Journal of Universal Computer Science, vol. 19, no. 1 (2013), 157-182
submitted: 1/11/11, accepted: 28/12/12, appeared: 1/1/13 © J.UCS

A Catalog of Aspect Refactorings for Spring/AOP

Santiago A. Vidal
(ISISTAN Research Institute, UNICEN, Tandil, Argentina. Also CONICET.
svidal@exa.unicen.edu.ar)

Claudia Marcos
(ISISTAN Research Institute, UNICEN, Tandil, Argentina. Also CIC.
cmarcos@exa.unicen.edu.ar)

Abstract: The importance of enterprise applications in current organizations makes it necessary to facilitate their maintenance and evolution along their life. These kind of systems are very complex and they have several requirements that orthogonally crosscut the system structure (called crosscutting concerns). Since many of the enterprise systems are developed with the Spring framework, can be taken advantage of the benefit provided by the aspect-oriented module of Spring in order to encapsulate the crosscutting concerns into aspects. In this way, the maintenance and evolution of the enterprise systems will be improved. However, most of the aspect refactorings presented in the literature are not directly applicable to Spring systems. Along this line, in this work we present an adaptation of a catalog of aspect refactorings, initially presented for AspectJ, to be used with Spring/AOP. Also, we conduct a case study in which two enterprise applications developed with the Spring framework are refactored in order to encapsulate their crosscutting concerns into aspects.

Key Words: separation of concerns, refactoring, aspect-oriented programming
Category: D.3.3, D.2.3

1 Introduction

The concept of enterprise application is used to describe the software systems that are used by the companies to help to solve the problems of these organizations [Kayal (08)]. An enterprise system is a piece of software that provides functionality to support business logic for an enterprise. In general, these systems are used in organizations aimed at improving productivity and business efficiency. The features of this kind of systems are usually business tools such as shopping and online payment processing, interactive product catalogs, automated billing, and so on.

These enterprise applications are complex and mission-critical. To design and implement this kind of systems hundreds of requirements must be satisfied. Each implementation decision, made to satisfy a requisite, usually affects other requisites. Sometimes, the relationships between requirements are not clear and the violation of any of these requirements may result in the failure of an entire project [Johnson (03)]. In fact, the maintenance and evolution of these systems is even more complex once they are deployed.

A traditional approach to deal with complexity and simplify the evolution of complex systems, such as enterprise systems, is modularization [Parnas 72]. When complex software requisites are found, generally they are divided into several parts, such as business logic, data access and presentation logic. These features represent different system concerns that are encapsulated into different modules. However, there exist some concerns, called crosscutting concerns (CCCs), whose encapsulation is almost unviable because they crosscut the modules of a system [Kiczales et al. 97]. This kind of concerns can be encapsulated into a new component called aspects by means of Aspect Oriented Programming (AOP) [Elrad et al. 01].

AOP is a software paradigm that complements object-oriented programming (OOP) to address the problem of separation of concerns [Kiczales et al. 97]. AOP allows the encapsulation of CCCs into new components called aspects. In this way, AOP increases the software modularization and potentially reduces the impact of change propagation when systems are modified [Garcia et al. 05].

Since the appearance of AOP various aspect languages have been developed, such as AspectJ[1], or frameworks with support for aspects such as Spring/AOP[2]. With regard to the latter, there is a wide and growing use of the Spring framework to develop enterprise systems. In order to improve the separation of concerns by means of AOP, the systems should be refactored to encapsulated the CCCs into aspects. In this way more flexible and modular enterprise systems will be potentially achieved improving the long-term evolution of the applications. To achieve the goal of encapsulating the CCCs into aspects, Spring provides an AOP module for writing aspects and that promotes low coupling of code.

For these reasons, in order to improve the separation of concerns and software evolution in enterprise applications, AOP can be adopted with low cost specially in those legacy systems that have been developed with Spring and that are continually evolving. In this way, the process to separate the CCCs involves the identification of them in the legacy systems, in a process known as aspect mining [Kellens et al. 07], and then, the encapsulation of the CCCs into aspects in a process known as aspect refactoring [Kellens et al. 07]. Most of the existing aspect mining techniques can be applied to Spring since they are based on Java code [Abait and Marcos 09, Marin et al. 07]. However, since most of the aspect refactoring techniques transform CCCs to aspects using a specific AOP language and a big number of systems are implemented in Java, most of the aspect refactoring techniques that have been presented are for AspectJ [Hannemann and Kiczales 02, Iwamoto and Zhao 03, Marin 04, Malta and de Oliveira Valente 09, Monteiro 04]. These refactorings transform CCCs of Java systems to aspects written in AspectJ. However, few refactorings have been proposed to generate

[1] http://www.eclipse.org/aspectj/
[2] http://www.springsource.org/

aspects in Spring/AOP [Laddad (09)].

Along this line, in this paper we present the adaptation of a set of aspect refactorings initially presented for AspectJ to be used with the Spring/AOP framework. We think that the possibility of adapting a catalog of aspect refactorings to be used in Spring/AOP will simplify the refactoring task of these systems. The task of adapting the refactorings is not trivial since Spring/AOP does not include all the feature presented by AspectJ. In this work we focus in the adaptation of those refactorings that allow the encapsulation of CCCs into aspects. In order to validate the adapted refactorings, we present the refactoring of two enterprise applications developed with Spring achieving the encapsulation of their CCCs into aspects. In this way, the main contribution of this paper is the adaptation of a set of aspect refactorings to be used in Spring/AOP.

The rest of this paper is structured as follows: [Section 2] presents the main differences between Spring/AOP and AspectJ; [Section 3] presents the adaptation of a catalog of aspect refactorings to be used in Spring/AOP; [Section 4] details the refactoring process of two enterprise applications in order to encapsulate their CCCs into aspects; [Section 5] analyzes some related work; and [Section 6] presents the conclusions.

2 Spring/AOP and AspectJ

Spring is a popular framework used to developed enterprise applications. With the goal of creating more modular systems, Spring includes an AOP module based on interceptors and proxies. This AOP module is based on AspectJ, which is the most complete language implementation of AOP. However, there are important differences between Spring/AOP and AspectJ.

Spring/AOP is based on the use of proxies which avoid the need of an explicit weaving step (at loading or building time) such as AspectJ. This kind of approach presents a minor obstacle to adopt AOP because the implementation of the system is achieved in a pure Java environment. Nevertheless, Spring/AOP presents a subset of the features of AspectJ. For example, Spring/AOP only intercepts one kind of join points: the execution of non static public methods. On the other hand, AspectJ allows the definition of other kind of join points such as the execution of a method, the load of classes, the access to a field, or the handling of an exception [Laddad (09)].

As a result of applying an approach based on proxies, the weaving of the aspects with the beans (Java classes with getters and setters used to configured the aspects) in Spring is done at runtime encapsulating the aspects in a Proxy class. This class intercepts the calls to the methods, performs the additional logic of the aspects, and then transmits those calls to the target bean. As a consequence of the use of these mechanisms, the performance of the Spring/AOP modules is lower than the bytecode weaving implemented by AspectJ.

Another important difference is that Spring/AOP restricts the exposure of join points only to public methods declared on the beans. For this reason, the number of potential methods to be advised by the aspects is limited. Since the methods must be declared on the beans, the configuration of the aspect is achieved using the regular syntax definition of beans.

Regarding the programming styles, Spring/AOP presents two alternatives. A configuration style based on XML called "Schema-style" and a style based on annotations called "@AspectJ-style". The Schema style allows the transformation of Java classes to aspects by the specification of the metadata related to the aspect using an XML configuration file. On the other hand, the @AspectJ style allows the annotation of Java classes with Java 5 annotations using a subset of the syntax of AspectJ. The main advantage of the schema style is that it can be used with previous versions to Java 5. However, using this style, the logic of the aspects is more difficult to understand than using the @AspectJ style because in order to know what an aspect does is necessary to examine the configuration file and the Java code. Using the @AspectJ style the logic of the advices and pointcuts remains in the annotated aspect. For this reason, the @AspectJ style is used in the rest of this paper.

3 Spring/AOP Refactorings

In order to encompass a wide range of situations that may occur during the refactoring process a catalog of aspect refactorings is needed. Most of the aspect refactorings proposed in the literature are centered to transform Java code to AspectJ. However, a large number of the developed enterprise applications are implemented in Spring [Walls and Breidenbach(2005)]. The evolution of these systems should be accomplished with a low cost. An alternative to achieve this goal is the use of AOP. Since there is not a specific Spring/AOP catalog, it is necessary to adapt one of the existing AspectJ catalogs. With this goal in mind, in this section a subset of the aspect refactorings presented by Monteiro [Monteiro 04] are adapted to Spring/AOP using the *@AspectJ* syntax [Laddad (09)]. Specifically, the refactorings adapted are those whose purpose is the encapsulation of a CCC into an aspect (called "refactorings for feature extraction" [Monteiro 04]).

Since Spring/AOP only implements a subset of the features of AOP [Laddad (09)], the task of adaption of the refactorings is not trivial. As a result of this constraint most of the aspect refactorings to be adapted need major changes.

As is shown in Table 1 only one refactoring cannot be adopted to Spring/AOP (because of differences in the implementation between Spring/AOP and AspectJ). Additionally, while three refactorings are directly applicable, most of the refactorings needs a major adaptation.

Feasibility	Refactorings
Adapted	Move Method from Class to Inter-type, Extract Fragment into Advice, Encapsulate Implements with Declare Parents, Move Field from Class to Inter-type, Inline Class within Aspect, Extract Feature into Aspect
Directly Applicable	Extract Inner Class to Standalone, Inline Interface within Aspect,as comprise on Change Abstract Class to Interface
Not Applicable	Split Abstract Class between Aspect and Interface

Table 1: Aspect refactorings adapted.

In what follows, the adaptation of the aspect refactorings for feature extraction is presented in detail grouped for the effort demanded by the adaptation and the feasibility of it. For each refactoring, the possibility of applying this refactoring in the context of Spring/AOP is analyzed and the changes in comparison with AspectJ are presented. In order to show the differences between Spring and AspectJ refactorings, we present the changes using the same code examples used by Monteiro [Monteiro 04].

3.1 Adapted Refactorings

In this section the aspect refactorings that were successfully adapted are presented. For each of them the main difficulties found during the adaptation are discussed and the mechanisms to apply the refactorings are presented (code examples are shown for those with major changes).

3.1.1 Move Method from Class to Inter-type

This aspect refactoring encapsulates a method related to a CCC into an aspect. In order to achieve this goal an inter-type declaration is used in AspectJ. That is, this refactoring moves a method into an aspect as an inter-type declaration. This refactoring needs to be modified because Spring/AOP only allows the introduction of new interfaces to beans managed by the framework. For this reason, it is desirable that all members that are accessed by the method will be available on an inner class inside the aspect. To accomplish the encapsulation of the method the next steps should be followed:

1. If the method to be moved does not belong to an interface implemented by the class, an inner interface to the aspect containing the declaration of the method must be created.

2. Create an inner class inside the aspect that implements the interface above-mentioned (add the implements declaration to the interface if the inner class was created in a previously refactoring).

3. Move the method to the new class changing the method access to public if necessary.

4. Add a private static field to the aspect whose type is the same as the interface that contains the method.

5. Add the annotation @DeclareParents to the field. The annotation *value* attribute must be the original class in which the method was defined (in order to introduce the method into the class when it is invoked) and the *defaultImpl* attribute must contains the inner class name.

In order to shown the mechanism of this aspect refactoring the following example in which the method *display()* is encapsulated is presented:

```
public class TangledStack{
  public void display() {//...}
  //...
}
```

The method is deleted from *TangledStack* and it is encapsulated into a new aspect called *DisplayAspect* which contains an interface *Display* and an inner class *DisplayImpl* in which the method is declared. Also, a static variable of type *Display* is added to the aspect in order to create the inter-type declaration using the @DeclareParents annotation.

```
@Aspect
public class DisplayAspect {
  @DeclareParents(value=``TangledStack'', defaultImpl= DisplayImpl.class
      )
  public static Display mixin;
  interface Display {
    void display();
  }
  static class DisplayImpl implements Display {
    public void display(){// ...}
  }
}
```

Finally, the *application-context.xml* file must be changed in order to add the reference to the aspect *DisplayAspect* (Since the refactoring of this file is similar for all the aspect refactorings, they will be omitted in the following examples).

```
<beans ...>
  <bean id=``stack'' class=``TangledStack''>
  </bean>
  < bean class="DisplayAspect" />
</beans>
```

3.1.2 Extract Fragment into Advice

This aspect refactoring is used to encapsulate into an aspect a fragment of code inside a method. So, a pointcut that captures the joinpoint and its context is created and the fragment of code is extracted into an advice in the AspectJ refactoring. As said before, unlike AspectJ, Spring/AOP only allows join points capturing the execution of public methods handled by the framework; hence the cases in which this refactoring can be applied are limited. The cases that can be refactored without problems are those in which the fragment of code is at the beginning or at the end of a method. When the fragment of code is mixed with other statements of the method, it will be necessary to apply OO refactorings in order to modify the code of the method an enable the aspect refactoring application.

In order to facilitate the application of the refactoring it is convenient to start the encapsulation of the fragment of code by extracting it into a method using *Extract Method* [Fowler (99)] and then apply *Move Method from Class to Inter-type* (3.1.1) to move the method to an aspect. In this way, the join point context (such as temporal variables, parameters, or instance variables) is easily captured. Next, the following steps should be accomplished:

1. A method with the @Pointcut annotation mark must be added to the aspect. This pointcut must capture the execution of the method that contains the fragment of code to be extracted.

2. Proper joinpoint context capturing should be ensured (as in AspectJ).

3. A method with the proper advice annotation mark (@Before, @After, @AfterReturning, @AfterThrowing, or @Around) must be added to the aspect. The advice *value* attribute must be the pointcut previously created.

In the next example, we extract the call to method *display()* from the method *push(Object)*. Since this call is at the end of the method, the refactoring can be applied without problem.

```
public class TangledStack{
  private int _top = -1;
  private Object[] _elements;
  public void display(){
    //...
  }
  public void push(Object element){
    _elements[++_top] = element;
    display();
  }
  //...
}
```

First, we define a new aspect (called *WindowView*) with a pointcut that captures all the calls to the *push(Object)* method using the @Pointcut annotation.

Also, with this pointcut the context of the joinpoint is captured by means of the *stack* variable. Finally, since the call to the *display()* method is at the end of the *push(Object)* method, an advice is created with the @AfterReturning annotation and the call to the *display()* method is deleted from the *push(Object)* method.

```
@Aspect
public class WindowView {
  @Pointcut(``(execution(public void TangledStack.push(Object))) && this
      (stack)'')
  private void stateChanged(TangledStack stack) {}
  @AfterReturning(``stateChanged(stack)'')
  public void displayState(TangledStack stack) {
    stack.display();
  }
}
```

3.1.3 Encapsulate Implements with Declare Parents

The goal of this aspect refactoring is the encapsulation of the CCC roles played by the declaration of interfaces. For this reason, the refactoring in AspectJ moves the implements declaration of the interface from the class to an aspect using the *Inter-type declaration* mechanism. In Spring/AOP the mechanism of introductions must be used which is similar to the *Inter-type declaration* mechanism of AspectJ. However, the main limitation in Spring/AOP, with regard to AspectJ, is that it is necessary to move into the aspect all the methods inherited from the interface. To accomplish the encapsulation of the implements declaration the next steps should be followed:

1. Add a private static field to the aspect. The type of the field must be the interface to encapsulate.

2. Add the annotation @DeclareParents to the field. The annotation *value* attribute must be the class that implements the interface. In this way, the interface is introduced into the class avoiding the explicit declaration of the interface into the class.

3. Add an inner class to the aspect that implements the interface. The name of the inner class must be added to the *defaultImpl* value of the @DeclareParents annotation previously created. This indicates that the inner class provides the implementation of the role.

In the next example, we extract the secondary role played by *TargetInterface* from a class.

```
public class class SomeImplementingClass implements TargetInterface {
  //...
}
```

When the aspect refactoring is applied, the implements declaration is removed from the class and the annotation *@DeclareParents* is included in an aspect as is shown in the following code.

```
@Aspect
public class Implementation {
  @DeclareParents(value= ''SomeImplementingClass'',defaultImpl=
      InterfaceImplementation.class)
  private static TargetInterface mixin;

  static class InterfaceImplementation implements TargetInterface {
    // ...
  }
}
```

3.1.4 Move Field from Class to Inter-type

This aspect refactoring encapsulates a field related to a CCC by means of an aspect. The main difference between the AspectJ version of the refactoring and the Spring/AOP one is that in the latter the field must be declared as an instance variable of the aspect (instead of introducing the field into the target class). To apply the refactoring the next steps should be followed:

1. Move field declaration from the class to the aspect. If the field belongs to an internal type of the class, it is necessary to extract the inner class of this type into a new class using *Extract Inner Class to Standalone* (3.2). If the code of the CCC to which the field belongs had been encapsulated into an inner class of the aspect, such as the application of *Encapsulate Implements with Declare Parents* (3.1.3), the instance variable must be encapsulated into the inner class.

2. For each fragment of code in which the code is used, decide if the whole method or only a fragment of it must be moved to the aspect, and apply the appropriate refactoring: *Move Method from Class to Inter-type* (3.1.1) or *Extract Fragment into Advice* (3.1.2).

In contrast with AspectJ, it is not necessary to check at execution time if there is any access to the field from sources external to the aspect. This is because the field is a private instance variable of the aspect (and not of the target class), whereupon errors are detected at compilation time.

For example, consider the variables *_label* and *_text* of the following source code fragment.

```
public class TangledStack {
  private int _top = -1;
  private Object[] _elements;
  public final static int S_SIZE = 10;
  private JLabel _label = new JLabel("Stack ");
  private JTextField _text = new JTextField(20);
  // ...
}
```

If these variables belongs to a CCC, the *Move Field from Class to Inter-type* refactoring should be applied. In this case, the refactoring just move the instance variables into the aspect.

```
@Aspect
public class WindowViewAspect {
  private JLabel _label = new JLabel(``Stack '');
  private JTextField _text = new JTextField(20);
  // ...
}
```

3.1.5 Inline Class within Aspect

This aspect refactoring is applied in order to encapsulate a class into an aspect when the class is only used in that aspect. So, the class is moved into the aspect as an inner class. The only difference with AspectJ is that the new location of the class to be migrated must be updated in the *application-context.xml* file. In this way, the steps of the refactoring are the following:

1. Move the class into an aspect.
2. Change the public visibility of the class to static.
3. Change the attribute *class* of the beans file to reference the new location of the class.

For example, consider the following class.

```
public class OpenNotifier extends Observable {
  private Flower _enclosing;
  private boolean _alreadyOpen = false;
  public OpenNotifier(Flower enclosing) {
    this._enclosing = enclosing;
  }
  //...
}
```

After encapsulating this class into an aspect using the refactoring the result is as follows.

```
@Aspect
public class SomeAspect {
  // ...
  static class OpenNotifier extends Observable { /
    private Flower _enclosing;
    private boolean _alreadyOpen = false;
    public OpenNotifier(Flower enclosing) {
      this._enclosing = enclosing;
    }
    //...
  }
}
```

3.1.6 Extract Feature into Aspect

This is the main refactoring proposed by Monteiro [Monteiro 04] because the goal of it is the extraction of a CCC spread out over several Java elements such as methods, fields. To accomplish this goal, this refactoring uses other refactorings presented in the catalog, such as *Move Method from Class to Inter-type* (3.1.1) or *Extract Fragment into Advice* (3.1.2), and OO refactorings as the proposed by Fowler [Fowler (99)]. For this reason the main constraints to apply this refactoring (i.e. the differences with the AspectJ refactorings) are the same that were presented for the previous aspect refactorings.

3.2 Directly Applicable Refactorings

Some aspect refactorings presented by Monteiro [Monteiro 04] can be easily applied using Spring/AOP without adaptation. They are *Extract Inner Class to Standalone*, *Inline Interface within Aspect*, and *Change Abstract Class to Interface*. The main reason because they are directly applicable into Spring/AOP is the fact that this aspect refactorings do not involve AOP language elements.

3.3 Not Applicable Refactorings

The only aspect refactoring that could not be adapted was *Split Abstract Class between Aspect and Interface* [Monteiro 04]. This refactoring is used when one or more classes inherit from an abstract one impeding the inheritance from another class. Additionally, the abstract class should be changed into an interface because it defines concrete members. The AspectJ solution for this problem is to move all the concrete members defined in the abstract class into an aspect. Then, the abstract class can be changed into an interface.

The Spring/AOP adaptation of this aspect refactoring is not possible mainly because the differences between the implementation of the mechanisms of inter-type declaration of Spring/AOP and AspectJ. While the introduction mechanism of Spring/AOP could be used to provide those methods of the abstract class to its subclasses, Spring/AOP requires the implementation of the interface with all its methods (and in this case, they are methods that have no implementation defined).

A possible alternative is to define a hierarchy of aspects in which an aspect must define the methods of the abstract class and a sub-aspect for each concrete class. These sub-aspect must contain the implementations of the concrete methods and the *introduction* of the interface in the original subclasses. However, this approach is only applicable if the abstract class represents a CCC. In this case, the solution is the refactoring of the whole CCC into a hierarchy of aspects using the aforementioned restructuring.

4 Case Studies

In this section, the refactoring of the CCCs of two enterprise applications (Finance Manager and GridGain) is presented. Both applications use Spring but no aspects are defined. In this way, the goal of this case study is to validate the proposed aspect refactorings by encapsulating the CCCs of these applications using AOP mechanisms.

During the refactoring of a system a wide variety and quantity of refactorings needs to be applied. So, in order to simplify the refactoring process the aspect refactorings for Spring/AOP presented in [Section 3] were implemented in a tool. Specifically, our existing tool called AspectRT [Vidal and Marcos 09, Vidal and Marcos 12] that supports aspect refactorings for AspectJ was extended to support the adapted refactorings. In the following subsection we introduce AspectRT.

Next, the refactoring process of the CCCs of the two enterprise applications is described in detail. In order to find the possibles CCCs of the systems the aspect mining activity was executed by means of fan-in analysis [Marin et al. 04]. This first step has as a result the identification of those method with the highest number of invocations and the classes and methods involved in those calls. Then a manual analysis of these methods was accomplished in order to filter false positives.

In both case studies, for each CCC refactored the following information is presented:

- Description of the CCC.
- Involved fragments of code, methods, classes, etc.
- Aspect refactorings applied.
- Discussion of the refactoring process.

4.1 AspectRT

AspectRT (Aspect Refactoring Tool) is a plug-in for the Eclipse IDE[3] and it is integrated with AspectJ plug-in (AJDT)[4]. AspectRT helps architects, designers, and developers to migrate object-oriented systems to aspect-oriented ones, providing a set of aspect refactorings. The tool is based on graphical wizards that assist the developer during the refactoring process simplifying this task. This tool was initially developed to allow the generation of AOP code to be used in AspectJ. We extended AspectRT to support Spring/AOP refactorings.

[3] http://www.eclipse.org/
[4] http://www.eclipse.org/aspectj/

In the tool the aspect refactorings are automated in a similar fashion to the Eclipse refactoring tool in which, when an element is selected, all applicable refactorings for that element are proposed. Then, when a refactoring is selected it is applied automatically requesting minimal information from the developer when it is necessary.

AspectRT follows a process that relies on a set of steps [Vidal and Marcos 09, Abait and Marcos 09] to accomplish the refactoring of a system. The purpose of these steps are three-fold: encapsulate a CCC into an aspect, enable the extraction of a CCC when it is not possible the application of a refactoring of CCC, and improve the internal structure of aspects.

With the goal of assisting the developer during the refactoring process, artificial intelligence techniques, such as association rules and Markov models, are used in some steps of the process [Vidal and Marcos 12].

4.2 Finance Manager System

Spring Finance Manager[5] is a simple finance application that was built with the goal of demonstrating the Spring framework capabilities. Also, it illustrates the use of good design practices for Spring.

4.2.1 Crosscutting Concerns Refactoring

After the aspect mining phase three CCCs were detected: Null Checking, Constraint Validation, and String Conversion. Next, the details of this CCCs and its refactoring process to Spring/AOP are explained.

Null Checking Concern This CCC checks if a variable is null using the method *notNull(Object,String)* of the class *Assert*. If so, an exception is thrown with a message. All the invocations to this method were found at the beginning of the methods, to verify if one or more variables were null, as is shown in the next code:

```
public class AccountController {
  // ...
  @RequestMapping(value = ''/account'', method = RequestMethod.POST)
  public String create(@ModelAttribute(''account'') Account account,
      BindingResult result) {
    Assert.notNull(account, "Account must be provided.");
    for (ConstraintViolation<Account> constraint : validator.validate(
        account)) {
      //...
    }
    //...
  }
}
```

[5] http://code.google.com/p/spring-finance-manager/

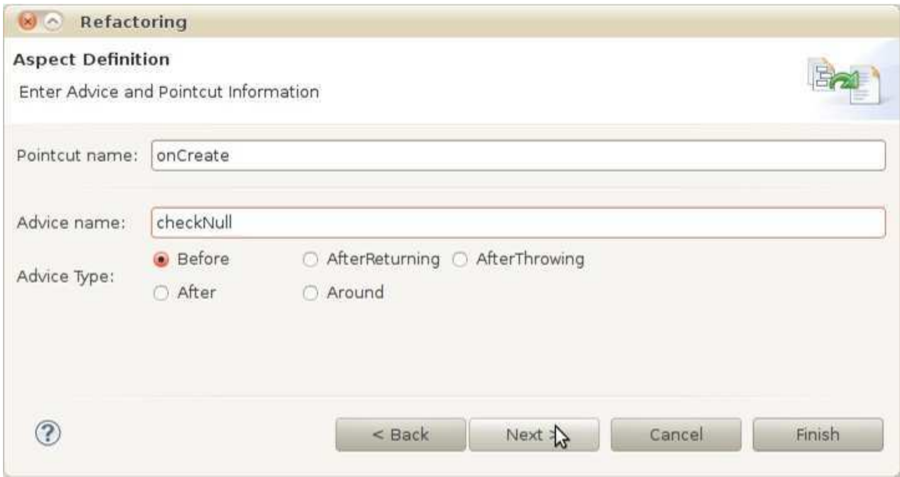

Figure 1: Extract Fragment into Advice wizard.

In total, 21 calls to this method were found distributed in 20 methods in the classes *AccountController*, *PersonController*, and *ProductController* of the package *net.stsmedia.financemanager.web.mvc*. The benefit of refactoring this CCC by its encapsulation into an aspect is twofold, first the code repetition is avoided and second, the methods refactored are simplified because they only deal with the main concern.

To encapsulate this concern into an aspect each call to the method *notNull* is iteratively refactored applying the refactoring *Extract Fragment into Advice* (3.1.2) using the tool [Fig. 1].

As a result of refactoring all the calls to the method an aspect containing all the checking for null parameters is obtained. To avoid code repetition in the aspect some manual refactorings were applied to it. Specifically, is noticed that the parameters that are checked for a null value are of two types: (1) identifiers of type Long or (2) objects that its type contains the Java annotation @Entity. Also, it was noticed that all the checking is done in methods of classes with the suffix *Controller*. In this way the aspect can be summarized in two pointcuts with two advices. Next, the final aspect is shown.

```
@Aspect
class CheckNulls {
  @Pointcut(``(execution(* net.stsmedia.financemanager.web.mvc.*
       Controller.*(..,Long,..))) && args(id,..)'')
  public void onControllerOpId(Long id) {}

  @Before(``onControllerOpId(id)'')
  public void checkNull(Long id) {
     Assert.notNull(id, "identifier must be provided.");
  }
```

```
  @Pointcut(''(execution(*net.stsmedia.financemanager.web.mvc.*
      Controller.*(..,@Entity*,..))) && @args(ann,..) && args(value,..)'
      ')
  public void onControllerOp(Entity ann, Object value) {}

  @Before(''onControllerOp(arg, param)'')
  public void checkNull(Entity arg, Object param) {
    Assert.notNull(param, arg.name()+"must be provided.");
  }
}
```

Constraint Validation Concern This CCC accomplishes the validation of the integrity constraints on an entity to be created or updated. To achieve this validation the set of methods {*validate(), rejectValues(), getPropertyPath(), getMessage()*} and a variable of type *Validator* are used as is shown in the next source code fragment.

```
public class ProductController {
  private Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
  // ...
  @RequestMapping(value = ''/product/cash'', method = RequestMethod.POST
      )
  public String createCash(@ModelAttribute(''cash'') Cash product,
      BindingResult result) {
    for (ConstraintViolation<Cash> constraint:validator.validate(product)){
      result.rejectValue(constraint.getPropertyPath().toString(), "", constraint.getMessage());
    }
    // ...
  }
}
```

This CCC was found in 10 methods with the prefixes create and update in the classes *AccountController*, *PersonController*, and *ProductController*. By the encapsulation of the Constraint Validation Concern the methods in which is found were simplified.

To refactor this concern all the fragments of code involved and the variable *validator* were moved into an aspect. In order to accomplish this task, the aspect refactoring *Move Field from Class to Inter-type* (3.1.4) was applied to encapsulate the variable [Fig. 2]. Also, the refactoring *Extract Fragment into Advice* (3.1.2) was applied to move the fragments of code, of the methods related to the CCC, into an aspect.

Since the declaration of the variable is the same in the three classes, it was only declared once in the aspect. This declaration was automatically detected by the *Move Field from Class to Inter-type* wizard so when it applied the refactoring the field was deleted from the class but it was not declared again in the aspect. The resulting aspect of this refactoring is shown next.

```
@Aspect
class Validation {
  private Validator validator = javax.validation.Validation.
      buildDefaultValidatorFactory().getValidator();
```

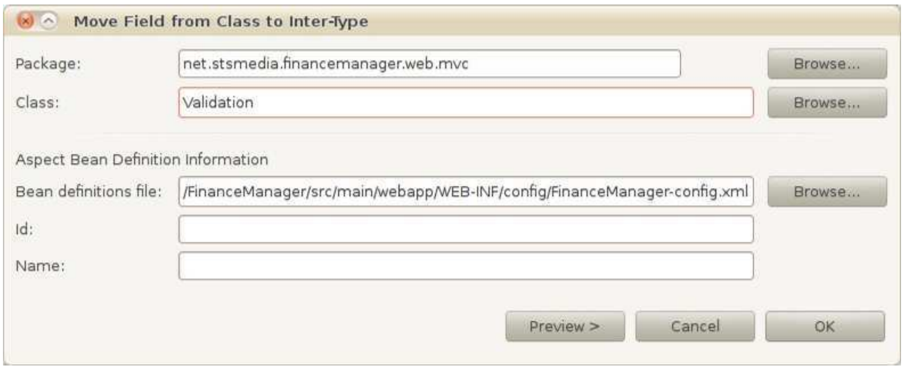

Figure 2: Move Field from Class to Inter-type wizard.

```
  @Pointcut(``(execution(public String net.stsmedia.financemanager.web.
      mvc.*Controller.*(*, BindingResult))) && args(entity, result)'')
  public void onCreate(Object entity, BindingResult result) {}

  @Before(``onCreate(entity, result)'')
  public void adviseCreate(Object entity, BindingResult result){
    for(ConstraintViolation<?> constraint : validator.validate(entity)){
      result.rejectValue(constraint.getPropertyPath().toString(), ``'',
          constraint.getMessage());
    }
  }
}
```

String Conversion Concern This concern checks if a value is null or invalid. Then this value is converted into an object. Specifically this behavior is found in the methods *getAsText()* and *setAsText()* that are inherit from *PropertyEditorSupport* class. The next source code shows an example of the CCC.

```
public class PersonEditor extends PropertyEditorSupport {
  // ...
  public String getAsText() {
    Object obj = getValue();
    if (obj == null) {
      return null;
    }
    return (String) typeConverter.convertIfNecessary(((Person) obj).
        getId(), String.class);
  }
  public void setAsText(String text) {
    if (text == null || text.length() == 0) {
      setValue(null);
      return;
    }
    Long identifier = (Long) typeConverter.convertIfNecessary(text, Long
        .class);
    //...
  }
}
```

Spring/AOP aspect refactoring	No. of Applications
Extract Fragment into Advice	35
Move Field from Class to Inter-type	3
Total	*38*

Table 2: Aspect refactorings applied to Finance Manager System.

The concern was found in 4 occasions, that is, in two classes: *PersonEditor* and *ProductEditor*. The benefits of refactoring this CCC are similar to previous cases since the code repetition is avoided while the legibility of the original methods is improved.

To refactor the String Conversion Concern the four fragments of code found were refactored by applying *Extract Fragment into Advice* (3.1.2). The type of the advice must be *Around* because in some occasion, when the value to set or get is null (or empty) is not necessary the invocation of the target method.

After the refactoring of the four fragments of code some manual refactorings were needed in the final aspect in order to simplified the pointcut expressions.

4.2.2 Refactoring Results

During the refactoring process of this system the refactorings for Spring/AOP applied were *Extract Fragment into Advice* (3.1.2) and *Move Field from Class to Inter-type* (3.1.4) [Tab. 2]. These refactorings, as was said in [Section 3.1], needed to be adapted with several changes regarding the AspectJ version.

An important fact to remark is that the Finance Management System has a suite of unit tests. We executed the test cases before and after refactoring the system in order to validate the aspect refactorings applied and the preservation of the behavior of the system.

4.3 GridGain

GridGain[6] is a Java platform for the development of cloud computing applications. This system uses internally the Spring framework and allows the development of applications that can use all the features of Spring (as for example the dependency injection mechanism). For these reasons GridGain is a good case study for this work.

4.3.1 Crosscutting Concerns Refactoring

Six CCCs were discovered in this system: Logging, Bean Management, Marshalling, Arguments Checking, Before/After Calls, and Bean Registration. Next

[6] http://www.gridgain.com/

the discussion of the refactoring of each CCC to AOP is presented.

Logging Concern This CCC deals with the logging of messages in a large number of methods across the whole system using the methods provided by the interface *GridLogger*. For example, a consistent behavior can be observed in the *start()* methods implemented in the three classes that inherit from *GridDeploymentStore*. At the beginning of these methods the no crosscut functionality is executed and next is executed the logging code. This situation is shown in the following code.

```
class GridDeploymentLocalStore extends GridDeploymentStoreAdapter
    implements GridDeploymentStore {
  //...
  public void start() throws GridException {
    spi.setListener(new LocalDeploymentListener());
    if (log.isInfoEnabled() == true) {
      log.info(startInfo());
    }
  }
}
```

In total, 104 instances of this CCC were found in the methods *start()* and *stop()* defined by the 3 subclasses that inherit from *GridDeploymentStore* and 11 subclasses that extend from *GridManagerAdapter*. Also, the Logging concern was found in the methods *spiStart()* and *spiStop()* that are defined in 38 subclasses of *GridSpiAdapter*. The refactoring of this concern centralizes all the calls to the data logger into a single aspect avoiding the tangled code.

To encapsulate this CCC into an aspect the aspect refactoring *Extract Fragment into Advice* (3.1.2) must be applied over all the statements in which a message is logged. Also, the methods used to obtain the message to be logged (called *startInfo()* and *stopInfo()*) are also encapsulated into an aspect using the aspect refactoring *Move Method from Class to Inter-type* (3.1.1). This is done because these methods are only called from the aspect.

Next a fragment of the code of the final aspect is shown. To obtain it, a few manual restructurings were applied in order to properly obtain the *log* variable and to simplify the pointcut expressions.

```
@Aspect class LoggingStart {
  @Pointcut(``(execution(public void org.gridgain.grid.spi.
      GridSpiAdapter.spiStart(String))) && this(_this) && args(gridName)
      '')
  public void onSpiStart(GridSpiAdapter _this, String gridName) {}

  @After(``onSpiStart(_this, gridName)'')
  public void adviseSpiStart(GridSpiAdapter _this, String gridName)
      throws GridSpiException {
    if (_this.getLog().isInfoEnabled() == true){
      _this.getLog().info(startInfo());
    }
  }

  public final String startInfo() {
```

```
    return ``Manager started ok: '' + getClass().getName();
  }
}
```

The *log* variable is not in the final aspect. That is because it was not possible to fully modularize all the code found for the Logging concern. The logging statements that were not restructured presents an inconsistent behavior, for example, in some cases the logging is executed during a calculation. This type of logging can not be extracted into an aspect therefore it remains unchanged in the base code.

Bean Management Concern This CCC records in a *startTstamp* field the starting time of the services represented by the class *GridSpiAdapter* using the method *startStopwatch()*. All the invocations to this method are done in the first statement of the method *spiStart()* in the subclasses of *GridSpiAdapter* as is shown in the following code.

```
public void spiStart(String gridName) throws GridSpiException {
  startStopwatch(); // Start SPI start stopwatch.
  assertParameter(dataSource = null, "dataSource= null'');
  //...
}
```

Also, it is important to analyze the *startTstamp* variable in which the method *startStopwatch()* records information. This variable is also accessed by three methods of the interface *GridSpiManagementMBean*. This interface is implemented by all the subclasses of *GridSpiAdapter* and represents a CCC used to implements JMX[7] and should be encapsulated.

To summarize, the CCC is spread over *spiStart()* method which is implemented in the 38 subclasses of *GridSpiAdapter*, the method *startStopwatch()*, and 11 methods of the interface *GridSpiManagementMBean*.

To refactor this concern the calls to *startStopwatch()* were encapsulated into an aspect using *Extract Fragment into Advice* (3.1.2). Also, the implementation of the interface *GridSpiManagementMBean* with all its methods and fields were refactored applying *Encapsulate Implements with Declare Parents* (3.1.3).

Once all the refactorings are applied, some manual restructurings are needed in order to troubleshoot casting issues in the advices. The final aspect is shown below.

```
@Aspect
class BeanManagement {
  @DeclareParents(value = ``org.gridgain.grid.spi.GridSpiAdapter'',
      defaultImpl = GridSpiManagementMBeanImpl.class)
  private static GridSpiManagementMBean mixin;

  class GridSpiManagementMBeanImpl implements GridSpiManagementMBean {
    private long startTstamp = 0;
    private GridSpiInfo spiAnn = null;
```

[7] Java Management Extensions

```
    protected void startStopwatch() {
      startTstamp = System.currentTimeMillis();
    }

    public final String getAuthor() {
      return spiAnn.author();
    }
    public final String getVendorUrl() {
      return spiAnn.url();
    }
    // ...
  }

  @Pointcut(``(execution(public void org.gridgain.grid.spi.
      GridSpiAdapter.spiStart(String))) && this(_this) && args(gridName)
      '') public void onSpiStart(GridSpiManagementMBean _this, String
      gridName) {}

  @Before(``onSpiStart(_this, gridName)'')
  public void callStopwatch(GridSpiManagementMBean _this, String
      gridName) {
    ((GridSpiManagementMBeanImpl)_this).startStopwatch();
  }
}
```

Marshalling Concern This CCC transforms the objects into a proper format when they needs to be sent across a net (this process is known as marshalling). This functionality is required whenever a communication between the nodes involved in a distributed application is needed. This concern was found in the invocations to the methods *marshal()* an *unmarshal()* defined by the class *GridMarshalHelper*. Examples of these calls are shown in the following methods.

```
public void sendMessage(GridNode destNode, Serializable msg) throws
    GridSpiException {
  // ...
  if (nodeId.equals(destNode.getId()) == true) {
    // ...
  } else { try {
    GridByteArrayList buf =
    GridMarshalHelper.marshal(marshaller,
    new GridMuleCommunicationMessage(nodeId, msg));
    muleClient.dispatch(addr, buf.getArray(), null);
  } catch (GridException e) { // ... } }
}
```

```
private void handleException(GridJobExecuteRequest req, GridException ex
    , long endTime) {
  //...
  try {
    GridJobExecuteResponse jobRes = new GridJobExecuteResponse(locNodeId
        , req.getSessionId(), req.getJobId(),
        GridMarshalHelper.marshal(marshaller, ex) ,
        GridMarshalHelper.marshal(marshaller, null) ,
        GridMarshalHelper.marshal(marshaller, Collections.emptyMap()) , false);
    //...
  }
  catch (GridException e) {
    //...
  }
}
```

In total, 39 invocations were found to the *marshal()* method and 24 to the *unmarshal()* method. However, it was not possible to identify any pattern or consistent behavior in their use. For this reason the Marshalling concern cannot be refactored into an aspect oriented solution.

Argument Checking This CCC verifies if an argument fulfills a set of conditions such as not to be null or be within a range. The following code shows how the methods *checkNull()* and *checkRange()* of the class *GridArgumentCheck* are used for this task.

```
public GridNode getBalancedNode(GridTaskSession ses, List<GridNode> top,
      GridJob job) throws GridException {  GridArgumentCheck.checkNull(
    ses, ''ses'');
  GridArgumentCheck.checkNull(top, ''top'');
  GridArgumentCheck.checkNull(job, ''job'');
  GridArgumentCheck.checkRange(top.isEmpty() == false, "top.isEmpty() == false");
  // ...
}
```

For this concern 5 calls were found to the *checkNull()* method and 20 to *checkRange()*. However, no pattern of the use of this CCC could be inferred because each method has a unique logic to decide which checks are needed. Specifically, this occurs in the use of the *checkRange()* method for which the values depend only on the variables that are checked. This situation can be seen in the highlighted statement of the method shown. In the case of the *checkNull()* method, an aspect could be created that verifies that any parameter of a method is null but clearly this solution would not preserve the original behavior. For these same reasons this concern can not be refactored using AspectJ.

Before/After Calls Concern This CCC ensures the proper access and modification of the class *GridKernal* by means of a protocol of mutual exclusion. This protocol lies in the invocation of the methods *beforeCall()*, at the beginning of the operation, and the invocation of the method *afterCall()* at the end of the operation. An example of this protocol is shown in the next code.

```
public Collection<GridNode> getAllNodes() {
  beforeCall();
  try {
    return mgrReg.getDiscoveryManager().getAllNodes();
  } finally {
    afterCall();
  }
}
```

A total of 29 invocations to the methods *beforeCall()* and *afterCall()* were found in the class *GridKernal*. By the encapsulation of the concern Before/After Calls the protocol of mutual exclusion will be moved into an aspect leaving only the main concern in the methods.

To encapsulate this CCC into an aspect the aspect refactoring *Extract Fragment into Advice* (3.1.2) must be applied to the invocations of *beforeCall()* and *afterCall()*. Even though the methods *beforeCall()* and *afterCall()* should also be moved into the aspect, this is not possible because these methods use instance variables that belong to the main concern and are used by other methods. This refactoring could be performed properly with AspectJ because the pointcut and advice mechanism of this language allow the access to instance variables of a class.

After the application of the aforementioned refactorings the visibility of the the methods *beforeCall()* and *afterCall()* should be changed to private to public.

Bean Registration Concern This CCC registers a service of GridGain as a JMX bean when the service is starting and unregisters it when it is ended. For this task the methods *registerMBean()* and *unregisterMBean()* of the class *GridSpiAdapter* are used. Following, an example of this concern is shown.

```
public void spiStart(String gridName) throws GridSpiException {
  startStopwatch();
  if (log.isInfoEnabled() == true) {
    log.info(configInfo(``cacheName'', cacheName));
  }
  cache = CacheFactory.getCache(cacheName);
  if (cache == null) { // ... }
  registerMBean(gridName, this, GridCoherenceCheckpointSpiMBean.class);
}
public void spiStop() throws GridSpiException {
  cache = null;
  unregisterMBean();
}
```

Besides the methods *registerMBean()* and *unregisterMBean()* that compose this concern, the 38 invocations to these methods must be refactored. These invocations were found in the methods *spiStart()* and *spiStop()* that are located into the subclasses of *GridSpiAdapter*. In addition, the instance variables *spiMBean* and *jmx* are also related with the concern because they are only used by the registration methods.

The refactoring of this concern is done by applying the aspect refactorings *Move Method from Class to Inter-type* (3.1.1), *Move Field from Class to Inter-type* (3.1.4), and *Extract Fragment into Advice* (3.1.2).

Since this CCC is related with the Bean Management concern the code was encapsulated into the same aspect of the latter.

4.3.2 Refactoring Results

During the refactoring of the CCCs of GridGain 215 aspect refactorings were applied. Most of then were *Extract Fragment into Advice* (3.1.2) refactorings as is shown in [Tab. 3]. However, three other kinds of refactorings were applied too. Similarly to the results of Finance Manager, the set of Spring/AOP aspect refactorings applied are those that needed to be adapted with several changes regarding the AspectJ version. That is, it was not necessary to apply other

Aspect refactoring	No. of Applications
Extract Fragment into Advice	208
Move Field from Class to Inter-type	2
Move Method from Class to Inter-type	4
Encapsulate Implements with Declare Parents	1
Total	*215*

Table 3: Aspect refactorings applied to GridGain System.

refactorings than those that were adapted. We think that more experimentation is needed with different CCCs in order to know how often are used the no applied refactorings.

5 Related Work

Most of the studies of migrating object-oriented systems to aspect-oriented ones have been centered on the AspectJ language. In this way, Hanneman and Kiczales [Hannemann and Kiczales 02] proposes the refactoring of the CCCs found in the implementation of several design patterns. For a set of patterns taken from the GoF's book [Gamma et al. (95)], the crosscutting code is analyzed and a solution based on AspectJ is proposed. Tonella and Ceccato [Tonella and Ceccato 05] presents an approach restricted to refactoring scattered methods declared by interfaces (called aspectizable interfaces) and to encapsulating portions of code by means of pointcuts. Similar to our work, this work uses a small set of aspect refactoring to perform the restructuring. Monteiro and Fernandes [Monteiro and Fernandes 08] runs an example of the refactoring of a sample application using the catalog of refactorings presented by Monteiro [Monteiro 04] for AspectJ. Marin et al. [Marin et al. 09] propose a refactoring strategy based on crosscutting concern sorts. Once the CCC are described by means of concern sorts, they are refactored through interaction with the developer. Also, in this group of studies based on AspectJ, some authors present case studies of enterprise applications such as [Mesbah and van Deursen 05, Marin et al. 07].

Finally, some authors describe aspect-oriented refactorings using Spring/AOP. Laddad [Laddad (09)] describes several specific uses of aspect orientation in the development of enterprise application using Spring. However, refactorings that can be used for encapsulated CCC into an aspect in different circumstances are not presented. Similarly, Ghag [Ghag 07] discusses the implementation of commons CCC (such as, logging, security, and transactionality) on Spring/AOP.

6 Conclusion

In this article, we presented the adaptation of a set of aspect refactorings, initially presented to AspectJ, to be used in Spring/AOP. For each refactoring, we have explained the main differences with AspectJ, specified the application of the refactoring, and presented an example of its application. The adaption of these refactorings is a relevant task because it improves the time consumed by the refactoring process since there was no specific aspect refactoring catalog for Spring/AOP.

We validated the aspect refactorings by conducting two case studies in which two enterprise applications were refactored. We found that the aspect refactoring most used during the refactoring process were those that needed more changes regarding their AspectJ version.

In regards to the percentage of CCCs that could be encapsulated, more than 75% of the total CCCs were refactored by means of Spring/AOP. The main reason why the remaining CCCs can not be encapsulated is because the limited set of joinpoints supported by Spring/AOP. Nevertheless, we found that one advantage that simplified the refactoring process was the fact that most of the instances of a same CCC share a pattern code. For this reason, as future work, we will try to evaluate the refactoring of enterprise systems, specially, those whose CCC structure is not regular. We think that in these cases major refactorings will be needed before the encapsulation of the concerns into aspects. That is why it must be assessed in advance the costs and benefits involved in the encapsulation of the CCCs. Another future work is the analysis of the need of specific Spring/AOP refactorings other than those that have already been defined for AspectJ.

Acknowledgements

We would like to thank Guillermo Zunino who collaborated in the implementation and evaluation of the aspect refactorings described in this article.

References

[Abait and Marcos 09] Abait, E., Marcos, C.: "Combining aspect mining techniques based on crosscutting concern sort"; III Latin American Workshop on Aspect-Oriented Software Development (23th BSSE); 18–23; Fortaleza, Brazil, 2009.

[Chidamber and Kemerer 94] Chidamber, S., Kemerer, C.: "A metrics suite for object oriented design"; IEEE Transactions on Software Engineering; 20 (1994), 6, 476–493.

[Elrad et al. 01] Elrad, T., Filman, R. E., Bader, A.: "Aspect-oriented programming: Introduction"; Commun. ACM; 44 (2001), 10, 29–32.

[Fowler (99)] Fowler, M.: Refactoring: improving the design of existing code; Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Gamma et al. (95)] Gamma, E., Helm, R., Johnson, R.: Design Patterns. Elements of Reusable Object-Oriented Software.; Addison-Wesley Longman, Amsterdam, 1995.

[Garcia et al. 05] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: "Modularizing design patterns with aspects: a quantitative study"; AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development; 3–14; ACM, NY, USA, 2005.

[Ghag 07] Ghag, G.: "Implement crosscutting concerns using Spring 2.0 AOP"; Javaworld. http://www.javaworld.com/javaworld/jw-01-2007/jw-0105-aop.html; (2007).

[Hannemann and Kiczales 02] Hannemann, J., Kiczales, G.: "Design pattern implementation in Java and AspectJ"; SIGPLAN Not.; 37 (2002), 11, 161–173.

[Iwamoto and Zhao 03] Iwamoto, M., Zhao, J.: "Refactoring aspect-oriented programs"; The 4th AOSD Modeling With UML Workshop, UML'2003; ACM, 2003.

[Johnson (03)] Johnson, R.: Expert One-on-one J2EE Design and Development; Wiley & Sons, 2003.

[Kayal (08)] Kayal, D.: Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework; Apress, 2008.

[Kellens et al. 07] Kellens, A., Mens, K., Tonella, P.: "A survey of automated code-level aspect mining techniques"; Transactions on Aspect-Oriented Software Development (TAOSD); IV (2007), Springer-Verlag.

[Kiczales et al. 97] Kiczales, G., Lamping, J., Mendheka, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.: "Aspect-Oriented Programming"; Proceedings of the European Conference on Object-Oriented Programming (ECOOP); number 1241 in Lecture Notes in Computer Science; Springer, Finland, 1997.

[Laddad (09)] Laddad, R.: AspectJ in Action: Enterprise AOP with Spring Applications; Manning Publications Co., Greenwich, CT, USA, 2009; 2nd edition.

[Lopez-Herrejon and Apel 07] Lopez-Herrejon, R., Apel, S.: "Measuring and characterizing crosscutting in aspect-based programs: basic metrics and case studies"; Proceedings of FASE'07; 423–437; Springer-Verlag, Berlin, 2007.

[Malta and de Oliveira Valente 09] Malta, M., de Oliveira Valente, M.: "Object-oriented transformations for extracting aspects"; Inf. Softw. Technol.; 51 (2009), 1, 138–149.

[Marin 04] Marin, M.: "Refactoring JHotdraw's undo concern to AspectJ"; Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE2004).; 2004.

[Marin et al. 07] Marin, M., Deursen, A., Moonen, L.: "Identifying crosscutting concerns using fan-in analysis"; ACM Trans. Softw. Eng. Methodol.; 17 (2007), 1, 1–37.

[Marin et al. 09] Marin, M., Deursen, A., Moonen, L., Rijst, R.: "An integrated crosscutting concern migration strategy and its semi-automated application to JHotdraw"; Automated Software Eng.; 16 (2009), 2, 323–356.

[Marin et al. 04] Marin, M., van Deursen, A., Moonen, L.: "Identifying aspects using fan-in analysis"; WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering; 132–141; IEEE Computer Society, Washington, USA, 2004.

[Mens and Tourwe 04] Mens, T., Tourwe, T.: "A survey of software refactoring"; IEEE Trans. Softw. Eng.; 30 (2004), 2, 126–139.

[Mesbah and van Deursen 05] Mesbah, A., van Deursen, A.: "Crosscutting concerns in J2EE applications"; Proceedings of WSE '05; 14–21; IEEE Computer Society, Washington, USA, 2005.

[Monteiro and Fernandes 08] Monteiro, M., Fernandes, J.: "An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms"; Softw. Pract. Exper.; 38 (2008), 4, 361–396.

[Monteiro 04] Monteiro, M. P.: "Catalogue of refactorings for aspectj"; Technical Report UM-DI-GECSD-200402; Universidade do Minho (2004).

[Parnas 72] Parnas, D. L.: "On the criteria to be used in decomposing systems into modules"; Commun. ACM; 15 (1972), 12, 1053–1058.

[Sirbi and Kulkarni 10] Sirbi, K., Kulkarni, P. J.: "Aspect oriented software metrics-an empirical study"; International Journal of Computer Applications; 7 (2010), 4, 17–22; published By Foundation of Computer Science.

[Tonella and Ceccato 05] Tonella, P., Ceccato, M.: "Refactoring the aspectizable interfaces: An empirical assessment"; IEEE Transactions on Software Engineering; 31 (2005), 10, 819–832.

[Vidal and Marcos 09] Vidal, S., Marcos, C.: "Un proceso iterativo para la refactorizacin de aspectos"; Revista Avances en Sistemas e Informtica; 6 (2009), 1, Escuela de Ingeniera de Sistemas. Facultad de Minas.

[Vidal and Marcos 12] Vidal, S., Marcos, C.: "Building an expert system to assist system refactorization"; Expert Systems with Applications; 39 (2012), 3, 3810 – 3816.

[Walls and Breidenbach(2005)] Walls, C., Breidenbach, R.: Spring in Action; Manning, 2005.