

Patterns for Refactoring to Aspects: An Incipient Pattern Language

Miguel P. Monteiro
Departamento de Informática
Faculdade de Ciência e Tecnologia
Universidade Nova de Lisboa
Tel +351-21 294 8536 ext: 0708
mmonteiro@di.fct.unl.pt

Ademar Aguiar
INESC Porto
Faculdade de Engenharia
Universidade do Porto
Tel. +351-22 508 1518 ext: 3212
ademar.aguiar@fe.up.pt

ABSTRACT

Aspect-Oriented Programming is an emerging 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 prerequisites for the refactoring process.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.3.3 [Programming Languages]: Language Constructs and Features – *patterns*.

General Terms

Design, Human Factors, Languages.

Keywords

Software refactoring, Aspect-oriented programming.

1. 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 cannot be modularized with the traditional mechanisms of object-oriented (OO) languages such as Java. Such concerns are usually called *crosscutting concerns* (CCCs) [6]. Examples include systems that use the services provided by middleware and the implementation of various well-known design patterns (e.g. *Observer* and *Visitor*) [11]. 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* [17]. 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) [17] 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* [17]. Currently, many aspect-oriented languages are backwards compatible extensions to existing languages. Of those, the most mature is AspectJ [18][19][7], 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 that use AOP technology [16]. 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][10] 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*) that are intended to focus on the initial issues that arise when considering the option to refactor an existing OO system to AOP.

The rest of the paper is structured as follows. Section 2 overviews the proposed pattern language, section 3 surveys the main concepts of AOP and section 4 describes three of the patterns.

2. The Pattern Language

The pattern language – outlined in Figure 1 – 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 using AOP in the future evolution of their systems. The patterns originate from reading the existing literature, experience gained by the authors and ongoing experiments.

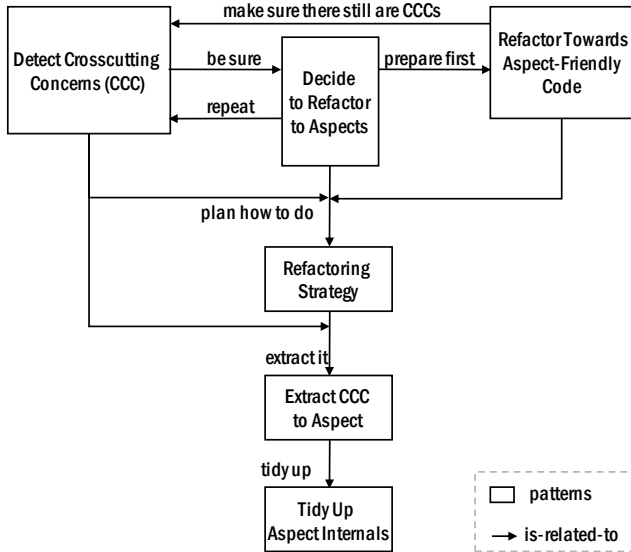


Figure 1. Relationships between the patterns.

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 (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.

Detect Crosscutting Concerns helps developers in diagnosing the presence of CCCs in their systems, 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 whether 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 helps developers to decide if they should first perform some preparatory, traditional OO refactorings or if they can jump straight into AOP refactoring.

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. It is significantly harder (or even impossible) to perform certain kinds of “tidying up” transformations before the modularization phase, i.e., while the implementation elements are scattered throughout multiple modules. Thus, *Refactoring Strategy* proposes that priority be given to the extraction of all elements of the target CCC to a new aspect module – using *Extract CCC to an Aspect* – after which it becomes feasible to use *Tidy Up Aspect Internals* to perform a comprehensive tidying up

of the of the extracted aspect by focusing on improving its internal structure.

As regards *Extract 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 behaviour is broken during the process. Thus, *Extract CCC to an Aspect* follows the principles and spirit advocated in Fowler’s book [10]. The refactoring process proposed by *Extract CCC to an Aspect* was first proposed in [28]. A detailed example is described in [26].

Tidy Up Aspect Internals gives tips on dealing with potential inadequacies in the internal structure of the aspect obtained through *Extract 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 (e.g., the same class member duplicated in multiple classes). Such designs tend to be strongly influenced by the original presence of the scattering and tangling phenomena and may no longer make sense after the CCC is modularized. *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 refactorings should strive for.

3. CONCEPTS OF AOP

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

3.1 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).

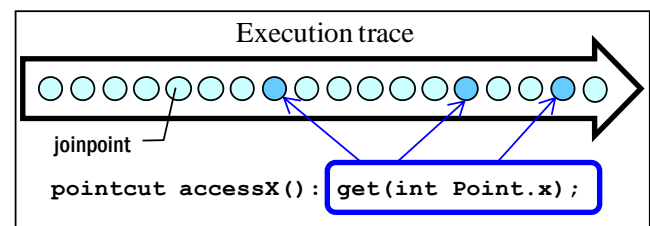


Figure 2. Execution trace as a sequence of joinpoints, some selected by a pointcut.

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 aspect-oriented

languages (AOLs) are characterized is its *joinpoint model*, i.e., the set of joinpoints it supports. AOLs 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 AOL 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.

3.2 Pointcut

A pointcut is a declarative clause that specifies sets of joinpoints (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”. The following example shows a pointcut capturing all calls to the public methods of `java.io.PrintStream` having any number of arguments, void as return type, and a name starting with “print”:

```
public pointcut allCalls2SystemOutPrints():
    call(public void java.io.PrintStream.print*(..));
```

As the places in the source code relating to the specified joinpoints are non-contiguous, the set of *captured joinpoints* cuts across the system’s structure. Pointcuts can be composed like predicates, using operators `&&`, `||` and `!`, which express (joinpoint) set union, set intersection and set complement respectively. Some pointcuts serve 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 (see section 3.3 for an example).

In addition, many AOLs also provide to restrict the set of joinpoints captured by other pointcuts, rather than specifying sets of their own. The following example shows a pointcut similar to the one above, but complemented with a `within` pointcut that restricts 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));
```

3.3 Advice

AOLs have the ability to execute additional behaviour before, after, and, in some cases, instead of the captured joinpoints (Figure 3). In some AOLs, 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 true first-class entities. In AspectJ, an `around` advice executes instead of the captured joinpoint and can optionally execute the original joinpoint by means of a call to `proceed()`. Next follows the example of an `around` advice that executes instead of each method call captured:

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

To illustrate how context from the joinpoints can be captured and 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);
```

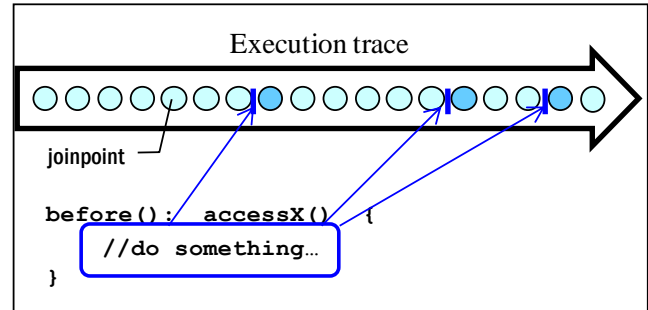


Figure 3. Before advice executes before each captured joinpoint.

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

```
void around(Object msg): msgsFromSystemOutPrint(msg) {
    proceed("[ " + msg.toString() + " ]");
}
```

3.4 Aspect

In AOP, the term *aspect* is used to refer to the modular implementation of a concern whose implementation would otherwise cut across multiple modules. Often, *aspect* can be used to refer to a concern that does not make sense to consider by itself. Examples of such concerns include persistence, synchronization, and indeed the most or all of the services provided by middleware.

3.5 Dynamic and static crosscutting

One well-known characteristic definition of objects is that objects encapsulate state and behaviour, both of which take the form of class members, typically fields and methods. Often, the presence a CCC gives rise to duplication and scattering of such members throughout multiple classes. A distinction between *static structure* and *dynamic behaviour* is often applied to the elements that an aspect composes to the remaining modules of a system. The ability of aspects to compose crosscutting behaviour to a given system, e.g., by means of pointcuts and advices, is called *dynamic crosscutting*. This ability is about composing behaviour. In addition, many AOLs also have the ability to change or extend the existing *static 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 mechanisms to modify the static structure of existing modules are called *static crosscutting*. In AspectJ, static crosscutting is mostly supported by *inter-type declarations*, which provide aspects with the ability to introduce additional members 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.

```
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 information hiding on the aspect side. More detailed information about AOP concepts and AspectJ can be found in the literature, namely in [27][17][18][19][7].

3.6 Weaving

Weaving is the name given to the phase in which aspect functionality is composed with the remaining modules of the system. The concrete stage when composition takes place (e.g., compile-time, load-time, runtime) and how that impacts on language mechanisms depend on the language/tool design and the implementation technologies. Some AOLs, 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 [12]. In addition, some AOLs 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 concrete phases when aspects are active. Other AOLs (AspectJ included) support *implicit instantiation*, in which case aspects are always active by default and finer control must be supported programmatically. Figure 4 shows the weaving process of AspectJ.

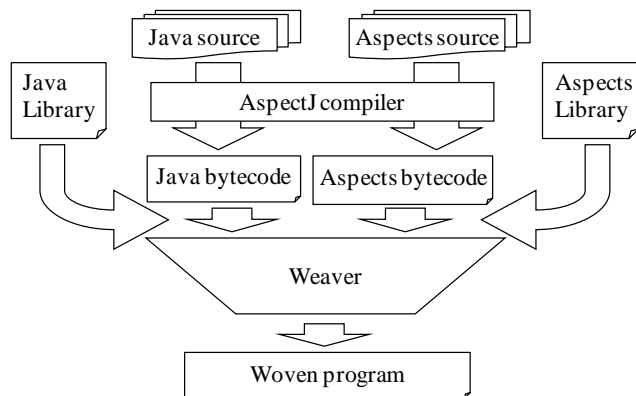


Figure 4. Compilation process with aspect weaving

4. THE PATTERNS

This section proposes the patterns *Detect Crosscutting Concerns*, *Decide to Refactor to Aspects* and *Refactor Towards Aspect-Friendly Code*.

4.1 Detect Crosscutting Concerns

4.1.1 Context

You are evolving an existing OO system. You notice symptoms in the system that may be indicative of the presence of CCCs.

4.1.2 Problem

You want to assess if the symptoms do 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?

4.1.3 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 refactoring. In many situations, improving the design and structure of the system exactly corresponds to *Refactor Towards Aspect-Friendly Code*. In general, it is counter-productive to attempt to extract aspects from systems before they are properly refactored and decomposed.

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.

4.1.4 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 clean not, *Refactor Towards Aspect-Friendly Code* before considering extracting CCCs from it. This may amount to simply applying traditional refactoring [10] 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 showing the symptoms of crosscutting after using *Refactor Towards Aspect-Friendly Code*, you most likely detected a CCC waiting to be extracted to an aspect.

Duplication. See if the system has multiple snippets of boilerplate 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.

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 [10] 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* (one change that alters many classes) is indicative of code scattering. In [15], 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 *Shogun Surgery* by doing it”. 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 [11], 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 relates to a CCC.

4.1.5 Examples

4.1.5.1 Duplicated boiler-plate code

In the past, it was suggested that middleware provides “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 AOP¹, Aspectwerkz², and Spring³. Already in her seminal work, Lopes [21] uses the examples of synchronization and distribution to illustrate the causes of code tangling.

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

    //Authorization:
    //Security check for authorization of 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. Method with CCCs security, logging and transactionality.

¹ <http://labs.jboss.com/jbossaop/>

² <http://aspectwerkz.codehaus.org/>

³ <http://www.springframework.org/>

In Listing 1, a simple code sketch of a CCC is shown, comprising a method with three typical CCCs – security, logging and transactionality – in which code related to CCCs is shaded. The example is taken from an online article by Ghag [12], which shows how the Spring 2.0 framework can be used to modularize the three CCCs involved. The example is here adapted to AspectJ.

The method from Listing 1 illustrates the typical CCCs that early AOPs 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 phenomena of tangling and scattering are observable even in systems that are well decomposed (e.g. according to the style proposed by Fowler et al [10]).

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

Listing 2. Logging aspect for the example of Listing 1.

```
public aspect Authorization {
    pointcut operations():
        execution(* com.myorg.springaop.examples.MyServices.*(..));
    void around(): operations() {
        boolean permissionGranted;
        //Authorization:
        //Security check for authorization of business-method
        if(permissionGranted)
            proceed(); //proceed with the operation
        else
            //notify that permission is denied
    }
}
```

Listing 3. Authorization aspect for the example of Listing 1.

```
public aspect Transactionality {
    pointcut operations():
        execution(* com.myorg.framework.MyServices.*(..));
    void around(): operations() {
        //transactionality
        try {
            System.out.println("Transaction: Start new transaction");
            proceed(); //carry out the core logic
            System.out.println("Transactionality: Commit");
        }
        catch(Exception e) {
            System.err.println("Transactionality: Rollback");
        }
        finally { System.out.println("Transactionality: Close session"); }
        return result;
    }
}
```

Listing 4. Transactionality aspect for the example of Listing 1.

Listings 2–4 show each of the CCCs modularized into its own AspectJ module. Listing 5 shows the Java method with the core logic, after the extraction of the CCCs.

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

Listing 5. Method clean of CCCs.

Finally, there remains the issue of dealing with the order with which aspects compose their functionality (briefly mentioned but not considered in [12]). 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. Specifying precedence of aspects.

4.1.5.2 Duplicated boiler plate code

Bruntink et al [4] 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.

4.1.5.3 Role-based collaborations

One hint on the existence of secondary roles is provided by the implementation of Java interfaces. Java programmers often use interfaces to model secondary roles played by objects. Such secondary roles are generally hard-wired to the core concern and cannot be unplugged from it. Often, more than one class implements the interface, in which case the implementation of the interface is crosscutting [30].

4.1.5.4 OO implementation of design patterns

The traditional OO implementations of several design patterns are also CCCs [14][24]. 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. Thus, the implementations of some patterns comprise examples of role-based collaborations.

Observer [11] is among the most often cited examples, as it defines two different roles, typically assigned to different classes. *Observer* models a collaboration between objects playing one of two roles, *subject* and *observer*. Aspects can effectively modularize such collaborations [14]. 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 to some extent. However, it is important to note that the derived benefit from implementing the pattern with AOP may depend on the particular requirements and circumstances of the instance of the pattern [25].

4.1.6 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).

4.1.7 Known uses

The original paper that proposes AOP [17], Kiczales et al describes several instances of code tangling, identifying CCCs as their root cause. They distinguish between *components* and *aspects*. In this context, components are the units of modularity supported by the base language, such as objects, procedures and APIs; and (latent) aspects are 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. In their seminal paper, Kiczales et al [17] propose the examples shown in Table 1. Note that not all the examples of CCCs from Table 1 could be suitably implemented with a language such as AspectJ (e.g., loop fusion). The purpose of the examples is to provide a broad idea of what *can* be a CCC.

Table 1. Examples of crosscutting concerns proposed in [17]

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 handling
matrix algorithms	procedural	linear algebra operations	matrix representation, permutation, floating point error

4.2 Decide to refactor to aspects

4.2.1 Context

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

4.2.2 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?

4.2.3 Forces

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

Maturity of the tool used. Many AOLs are proof-of-concept tools developed in the context of academic and research 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 of the AOL.

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 that a first phase entails more than mere extraction, this may be a sign that *Refactor Towards Aspect-friendly Code* 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 as the team goes along with the refactoring process. Only after making a thorough 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 [10] or AOP-specific refactorings [28][20][13]. 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 all refactoring processes and 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 preparatory refactorings. The need for such refactorings is another case for using *Refactor Towards Aspect-friendly Code*. See also [8] for techniques to deal with code devoid of unit tests.

Compositional power of the available AOL. CCCs can be classified into two broad categories: *homogeneous* and *heterogeneous* CCCs [22]. A homogeneous CCC is a concern in which the same or very similar behaviour 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 behaviour 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 is that the mechanisms for static crosscutting of those AOLs are not as powerful or expressive as their mechanisms for dynamic crosscutting. Therefore developers must assess whether the available AOL can effectively handle the CCCs discovered from applying *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.

Enhanced flexibility 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 representation of the concerns as first-class entities is enriched. Duplication is often reduced or eliminated. All this has positive consequences to the evolution and maintenance of both the core concerns and CCCs.

4.2.4 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 [10]. If it is not, first *Refactor Towards Aspect-Friendly Code*.
- Your team identified and located precisely the various scattered elements that relate to the CCC, or is confident that they can be located in a straightforward manner as the team goes along with the process.
- 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 [22]. 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.

4.2.5 Examples

Monteiro describes a CCC that proved to be inadequate for refactoring to AspectJ [23]. Some of the symptoms of the awkwardness of the result of an attempt to extract it to an aspect are described in [27].

4.2.6 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 the AOL used 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.

4.2.7 Known uses

AspectJ. AspectJ [1][18][19] is a good example of an AOL being used in industrial projects. Colyer and Clement describe lessons learned while refactoring a large IBM middleware platform [6]. Zhang and Jacobsen report on the aspectization of ORBacus⁴, an open source industrial implementation of the CORBA middleware platform [32]. Tonella and Ceccato treat the implementation of 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 [30].

Other AOLs. Published work about refactoring to AOP is not confined to the Java universe. Mortensen, Ghosh and Bieman report on their experiences of refactoring to AspectC++ two VLSI CAD applications written in C++ [29]. They also provide details on the techniques used for ensuring proper test coverage. Bruntink Deursen and Tourwé report on their experience in migrating CCCs of a large-scale C system into AspectC [5].

4.3 Refactor Towards Aspect-Friendly Code

4.3.1 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.

4.3.2 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?

4.3.3 Forces

Lack of joinpoint leverage. Extracting a code fragment from a method entails creating a pointcut that captures a joinpoint that corresponds to a point behaviourally equivalent to where the fragment is placed, or extending an existing pointcut. The code base may not expose suitable joinpoints. For instance, it is much simpler to move elements that are first-class members (such as fields and methods) than to move code fragments from the middle of long method. Prior refactoring may be required in such cases, to make the code base more amenable to the composition of aspects, using *Refactor Towards Aspect-Friendly Code*.

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 [23][31]. The more well-decomposed a system is, the greater the likelihood that it exposes all desirable joinpoints. Unfortunately, many existing systems are not well decomposed [9], again requiring prior use of *Refactor Towards Aspect-Friendly Code*.

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 but yields its own benefits.

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.

4.3.4 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 [10] 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. Typical examples of such smells are: *Duplicated Code*, *Long Method* and *Large Class*.

4.3.5 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 motivation 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.

4.3.6 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 [23] notes that good OO style – in the sense proposed by Fowler et al [10] – is a precondition for applying AOP and briefly discusses the subject. If, for instance, the system has many instances of the *Large Class* and *Long Method* code smells [10], the team risks facing situations in which most or all of the elements of the CCC are hard-to-reason-with and hard-to-

⁴ ORBacus, <http://www.iona.com>.

disentangle fragments “swimming” in a sea of unrelated code. Yuen and Robillard reach conclusions similar to those of Monteiro [23] on the basis of experiments that included locating and extracting two CCCs from an open-source Java project [31].

5. CONCLUSION

This paper proposes three patterns of an incipient pattern language for refactoring an existing system into the aspect-oriented paradigm. The patterns focus on the early assessment and decision stages. *Detect Crosscutting Concerns* provides advice on how to identify latent aspects in a software system. *Decide to Refactor to Aspects* describe the situations in which it is feasible to refactor to aspects. *Refactor Towards Aspect-Friendly Code* provides advice on how to assess whether the necessary prerequisites for a refactoring process are met.

6. ACKNOWLEDGMENTS

The authors would like to thank Peter Sommerlad (our PLoP'07 shepherd), Ralph Johnson, Atul Jain, Hriday Rajan, Berna L. Massingill and Mark Mahoney for the valuable feedback provided on earlier versions of this paper.

This work was partially supported by project AMADEUS (POCTI, PTDC/EIA/ 70271/2006) funded by Portuguese *Fundação para a Ciência e Tecnologia*.

7. 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] 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.
- [5] 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.
- [6] Colyer A., Clement A. (2004) Large-scale AOSD for Middleware. In proceedings of AOSD 2004, pages 56-65, Lancaster, UK.
- [7] Colyer A, Clement A., Harley G., Webster M. (2004) Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison Wesley.
- [8] Feathers M. (2004). Working effectively with legacy code. Prentice Hall.
- [9] Foote B., Yoder J. (1999). Big Ball of Mud. In proceedings of PLoP '97, Monticello, Illinois.
- [10] Fowler M., Beck K., Opdyke W., Roberts D. (1999). Refactoring – Improving the Design of Existing Code. Addison Wesley.
- [11] Gamma E, Helm R, Johnson R, Vlissides J. (1995) Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley.
- [12] Ghag, G. (2007). Implement crosscutting concerns using Spring 2.0 AOP. Javaworld. <http://www.javaworld.com/javaworld/jw-01-2007/jw-0105-aop.html>
- [13] Hanenberg S, Oberschulte C, Unland R. (2003) Refactoring of aspect-oriented software. In proceedings of Net.ObjectDays, Thuringia, Germany.
- [14] Hannemann J., Kiczales G. (2002). Design Pattern Implementation in Java and AspectJ. In proceedings of OOPSLA 2002, Seattle, USA, ACM press, pages 161-173.
- [15] Kerievsky J. (2004). Refactoring to Patterns, Addison-Wesley.
- [16] Kersten, M. (2005). AOP tools comparison, Part 1. Developerworks. <http://www.ibm.com/developerworks/java/library/j-aopwork1/index.html>
- [17] 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.
- [18] 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.
- [19] Laddad R. (2003) AspectJ in Action – Practical Aspect-Oriented Programming. Manning.
- [20] Laddad R. (2003) Aspect-Oriented Refactoring, parts 1 and 2, The Server Side, 2003. www.theserverside.com/
- [21] Lopes C. V. (1997). D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA.
- [22] 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.
- [23] Monteiro MP. (2005) Refactorings to evolve object-oriented systems with aspect-oriented concepts. PhD thesis, Universidade do Minho, Portugal.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] Monteiro M. P., Fernandes J. M., Object-to-Aspect Refactorings for Feature Extraction, Industry track paper at the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, UK, March 2004.

- [28] Monteiro M.P., Fernandes J. M. (2005) Towards a Catalogue of Aspect-Oriented Refactorings. In proceedings of AOSD 2005, pages. 111-122. Chicago, USA.
- [29] 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.
- [30] Tonella P., Ceccato M. (2004). Migrating Interface Implementation to Aspects. In proceedings of ICSM'04, pages 220-229, Chicago, USA.
- [31] Yuen I., Robillard M. (2007). Bridging the Gap between Aspect Mining and Refactoring. In proceedings of the LATE 2007 workshop, Vancouver, Canada.
- [32] 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.