Aspect-oriented Refactoring of Java Programs

Miguel P. Monteiro
CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

João M. Fernandes
Dep. Informática / Centro Algoritmi, Escola de Engenharia
Universidade do Minho, Portugal

1 Introduction

Aspect-Oriented Programming (AOP) is an emerging programming paradigm providing novel constructs to improve the overall modularity of a software system. The primary contribution of AOP is the modularization of crosscutting concerns (CCCs), which give rise to the negative symptoms of scattering and tangling, which make a system hard to reason with, maintain and evolve. The Java language and platform played a central role in the early years of research and development of AOP. Most aspect-oriented programming languages (AOPLs) proposed throughout the years are backwards compatible extensions to existing languages and the majority of such extensions use Java as the base language (Brichau & Haupt, 2005). Some of the proposed Java extensions were proof-of-concept projects whose development was later discontinued, but a few of them kept pace with the subsequent development of their base language and evolved into robust, full-fledged industrial languages in their own right (AspectJ, OT/J).

Refactoring (Fowler, 1999) is the task of modifying the internal structure of a software system without changing its externally observable behaviour. The availability of AOPLs that are backwards compatible extensions to Java opens the way to *aspectize* Java systems, i.e., refactor a Java system to turn it into an aspect-oriented version of that system. This is the task of *aspect-oriented refactoring* (Monteiro & Fernandes, 2006a, Laddad, 2003).

Aspectizing systems with symptoms of crosscutting concerns (CCCs) promises to bring several benefits, namely improved modularity and evolvability and the removal of negative symptoms associated to the presence of CCCs. Availability of robust, up-to-date, aspect-oriented extensions to Java makes it worthwhile to reengineer existing Java systems this way.

The purpose of this chapter is twofold. First, it provides a short overview of aspect-oriented programming in general. Second, it provides a survey of the current state-of-the-art in aspect-oriented refactoring. The rest of the chapter is organized as follows. Section 2 introduces the topic of crosscutting concerns as a fundamental problem in modern software. Section 3 introduces the topic of refactoring and establishes and clarifies the relation between CCCs and refactoring. Section 4 provides an introduction to AOP, using AspectJ as its typical representative, and discusses the fragile pointcut problem, which is possibly the primary technical problem in aspect-oriented systems. Section 5 surveys the various research fronts regarding aspect-oriented refactoring. Section 6 concludes the chapter with a summary.

2 Crosscutting Concerns

Modern software is complex, and the complexity is increasing. One of the traditional ways for programmers to cope with complexity is to use *paradigms*, i.e., principles and rules prescribing how to decompose a system into *units of modularity* and how those units are connected and composed (Parnas, 1972). Object-oriented programming (OOP) is currently the dominant programming paradigm. The reason for its success is that the decomposition units it supports (classes) are full-fledged modules that make a good fit to domain and implementation concerns. A number of benefits ensue, including abstraction, information hiding, modular substitution and pluggability. All these benefits positively contribute to the long-term evolvability of software systems. For instance, the advantage of information hiding and modular substitution is that the implementation of a concern (and corresponding module) can be improved without incurring the costs of updating and retesting the other concerns (modules) of the system.

However, it is also known that existing programming paradigms suffer from the *tyranny of the dominant decomposition* (Tarr et al., 1999), meaning that each paradigm provides a *single* decomposition criterion for a software system. As a consequence, concerns that do not align with the primary decomposition tend to cut across the decomposition units – *they are not modularized*. Though OOP proved a good paradigm through which to leverage abstraction and other positive properties, it is subject to the same shortcoming of a single decomposition like other paradigms. OOP gives rise to its own kind of non-aligning or crosscutting concerns, which tend to be non-functional since OOP is good at modularizing functional concerns.

The Java language follows the OOP paradigm and therefore shares the decomposition problems typical of OOP. In Java systems, it often happens that concerns such as persistence, exception handling, logging, distribution and indeed and most, if not all, services provided by Java middleware are scattered across the systems' classes. Traditional OOP mechanisms such as inheritance and aggregation are unable to localise the code related to such concerns within a single module. Consequently, the representation of such concerns takes the form of multiple, small code fragments that are scattered throughout the classes of the system, a phenomenon usually referred as *code scattering*. In addition, the various code fragments related to CCCs tend to be mixed with the code related to the primary functionality of the existing modules, hampering the comprehensibility and ease of evolution of all concerns involved, including the core concern. This negative effect is dubbed *code tangling*. Concerns that give rise to the scattering and tangling symptoms were named *crosscutting concerns* (CCCs) (Kiczales et al., 1997), which comprise a consequence of the tyranny of the dominant decomposition. Typical Java examples of CCCs include most services provided by Java middleware. **Figure 1** illustrates the tangling symptom with a code sketch.

The implementation of many design patterns (Gamma et al., 1995) are also examples of CCCs. Design patterns generally define a collaboration of objects that is represented in abstract terms by pattern *roles*, or *participants*. In each specific instance of the pattern in a software system, the roles are played by concrete classes from that system. In the majority of the cases, the pattern defines more than one role and the collaboration generally involves more than one class. The collaboration comprises an additional concern whose representation would ideally be localized in its own module, but with traditional OOP approaches is usually scattered throughout the participant classes.

3 Refactoring

Programmers have been performing ad hoc behaviour-preserving transformations for decades, though they did not use this name. Only at the start of 1990s did it become the subject of formal study. The earliest works on refactoring were carried out by Griswold and Notkin (Griswold, 1991, Griswold & Notkin, 1993), which focused on block-structured imperative programs and functional programs and by Opdyke and Johnson

(Opdyke & Johnson, 1990, Opdyke, 1992), which first coined the term *refactoring* and focused on object-oriented frameworks (Opdyke & Johnson, 1990). Some of the early researchers based their work on the assumption that the global restructuring of software systems could not be cost-effective, unless it was automated and separated from other qualitatively different maintenance activities (Griswold, 1991).

```
public class SomeBusinessClass extends OtherBusinessClass {
   ... Core data members
   ... Log stream
   ... Concurrency control lock
   ... Override methods in the base class

   public void someOperation1(<operation parameters>) {
      ... Ensure authorization
      ... Lock the object to ensure thread-safety
      ... Start transaction
      ... Log the start of operation
      ... Perform the core operation
      ... Log the completion of operation
      ... Commit or rollback transaction
      ... Unlock the object
   }
   ... More operations similar to above addressing multiple concerns
}
```

Figure 1: Sketch illustration of code tangling in a Java class as originated by the presence of crosscutting concerns, taken from (Laddad, 2010). Non-core concerns are highlighted in gray background and comprise middleware concerns including logging, authorization checking, transaction support and concurrency control.

3.1 Traditional Object-Oriented Refactoring

Refactoring became mainstream after the publication of Fowler's book on the subject (Fowler, 1999) and the advent of agile methodologies, most notably *extreme programming* (XP) (Beck, 2000). Refactoring is one key component of XP but their advocates did not regard automatic tool support as an essential prerequisite to refactoring. Unit testing, another key component of XP provided a minimum support for safety against regressions, which made manual refactoring feasible. Naturally, tools for automating code transformations were still needed to further enhance productivity and safety. Tool support for refactoring was largely unavailable at the time Fowler published his book. In fact, Fowler's book acted as a manifesto to challenge tool developers to provide support for refactoring. Some influential developers indeed responded positively to Fowler's challenge. As a result, present users of many IDEs provide automated support for many of the source code transformations proposed and documented in Fowler's book.

Fowler's book uses Java as the representative of OOP and modern programming. The book documents a catalogue of *refactorings* and *bad code smells*. Refactorings are descriptions of transformations in source code to produce a specific outcome such as extracting a new method or renaming a class field. Code smells describe undesirable, though frequently occurring, symptoms in source code that warrant the use of refactorings. The code smells described in Fowler's book reflect past experience of recurring patterns of bad design and/or badly

written code, e.g., *Duplicated Code* or *Long Method*. Code smells serve as guidelines for recognizing situations in which refactorings should be used. Fowler's book also associates each code smell to the group of refactorings that can remove the symptoms the smell represents. Together, the two catalogues advocate a specific style for designing and coding OOP systems. A system without code smells ought to be a well-designed system, coded in good style. Other authors also described some of the same symptoms and recommended correcting procedures (Johnson & Foote, 1988, McConnell, 2004).

3.2 Crosscutting Concerns and Traditional Refactoring

The books by Fowler and other authors from the refactoring community (Wake, 2004, Kerievsky, 2004) fail to acknowledge that even a well-designed system cannot be free of CCCs and associated symptoms. Consider the following scenario: given a software system, developers want to be able to modify the implementation of concern A without impacting on the implementation of concern B, for *any* concerns A and B. Such a scenario amounts to *perfect modularity*, which is impossible to achieve in practice and perhaps even in theory. During the life cycle of a typical system, many changes in requirements, assumptions, platforms and environments are likely to arise, and it is impossible to anticipate each and every one of them. An unimportant concern today may morph into a crucial CCC tomorrow and traditional approaches cannot provide guarantees against such scenarios.

Some of the code smells proposed throughout the years can be traced to CCCs, though the connection was never made explicit in the earliest publications on refactoring (Fowler, 1999, Wake, 2004, Kerievsky, 2004). *Shotgun Surgery* (Fowler, 1999) is described as "one change that alters many classes", which corresponds to code scattering. *Divergent Change* (Fowler, 1999) is "one class that suffers many kinds of changes", which corresponds to code tangling. Wake mentions configuration information, logging and persistence as possible causes to the *Shotgun Surgery* smell (Wake, 2004). All these concerns are often mentioned as examples of CCCs. Kerievsky proposes a variant of *Shotgun Surgery* that he calls *Solution Sprawl* (Kerievsky, 2004), claiming 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" (Kerievsky, 2004). Both variants of the smell are promising indicators of the presence of CCCs.

4 Aspect-Oriented Programming

This section provides an overview of the main concepts of AOP, using code examples in AspectJ to illustrate. The aspect-oriented paradigm is currently understood as primarily focusing on software modularity in a systematic way and with an emphasis on the modularization of crosscutting concerns (CCCs) (Rashid & Moreira, 2006). This is a holistic view of aspect-orientation across all phases of the software life cycle. In its earliest years, there were debates on the nature of AOP and what differentiates AOP from other paradigms. Filman and Friedman proposed *quantification* and *obliviousness* as the defining properties of aspect-orientation (Filman & Friedman, 2000). Quantification of a piece of code over a program is the ability to specify a set of separate points in the execution of the program and then execute the piece of code in every point of the set. In other words, quantification is to say, "whenever condition C arises, perform action A". The kinds of quantifications that an aspect-oriented language supports, as well as associated rules, comprise its *joinpoint model*. In turn, the joinpoint model of an AOPL influences its expressiveness. A typical quantification would be "whenever method M is called", but can be something more fine-grained like "whenever field F from class C is

accessed for reading". Obliviousness is the ability to quantify a piece of code over programs that were written by programmers oblivious to this code. We can distinguish between code obliviousness and programmer obliviousness. *Code obliviousness* pertains to cases in which the program's source code was not specifically prepared to be subject to aspect quantifications. *Programmer obliviousness* is a stronger form relating to cases in which programmers who originally wrote the program did not specifically prepare it to the subsequent addition of aspect modules. Thanks to quantification, it becomes theoretically possible to add, change, or circumvent behaviour from a program without changing its source code. Thanks to obliviousness, AOP opens the possibility to reuse code not originally intended to be reused, or reuse it in ways other than those initially intended.

4.1 Dynamic Crosscutting

The majority of implemented mechanisms for supporting quantification are based on the concept of *joinpoint*, as understood in the context of AspectJ. A joinpoint is a well-defined event in the execution of a program, such as the call to a method, the access to an object field, the execution of a constructor, or the throwing of an exception. The execution trace of a program can be approached as a sequence of such events (see **Figure 2**). Some joinpoints are atomic in that no other joinpoint can originate between the beginning of the joinpoint and its conclusion. Examples include joinpoints "field get" and "field set". Other joinpoints have nested joinpoints, e.g., "method call" and "method execution" joinpoints. Method call joinpoints are different from method execution joinpoints, due to polymorphism and dynamic method dispatch: the former are associated to the points where calls are made, the later relate to the instruction blocks that actually execute. Joinpoints are always properly nested: two joinpoints are either disjoint or one is included in the other. One of dimensions through which AOPLs are characterized is its joinpoint model. AOPLs strive to support joinpoints that are relatively robust, i.e., joinpoints that do not break with trivial editing operations on the source code. The kinds of joinpoints supported by a given AOPL comprise an *open set*, in the sense that one can always discover one more kind – devising new, more high-level and robust joinpoints is currently the subject of research.

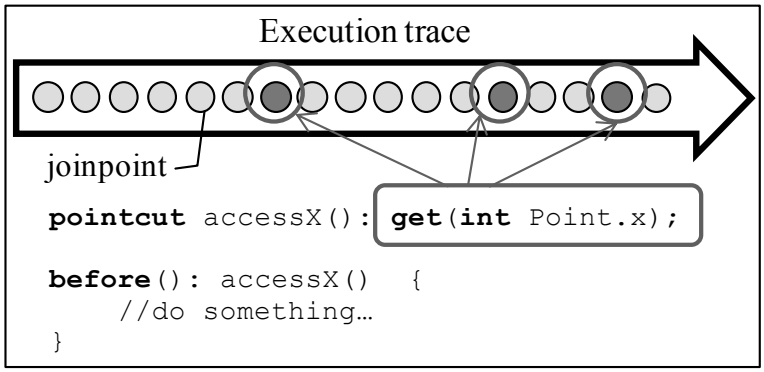


Figure 2: A program's execution trace as a sequence of joinpoints. Each small circle specifies an execution event covered by the AOPL's joinpoint model. The joinpoints that match the definition of a pointcut are highlighted in darker gray. When the program is running, the *before* advice below the pointcut executes just before each of the captured joinpoints.

Joinpoints are supposed to be specified in sets, not in isolation. Therefore AspectJ-like AOPLs support the *pointcut* construct, a declarative clause that specifies sets of joinpoints (see Figure 2). A pointcut can, for instance, specify all calls to public methods, or the execution of methods that belong to a given class and whose name starts with "set". As the places in the source code relating to the specified joinpoints are non-contiguous, the set of captured joinpoints cuts across the structure of system. In addition, many AOPLs provide pointcuts with the ability to capture useful values from the context of the joinpoint, such as method arguments, the reference to the currently executing object, or the target of a method call. Pointcuts can be composed like predicates, using operators &&, || and !, which express (joinpoint) set union, set intersection and set complement respectively. In many AOPLs, certain pointcuts serve to restrict the set of joinpoints captured by other pointcuts, rather than specifying sets of their own. AOPLs have the ability to execute additional behaviour *before*, *after*, and even *instead* of the specified joinpoints. For this reason, joinpoints are often said to be *captured*. In many AOPLs, the construct specifying the additional behaviour is called *advice*. In AspectJ, advices are nameless blocks of code, with the consequence that AspectJ advices are not first-class entities.

The ability of aspects to compose crosscutting behaviour to a given system, e.g., by means of pointcuts and advices, is called *dynamic crosscutting*. The term *aspect* is used to refer to the modular implementation of a concern whose implementation would otherwise cut across multiple modules. In AspectJ, aspects are class-like modules that can contain the aforementioned constructs (plus a few others) and are subject to a special *implicit* instantiation model that avoids the explicit use of new. Other definitions of aspect exist. For instance, the term *aspect* can at a higher conceptual level be used to refer to a concern in software that does not make sense to consider by itself. Middleware frameworks enabled by AOP technology can provide many of their services to POJOs (plain old Java objects) directly, without requiring their classes to follow burdensome rules of a given object model.

A very simple example[1] is next shown to illustrate some of the topics covered above – see **Figure 3**. The example is based on a GUI figure that can be displayed, stretched and moved. The central concept is represented by Java interface FigureElement, with Point and Line as two implementing classes. The left side **Figure 3** outlines the base system and the right side presents an aspect whose purpose is to keep an up-to-date. Class Display manages the display and provides method update to display the current states of all figure elements and that must be called whenever the state on one of the figure elements is changed. The aspect module DisplayUpdating uses pointcut move to capture all calls to the state-changing of Point and Line. This is done through pointcut designator call. The before advice next to the pointcut bound to a second pointcut expression (i.e., an anonymous pointcut) that uses another pointcut designator to restrict the joinpoints specified by move to just those cases in which target of a call is of type FigureElement. When the program is running with the aspect, its advice executes every time one of the these joinpoints is reached.

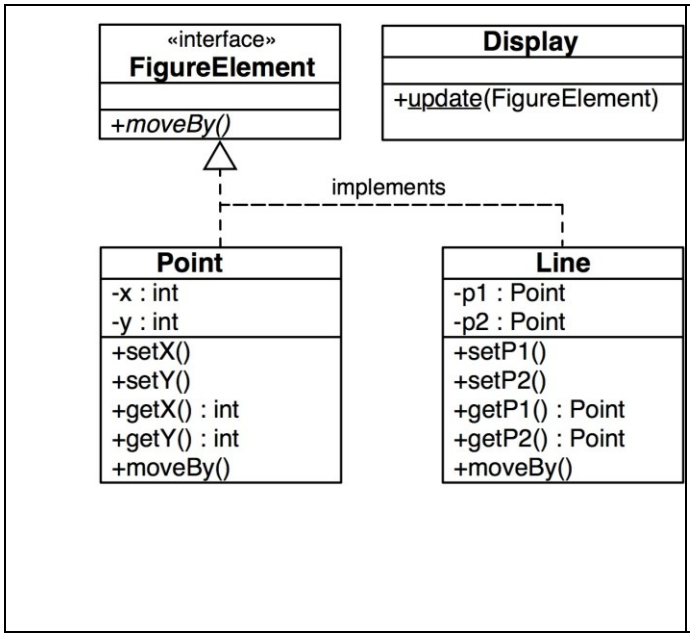
4.2 Static Crosscutting

In addition to dynamic crosscutting, AspectJ has mechanisms to modify the static structure of existing modules, an ability generally called *static crosscutting*. These include the ability to change or extend the existing structure of target classes, by declaring additional fields and methods, or modifying subtype relationships (e.g., by making a class to implement an extra interface). In AspectJ, the primary mechanism for static crosscutting is the *inter-type declaration*, which provides the ability to introduce additional members to a set of target classes. Though inter-type declarations are placed within the aspects, the target classes are the owners of the introduced members. For instance, the inter-type declaration shown next introduces to every instance of interface

[1] This example was inspired by, and remains very similar to, a well-known *Figures* example from the *AspectJ Programming guide* http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

FigureElement an additional field disabled. AspectJ also supports inter-type declarations of complete method definitions.

```
private Display FigureElement.display = displayObject;
```




```
public aspect DisplayUpdate {

  pointcut move():
    call(void Point.setX(int))
||
    call(void Point.setY(int))
||
    call(void Line.setP1(Point))
||
    call(void Line.setP2(Point));

  after(FigureElement f):
  move() && target(f) {
    Display.update(f);
  }

}
```

Figure 3: Illustrating example of the pointcut and advice mechanisms of AspectJ.

The visibility of inter-type members is relative to the aspect, not to the target classes. When an AspectJ aspect declares an inter-type member as private, only code within the aspect can use those members, further reinforcing encapsulation and information hiding. More detailed information about AOP concepts and AspectJ can be found in the literature (Kiczales et al., 1997, Kiczales et al., 2001, Laddad, 2010).

One other feature of AspectJ worth pointing out are declare error and declare warning clauses through which an AspectJ program can configure the compiler to issue new kinds of errors or warnings. These clauses are driven by a given pointcut and the compiler issues an error/warning upon reaching one of the joinpoints specified by the pointcut. To illustrate, suppose we want to enforce the rule that classes from **Figure 3** cannot make calls to use System.out. The following declare error clause can be used to enforce that rule, by making the compiler issue an error for each access to any field of type PrintStream – the type of System.out.

```
declare error: get(java.io.PrintStream *) && within(Capsule):
"Illegal input/output";
```

Among other things, this feature proved very useful in implementing policy enforcements during project developments. Curiously, many new AOPLs inspired by the example of AspectJ do not provide it.

4.3 The Fragile Pointcut Problem

The first generation of pointcut-based languages, which includes AspectJ, selects joinpoints by naming members of target classes from the base system. This has several disadvantages. First, it narrows the scope for

reusability of the aspects. Second, though the system's classes generally remain oblivious to the aspect modules, the aspects themselves are tightly coupled to the base system. The problem and some solutions are next illustrated with the example from **Figure 3**. That figure shows a pointcut to specify all the joinpoints at which a FigureElement moves. Apart from being rather verbose, expressions in such a style of comprehensive joinpoint enumeration are brittle in the face of many reasonable changes to the system classes. For example, adding a void setXY(int, int) method[2] to Point would break the pointcut because calls to the new method wouldn't be captured by the pointcut. To avoid these problems, programmers strive to produce pointcut expressions that focus on the high-level semantics rather than on transitory low-level details. Use of wildcards in pointcut expressions can ameliorate these problems to some extent. The wildcard-based style abstracts pointcut expressions from irrelevant names and exploits patterns in naming conventions, yielding expressions that are robust enough to withstand a number of modifications to the base system, namely some name changes that do not affect some assumed name pattern, or the addition and removal of classes and class members. **Figure 4(a)** shows an example. In this version, the quantification is expressed in terms of the FigureElement interface. The "+" after the interface name ensures that events from all sub-types of FigureElement are captured, which includes implementing classes. The expression set*(..) specifies methods with any number of parameters and whose names start with "set". This way, classes can indeed be added or removed without the need for updating the pointcut. However, this style does not isolate the expression against *all* kinds of changes. For instance, if later some method unrelated to the movement of figures is added to FigureElement, e.g., void setup() or void setting(), call events associated to those methods will be captured as well. An alternative would be to express the intended effect in terms of accesses to fields rather than of calls to methods, as illustrated in **Figure 4(b)**. However, this approach couples the pointcut to private members (as that is the usual visibility of class fields). It also risks capturing accesses to unrelated fields, e.g., some display field. And so on.

a)	`pointcut move(): call(void FigureElement+.set*(..));`
b)	`pointcut move(): set(* FigureElement+.*);`

Figure 4: Illustrating examples wildcard-based pointcuts in AspectJ.

The above problem was first pointed out at length in (Koppen & Störzer, 2004) and is generally referred to as the *fragile pointcut problem*. The root cause is that even carefully designed wildcard-based pointcut expressions do not quite express the high-level intentions of the programmer. In addition, relying on naming conventions yields code dependent on knowledge out of reach of the type checker. This has the consequence that changes on the elements on which the name conventions are based are not signalled by the compiler. In some limited cases, the compiler may be configured to issue a warning, such as when a pointcut captures zero joinpoints – but that may be exactly the right outcome for some cases. The consequences of silence on the part of the compiler are exacerbated by the fact that pointcut-based behaviour is non-local and therefore changes of such behaviour can have system-wide implications. This problem is especially acute when considering long-term system evolution. In general, the aspect programmer needs global system knowledge to assure her pointcuts work as expected, as well as ways to automatically signal unexpected mismatches and corresponding changes in behaviour. Though there are a few more techniques available to the programmer to guard against these problems (cf. section 5.1), long-term evolution of large systems with aspects exerting system-wide effects is impossible to achieve in practice without tool support.

[2] This method is actually used in the *Figures* example from the AspectJ Programming guide.

5 Aspect-Oriented Refactoring

Making refactoring tools be aware of aspects presents significant scientific and technical challenges. At the time of this writing, seven years have passed since the publication of the earliest work on the topic (September 2003: (Hanenberg et al., 2003)). Yet, no fully-fledged support for pointcut-based AOPLs is available in industrial-strength tools. One of the earliest hurdles to overcome was conceptual rather than technical. Early attempts to develop techniques for supporting aspect-oriented refactoring took off at a time when a notion of good style and design for AOP systems was in an incipient state. Yet, tool developers needed clear notions of style and associated refactoring steps as a basis for deciding which code transformations should be supported, which ones should be given priority, and possibly how to enable a tool to assist in selecting a specific refactoring. Prior conceptual work had to be carried out. Research on refactoring of AOP software can be categorized into the following work fronts:

- Developing notions of good style and design for AOP. One of the ways such notions can be expressed is through new refactorings and code smells for AspectJ-like languages.
- Development of tool support for AOP-specific refactoring activities.
- Reports on refactoring experiments involving the aspect-oriented paradigm.

Research on tool support can be further classified into:

- Extraction of CCCs from existing OOP systems to aspect modules and refactoring AOP systems. The two activities are really the same: any process of extracting existing OOP elements into aspects very soon starts creating and adding aspect-specific elements, in which case the resulting system must be approached as a AOP system.
- Making traditional OOP refactorings *aspect-aware*, i.e., providing the tool carrying out the refactoring with the ability to detect changes on the effect of pointcuts and possibly assist in corrective actions.

On a different front from all other works surveyed here, Monteiro and Aguiar cover the early assessment and decision stages: identifying latent aspects in existing systems, knowing when it is feasible to refactor to aspects and what to look for in an assessment of the prerequisites for carrying out a refactoring process (Monteiro & Aguiar, 2007).

Before discussing the various work fronts outlined above, the following additional concepts should be introduced:

- *Aspect mining* is a research topic focusing on tools and techniques for the automatic or semi-automatic detection of latent aspects in existing non-AOP systems. This topic is out of scope of this chapter, but it is mentioned a number of times in the survey that follows.
- *Heterogeneous and homogeneous concerns* are both characterized by broad scattering. The distinction is that with homogeneous CCCs, the scattered logic is identical or very similar in each location (e.g., “boiler-plate code”) while in the case of heterogeneous CCCs exhibit different code in each location.

5.1 Aspect-oriented Refactorings and Code Smells

It is important to note this section aims at providing a short *survey*. Unfortunately, the substance of these publications – notions of style for a new programming paradigm – is not an easy topic to summarize in a short

space. Readers interested in gaining a deeper knowledge on this topic are referred to the various publications cited. This particularly applies to the minute details of the many refactorings mentioned, which here are mentioned by their name alone. It should also be noted that there is some overlap among the refactorings proposed by various authors. As pointed out in (Hannemann, 2006), each group of authors tends to "reinvent the wheel" and does not use refactorings by others when proposing theirs. For instance, each group proposes their own variant of an *Extract Advice* refactoring. Hannemann points out a number of such overlaps (Hannemann, 2006).

The two earliest publications on the topic of aspect-oriented refactoring call into attention the fragile pointcut problem and illustrate it with some code examples (Hanenberg et al., 2003, Iwamoto & Zhao, 2003). They also focus partially or totally on proposing new refactorings for AOP systems or similar techniques. All authors agree that refactorings tools for OOP systems cannot be applied to systems containing pointcut-based aspect modules. Both publications also announce ongoing work on tools providing some of the functionality deemed desirable, but no subsequent developments were announced regarding such tools. Hanenberg et al. propose the notion of aspect-aware refactorings and propose a set of enabling conditions to preserve the observable behaviour. They point out that an aspect-aware tool must automatically verify these conditions, as manual verification is an exhausting task, even in small systems. In their analysis of the fragile pointcut problem, Iwamoto and Zhao present an analysis of all refactorings proposed by Fowler in light of the possible impact they can have on aspects. They conclude that only three can be safely applied without the need for checking possible effects on aspect behaviour. The reason is that those are the sole refactorings that involve elements out of reach of AspectJ's pointcut protocol. Iwamoto and Zhao also announce their intention to build a catalogue of AO refactorings and present a list of 24 refactorings, but just the names of the refactorings are provided.

Laddad is strictly focused on refactoring techniques performed manually but provides a particularly rich contribution in this front (Laddad, 2003). Laddad presents a collection of refactorings 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 Method Calls*, *Extract Interface Implementation*, and *Replace Override with Advice*, but also concern-specific refactorings such as *Extract Concurrency Control* and *Extract Contract Enforcement*. Other refactorings fit into the larger, pattern-related category of refactoring proposed by Kerievsky – *Extract Worker Object Creation* and *Replace Argument Trickle by Wormhole* (Kerievsky, 2004). These two refactorings are based on two of the design patterns – *Worker Object* and *Wormhole* respectively – described by Laddad in his book (Laddad, 2010). Kerievsky also proposes the idea of a refactoring that goes *toward* a pattern rather than yielding a full-fledged implementation of the pattern at one go (Kerievsky, 2004). Laddad proposes a refactoring in that vein: using *Extract Exception Handling* goes towards an instance of the *Exception Introduction* design pattern (Laddad, 2010). One of the more general refactorings proposed by Laddad – *Extract Method Calls* – is the subject of an analysis in light of the issues that can arise in the task of automating it (Kellens & Gybels, 2005).

Some of Laddad's recommendations are largely influenced by the fragile pointcut problem. Laddad recommends that aspects be put as close to the target code as possible. In most cases, this amounts to deploying them as nested, inner aspects within a class. This makes it more likely that the programmer will examine the aspects when the base code changes. Another recommendation is that new 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. In addition, Laddad provides several guidelines to programmers 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, wildcard-based pointcut. Laddad also proposes an idiom based on the declare error mechanism to make the

compiler flag cases when the two versions capture different sets of joinpoints. Given two pointcuts p1 and p2, the following expression issues an error in such cases:

```
declare error: (!p1() && p2()) || (p1() && !p2()) : "Mismatch in join points captured";
```

Tonella and Ceccato base their work on the assumption that Java interfaces are often, though not always, related to concerns other than the one pertaining to the system's main decomposition (Tonella & Ceccato, 2004). 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. The authors also point out various issues that can arise in a typical extraction of an interface implementation into an aspect.

Deursen et al. provide a short overview of the state of art of aspect mining and refactoring (Deursen et al., 2003). Although their main concern seems to be aspect mining, they also mention several open questions about refactoring to aspects, including the issues of how existing code smells can be used to identify candidate aspects and how can the introduction of aspects be described in terms of new refactoring catalogues. Monteiro and Fernandes contribute to answering the questions posed by Deursen et al (Monteiro & Fernandes, 2006a). They are the first authors to tackle the topic of aspect-oriented refactoring specifically from the view of programming style. They argue that the advent of AOP makes some practices of OOP obsolete, which should be approached as bad-style AOP and argue that notions of good style for AOP are needed by both programmers and tool developers, for different reasons. They point out that both refactorings *and* smells play a part in expressing notions of good style. Until then, publications by other authors were proposing new refactorings or presenting work on tool support, but none was complementing them with code smells. They propose a catalogue of 27 refactorings plus three code smells, presented using the same structure used by Fowler (Fowler, 1999) and Kerievsky (Kerievsky, 2004), including source code examples. As a consequence, these refactorings are the ones more thoroughly documented.

The catalogue proposed by Monteiro and Fernandes deals with the extraction of CCCs into aspects and subsequent tidying up of the structure of the resulting aspects. This is expressed by a refactoring strategy that provides a framework for using the refactorings from the catalogue and divides the refactoring process into three phases. The refactorings from the catalogue are accordingly organized in three groups, each focusing on one of the phases and headed by a higher-level, composite refactoring serving as a framework. The composite refactorings and the refactoring phases they represent are as follows:

1. *Extract Feature into Aspect*: extraction of CCCs from a given Java system. All elements related to a CCC are moved to a single aspect module.
2. *Tidy Up Internal Aspect Structure*: the internal structure of the resulting aspect is improved, by removing duplication and performing various tidying up operations. According to the strategy, this phase should be carried out after completing the first phase to take advantage of the newly-gained modularization. The tidying up steps are easier to perform after all related elements are localized within a single module.
3. *Extract Superaspect* deals with the cases in which several different but related aspect modules were extracted and duplication is detected among them. In that case, a generalisation phase is performed in which commonalities are pulled up into some super-aspect that may be reusable in some cases.

Monteiro and Fernandes provide a detailed description of a complete refactoring process through which most of the refactorings from their proposal are used (Monteiro & Fernandes, 2008). Of the three smells proposed, two apply to OOP code and one is specific to AOP systems. It is also explained how to remove the smells by using the refactorings from the catalogue. The two smells for OOP are *Double Personality*, which expresses the notion that classes should have just one core set of responsibilities (i.e., a core concern), and

Abstract Classes, which expresses the notion that implementation code in abstract classes should be moved to aspects, leaving room to turn the abstract class into a Java interface. Subclasses become free to inherit from some other class and the new interface can still acquire a default, concrete implementation thanks to the ability of AspectJ to emulate mixin composition. In addition, Monteiro and Fernandes analyse the refactorings proposed in prior publications (mentioned above) and propose corresponding code smells. In another publication (Monteiro & Fernandes, 2006b), the same authors argue that the use of a number of design patterns (Gamma et al., 1995) becomes likewise obsolete in light of AOP and suggests further work for deriving new refactorings through which pattern instances are replaced with aspects. Some suggestions for a few refactorings are provided, mentioning a name and short outline.

Malta and Valente propose a number of OOP refactorings to complement the aforementioned collections (Malta & Valente, 2009). They note that a number of OOP refactorings need to be performed on existing OOP systems to adequately prepare them for the extraction of CCCs. Many CCCs take the form of a pattern in code that appears in multiple points of the system but often such patterns are placed in points not covered by the joinpoint models of current AOPLs and present irregularities among the various points, e.g., in the relative order of statements or in the position of a fragment within a method's body (sometimes in the beginning, sometimes in the middle or in the end). The refactorings proposed by Malta and Valente aim to place such code patterns within the reach of the AOPL's pointcuts and to produce the required regularity. The refactorings proposed are divided into groups for *statement reordering* and *method extraction*. Malta et al. complement that work with a set of guidelines to assist developers in the application of the refactorings (Malta et al., 2009). They also announce a prototype tool (TransformationMapper) providing partial support to developers in applying the guidelines.

Though it uses neither refactorings nor code smells as a means of expression, the following work deserves to be mentioned. Griswold et al. propose a design rule in which an aspect is used to declare a special interface based on pointcuts, to create a layer between aspect modules and the concrete classes of the base system (Griswold et al., 2006). This kind of interface is named *crosscut programming interface* (XPI) and its purpose is to provide a stable contract on the basis of which both base programmers and aspect programmers can work independently. The idea of XPIs results from a study that showed that although aspects can very aggressively hide the details of their implementations to the remaining modules of a system, this encapsulation is one-sided (Sullivan et al., 2005). In many cases, the aspects themselves were extremely exposed to minute details of the remaining modules in the system.

Apart from issues related to pointcut coding techniques, the refactoring space for AspectJ seems to have been explored to a reasonable extent already. The current body of knowledge can be used as a basis for developing refactoring catalogues tailored for other AOPLs.

5.2 Tool Support for Aspect-Oriented Refactoring

The earliest works on tool support (Hanenberg et al., 2003, Hannemann et al., 2003, Hannemann et al., 2005, Binkley et al., 2005) are focused on the extraction of CCCs from existing Java systems. In common to many approaches on tool support is the recognition that the crosscutting nature of the concerns to be extracted places requirements that are hard to meet adequately when using the incremental approaches generally favoured in traditional OOP refactorings. Often, these comprise applying multiple transformation steps requiring knowledge of the system that is either local or clearly circumscribed. By contrast, the process of migrating OOP systems to AOP is more knowledge-intensive and any refactoring tool needs to involve the developer in a workflow. This also applies to the aspect aware versions of OOP refactorings: a given refactoring can affect multiple, scattered points in a system and the developer may need to deal with each case individually (Hanenberg et al., 2003, Hannemann et al., 2003, Wloka et al., 2008).

Two approaches are focused on CCC extraction though none tackles the issue of the impact of the refactorings on existing pointcuts (Hannemann et al., 2005, Binkley et al., 2006). Hannemann et al. proposes an approach conceptually based on the notion of role as used in design patterns (Hannemann et al., 2005, Hannemann et al., 2003). The approach is likewise tailored for the migration of the implementation of a design pattern to an equivalent implementation in AspectJ, though it seems capable of being applied to other, more general cases of CCCs. It is based on a role-based abstract description of the target crosscutting structure, used to differentiate between individual concrete program elements. To perform a concrete refactoring, those elements must be mapped to the abstractions for the description at hand. One such abstract description must be developed for each pattern or crosscutting structure. The tool drives a workflow whose main purpose is to perform the mapping, comprising the following steps:

- *Selection*: a developer selects a refactoring from the collection provided by the tool.
- *Mapping*: a mapping of program elements to pattern roles is performed.
- *Configuration*: static analysis is used to identify possible alternative decisions in the refactoring process. A number of issues may arise in this phase (e.g., name conflicts), in which the developer must make decisions for each case presented.
- *Execution*: the tool performs the sequence of transformations on the basis of the abstract representation, configured and concretized with the developer's inputs and decisions. Possible transformations include moving methods from classes to aspects, creation of new program elements and removal of existing elements. Note that this can have a direct impact on the program's interface.

Binkley et al. present an approach that similarly extracts a CCC into an aspect module and provide a tool called AOP-Migrator that implements the approach (Binkley et al., 2006). Unlike the approach by Hannemann et al., AOP-Migrator also performs aspect mining, using it to mark code regions that represent instances of the candidate elements to be extracted. Supported regions include whole methods, method calls and sequences of statements (i.e., blocks of code). The tool supports a small number of semi-automated refactorings that extract the marked fragments, which can be automatically migrated to aspects yielding a semantically equivalent AOP implementation of the program. As with the above approach, the tool drives a workflow, comprising a loop iterating over all the marked portions of code. In turn, each marked segment of code is analyzed in isolation and moved to an aspect after preparations made with the involvement of the developer. The iteration goes on until no more marked statements remain in the base code. The loop comprises the following steps:

- *Discovery*: the tool determines the applicable refactorings for a marked code fragment.
- *Enabling transformation*: In case a selected fragment cannot be extracted through existing refactorings, OOP refactorings transformations are suggested (e.g., *Extract Method*) to make an extraction into aspects generally feasible.
- *Selection*: the selection of an appropriate refactoring is guided through a pre-set prioritization scheme that the developer can accept or override.
- *Execution*: Fully automated generation of aspect template code, move of marked fragments to the aspect and creation of necessary pointcuts and advices.

For every extracted program element it generates a single pointcut bound to a single advice which contains the extracted element. However, AOP-Migrator also supports AOP-specific refactorings named "pointcut abstraction" that aim at improving the quality of the automatically generated pointcut code, by unifying definitions shared by different generated elements.

Wloka et al. tackle a different work front than the one above, focusing on automating the detection of change effects on pointcuts and the generation of pointcut updates, i.e., supporting aspect aware refactorings (Wloka et al., 2008). This approach is based on a model for representing pointcuts, explaining the change impact analysis and introducing an approach for automating pointcut update decisions. A tool based on this model is presented – named SoothSayer – that performs program analysis to detect affected or broken pointcuts, assess ways to derive suitable pointcut adjustments and determine when developer feedback must be provided. Since all approaches have to cope with changes to the base system that may affect existing pointcuts, this work can be seen as complementing those previously mentioned.

Anbalagan and Xie propose an automated approach for mining aspects and inferring pointcuts from legacy Java applications and present a prototype implementation (Anbalagan & Xie, 2007). The tool automatically identifies aspects and their joinpoints, clusters the join points based on common characteristics in the method names, using lexical matching, and infers a pointcut expression for each cluster of joinpoints. An additional testing mechanism checks that the inferred pointcut expressions are of correct strength. This work is differs from that by Wloka et al. in that it does not consider the maintenance of the pointcut when the base system undergoes some evolution. Piveta at al discuss representations of refactoring opportunities for AOP software suitable for automation tools and propose a set of representation mechanisms to provide a precise way to describe refactoring conditions, the association of conditions with refactorings and how to perform searches for refactoring opportunities (Piveta et al., 2009).

5.3 Refactoring Experiments involving the Aspect-Oriented Paradigm

This sub-section covers case studies and refactoring experiments of various natures involving AOP. Note that the work surveyed in the previous sections also generally includes experiments but there the focus is not on the experiments themselves.

Middleware seems to be the "killer application" of AOP and middleware systems feature prominently in refactoring experiments. Colyer and Clement describe an experiment in using AspectJ to refactor both homogeneous and heterogeneous concerns in a industrial middleware product-line platform (Colyer & Clement, 2004). They report having significantly more difficulty with heterogeneous concerns than with homogeneous concerns. Zhang and Jacobsen describe an analysis based on aspect mining of a CORBA-based middleware system that suggests that middleware architectures inherently suffer from having coordinate CCCs (Zhang & Jacobsen, 2003). They also report on the use of AspectJ to modularize system's CCCs. A subsequent quantitative analysis based on software engineering metrics suggests that AOP lowers the complexity of the architecture, increases modularity and preserves performance.

Marin et al. propose a general strategy to reengineer legacy OOP systems into aspect-oriented versions of those systems (Marin et al., 2009). The various phases of the strategy are driven by a classification of CCCs into *crosscutting concern sorts* proposed in an earlier work (Marin et al., 2005). Sorts are CCCs described by their specific symptoms (i.e., implementation idiom in a non-AOP language), and a (desired) aspect mechanism to modularize a sort instances with an AOP solution. The strategy covers the following phases:

- *Identification of the primary CCCs*, using aspect mining techniques.
- *Concern exploration*, in which the outputs from the previous phase are used to acquire a comprehensive identification of the implementation elements of the identified CCCs.
- *Query-based concern modelling and documentation*, in which various modelling tools are used to obtain coherent representations of the CCCs under analysis. A classification of the CCCs into sorts is also performed in this phase.

- *Refactoring* of the OOP idioms modelled and documented in the previous phases into equivalent AOP implementations.

Marin et al. mention a number of tools to assist some of the phases but what comes to the fore is the strategy, which was tested with the JHotDraw open source project, yielding a version of JHotDraw in which a few CCCs were migrated to AspectJ aspects. That version was made publicly available as an open-source project called AJHotDraw (sourceforge.net/projects/ajhotdraw). However, the strategy was applied to JHotDraw without the tools.

Kulesza et al. report on the use of refactorings from the catalogue by Monteiro and Fernandes to aspectize the JUnit framework (Kulesza et al., 2005, Monteiro & Fernandes, 2006a). Benn et al. report on an experiment in refactoring of a medium-scale distributed system to AspectJ. A suite of metrics for internal quality characteristics of the code is used to perform an assessment of the results, which suggests that the refactoring improved the internal quality of the subject system (Benn et al., 2005).

6 Summary

This chapter presents a survey of topic of refactoring aspect-oriented software. The survey divides the topic in several research fronts that include development of refactorings and code smells, refactoring experiments and approaches for automating the refactorings.

Catalogues of refactorings and code smells contribute to express and mature notions of good style and design. That front is steadily advancing but existing work is almost exclusively focused on AspectJ. For instance, Object Teams is currently the primary candidate to become the second industrial-strength aspect-oriented Java extension and yet, we have no knowledge of a published collection of refactorings for Object Teams at this point. On the automation and tool support front, much remains to be done. The major Achilles' heel of these languages remains the fragile pointcut problem. Pointcut technology, including the AspectJ variety, continues to feature prominently in the latest editions of the primary conference focused on the topic (AOSD; www.aosd.net), covering topics such as higher-level pointcut protocols less tied to the low-level lexical components of a given system. Aspect-oriented systems devoid of explicit pointcuts are the exception rather than the rule. Indeed, the marked difference between Object Teams with respect to AspectJ was used as an argument in Object Teams successful candidacy to become an Eclipse project (www.eclipse.org/proposals/object-teams). Naturally, one obvious route is not to provide wildcard-based pointcuts to the language. That is the case of Object Teams (OT/J), whose model is based on multiple dimensions of polymorphism and does not pose technical hurdles to implementing refactoring support. Its development environment has been supporting refactoring for years.

Acknowledgement

Work partially supported by FCT, under project AMADEUS (POCTI, PTDC/EIA/70271/2006).

References

Anbalagan, P. & Xie, T. (2007). Automated Inference of Pointcuts in Aspect-Oriented Refactoring. Proceedings of the 29th International Conference on Software Engineering (pp. 127-136).

(AspectJ) AspectJ project home page www.eclipse.org/aspectj/

Extreme Programming Explained: Embrace Change. Addison-Wesley.

Benn, J., Constantinides, C., Padda, H., Pedersen, K., Rioux, F. & Ye, X. (2005). Reasoning on Software Quality Improvement with Aspect-Oriented Refactoring: A Case Study. *Proceedings of the IASTED International Conference on Software Engineering and Applications* (pp. 309-315).

Brichau, J. & Haupt, M. (2005). M. Report describing survey of aspect languages and models. AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01.

Binkley, D., Ceccato, M., Harman, M., Ricca, F. & Tonella, P. (2006). Tool-supported Refactoring of Existing Object-Oriented Code into Aspects. *IEEE Transactions on Software Engineering, vol. 32(9)* (pp.698-717).

Colyer, A. & Clement, A. (2004). Large-scale AOSD for middleware. *Proceedings of the 3rd international conference on Aspect-Oriented Software Development* (pp.56-65).

Deursen, A.v., Marin, M. & Moonen, L. (2003). Aspect Mining and Refactoring. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03).

Filman, R. E. & Friedman D. P. (2000). Aspect-Oriented Programming is Quantification and Obliviousness. *Workshop on Advanced Separation of Concerns at OOPSLA 2000*.

Fowler, M. (with contributions by Beck, K., Opdyke, W., Roberts, D.) (1999). Refactoring – Improving the Design of Existing Code. Addison Wesley.

Gamma, E.; Helm R., Johnson R. & Vlissides J. (1995). Design Patterns, Elements of Reusable Object-Oriented Software. *Addison Wesley*.

Griswold, W. G. (1991). Program restructuring as an aid to software maintenance. *PhD thesis, University of Washington, USA*.

Griswold, W. G. & Notkin D. (1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology, 2(3)* (pp. 228–269).

Griswold, W., Sullivan, K., Song, Y, Shonle, M., Tewari, N., Cai, Y. & Rajan, H. (2006). Modular software design with crosscutting interfaces. IEEE Software, vol.23 (1) (pp. 51–60).

Hanenberg, S., Oberschulte, C. & Unland, R. (2003). Refactoring of Aspect-Oriented Software. Proceedings of NetObjectDays, Vol. 2591 of Lecture Notes in Computer Science, Springer Verlag.

Hannemann, J., Thomas, F. & Murphy, G. (2003). Refactoring to aspects: an interactive approach. *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (pp. 74-78).

Hannemann, J., Murphy, G., & Kiczales G. (2005). Role-Based Refactoring of Crosscutting Concerns. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, ACM press (pp. 135-146).

Hannemann, J. (2006). Aspect-Oriented Refactoring: Classification and Challenges. *Workshop on Linking Aspect Technology and Evolution revisited (LATEr) at AOSD 2006*.

Iwamoto, M. & Zhao J. (2003). Refactoring Aspect-Oriented Programs. *Proceedings of 4th AOSD Modeling With UML Workshop at UML'2003*.

Johnson, R. E. & Foote B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming*, (pp.22-35).

Kellens, A. & Gybels, K. (2005). Issues in Performing and Automating the 'Extract Method Calls' Refactoring. *Software Engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD 2005*.

Kerievsky, J. (2004). Refactoring to Patterns. *Addison-Wesley*.

Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. & Irwin J. (1997). Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*. Vol. 1241 of Lecture Notes in Computer Science, Springer Verlag (pp. 220-242).

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. (2001). An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming*. Vol. 2072 of Lecture Notes in Computer Science, Springer Verlag (pp. 327-353).

Koppen, C. & Störzer M. (2004). PCDiff: Attacking the Fragile Pointcut Problem. Interactive Workshop on Aspects in Software (EIWAS).

Kulesza U., Sant'Anna, C. & Lucena, C. (2005). Refactoring the JUnit Framework using Aspect-Oriented Programming. *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 136-137).

Laddad, R. (2003). Aspect-Oriented Refactoring, parts 1 and 2. The Server Side.
www.theserverside.com/news/1366873/Part-1-Overview-and-Process
www.theserverside.com/news/1365184/Part-2-The-Techniques-of-the-Trade

Laddad, R. (2010). AspectJ in Action, 2nd edition. *Manning Publications*.

Malta, M. & Valente, M. (2009). Object-oriented transformations for extracting aspects. *Information and Software Technology, vol. 51* (pp. 138–149).

Malta, M., Oliveira, S. & Valente, M. (2009). Guidelines for Enabling the Extraction of Aspects from Existing Object-Oriented Code. *Journal of bject Technology, Vol. 8, No. 3* (pp. 101–119).

Marin, M., Moonen, L. & Deursen, A.v. (2005). A classification of crosscutting concerns. *Proceedings of the 21st International Conference on Software Maintenance*, (pp. 673-677).

Marin, M., Moonen, L. & Deursen, A.v. (2009). An Integrated Crosscutting Concern Migration Strategy and its Semi-Automated Application to JHotDraw. *Automated Software Engineering, vol. 16 (1), Springer* (pp. 323-356).

McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction, 2nd edition. *Microsoft Press*.

Monteiro, M. P. & Fernandes, J. M. (2006). Towards a Catalogue of Refactorings and Code Smells for AspectJ. *Vol. 3880 of Lecture Notes in Computer Science, Springer Verlag* (pp. 214-258).

Monteiro, M.P. & Fernandes, J. M. (2006). Using Design Patterns as Indicators of Refactoring Opportunities (to Aspects). Workshop on Linking Aspect Technology and Evolution revisited (LATEr) at AOSD 2006.

Monteiro, M.P. & Fernandes, J. M. (2008). An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms. *Software: Practice and Experience vol.38 (4), John Wiley & Sons* (pp. 361-396).

Opdyke, W. F. (1992). Refactoring Object-Oriented Frameworks. *Ph.D. Thesis, University of Illinois at Urbana-Champaign, USA*.

Opdyke, W., Johnson, R (1990). Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*.

(OT/J) Object Teams home page.eclipse.org/objectteams/ (see also the original home page: www.objectteams.org/)

Parnas D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15 (12)* (pp. 1053-1059).

Piveta, E., Pimenta, M., Araújo, J., Moreira, A., Guerreiro, P. & Price, T. (2009). Representing Refactoring Opportunities. *Proceedings of the 2009 ACM symposium on Applied Computing* (pp. 1867-1872).

Rashid, A. & Moreira, A. (2006). Domain Models Are NOT Aspect Free. *Proceedings of the 9th Internation Conference on Model-Driven Engineering Languages and Systems* (pp. 155-169).

Sullivan, K., Griswold, Song, Y., W., Cai, Y., Shonle, M., Tewari, N. & Rajan, H. (2005). Information hiding interfaces for aspect-oriented design. *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 166-175)

Tarr, P., Ossher, H., Harrison, W. & Sutton Jr., S.M. (1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the 21st International Conference on Software Engineering* (pp. 107-119).

Tonella P. & Ceccato M. (2004). Migrating Interface Implementation to Aspects. *Proceedings of 20th IEEE International Conference on Software Maintenance* (pp. 220-229).

Wake, W. (2004). Refactoring Workbook. *Addison Wesley*.

Wloka, J., Hirschfeld, R. & Hänsel, J. (2008). Tool-supported refactoring of aspect-oriented programs. *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, ACM press (pp. 132-143).

Zhang, C. & Jacobsen, H.-A. (2003). Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems, vol.14(11)* (pp. 1058-1073).