

Patterns for Refactoring to Aspects: An Incipient Pattern Language

Authors Miguel P. Monteiro, Ademar Aguiar
FCT, Universidade Nova de Lisboa
INESC Porto, Faculdade de Engenharia, Universidade do Porto
E-mail: mmonteiro@di.fct.unl.pt, adem.ar.aguiar@fe.up.pt

Aspect-Oriented Programming is an emergent programming paradigm providing novel constructs that eliminate code scattering and tangling by modularizing crosscutting concerns in their own aspect modules. Many current aspect-oriented languages are backwards compatible extensions to existing, popular languages, which opens the way to aspectize systems written in those languages. This paper contributes with the beginnings of a pattern language for refactoring existing systems into aspect-oriented versions of those systems. The pattern language covers the early assessment and decision stages: identifying latent aspects in existing systems, knowing when it is feasible to refactor to aspects and assessment of the necessary pre-requisites for the refactoring process.

Introduction When developing modern complex software, good design and coding style are necessary but not sufficient prerequisites for yielding optimal separation of concerns. Modern software often includes concerns that traditional modularization mechanisms of object-oriented (OO) languages such as Java cannot modularize. Such concerns are usually called *crosscutting concerns* (CCCs) [7]. Examples include systems that use the services provided by middleware and the implementation of some well-known design patterns [12]. The source code related to CCCs takes the form of multiple, duplicated code fragments that are scattered throughout the modules of the system (e.g., methods, classes and packages), a phenomenon known as *code scattering* [19]. In addition, CCCs give rise to *code tangling*, i.e., the scattered code fragments tend to be intertwined with the code related to the primary functionality of the system, harming the comprehensibility and ease of evolution of all concerns present in the affected modules.

Aspect-Oriented Programming (AOP) [19] is an emergent programming paradigm providing novel constructs that are capable of eliminating code scattering and tangling by modularizing CCCs in their own modules – called *aspects* [19]. Currently, many aspect-oriented languages are backwards compatible extensions to existing languages. Of those, the most mature is AspectJ [20][21][8], an extension to Java. Many design dimensions of many of the more recent AOP tools betray a strong influence from AspectJ. In addition to programming languages, there are other kinds of tools, namely frameworks for middleware services [18] that use AOP technology. Many of these tools use plain Java and compose their services by way of XML files and Java 5 annotations.

The availability of aspect-oriented extensions to existing languages opens the way to refactor existing systems into aspect-oriented versions of those systems. This paper contributes with the beginnings of a pattern language for refactoring [3][11] existing OO systems into AOP. To this purpose, the paper proposes three patterns (DETECT CROSSCUTTING CONCERNS, DECIDE TO REFACTOR TO ASPECTS, and REFACTOR TOWARDS ASPECT-FRIENDLY CODE BASE) that are intended to focus on the initial issues that arise when considering the option to refactor an existing OO system to AOP.

Pattern language The pattern language comprises a set of interdependent patterns that aim to help people developing and/or maintaining software systems become aware of the problems they will typically face when considering the possibility to use AOP in the future evolution of their systems. The patterns originate from reading the existing literature, experience gained by the authors and ongoing experiments.

Overview of the patterns To describe the patterns, we use the tried-and-tested format of *Name-Context-Problem-Forces-Solution-Examples*. Prior to describing the three patterns documented in this paper we start by presenting an overview of the envisioned pattern language by summarizing the intent of each pattern (see also Figure 1). Note that what follows is a conservative estimation of the patterns, as it is likely that more patterns will emerge from the ongoing process of characterizing them.

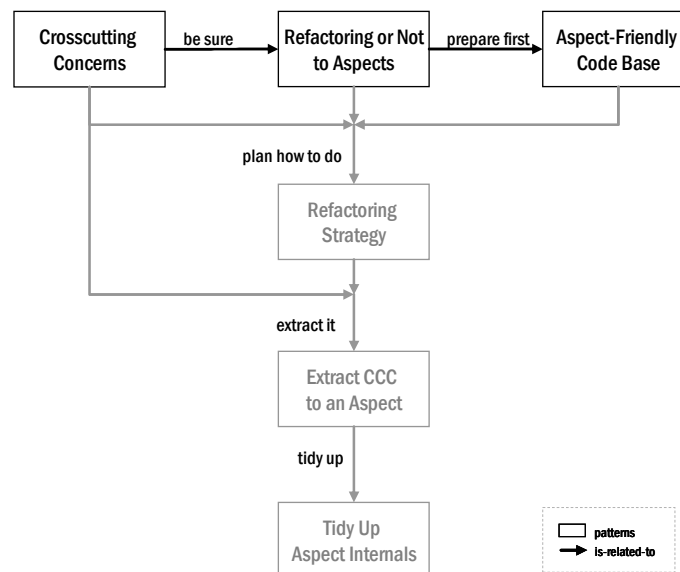


Figure 1 Relationships between the patterns.

Detect Crosscutting Concerns (CCC) helps developers on diagnosing the presence of CCCs in her system, by describing the symptoms and characteristics in source code that can serve as indicators for the developer.

Decide to refactor to aspects helps developers to make an informed decision about to use or not AOP refactoring to extract aspects identified through DETECT CROSSCUTTING CONCERNS, on the basis of the rough category of the detected CCC and the capabilities of the available AOL. It calls into attention some situations where it is advisable to avoid such a course of action.

Refactor Towards Aspect-Friendly Code Base helps developers to decide if they should first perform some preparatory, traditional OO refactorings or if they can jump straight into AOP refactoring – by using `REFACTORING STRATEGY` or `EXTRACTING CCC TO AN ASPECT`.

Refactoring Strategy helps developers to plan the refactoring process, by pointing out the most typical phases and by providing information about each phase. `REFACTORING STRATEGY` is motivated by the insight that modularity is a prerequisite for performing certain kinds of code transformations, namely those that target AOP specific constructs that compose aspect functionality to multiple modules. In addition, it is significantly easier (or possible at all) to ensure that the transformations are behavior preserving when they focus on the implementation elements of a single module. Thus, `REFACTORING STRATEGY` proposes that priority be given to extraction to all elements of the target CCC to a new aspect module – using `EXTRACT CCC TO AN ASPECT` – before focusing on improving the internal structure of the extracted aspect, which is the purpose of `TIDY UP ASPECT INTERNALS`.

Extracting CCC to an Aspect. It is important to keep in mind that the first phase of refactoring to AOP is always one of *extraction*, i.e., moving all elements of the target CCC to a new aspect module. `EXTRACT CCC TO AN ASPECT` focuses on the specific details of the aspect extraction process and provides tips on what should be the order with which code fragments and class members should be moved. The stress is on *safety*, i.e., on minimizing the chances that existing behavior is broken during the process. Thus, `EXTRACT CCC TO AN ASPECT` follows the principles and spirit advocated in Fowler's book [11]. The refactoring process proposed by `EXTRACT CCC TO AN ASPECT` was first proposed in [30]. A detailed example is described in [28].

Tidy Up Aspect Internals gives tips to deal with potential inadequacies in the internal structure of the aspect obtained through `EXTRACTING CCC TO AN ASPECT`. `TIDY UP ASPECT INTERNALS` is motivated by the insight that the internal structure of extracted aspects still corresponds to the original design of the CCC. Such designs tend to be strongly influenced by the presence of the scattering and tangling phenomena and therefore may no longer make sense when the CCC is modularized. Thus, `TIDY UP ASPECT INTERNALS` calls into attention some symptoms that indicate that a restructuring of the new module is desirable and provides tips on what such restructurings should strive for.

Concepts of AOP In this section, we provide an overview of the main concepts of AOP. Code examples in AspectJ are used to illustrate.

Joinpoint. 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 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).

pointcut that restricts² captured calls to those that originate within the lexical boundary of class Capsule:

```
public pointcut callsFromCapsule2SystemOutPrints() :
    call(public void java.io.PrintStream.print*(..)) &&
    within(Capsule);
```

Advice. AOLs have the ability to execute additional behavior before, after, and, in some cases, instead of the captured joinpoints. In many AOLs, the blocks of code³ that specify the additional behavior are called an *advice*. In AspectJ, around advice execute instead of the captured joinpoint. Around advice can execute the original joinpoint by means of a call to `proceed()`. Next follows the example of a piece of around advice that executes instead of each method call captured:

```
void around(): allCalls2SystemOutPrints() {
    System.out.println("message printed.");
}
```

To illustrate how context captured from the joinpoints can be used, we next show a pointcut similar to the first one but that also captures the argument to the print method. In doing so, it also restricts the set of captured joinpoints to those calls that receive one argument of the specified type.

```
public pointcut messagesFromSystemOutPrint(Object message) :
    allCalls2SystemOutPrints() && args(message);
```

The advice shown next adds square brackets to the beginning and end of the messages sent to the console:

```
void around(Object message) : msgsFromSystemOutPrint(message) {
    String newMessage = "[" + message.toString() + "];";
    proceed(newMessage);
}
```

Dynamic crosscutting. This is the name given to the ability of aspects to compose crosscutting behavior to a given system, by means of pointcuts and advices.

Static crosscutting. Many AOLs also have 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). These structural mechanisms are called *static crosscutting*. In AspectJ, static crosscutting is mostly supported by *inter-type declarations*, which provide aspects with the ability to introduce additional state and behavior to a set of target classes. Though the 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 class Server an additional field `disabled`, of type boolean, initialized to false. Similar declarations can be made of methods.

² Several of the AspectJ pointcuts serve to restrict the set of joinpoints captured by other pointcuts, rather than specifying sets of their own. It is the case of `within()`, which restricts joinpoints to those originating from the lexical boundaries of the specified

³ In AspectJ, advice blocks of code are nameless and do not comprise first-class entities.

```
private boolean Server.disabled = false;
```

The visibility of inter-type members is relative to the aspect, not to target classes. When an AspectJ aspect declares an inter-type member as private, only code within the aspect can use those members, further reinforcing modularity. More detailed information about AOP concepts and AspectJ can be found in the literature, namely in [19][20][21][8].

Weaving. This is the name given to the phase during which aspect functionality is composed to the remaining modules of the system. The exact moment when composition takes place and how that impacts on language mechanisms depends on the language/tool design and the implementation technologies. Some AOPs, including AspectJ, were designed so that weaving is orthogonal to other facets of the language. In AspectJ, weaving takes place at static time, which can be compile time or class load time. On the other hand, the weaving of Spring AOP is based on dynamic proxies and occurs at runtime [13]. In addition, some AOPs provide specific mechanisms for aspect instantiation (e.g., through the `new` keyword), which entails some form of dynamic weaving and enables programmers to control the times when aspects are active. Other AOPs (AspectJ included) support implicit instantiation, in which case aspects are always active by default and finer control must be supported programmatically. Figure 3 shows the weaving process of AspectJ.

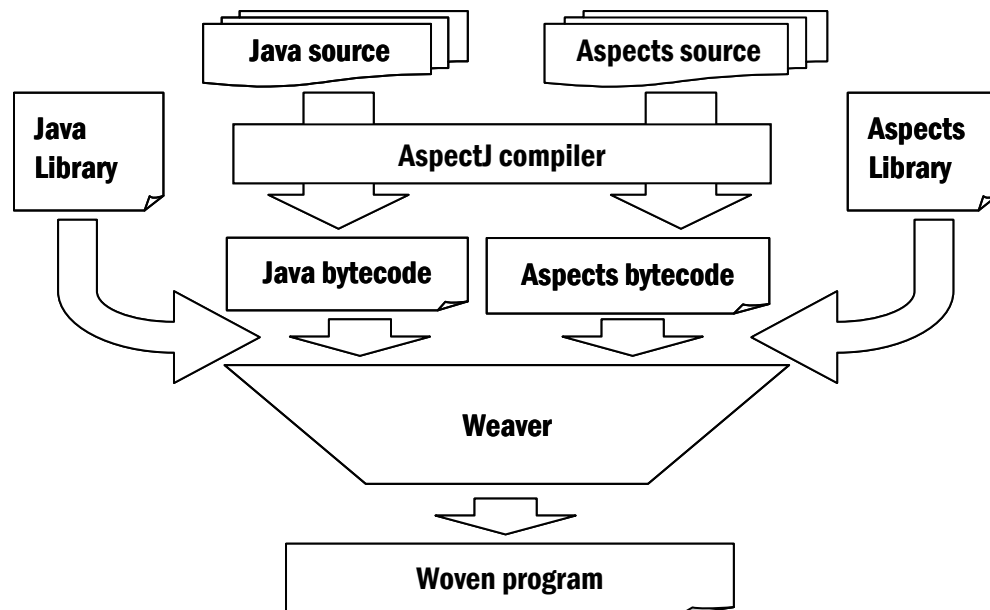


Figure 3 Compilation process with aspect weaving

Pattern **Detect Crosscutting Concerns**

Context You are evolving an existing OO system.

Problem You want to assess if a set of symptoms in the source code does correspond to a CCC. In addition, you want to be certain that a given concern, feature or functionality comprises a CCC that is amenable to modularization using the available AOL.

How does one recognize a CCC from hints and symptoms in the source code?

How does one know that the CCC is of a kind that can be effectively handled by the available AOL?

Forces **Paradigm shift.** It may be hard for people not familiar with AOP to recognize that a given concern can be further modularized using aspects, even when she is facing problems of software evolution caused by the presence of CCCs.

Confusion with bad style. Some of the symptoms that indicate the presence of a CCC (e.g., code duplication) can also be observed in code written in bad style or corresponding to a bad design. In such cases, the right approach is to perform traditional refactoring⁴. To attempt to extract aspects from systems in such a state is generally counter-productive.

Tool (im)maturity. The activity of discovering latent aspects in existing systems is named *aspect mining*. Though aspect mining is a vibrant research area, there are currently no mature tools to discover aspect candidates, or such tools are not currently integrated into existing development environments.

Solution **Ensure that the system is well decomposed.** Start by making sure that the system is written in good style and its design is sound and up to date. If it is not, REFACTOR TOWARDS ASPECT-FRIENDLY CODE BASE before considering extracting CCCs from it. This may amount to simply applying traditional refactoring [11] to the system in order to make it better decomposed, as there is a fortunate alignment between good OO style and aspect-friendliness. If, however, the system is already well decomposed or it keeps betraying the symptoms of crosscutting after using REFACTOR TOWARDS ASPECT-FRIENDLY CODE BASE, you most likely detected a CCC waiting to be extracted to an aspect.

Duplication. See if the system has multiple snippets of boiler-plate code scattered throughout multiple methods that are clearly related to the same concern (e.g., writing the object states on the database, registering the event on the logger or interfacing with some middleware). If the code snippets are all similar or identical, you are in the presence of a *homogeneous CCC*, the most straightforward example of a CCC.

⁴ In many situations, improving the design and structure of the system exactly corresponds to REFACTORING TOWARDS ASPECT-FRIENDLY CODE BASE.

Product Line Feature. If, on the other hand, the concern comprises multiple code fragments that are all related but dissimilar, you may be in the presence of a *heterogeneous CCC* (most likely a *product line feature*). Not all AOLs handle that kind of concern effectively. Before refactoring, be sure that your AOL is one that does. If it isn't, don't DECIDE TO REFACTOR TO ASPECTS.

Code smells. Two of the *code smells* proposed in Fowler's book on refactoring [11] are indicative of the presence of CCCs: *Divergent Change* (a class or method that suffers many kinds of changes) is indicative of code tangling, while *Shotgun Surgery*⁵ (one change that alters many classes) is indicative of code scattering. Though these smells can also be the result of bad style in design and programming, it is worth checking whether they are indicative of a CCC.

Role based collaborations. Role based collaborations between objects, such as those that usually result from the implementation of some well-known design patterns [12], may be also CCCs. If you notice that several classes contain code that is not related to their core functionality (e.g., they also notify observer objects of changes in state or also send messages to a mediator object), check whether the code associated with the secondary role comprises a CCC.

Examples **Duplicated boiler-plate code (middleware concerns).** It has been suggested that middleware comprises “the killer application” of AOP, since most or all of the services provided by middleware are crosscutting by nature. Probably as a consequence, some of the most widely adopted AOP tools are frameworks for middleware services, namely JBoss AOP⁶, Aspectwerkz⁷, and Spring⁸. Already in her seminal work, Lopes [23] uses the examples of synchronization and distribution to illustrate the causes of code tangling.

In Listing 1, a simple code example of CCCs is shown, comprising a method with three CCCs – security, logging and transactionality – in which code related to CCCs is shaded. The example is taken from [13], which shows how the Spring 2.0 framework can be used to modularize the three CCCs involved. The example shown next uses toy solutions to the same effect, coded in AspectJ.

The method from Listing 1 illustrates the typical CCCs that early AOLs such as AspectJ are very effective in modularizing – fragments of boiler-plate code tangled with the core (business) logic of the method. To get an idea of the full impact of the CCCs across the whole system, picture many such fragments duplicated in many methods of the class, and the same scenario taking place in many of the other classes of the system. An important point to be taken is that such tangling and scattering phenomena is observable even in systems that are well decomposed (e.g. according to the style proposed in [11]). Listings 2–4 show each of the CCCs modularized into its own aspect module. Listing 5 shows the method with the core logic, after the extraction of the CCCs.

⁵ In [17], Kerievsky proposes a variant of *Shotgun Surgery* that he calls *Solution Sprawl*, noting that they are similar but sensed differently: “we become aware of *Solution Sprawl* by observing it, while we detect *Shotgun Surgery* by doing it”.

⁶ <http://labs.jboss.com/jbossaop/>

⁷ <http://aspectwerkz.codehaus.org/>

⁸ <http://www.springframework.org/>

```

public String doSomething(String input) {
    //Logging
    System.out.println(
        "Logging: entering business method with:" + input);

    System.out.println("Authorization: Security check for
authorization of action (business-method)");

    //transactionality
    try {
        System.out.println(
            "Transactionality: Start new session and transaction");

        System.out.println("\nSome business logic\n");

        System.out.println(
            "Transactionality: Commit transaction");
    } catch (Exception e) {
        System.err.println(
            "Transactionality: Rollback transaction");
    } finally {
        System.out.println("Transactionality: Close session");
    }
    //Logging
    System.out.println(
        "Logging: exiting business method with:" + input);
    return input;
}

```

Listing 1 - Example of method with crosscutting concerns security, logging and transactionality.

```

public aspect Logging {
    pointcut operations():
        execution(* com.myorg.springaop.examples.MyServices.*(..));
    //Logging
    before(String input): operations() && args(input) {
        System.out.println("Logging: entering business method with:"
            + input);
    }
    after(String input): operations() && args(input) {
        System.out.println("Logging: exiting business method with:"
            + input);
    }
}

```

Listing 2 - Logging aspect.

```

public aspect Authorization {
    private String permissionDenied = "permission denied";
    pointcut operations():
        execution(* com.myorg.springaop.examples.MyServices.*(..));
    String around(): operations() {
        boolean someCondition = true;
        System.out.println("Authorization: Security check for
authorization of action (business-method)");
        if(someCondition)
            return proceed();
        else
            return permissionDenied;
    }
}

```

Listing 3 - Authorization aspect.

```

public aspect Transactionality {
    pointcut operations():
        execution(* com.myorg.springaop.examples.MyServices.*(..));

    String around(): operations() {
        String result = "";
        //transactionality
        try {
            System.out.println("Transactionality: Start new session
and transaction");
            result = proceed();
            System.out.println("Transactionality: Commit
transaction");
        }
        catch(Exception e) {
            System.err.println("Transactionality: Rollback
transaction");
        }
        finally {
            System.out.println("Transactionality: Close session");
        }
        return result;
    }
}

```

Listing 4 - Transactionality aspect.

```

public String doSomething(String input) {
    System.out.println("\nSome business logic\n");
    return input;
}

```

Listing 5 - Method clean of crosscutting concerns.

Finally, there remains the issue of dealing with the order with which aspects compose their functionality (briefly mentioned but not considered in [13]). The AspectJ solution uses the **declare precedence** mechanism shown in Listing 6. In order to further ensure good separation of concerns, it is placed in a separate aspect in this case.

```

public aspect AspectPrecedence {
    declare precedence: Logging, Authorization, Transactionality;
}

```

Listing 6 - Version of method, clean of CCCs.

Duplicated boiler plate code. In [5], Bruntink et al report on the use of the technique of clone detection to automatically identify and locate CCCs in source code. The reported case study is an industrial C system. The authors conclude that looking for boiler plate code is indeed a promising approach to detect latent aspects in existing systems.

Traditional OO implementation of design patterns. The implementations of several design patterns are also CCCs [15][26]. The benefits brought from enhanced modularity tend to be felt most strongly in patterns whose solution gives rise to crosscutting of some form, including one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances. *Observer* [12] is among the most often cited examples, as it represents a whole collaboration between objects that aspects can completely modularize [15]. Other patterns also

derive similar benefits, namely *Chain of Responsibility* and *Mediator*. In some cases, the implementation of the pattern completely disappears, as the language mechanisms can directly implement the intended functionality. *Decorator*, *Strategy* and *Visitor* are examples. However, it is important to note that the derived benefit from implementing the pattern with AOP may depend on the particular circumstances of the instance of the pattern [27].

Role based collaborations. One hint on the existence of secondary roles is provided by the implementation of Java interfaces [32], as it is common for Java programmers to use interfaces to model secondary roles played by objects.

Consequences By being aware of the CCC, developers are in a better position to handle it in a suitable manner throughout the future evolution of the system, independently of whether they DECIDE TO REFACTOR TO ASPECTS or not.

Known Uses The original paper that proposes AOP [19], Kiczales et al describe several instances of code tangling, identifying CCCs as their root cause. They distinguish between *components* (i.e., the units of modularity supported by the language at hand such as objects, procedures and APIs) and *aspects* – concerns that tend not to be units of modularity in the system’s functional decomposition, but rather be properties that affect the performance or semantics of the components in systemic ways. Kiczales et al they provide the following table with further examples:

application	component language	component	aspects
image processing	procedural	filters	loop fusion, result sharing, compile-time memory allocation
digital library	OO	repositories, printers, services	minimizing network traffic, synchronization, failure hadling
matrix algorithms	procedural	linear algebra operations	matrix representation, permutation, floating point error

Pattern **Decide to refactor to aspects**

Context You are evolving an existing OO system in which you detected the presence of CCCs.

Problem You would like to know whether the combination of your system and the available AOL make for a good candidate for resorting to aspect-oriented refactoring.

What are the conditions that a given system must meet to be a good refactoring candidate? Do the available tools make it feasible to undertake a refactoring to aspects?

Forces **Availability of an AOP extension.** Refactoring to aspects requires the availability of an aspect-oriented extension of the original OOP language in which the system is written.

Maturity of the tool used. Many AOLs are proof-of-concept tools developed in the context of university and research centre projects. In most cases, it is not practical to rely on such immature and untested tools. One possible exception to this rule is when the team developing the application is also developing the language or has a close relationship with the developers.

Skills of the team members. Using AOP technology requires specialized skills that cannot be taken for granted on the part of the majority of programmers. Acquiring AOP skills is a time-consuming task that involves a paradigm shift and requires a non-trivial effort. The upfront cost may not warrant the switch to AOP in some cases.

Cost. Refactoring takes time and effort to perform.

Assessment issues. The first phase of refactoring an existing OO system to AOP is not about changing the existing decomposition, but merely about extracting code that relates to the target CCC. If you feel it is about more than that, it is a sign that REFACTOR TOWARDS ASPECT-FRIENDLY CODE BASE should be used first. Prior to extraction, the team must precisely identify and locate all elements relating to a given CCC, or be confident that they can be easily detected was the team goes along with the refactoring process. Only after making a good assessment of the target CCC is the team in a position to make a reasonably accurate estimate of how much effort and cost it takes to perform the extraction.

Flexibility of the refactoring process. There is no need to perform a large refactoring at one go. Often, a large and complex refactoring can be performed as a series of small contributions possibly spanning many weeks or months. This provides the opportunity to perform the refactorings when time is more readily available.

(lack of) Tool support. Presently, there is no tool support for AOP refactoring, be they *aspect-aware* versions of traditional refactorings [14] or AOP-specific refactorings [30][22][25]. Though developers can still use present tools to perform traditional OO refactorings, that is unsafe and developers will need to check whether the logic of existing aspects was affected in each case. In practice, refactoring to

aspects presently entails performing the refactorings manually, without the support from tools or with only limited and unsafe support.

Test coverage. Good test coverage is a prerequisite for any refactoring process: AOP does not change this. In the case of legacy systems that are not covered by tests, developers face a chicken-and-egg problem, as experience shows that code not covered by tests is often not as amenable to unit-testing, and it usually requires prior refactorings to be. That, however, is another case for using REFACTOR TOWARDS ASPECT-FRIENDLY CODE BASE. See also [9] for techniques to deal with dealing with code devoid of unit tests.

Compositional power of the available AOL. CCCs can be classified into two broad categories: *homogeneous* and *heterogeneous* CCCs. A homogeneous CCC is a concern in which the same or very similar behavior needs to occur at multiple points in the control flow of a software system. A heterogeneous CCC is a concern that impacts multiple points in a software system, but where the behavior that needs to occur at each of those points is different. This distinction is important, because early AOLs (including AspectJ) do not modularize heterogeneous CCCs as effectively as homogeneous CCCs. One reason for this being so, is that the mechanisms for static crosscutting of those AOLs are not as powerful as their mechanisms for dynamic crosscutting. Therefore developers must assess whether the available AOL can effectively handle the CCCs discovered from using DETECT CROSSCUTTING CONCERNS, lest they fall into the trap of trying to extract a kind of CCC that is not handled effectively by the AOL at hand.

Legal issues. Though aspects modularize CCCs at the conceptual and source code levels, this is often not the case at the binary level, depending on the weaving technology used. In many cases, the weaving phase inserts new sections of code into the binary representations of the modules affected by the aspect. For this reason, weaving a third-party component often violates the license under which the component is provided. Though many such legal hurdles can be solved by technical solutions (for instance, by judiciously selecting pointcuts that affect only code to which the team is legally entitled to change), there are cases in which easy turnarounds are awkward or unavailable, making it impractical to perform the refactoring.

Greater ease of evolution and maintenance. A system with CCCs localized within aspects has an improved modularity and is devoid of the scattering and tangling effects. The number of modules is likely to increase, as more concerns are placed in their own modules and the overall level of abstraction is therefore raised. In many cases, duplication is reduced. All this has positive consequences to the evolution and maintenance of both the core concerns and CCCs.

Solution Assess whether all conditions to make a refactoring feasible are met and be aware of its positive and negative consequences. Then make a decision, and, if yes, proceed with the refactoring process.

Your OOP system is a good candidate for refactoring to AOP if the members of your team are aware of the presence of CCCs in your system, whose evolution is proving costly and/or troublesome.

If no AOP extension to the language in which your system is written is available, don't DECIDE TO REFACTOR TO ASPECTS. If an AOP extension is available, the option of going ahead can be justified if the following conditions apply:

- The AOP extension to the language in which your system is written to is considered mature enough for your purposes.
- Your system is already well-decomposed according to the design principles and style proposed in [11]. If it is not, first REFACTOR TOWARDS ASPECT-FRIENDLY CODE BASE.
- Your team identified and located precisely the various scattered elements that relate to the CCC, or is confident that they can be easily located as the team goes along with the process.
- Some team members possess the required skills in the field of AOP and the team is willing to switch from the OOP version of the language to the AOP extension.
- There is good coverage of unit tests, at least in the areas of functionality affected by the CCC.

Most CCCs lie between the two extremes of a continuum between entirely homogeneous CCCs and entirely heterogeneous CCCs [24]. The developer team must decide whether, in their particular case, the CCC warrants its extraction to an aspect. As a rule, CCCs that require mostly dynamic crosscutting are handled effectively by most AOLs.

Examples [25] describes a CCC in that proved to be inadequate for refactoring to AspectJ. Some of the symptoms of the awkwardness of the result of an attempt to extract it to an aspect are described in [29].

Consequences **No turning back.** Once the system is made to evolve to AOP, it is hard and costly to reverse this particular evolution step.

Less mature tool support. Available tool support for evolving the system may be less mature than for the OO version of the system.

Permanent need for AOP skills within the team. In order to maintain the system, the team will need to permanently include one or several members knowledgeable in AOP and associated tools.

Enhanced evolvability. The source code of the system is cleaner and free from the scattering and tangling effects, and therefore understandability and maintainability are made easier.

Known uses **AspectJ.** AspectJ [1][20][21] is a good example of an AOL being used in industrial projects. In [7], Colyer and Clement describe lessons learned while refactoring a large IBM middleware platform. Zhang and Jacobsen report on the aspectization of ORBacus⁹, an open source industrial implementation of the CORBA middleware platform [34]. Tonella and Ceccato [32] treat the implementation of

⁹ ORBacus, <http://www.iona.com>.

Java interfaces as a CCC and report on the results of an extraction experiment targeting three packages from the standard library of the Java 2 Runtime Environment Standard Edition.

Other AOLs. Published work about refactoring to AOP is not confined to the Java universe. In [31], Mortensen, Ghosh and Bieman report on their experiences of refactoring to AspectC++ two VLSI CAD applications, written in C++. They also provide details on the techniques used for ensuring proper test coverage. In [6], Bruntink Deursen and Tourwé report on their experience in migrating CCCs of a large-scale C system into AspectC.

Pattern **Refactor Towards Aspect-Friendly Code** Base

Context You have an OO system with one or several CCCs and you decided that you want to refactor it to an AOP extension of the existing language.

Problem You would like to assess whether the system in its current form is ripe for refactoring to that AOL as it is, or whether some prior refactoring is required.

Will the present structure of the system constrain the refactoring process? If so, what should be the course of action?

Forces **Lack of joinpoint leverage.** It is much simpler to move elements that are first-class members (such as fields and methods) than to move code fragments. Extracting a code fragment from a method entails creating a pointcut that captures a joinpoint that corresponds to the exact point where the fragment is placed, or extending an existing pointcut with the relevant joinpoint. The code base may not expose suitable joinpoints. Prior refactoring is required in those cases, to make the code base more amenable to the composition of aspects.

Present design and style of the target system. Experience gained in the latest few years tells that good OO design and coding style are important prerequisites for refactoring to AOP [25][33]. The more well-decomposed a system is, the greater the likelihood that it exposes all desirable joinpoints. However, many existing systems are not well decomposed [10] and may place additional hurdles.

Time available to refactor. Refactoring takes time but can be performed in phases. By itself, refactoring just to make a system aspect friendly yields no aspects at all.

Benefits of traditional OO refactoring. Refactoring to a better style or design brings benefits to understandability, maintainability and ease of evolution, that are independent of whether you DECIDE TO REFACTOR TO ASPECTS or not.

Solution Before going ahead with AOP refactorings, ensure that your system is already well-decomposed according to the current notions of good OO style. Fowler's book [11] provides a catalogue of 72 refactorings that can be used to perform such a decomposition, as well as a collection of 22 bad smells that indicate the kinds of situation in the code that warrant the use of the refactorings. Good examples of such smells are: DUPLICATED CODE, LONG METHOD and LARGE CLASS.

Consequences Once the code base is further decomposed, it is more likely to expose the joinpoints needed by potential aspects.

In some cases, duplication initially deemed to be caused by CCCs may be removed, eliminating the need to refactor to aspects.

The resulting system is easier to reason with and evolve, independently of the initial motivation being to make the system more aspect-friendly.

Known uses **Large repository of testimonies.** The *aspectj-users mailing list* [2] has lots of posts describing awkward situations that are solved by refactoring the code base in order to expose the desirable joinpoints.

When discussing insights acquired from analyzing a Java framework, Monteiro notes that good OO style – in the sense of [11] – is a precondition for applying AOP and briefly discusses the subject [25]. If, for instance, the system has many instances of the *Large Class* and *Long Method* code smells [11], the team risks facing situations in which most or all of the elements of the CCC are hard-to-reason-with and hard-to-disentangle fragments “swimming” in a sea of unrelated code.

Yuen and Robillard reach conclusions similar to those of [11], on the basis of experiments that included locating and extracting two CCCs from an open-source Java project [33].

Credits (to be written)

References

- [1] AspectJ home page. <http://www.eclipse.org/aspectj/>
- [2] AspectJ users mailing list, <https://dev.eclipse.org/mailman/listinfo/aspectj-users>
- [3] Refactoring home page. <http://www.refactoring.com/>
- [4] Bracha G., Cook W. (1990) Mixin-Based Inheritance. In proceedings of ECOOP/OOPSLA 1990, ACM press, pages 303-311.
- [5] Bruntink M., Deursen A.v., Engelen R., Tourwé T. (2005). On the Use of Clone Detection for Identifying Crosscutting Concern Code. In IEEE Transactions of Software Engineering, (Vol. 31, No. 10), pages 804-818.
- [6] Bruntink M., Deursen A.v., Tourwé T. (2004). Isolating Crosscutting Concerns in System Software, In proceedings of the WCRE 2004 Workshop on Aspect Reverse Engineering (WARE), Delft, The Netherlands.
- [7] Colyer A., Clement A. (2004) Large-scale AOSD for Middleware. In proceedings of AOSD 2004, pages 56-65, Lancaster, UK.
- [8] Colyer A, Clement A., Harley G., Webster M. (2004) Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison Wesley.
- [9] Feathers M. (2004). Working effectively with legacy code. Prentice Hall.
- [10] Foote B., Yoder J. (1999). Big Ball of Mud. In proceedings of PLoP '97, Monticello, Illinois.
- [11] Fowler M., Beck K., Opdyke W., Roberts D. (1999). Refactoring – Improving the Design of Existing Code. Addison Wesley.
- [12] Gamma E, Helm R, Johnson R, Vlissides J. (1995) Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley.
- [13] Ghag, G. (2007). Implement crosscutting concerns using Spring 2.0 AOP. Javaworld. <http://www.javaworld.com/javaworld/jw-01-2007/jw-0105-aop.html>
- [14] Hanenberg S, Oberschulte C, Unland R. (2003) Refactoring of aspect-oriented software. In proceedings of Net.ObjectDays, Thuringia, Germany.
- [15] Hannemann J., Kiczales G. (2002). Design Pattern Implementation in Java and AspectJ. In proceedings of OOPSLA 2002, Seattle, USA, ACM press, pages 161-173.
- [16] Hannemann J., Murphy G., Kiczales G. (2005). Role-Based Refactoring of Crosscutting Concerns. In proceedings of AOSD 2005, Chicago, USA, ACM press, pages 135-146.
- [17] Kerievsky J. (2004). Refactoring to Patterns, Addison-Wesley.
- [18] Kersten, M. (2005). AOP tools comparison, Part 1. Developerworks. <http://www.ibm.com/developerworks/java/library/j-aopwork1/index.html>
- [19] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. (1997) Aspect-oriented programming. In Proceedings of ECOOP 1997, Jyväskylä, Finland (LNCS, vol. 1241), Springer; pages 220–242.
- [20] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In proceedings of ECOOP 2001, Budapest, Hungary, (LNCS, vol. 2072), Springer; 327–335.
- [21] Laddad R. (2003) AspectJ in Action – Practical Aspect-Oriented Programming. Manning.
- [22] Laddad R. (2003) Aspect-Oriented Refactoring, parts 1 and 2, The Server Side, 2003. www.theserverside.com/
- [23] Lopes C. V. (1997). D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA.

- [24] Lopez-Herrejon R., Apel S. (2007). Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In proceedings of FASE 2007 at ETAPS 2007, pages 422-437.
- [25] Monteiro MP. (2005) Refactorings to evolve object-oriented systems with aspect-oriented concepts. PhD thesis, Universidade do Minho, Portugal.
- [26] Monteiro M.P. (2006). Using Design Patterns as Indicators of Refactoring Opportunities (to Aspects). In proceedings of the LATEr 2006 workshop at AOSD 2006, Bonn, Germany.
- [27] Monteiro M. P., Fernandes J. M. (2004) Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. In proceedings of the DSOA'2004 workshop at JISBD 2004, Málaga, Spain.
- [28] Monteiro M. P., Fernandes J. M. (2005) Refactoring a Java Code Base to AspectJ – An Illustrative Example. In proceedings of ICSM'05 pages 17–26, Budapest, Hungary.
- [29] Monteiro M. P., Fernandes J. M. (2003) Some Thoughts On Refactoring Objects to Aspects. In proceedings of the DSOA'2003 workshop at JISBD 2003, Alicante, Spain.
- [30] Monteiro M.P., Fernandes J. M. (2005) Towards a Catalogue of Aspect-Oriented Refactorings. In proceedings of AOSD 2005, pages. 111-122. Chicago, USA.
- [31] Mortensen M., Ghosh S., Bieman J.M. (2006). Testing During Refactoring: Adding Aspects to Legacy Systems. In proceedings of the Industry track of AOSD 2006, Bonn, Germany.
- [32] Tonella P., Ceccato M. (2004). Migrating Interface Implementation to Aspects. In proceedings of ICSM'04, pages 220-229, Chicago, USA.
- [33] Yuen I., Robillard M. (2007). Bridging the Gap between Aspect Mining and Refactoring. In proceedings of the LATE 2007 workshop, Vancouver, Canada.
- [34] Zhang C., Jacobsen H.A. (2003). Refactoring Middleware with Aspects. IEEE Transactions on Parallel and Distributed Systems, November 2003 (Vol. 14, No. 11), pages 1058-1073.