserver implements Observer{
13     public void update(Observable ob, Object a) {
14       System.out.println("Bee " + name + "'s bed time!");
15     }
16   }
17   public Observer openObserver() {
18     return openObsrv;
19   }
20   public Observer closeObserver() {
21     return closeObsrv;
22   }
23 }
```

Fig. 1. Bee class as observer in the implementation of the observer pattern from [8]

by defining, for each relationship, an inner class inside each participant. Thus, Flower defines two inner classes (Fig. 2, lines 25–37 and 38–50, respectively) that inherit from java.util.Observable. The classes within Flower use two inherited methods: (1) setChanged (used in lines 29 and 42), which marks a subject as having been changed, and (2) notifyObservers, which notifies all its observers if subject was changed. Though notifyObservers is overridden (lines 27–33 and 40–46), its functionality is reused (in lines 30 and 43).

Each observer likewise encloses one inner class implementing java.util.Observer for each observing relationship (Fig. 1, lines 7–11 and 12–16, respectively). As prescribed by the interface, each inner class defines an update method (lines 8–10 and 13–15). All participants in the pattern betray strong doses of *double personality*.

The example shows that OO does not cope well with concerns affecting multiple objects and classes, forcing programmers to produce decentralised designs for CCCs, when they would rather centralise the concern's implementation within some module. Such designs lead to duplicated code in every class playing some role in the concern.

OO 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 organise 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's 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 [35].

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

Fig. 2. Flower class as subject in the implementation of the observer pattern from [8]

Figure 3 shows the participants from Figs. 1 and 2, after each of the two observing relationships was extracted to its own aspect, using the refactorings from Table 1. During the extraction of both observing relationships [29] the isOpen field (Fig. 3, line 4) was encapsulated, yielding two new methods for the Flower class: isOpen

(lines 7–9) and setIsOpen (lines 10–12). The code for the reaction of the observers when they are notified of open and close events was likewise extracted to methods breakfastTime (lines 28–30) and bedtimeSleep (lines 31–33) respectively.

```
01 public class Flower {
02    private boolean _isOpen;
03
04    public Flower() {
05    _isOpen = false;
06    }
07    boolean isOpen() {
08       return _isOpen;
09    }
10    private void setIsOpen(boolean newValue) {
11       _isOpen = newValue;
12    }
13    public void open() { // Opens its petals
14       System.out.println("Flower open.");
15       setIsOpen(true);
16    }
17    public void close() { // Closes its petals
18       System.out.println("Flower close.");
19       setIsOpen(false);
20    }
21 }
```

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

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

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. We can see from Figs. 3 and 4 that the code for implementing the Observer pattern is no longer spread across the participant classes. However, the structure of the aspect resulting from the extraction still hardly resembles the one presented in [16], as ideally would be the case. The internal structure of the extracted aspect (Fig. 4) still reflects the original, decentralised design. The aspect betrays *duplicated code* ([10], p. 76), as it introduces identical fields (Fig. 4, lines 9 and 10–11) and methods (lines 16–18 and 19–21) to the two observer participants. The duplication was always present, but now that the code is modularised, it is clearly exposed. After modularisation, the original design is no longer justified and the inner classes comprise a needlessly complicated structure. The

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

Fig. 4. Part of the extracted aspect ObservingOpen modularising observations of Flower's open operation

code also betrays *aspect laziness*. 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.

Hannemann and Kiczales mention four modularity properties [16] for their implementation of the Observer pattern: locality, reusability, composition transparency and (un)pluggability. Just after the extraction, the aspect (Fig. 4) has only the first and last of these properties. Figure 5 shows a refactored aspect whose structure is close to that presented in [16].

The static nature of intertype declarations can lead to the *aspect laziness* smell. At the very least, the extracted aspect will need a tidying up. In some cases, including the present one, it requires a complete redesign. Intertype declarations are one of the reasons why the structure of aspects resulting from extraction processes is often unsuitable. Intertype 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 intertype declarations) and therefore make it simple to move members from classes to aspects. However, only the source code is modularised: the intertype members still belong to their respective target classes at the binary and runtime levels.

```
public aspect ObservingOpen {
    private interface Subject {}
    private interface Observer {}
    public abstract boolean Subject.isOpen();
    private boolean Subject.alreadyOpen = false;
    public abstract void Observer.breakfastTime();

    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;
}
```

Fig. 5. Aspect ObservingOpen after being tidied up

The transformations prescribed by *tidy up internal aspect structure* (Table 2 and Sect. 3.3) can transform the ObservingOpen aspect from Fig. 4 to the one shown in Fig. 5. In this example, we use the same implementation as in the reusable aspect for the Observer pattern [16], based on a weak hash map. The abstract declarations of methods isOpen and breakfastTime (Fig. 5) result from using *extend marker interface*

with signature, which was needed to separate generically applicable code from case-specific code.

7 Related Work

Deursen et al. [6] 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 aspects?" and "how can the introduction of aspects be described in terms of a catalogue of new refactorings?" In this paper, we contribute to answering both questions.

Iwamoto and Zhao announced in [18] their intention to build a catalogue of AOP refactorings. They present a catalogue of 24 refactorings, but the information provided about them is limited to the names of the refactorings. The refactorings we describe in this paper and in [27] include a description of the situations where the refactoring applies, mention of preconditions, detailed mechanics and code examples.

Several authors [15, 18, 24, 36, 38] call into attention the *fragile pointcut problem* (not always naming it this way), in some cases illustrating it with some code examples. The authors conclude that existing OO refactorings [10] cannot be applied to code bases with aspects. In [25], Laddad provides a few guidelines to ameliorate the problem, including suggestions on how to design and evolve pointcuts. 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 nontrivial systems.

Hanenberg et al. [15] propose *aspect-aware* refactorings – refactorings that take into account the presence of aspects and preserve behaviour by updating any pointcuts that may be affected by the transformation – and propose a set of enabling conditions to preserve the observable behaviour. 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.

In [14] and [34] Griswold, Sullivan and other authors propose a novel approach based on information-hiding interfaces for CCCs. Their approach entails hiding the implementation details (i.e., joinpoints) of code base behind *crosscut programming interfaces* (XPIs) [14] against which aspects are written. The XPIs prevent direct

dependencies of aspects on the code bases they advise and enforce design rules [34] that constrain the base code developers to honour the contract expressed through XPIs. Thus, this approach promises to decouple base code from aspect code in a more symmetric way and to solve the fragile pointcut problem. In [34], the authors discuss a comparative study they undertook of three implementations of a real software system, developed independently of the analysis. The authors refactored the system to both a version that conforms to the rules they propose and the more traditional nonsymmetric AOP approach that relies on obliviousness. The study suggests that the new approach brings benefits relative to the other two. To our knowledge, [14] is the first work attempting to provide clear rules on how to design base code for ease of advising. Though it is not expressed in terms of refactorings and code smells, the approach proposed in [14] and [34] contributes to developing a new style appropriate for AOP.

Hanenberg et al. [15] propose three AOP refactorings – *extract advice*, *extract introduction* and *separate pointcut*. Their *extract advice* corresponds to our *extract fragment into advice* (Table 1). Our collection of refactorings goes deeper in exploring the refactoring space; in this paper and in [27] we provide more detail and tackle 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 intertype* and *move method from class to intertype* (Table 1) 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 organise 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 should be a code smell.

In [25], Laddad presents a collection of refactorings [25] tailored to practitioners working in industry, particularly 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 enforcement*. In addition, some refactorings belong to the category of "refactoring to patterns" as presented by Kerievsky [21] – *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 [26] – *worker object creation* ([26], p. 247) and *wormhole* ([26], p. 256) respectively. The *extract exception handling* refactoring as presented in [25] goes towards a variant implementation of the *exception introduction* pattern ([26], p. 260).

Laddad's refactorings and ours cover different areas of the AOP refactoring space, providing different and complementing contributions to filling that space. Some of

Laddad's refactorings are presented with only a mention of their name and a brief motivating paragraph. We believe the refactorings would benefit if presented in the same format as used by Fowler et al. [10] and Kerievsky [21], and which we use as well [27, 28]. 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. In addition, several refactorings (namely the problem-specific ones) can be decomposed into simpler, lower-level steps, always an important thing with refactoring.

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 [35] 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 in Table 1 and document in [27]. 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 [17], Hannemann et al. propose that refactoring support for AOP be divided into three categories: aspect-aware OO refactorings (the concept proposed by Hanenberg et al.), aspect-oriented refactorings (i.e., refactorings that specifically target AOP constructs, such as those presented in this paper) and *refactorings of crosscutting concerns*, i.e., refactorings in which the scattered elements comprising a target CCC and their individual transformations are considered together, instead of handling each element separately. The latter category can only be carried out with the support of a suitable tool. The focus of [17] is to present one such tool. Some of Laddad's refactorings [25], such as *extract method calls*, *extract concurrency control* and *extract contract enforcement*, would be refactorings of CCCs if had some suitable tool support. Such refactorings tend to be concern-specific: these contrasts with ours, which aim to be applicable to multiple concerns, like those documented by Fowler et al. [10].

Like us, Hannemann et al. [17] use the Observer pattern ([11], p. 293) as a basis for an illustrating example. They provide the outline for a refactoring process comprising the extraction from a code base of a general implementation of Observer. The outline is much less detailed than the one we present in [29], which focuses on a specific Java implementation of Observer by Eckel. The outcome of their illustrating refactoring is

the AspectJ implementation [16] of Observer, which we also use in Sect. 6 and in [29]. Not surprisingly, there are similarities between some refactorings presented here and various refactorings that Hannemann et al. report using in their work:

- Their *add internal interface* is subsumed by our *generalise target type with marker interface* (Table 2 and Sect. 4.2).
- Their *replace object method with aspect method* is similar to our *replace intertype method with aspect method* (Table 2 and Sect. 4.5).
- Their *replace method call with pointcut and advice* corresponds to our *extract fragment into advice* (Table 1), the code fragment being a method call.
- Their *replace method with intertype method declaration* and *replace field with intertype field declaration* corresponds to ours *move method from class to intertype* and *move field from class to intertype* (Table 1), respectively.

In [3], 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 [15, 18, 25], 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 CCCs to aspects, and therefore there is some overlap between the refactorings they derive and our own extraction refactorings (Sect. 3.2). However, their focus is on providing proofs that the transformations are behaviour-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 generalisable, because its implementation of distribution is commonly used, and claim that it is possible to derive a concern-specific *extract distribution* refactoring. No details are given, though.

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 [35], we do not have knowledge of any other work covering the issue of AOP code smells.

8 Future Work

8.1 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. More complex refactoring experiments may expose problems and situations that should be taken into account in the preconditions and mechanics sections. Refactoring experiment targeting other languages should be performed to assess the validity of the refactorings beyond the Java/AspectJ space.

8.2 Expanding the Refactoring Space

Covering Other Language Characteristics. The refactorings we present here result from the two specific case studies, and do not use every available aspect construct, nor do they explore every possible combination. 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.

- *Nonsingleton 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 nonsingleton aspects, e.g., when should they be preferred and what refactorings should be used to transform singleton aspects.
- *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 [9] 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.

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. In IDEs and refactoring tools, the opposite of a refactorings correspond to the "undo" of that refactoring. In addition, opposites are often useful in their own right (e.g., pull up vs. push down refactorings).

Dealing with Published Interfaces. In this paper, we cover the restructuring of aspect code resulting from the extraction of CCCs, taking advantage of the newfound modularisation. It is also worth studying the impact of such extractions on the remaining code base and what actions would be desirable (e.g., post-extraction refactorings).

Restructuring the Remaining Base Code. In this paper, we cover the restructuring of aspect code resulting from the extraction of CCCs, taking advantage of the newfound modularisation. It is also worth studying the impact of such extractions on the remaining code base and what actions would be desirable (e.g., post-extraction refactorings). The XPI concept proposed by Griswold et al. [14] and associated design rules proposed by Sullivan et al. [34] provide new opportunities to expand and evolve the current refactoring space for AOP.

8.3 Other Code Smells

We believe many AOP smells wait to be discovered. For instance, use of privileged aspects is a candidate: The rationale for avoiding them is the same as for avoiding the use of public data. As Colyer and Clement remark in [4], 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 nonpublic (e.g., protected and package-protected) data. Refactoring the affected code bases to expose the nonpublic 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.

9 Summary

In this paper, we argue that collections of refactorings and code smells can be an effective way to express notions of style for AOP source code. We propose AOP-specific code smells, both for detecting CCCs in existing OO code and for improving the structure of extracted aspects – *double personality*, *abstract classes* and *aspect laziness*. 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.

Simply moving the members relating to a CCC does not yield a well-formed aspect. Extracted aspects expose problems caused by crosscutting, including *duplicated code* ([10], p. 76). *Aspect laziness* relates to the static nature of intertype declarations. We can take advantage of the newfound modularity to tidy up the aspect's internal structure with further refactorings.

We present a collection of AOP refactorings, which can remove these smells from source code, comprising the following groups:

- Ten refactorings to remove the smells related to CCCs from existing OO code. Besides covering common members such as fields and methods, these refactorings also deal with inner classes and interfaces. These refactorings are fully documented in [27].
- Six refactorings to remove problems found in extracted aspects, including *duplicated code* and *aspect laziness*. These refactorings are described in detail in this paper.
- Eleven refactorings to deal with the generalisation of aspects, i.e., the extraction of common code to superaspects. These refactorings are fully documented in [27].

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

References

[1] Beck K. Extreme programming explained: Embrace change. Addison-Wesley, Reading, MA, USA, 2000

[2] Bracha G. and Cook W. Mixin-based inheritance. In: *ECOOP/OOPSLA1990: Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications and European Conference on Object-Oriented Programming,* ACM, pp. 303–311, 1990

[3] Cole L. and Borba P. Deriving refactorings for AspectJ. In: *AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, ACM, pp. 123–134, 2005

[4] Colyer A. and Clement A. Large-scale AOSD for middleware. In: *AOSD 2004: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, ACM, pp. 56–65, 2004
[5] Cooper J. Java design patterns: A tutorial. Addison-Wesley, Reading, MA, USA, 2000. Also availabe at www.patterndepot.com/put/8/DesignJava.PDF
[6] Deursen A.v., Marin M., and Moonen L. Aspect mining and refactoring. In: *REFACE03: Workshop on REFactoring*: *Achievements, Challenges, Effects,* Waterloo, Canada, 2003
[7] Dijkstra E. Go-to statement considered harmful, *Communications of the ACM*, 11(3):147–148, 1968
[8] Eckel B. Thinking in Patterns, revision 0.9. book in progress, 2003. Available at http://www.pythoncriticalmass.com/downloads/TIPatterns-0.9.zip
[9] Filman R.E. and Friedman D.P. Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns at OOPSLA 2000*, Minneapolis, 2000
[10] Fowler M. et al. Refactoring – Improving the design of existing code, Addison-Wesley, Reading, MA, USA, 2000.
[11] Gamma E., Helm R., Johnson R., Vlissides J. Design patterns. *Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995
[12] Garcia A., Sant'Anna C., Figueiredo E., Kulesza U., Lucena C., and Staa A. Modularizing design patterns with aspects: A quantitative study. In: *AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development,* ACM, pp. 3–14, 2005
[13] Griswold W.G. Program restructuring as an aid to software maintenance. *PhD Thesis*, University of Washington, USA, 1991
[14] Griswold W.G., Sullivan K.J., Song Y., Cai Y., Shonle M., Tewari N., Rajan H. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming*, pp. 51–60, 2006
[15] Hanenberg S., Oberschulte C., Unland R. Refactoring of aspect-oriented software, net.objectdays 2003, Erfurt, Germany, 2003
[16] Hannemann J. and Kiczales G. Design pattern implementation in Java and AspectJ. In: *OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, pp. 161–173, 2002
[17] Hannemann J., Murphy G., and Kiczales G. Role-based refactoring of crosscutting concerns. In: *AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development,* ACM, pp. 135–146, 2005
[18] Iwamoto M. and Zhao J. Refactoring aspect-oriented programs. In: *4th AOSD Modelling With UML Workshop at UML'2003,* San Francisco, USA, 2003
[19] Jacobson I., Christerson M., Jonsson P., Övergaard G. Object-oriented software engineering: A use case driven approach, Addison-Wesley, Reading, MA, USA, 1992
[20] Kang K.C., Cohen S.G., Hess J.A., Novak W.E., Peterson A. Feature-oriented domain analysis feasibility study, SEI, Technical Report CMU/SEI-90-TR-21, 1990
[21] Kerievsky J. Refactoring to patterns. Addison-Wesley, Reading, MA, USA, 2004
[22] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W.G. An overview of AspectJ. In: *ECOOP 2001: Proceedings of the 15th European Conference on Object-Oriented Programming, LNCS vol. 2072*, Springer, pp. 327–353, 2001
[23] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., and Irwin J. Aspect-oriented programming. In: *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming, LNCS vol. 1241*, Springer, pp. 220–242, 1997

[24] Koppen C. and Störzer M. PCDiff: Attacking the fragile pointcut problem. In: *EIWAS 2004: Interactive Workshop on Aspects in Software*, Berlin, Germany, 2004
[25] Laddad R. Aspect-oriented refactoring, parts 1 and 2. The Server Side, 2003. www.theserverside.com/
[26] Laddad R. AspectJ in action – practical aspect-oriented programming, Manning, Greenwich, CT, USA, 2003
[27] Monteiro M.P. Refactorings to evolve object-oriented systems with aspect-oriented concepts. *Ph.D. Thesis*, Universidade do Minho, Portugal, 2005
[28] Monteiro M.P. and Fernandes J.M. Object-to-aspect refactorings for feature extraction. In: *AOSD 2004: Industry Track Paper at the 3rd International Conference on Aspect-Oriented Software Development*, Lancaster, UK, 2004
[29] Monteiro M.P. and Fernandes J.M. Refactoring a java code base to AspectJ – An illustrative example. In: *ICSM 2005: Proceedings of the IEEE International Conference on Software Maintenance 2005,* Budapest, Hungary, 2005
[30] Monteiro M.P. and Fernandes J.M. Towards a catalogue of aspect-oriented refactorings. In: *AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development,* ACM, pp. 111–122, 2005
[31] Opdyke W.F. Refactoring object-oriented frameworks. *Ph.D. Thesis*, University of Illinois at Urbana-Champaign, USA, 1992
[32] Orleans D. Separating behavioral concerns with predicate dispatch, or, if statement considered harmful. In: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA 2001*, Tampa Bay, USA, 2001
[33] Sabbah D. Aspects – From promise to reality. In: *AOSD 2004: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, ACM, pp. 1–2, 2004
[34] Sullivan K.J., Griswold W.G., Song Y., Cai Y., Shonle M., Tewari N., and Rajan H. Information hiding interfaces for aspect-oriented design. In: *ESEC/FSE 2005: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering,* ACM, pp. 166–175, 2005
[35] Tonella P. and Ceccato M. Migrating interface implementation to aspects. In: *ICSM'04*: *Proceedings of 20th IEEE International Conference on Software Maintenance,* IEEE Computer Society, Chicago, USA, pp. 220–229, 2004
[36] Tourwé T., Brichau J., and Gybels K. On the existence of the AOSD-Evolution paradox. In: *Workshop on Software-Engineering Properties of Languages for Aspect Technologies at AOSD 2003*, Boston, USA, 2003
[37] Wake W. Refactoring workbook, Addison-Wesley, Reading, MA, USA, 2004
[38] Zhang C. and Jacobsen H.-A. Quantifying aspects in middleware platforms. In: *AOSD 2003: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development,* ACM, Boston, USA, pp. 130–139, 2003
[39] Zhao J. Towards a metrics suite for aspect-oriented software. *Technical-Report*, SE-2002-136-25, Information Processing Society of Japan (IPSJ), 2002