Composition and Categorization of Aspect-Oriented Design Patterns

Radoslav Menkyna
Softec, s.r.o.
Kutuzovova 23, 83103 Bratislava 3, Slovakia
radoslav.menkyna@softec.sk

Valentino Vranić and Ivan Polášek
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology
Ilkovičova 3, 84216 Bratislava 4, Slovakia
vranic@fiit.stuba.sk, ipo@gratex.com

***Abstract*—This paper presents a composition of four particular aspect-oriented design patterns: Policy, Cuckoo's Egg, Border Control and Exception Introduction. The composition is studied in the context of the class deprecation problem in team development. Each of these four patterns is a representative of one of the three structural categories of aspect-oriented design patterns: pointcut, advice, and inter-type declaration pattern category. Although aspect-oriented patterns mostly can be composed with one another without having to modify the code of the pattern that has been applied first, this is not always so. Based on the structural categorization of aspect-oriented design patterns, a regularity in their sequential composition is uncovered and discussed in general and within a detailed example of Policy, Cuckoo's Egg, Border Control, and Exception Introduction composition and further examples of aspect-oriented design pattern compositions.**

I. Introduction

Although the notion of pattern in its original sense proposed by Alexander was indivisible of the notion of pattern language [1], software patterns are often perceived as more or less independently applied sublimated pieces of development experience [4]. Having it this way, we tend first to discover new patterns and then think of the opportunities of their composition rather then to aim at discovery of integral pattern languages that inherently comprise the ties between the patterns. This is so with object-oriented design patterns, and we may see this also applies to aspect-oriented design patterns that are just being discovered both on individual basis [10], [14], [16] and as pattern languages [7].

There are already a significant number of aspect-oriented design patterns discovered. Here we will go through a composition of four particular aspect-oriented design patterns towards some general assumptions on aspect-oriented design pattern composition based on their structure.

Aspect-oriented design patterns discussed here are related to the mainstream aspect-oriented approach established by PARC [9] whose main programming language representative is AspectJ. Numerous other aspect-oriented languages, such as AspectC++, AspectS, or Weave.NET, follow this paradigm. Most of existing frameworks that provide aspect-oriented programming support, such as Spring, JBoss, or Seasor, also follow the PARC approach.

The rest of the article is organized as follows. First, Section II states the problem of class deprecation in team development which we will use to illustrate the pattern composition. Section III describes the structure of the four specific aspect-oriented design patterns that can help in solving the class deprecation problem, and introduces a structural categorization of aspect-oriented design patterns. Section IV shows how these patterns can be actually composed to solve the class deprecation problem. Based on the structural categorization of aspect-oriented design, Section V devises a regularity in the sequential composition of aspect-oriented design patterns and discusses it. Section VI presents an overview of related work. Finally, in Section VII, we make some conclusions and indicate directions of further work.

II. Overcoming the Class Deprecation Problem in Team Development

This section will define the class deprecation problem on which we will study application and composition of patterns. Team development of software requires developers to obey some common rules and policies. A frequent example is the introduction of a new version of a class is to the framework used by application programmers. Sometimes, the old version of a class cannot be simply replaced with the new one at once.

All developers should be kept informed of the new class version and warned—or sometimes even forced—to use it. Just instructing developers to do so simply doesn't work. Developers often forget to obey policies or they overlook the information about a new class version. A better way is to incorporate this information into the build process. Compiler messages—warnings and errors—that notify developers of broken policies and rules have a better chance not to be overlooked.

In cases when the policy must be strictly fulfilled, the more radical steps may have to be taken. By introducing a new version of a class into the framework, its former version becomes deprecated. It would be useful not just to inform developers they are not allowed to use the old version any more, but also to automatically detect all attempts to instantiate the old class and swap them with the new class instantiation.

III. Structure of Aspect-Oriented Design Patterns

The main construct in PARC aspect-oriented programming is an aspect. It consists of pointcuts, which specify the join

points the aspect affects, advices that implement the affecting functionality, and inter-type declarations that statically affect types by introducing new fields and methods into them, inheritance relationship, warnings, compile errors, softened exceptions, and annotations. Here, we will take a closer look at four aspect-oriented design patterns whose composition we will study in further sections.

A. Border Control

The Border Control pattern [14] is used to define regions in the application. These regions are intended for use by other aspects to ensure they are applied only to appropriate places. In case of system changes, only declarations of regions in the Border Control aspect should be changed and the aspects using these declarations will be automatically redirected. As shown in Fig. 1, the Border Control pattern can be implemented by a single aspect consisting only of the pointcuts that define the regions. Regions may represent types or methods. For this, **within**() and **withincode**() primitive pointcuts are used, respectively.

```
public aspect MyRegions {
    public pointcut myTypes1(): within(mypackage1.+);
    public pointcut myTypes2(): within(mypackage2.+);
    public pointcut myTypes(): myTypes1() || myTypes2();
    . . .
}
```

Figure 1. The Border Control pattern.

B. Cuckoo's Egg

The Cuckoo's Egg pattern [14] enables to put another object instead of the one that the creator expected to receive, much similar to what a cuckoo does with its eggs. The pattern is implemented by an aspect that consists of a pointcut that captures constructor calls of the object to be swapped and an advice that actually does the swapping by simply creating and returning another object.

Figure 2 shows an example code of the Cuckoo's Egg pattern. Several types can be covered by swapping (the code in the figure shows two classes, MyClass1 and MyClass2) with constructor calls restricted with respect to where they occur. Note that the swapping object must be a subtype of the original object class; otherwise, we will get a class cast exception on the first attempt to instantiate the original class.

C. Policy

The main idea of the Policy pattern [14] is to define some policy or rules within the application. A breaking of such a rule or policy involves issuing a compiler warning or error. This is very useful in projects that involve many developers. The Policy pattern can be implemented by a single or several aspects. A single aspect approach is used to define project-wide rules or policies. If local rules or exceptions have to be addressed as well, project-wide rules and policies should be defined in an abstract aspect with an abstract pointcut. This

```
public aspect MyClassSwapper {
    public pointcut myConstructors():
        call(MyClass1.new()) || call(MyClass2.new());

    Object around(): myConstructors() {
        return new AnotherClass();
    }
}
```

Figure 2. The Cuckoo's Egg pattern.

pointcut is overridden in concrete aspects that inherit from the abstract aspect in order to implement local policies and rules [14].

```
public abstract aspect GeneralPolicy {
    protected abstract pointcut warnAbout();

    declare warning: warnAbout(): "Warning...";
}

public aspect MyAppPolicy extends GeneralPolicy {
    protected pointcut warnAbout():
        call(* *.myMethod(..)) || call(* *.myMethod2());
}
```

Figure 3. The Policy pattern.

D. Exception Introduction

If an advice calls a method that draws a checked exception, it is forced to cope with it. Sometimes, it is not possible to handle the exception in the advice, so it has to be drawn to a higher context. However, in AspectJ an advice cannot declare throwing a checked exception unless the advised joint point declared this exception, which is unlikely since base concerns are mostly not expected to be adapted to their aspects. The Exception Introduction pattern [10] shown in Fig. 4 solves this problem by catching a checked exception and wrapping it into a new concern-specific runtime exceptions. Such exceptions can be then thrown to a higher context where they can be unwrapped and the real cause of exception revealed [10].

E. Aspect-Oriented Design Pattern Categories

Each aspect-oriented design pattern comprises at least one aspect. By studying available aspect-oriented design patterns, one may notice that in the aspects of each pattern one of the three main parts of an aspect, i.e. a pointcut, advice, or inter-type declarations, prevails in achieving the purpose of the pattern. According to the element that dominates the structure of the aspects that implement them, aspect-oriented design patterns can be divided into three categories: pointcut patterns, advice patterns, and inter-type declaration patterns. Each of the patterns introduced so far is a representative of one of these categories. In the following text, we present further examples of patterns and categorize them.

```
public abstract aspect ConcernAspect {
    abstract pointcut operations();

    before(): operations() {
        try {
            concernLogic();
        } catch (ConcernCheckedException ex) {
            throw new ConcernRuntimeException(ex);
        }
    }
    void concernLogic() throws ConcernCheckedException {
        . . .
    }
}
```

Figure 4. The Exception Introduction pattern (adapted from [10]).

Pointcut Patterns: The Border Control pattern described in Section III-A is an example of a pointcut pattern. It actually contains no other elements than pointcuts. Other examples of pointcut patterns include Wormhole and Participant. The Wormhole pattern [10] employs pointcuts to connect a method callee with a caller in such a way that they can share their context information. It creates a direct connection between two levels in the call stack. This is very helpful when additional context information has to be added [10]. Without this pattern we would have to add extra parameters to each method in the control flow or to use a global storage.

Usually, aspects introduce some behavior to base concerns in such a way that the base concern is not aware of the aspect. In the Participant pattern [10], the roles swap: a class decides whether it will allow an aspect to affect it by declaring an appropriate pointcut. This may be useful when it is not possible to capture classes and methods that have to be affected by an aspect with the pointcut language in a reasonable way. For example, if an advice should affect only methods with some properties not reflected in their names, it is not possible to capture them by a pointcut other by literally listing them.

Advice Patterns: The Cuckoo's Egg pattern described in Section III-B is an example of an advice pattern. Other examples of advice patterns include Worker Object Creation and Exception Introduction. The Worker Object Creation pattern [10]—also known as Proceed Object [16]—captures the original method execution into a runnable object. This way it may be manipulated further. A typical use is to post its execution to another thread. This is very useful with Java Swing framework where all calls that update the GUI must be performed inside the event dispatch thread. Another example is improving responsiveness of GUI applications [10]. This pattern can also be used to advise the call to **proceed**(). This is desired when an aspect contains an around advice whose execution should be, for example, traced or logged [16].

Inter-Type Declaration Patterns: The Policy pattern described in Section III-C is an example of an inter-type declaration pattern. Another example of an inter-type declaration pattern is Default Interface Implementation [10] which employs inter-type declarations to introduce fully implemented methods into interfaces.[1] The classes that implement these interfaces inherit the method implementations and do not have to provide their implementation if the default one is satisfactory.

IV. Composing Aspect-Oriented Design Patterns

This section will show how Policy, Border Control, Cuckoo's Egg, and Exception Introduction can be composed to solve the class deprecation problem presented in Section II.[2] Suppose OldClass is deprecated. In our first approach to this problem we assume it is sufficient to issue a warning in case of deprecated class named instantiation. Developers are supposed to manually change to NewClass. Figure 5 shows how the Policy pattern can be applied to achieve this. The aspect Warning will detect every call to the OldClass constructor and show the provided warning text during compilation. Note that despite AspectJ 5 supports declaring annotations, so a standard @deprecated annotation declaration could have been introduced instead of a general warning with a custom message, this approach was not used because annotations declarations are made on type patterns, not pointcuts, which significantly limits the flexibility.

```
public aspect Warning {
    declare warning: call(*.OldClass.new()):
            "Class OldClass deprecated.";
}
```

Figure 5. Capturing instantiations of a deprecated class with the Policy pattern.

Subsequently, we realize that we have to allow the use of OldClass within the testing package and third party code. In this situation, the Border Control pattern (Section III-A) can be applied. This pattern defines regions in the application that can be used by other design patterns or aspects. Figure 6 presents an application of the Border Control design pattern to our problem. The aspect defines three public pointcuts which represent regions in our application. Afterwards, we will have to adapt the OldClassDeprecation aspect as shown in Fig. 7. By this, we actually composed a Policy with an existing Border Control.

Border Control is a pointcut pattern (Section III-A), while Policy is an inter-type declaration design pattern (Section III-C). As we saw in the example, composing a pointcut pattern with an existing inter-type declaration pattern requires changes in the existing inter-type declaration pattern. If we knew from the beginning there will be exemptions from banning the use of OldClass, it would be possible to apply the Border Control pattern first and the Policy pattern could be then added without having to change the existing code. This suggests that composing an inter-type declaration pattern with

[1] Laddad actually introduces Default Interface Implementation as an AspectJ idiom [10].

[2] Some preliminary results regarding the possibilities of composing aspect-oriented design patterns have been published in our earlier work [13].

```
public aspect Regions {
    public pointcut Testing():
        within(com.myapplication.testing.+);
    public pointcut MyApplication():
        within(com.myapplication.+);
    public pointcut ThirdParty():
        within(com.myapplication.thirdpartylibrary.+);
    public pointcut ClassSwitcher():
        within(com.myapplication.ClassSwitcher);
}
```

Figure 6. The Border Control pattern used to partition code into regions.

```
public aspect Warning {
    protected pointcut allowedUse():
        Regions.ThirdParty() || Regions.Testing();

    declare warning: call(Display.new()) && !allowedUse():
        "Class OldClass deprecated.";
}
```

Figure 7. Composing Policy with Border Control.

an existing pointcut pattern can be performed without having to change the existing pattern.

Assume now we would like to make a change from OldClass to NewClass automatic while still keeping developers informed of attempts to instantiate OldClass outside of the testing package and third party code. This may be achieved with the Cuckoo's Egg pattern (Section III-B). This pattern captures calls to a constructor of a particular class and employs an around advice to replace each such call with a call to a constructor of another class.

Figure 8 shows how Cuckoo's Egg may be applied to replace OldClass constructions with NewClass construction. A Cuckoo's Egg pattern uses the pointcuts defined in an existing Border Control pattern. Cuckoo's Egg is an advice design pattern and it was composed with Border Control without having to change it. This suggests that composing an advice pattern with an existing pointcut pattern can be made without changes in the existing pointcut pattern.

```
public aspect ClassSwitcher {
    public pointcut oldClassConstructor():
        call(*.OldClass.new()) &&
        !Regions.ThirdParty() && !Regions.Testing();

    Object around(): oldClassConstructor() {
        return new MyApplication.NewClass();
    }
}
```

Figure 8. Composing Cuckoo's Egg with Border Control.

Recall from Section III-B that NewClass must be a subtype of OldClass; otherwise, we will get a class cast exception on the first attempt to instantiate OldClass. Moreover, we need NewClass to be a subtype of OldClass to make it compatible with the existing references to OldClass to which it would be assigned. This can be achieved either by defining inheritance directly in NewClass or by using a declare parents inter-type declaration (presumably, though not necessarily, in the OldClassDeprecation aspect itself).

Assume there is a need to log the swapping of the deprecated class with the new one. This would seem a simple task. A logging code could be simply added to the CuckooEgg's advice. When there is a need to switch from deprecated class to new version this advice would be executed. But there is a problem. When the logging piece of code is added to the advice, assuming the logging is performed into a text file, it is needed to deal with an IOException that could occur during the execution of this code.

As mentioned in Section III-D, an aspect cannot declare throwing of an exception that was not declared by the advised join point. Another possibility is to throw a runtime exception. In this case the Exception Introduction pattern can be used. This pattern suggests to use a concern-specific runtime exception, which extends a runtime exception. By this, it becomes easier to distinguish between exceptions thrown by various aspects. Also, if a runtime exception has been used there would be no difference between exceptions thrown by various concerns [10]. Figure 9 presents an example implementation of a concern-specific exception and a use of the Exception Introduction pattern adapted to our problem.

```
public class SwitchLoggingException extends RuntimeException {
    public SwitchLoggingException(Throwable cause) {
        super(cause);
    }
}

public aspect SwitchLogging {
    before(): adviceexecution() && Regions.ClassSwitcher() {
        try {
            logSwapEvent()
        } catch(IOException e) {
            throw new SwitchLoggingException(e);
        }
}
```

Figure 9. Composing Exception Introduction with Cuckoo's Egg and Border Control.

Exception Introduction is considered to be an Advice pattern and it was added to the Cockoo's Egg pattern without having to make any change in it. As can be seen in Fig. 9, it can also reuse definitions from the already applied Border Control pattern.

V. Regularity in Aspect-Oriented Design Pattern Composition

As we will see in this section, the composition of aspect-oriented design patterns is substantially affected by their structural category (defined in Section III-E). Under a composition of two patterns we understand a subsequent interrelated

application of two patterns to a problem at hand. In other words, one of the patterns is applied to the problem, and afterwards another one is applied in connection to the artifacts of the former pattern.

Thanks to the crosscutting nature of aspects, most aspect-oriented design patterns can be composed with other patterns without the need to modify the already applied patterns. An example of a pattern that can be composed with almost any other pattern is Exception Introduction, which represents an advice pattern. Exception Introduction can simply be added to the program without having to make any change to already applied patterns.

Another pattern that can be used with other, already applied patterns without having to make any changes to them is Policy, which is an inter-type declaration pattern. This pattern defines a pointcut that captures the join points in a base concern or another pattern whose occurrence represents breaking of some policy. If such a joint point occurs, a compile error or warning is issued.

It is also possible to compose a pointcut pattern with another pointcut pattern without having to change it. In such a composition, the new pattern will actually use the pointcuts of the already applied pointcut pattern. A simple example of this would be composing a Wormhole pattern [10] with an already applied Border Control pattern. This way, the Wormhole pattern would be able to use regions defined by the Border Control pattern in its own pointcuts.

However, composing a pointcut pattern with an already applied advice or inter-type declaration pattern usually requires a change of this pattern. An example of composing a pointcut pattern with an already applied inter-type declaration pattern has been presented in Section IV where we had the Policy pattern applied and composed the Border Control pattern with it. Recall also from the same section that if we go the other way around, i.e. if we compose an inter-type declaration pattern (e.g., Policy) or advice pattern (e.g., Cuckoo's Egg) with an already applied pointcut pattern (e.g., Border Control), this can be done without having to change them.

Assume the pointcuts of a particular non-pointcut pattern in a developing application can no longer be defined in a simple way because it is not certain whether the pattern should be applied to new classes. Such a pattern can be composed with a Participant pattern, which is a pointcut pattern, which would enable individual classes to declare participation in this pattern application. However, the implementation of a Participant pattern requires the already applied pattern code to be altered.

Figure 10 presents schematically the compositions of the aspect-oriented design patterns we discussed. Pattern category is indicated graphically: oval nodes represent advice patterns, rectangular nodes are pointcut patterns, and rhomboid nodes stand for inter-type declaration patterns. Where any pattern of the given category is applicable, its name is shown as asterisk. The edge direction corresponds to the direction of pattern application: an edge originates in the pattern being applied and ends in the pattern to which this pattern is applied to achieve pattern composition. Dashed edges mean no change of the pattern at the edge end is required, while solid lines mean the change is necessary.

In Fig. 10 we see that a composition of a pointcut pattern with an existing pointcut pattern requires no change to the existing pattern: we simply define further pointcuts. Composing a pointcut pattern with an already applied advice or inter-type declaration pattern requires their change since the composition assumes the use of pointcuts defined in the pointcut pattern by the patterns of the latter two categories. This is caused by the nature of pointcut patterns: they define pointcuts to be used by other aspects in the application. On the other hand, a composition of an advice pattern or inter-type declaration pattern with another patterns of any category can be in most cases achieved without having to change the already applied design pattern.

VI. Related Work

Hanenberg et al. present a set of AspectJ idioms[3] and a scheme for their interrelated application [7]. Similarly to the well-known scheme of GoF patterns [6], it is represented by a graph in which patterns that can be composed are connected by directed edges. Each edge is annotated with the role of the pattern in which it originates plays in the pattern in which the edge terminates. However, no attempt is made to categorize the idioms.

There is a certain analogy between our categories and those proposed by Gamma et al. for object-oriented design patterns [6]. Advice patterns recall behavioral patterns since they affect behavior. Pointcut patterns deal with how aspects are composed with classes, objects, and other aspects, which is a paraphrase of the description of structural patterns. It has to be admitted that inter-type declarations correspond to creational patterns to a lesser extent, but we may see them as patterns of creating new elements and relationships.

Aspect-oriented design patterns represent a particular way of crosscutting concern realization. There have been attempts to make a classification of crosscutting concerns according to their invasiveness [15] or their purpose [11].

Cacho et al. studied aspect-oriented implementation of object-oriented design patterns and identified four categories of their composition: invocation-based, class-level interlacing, method-level interlacing, and overlapping [3]. This categorization can be applied to intrinsic aspect-oriented design pattern composition, too, in addition to the categorization of aspect-oriented design patterns as such proposed here.

Class deprecation, used as our case study, is applicable as an aspect-oriented change realization [2]. Captured in an aspect, a change becomes pluggable and reapplicable. The reapplication of a change implemented as an aspect to a new product version in its simplest form takes only including the aspect in a build, which dramatically improves product customization [5].

[3]Although denoted as idioms, they are applicable to PARC style aspect-oriented languages as much as the patterns presented in this paper.

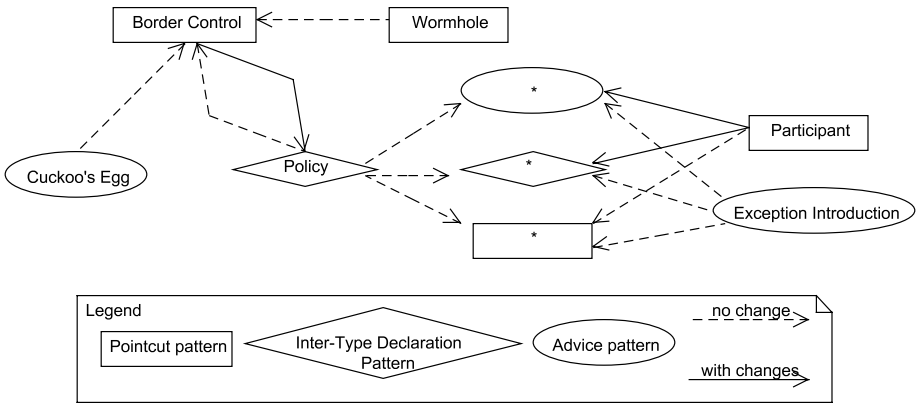

Figure 10. Composition of aspect-oriented design patterns and changes required by it.

VII. Conclusions and Further Work

In this paper, we proposed a categorization of aspect-oriented design patterns according to their structure into three categories: pointcut, advice, and inter-type declaration patterns. This categorization is particularly useful in determining whether a composition of an aspect-oriented design pattern with another, already applied pattern requires a change in this pattern.

We studied the composition of aspect-oriented design patterns of different categories with respect to the stability of the already applied patterns in detail on the composition of Policy, Cuckoo's Egg, Border Control, and Exception Introduction applied to the class deprecation problem in team development.

Our further work involves exploring the possibilities of employing aspect-oriented design patterns and their compositions in capturing changes in a pluggable and reapplicable way. Class deprecation treated in this paper may be seen as one such change. It would also be interesting to explore the possibilities of the guided design pattern instantiation [12] or feature modeling based design pattern instantiation [17] with respect to aspect-oriented patterns and making use of the composition constraints based on aspect-oriented pattern categories in such a process. We will also seek further parallels between categorization of GoF object-oriented design patterns and our categorization of aspect-oriented design patterns by exploring aspect-oriented implementations of GoF object-oriented design patterns [8].

Acknowledgements

The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

References

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] Michal Bebjak, Valentino Vranić, and Peter Dolog. Evolution of web applications with aspect-oriented design patterns. In Marco Brambilla and Emilia Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, pages 80–86, Como, Italy, July 2007.

[3] Nelio Cacho, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thais Batista, and Carlos Lucena. Composing design patterns: A scalability study of aspect-oriented programming. In *Proc. of 5th international Conference on Aspect-Oriented Software Development, AOSD 2006*, pages 109–121, Bonn, Germany, 2006. ACM.

[4] James O. Coplien. The culture of patterns. *Computer Science and Information Systems (ComSIS)*, 1(2), November 2004.

[5] Peter Dolog, Valentino Vranić, and Mária Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, December 2001.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] Stefan Hanenberg, Arno Schmidmeier, and Rainer Unland. AspectJ idioms for aspect-oriented software construction. In *Proc. of 8th European Conf. on Pattern Languages of Programs, EuroPLoP 2003*, Irsee, Germany, June 2003.

[8] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer.

[10] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[11] Marius Marin, Leon Moonen, and Arie van Deursen. Design pattern implementation in Java and AspectJ. In *Proc. of 21st IEEE International Conference on Software Maintenance, ICSM 2005*, pages 673–676, Budapest, Hungary, September 2005.

[12] Vladimír Marko. Template based, designer driven design pattern instantiation support. In *Proc. of 8th East European Conf. on Advances in Databases and Information Systems, ADBIS 2004*, Budapest, Hungary, September 2004.

[13] Radoslav Menkyna. Towards combining aspect-oriented design patterns. In Mária Bieliková, editor, *Proc. Informatics and Information Technologies Student Research Conference, IIT.SRC 2007*, pages 1–8, Bratislava, Slovakia, 2007.

[14] Russell Miles. *AspectJ Cookbook*. O'Reilly, 2004.

[15] Freddy Munoz, Benoit Baudry, and Olivier Barais. A classification of invasive patterns in AOP. Technical Report INRIA Research report 00266555, (IRISA/INRIA) Institut National de Recherche en Informatique et Automatique, INRIA Bretagne Atlantique, Rennes, France, March 2008. http://freddy.cellcore.org/files/pdf/report08a.pdf.

[16] Arno Schmidmeier. Patterns and an antiidiom for aspect oriented programming. In *Proc. of 9th European Conf. on Pattern Languages of Programs, EuroPLoP 2004*, Irsee, Germany, July 2004.

[17] Ľubomír Majtás. Tool based support of the pattern instance creation. *e-Informatica Software Engineering Journal*, 3(1):89–102, 2009.