Case Study Towards Implementation of Pure Aspect-oriented Factory Method Design Pattern

Žilvinas Vaira
Vilnius University
Institute of Mathematics and Informatics
Akademijos 4, LT-01108 Vilnius, Lithuania
zilvinas.vaira@ik.ku.lt

Albertas Čaplinskas
Vilnius University
Institute of Mathematics and Informatics
Akademijos 4, LT-01108 Vilnius, Lithuania
albertas.caplinskas@mii.vu.lt

Abstract—**The paper investigates a case of application of Factory Method design pattern in the context of aspect-oriented design. The case encompasses whole path of design pattern transformation from object-oriented to aspect-oriented design and its application to a particular design context. The main assumption is that there exist design patterns that solve design problems in a similar way for both programming paradigms and that such design patterns can be expressed in the terms of a corresponding programming paradigm. The paper uses design pattern classification and design pattern transformation technique proposing that 20 of Gang of Four 23 design patterns can solve design problems in a similar way to both Object-Oriented Programming and Aspect-Oriented Programming and can be successfully adapted for the needs of aspect-oriented design. The research presents a detailed explanation and examples how to apply proposed technique and discusses elaborated results.**

Keywords - Aspect-Oriented Programming; Object-Oriented Programming; Design Patterns; Factory Method; Framework design.

I. INTRODUCTION

The main intent of this paper is to present an exemplary case showing how object-oriented (OO) design patterns can be redesigned into pure aspect-oriented (AO) design patterns. The complete description of the theoretical discussion, design pattern classification and detailed pattern redesign technique description is proposed in [19]. The research is concentrated on Gang of Four (GoF) 23 [4] design patterns and investigates the question of the possible similarities between two programming paradigms – object-oriented programming (OOP) and aspect-oriented programming (AOP) [11]. As a result, it states that 20 of GoF 23 design patterns can be adapted to solve problems of aspect design. The main contribution of this paper is the experimental evaluation of the [19] proposed redesign technique. It is performed by detailed analysis of redesign technique application to a real life system design. Vaira and Čaplinskas [19] provided theoretical reasoning and models of the redesigned patterns. However, it does not give any insight of practical application of the technique except some hypothetical application context. The results of this paper provide strong evidence in the form of implementation diagrams and detailed description that such design patterns are applicable to real life systems design. It can be stated as a qualitative experimental evaluation of the previous theoretical research. To perform this evaluation, the Factory Method design pattern has been chosen for this research. The case of Factory Method design pattern can be treated as a critical case [16] because it corresponds to the creational design patterns, which are less to be likely acceptable for redesigning into aspects, because they are highly related to creation of objects. Creation of aspects is far different from creation of objects, because aspects are singletons by their nature and its creation in most AO language implementations is handled by aspect weaver automatically. Hence, this paper presents strong evidence that even creational OO design patterns can be adapted to design AO ones.

The major part of the paper includes details of the OO Factory Method design pattern redesign process into AO Factory Method pattern and investigates its application in the context of AO framework design. The main questions that we aim to answer in this paper include: are such AO design patterns applicable in real life applications and does AO representation of Factory Method design pattern change its purpose anyhow? All examples are presented using (Unified Modeling Language) UML class diagrams and stereotyped class diagrams for aspects. The resulted applications are implemented using Java and AspectJ [12] programming languages.

The remaining part of this paper is organized as follows. Section 2 describes the process of Factory Method pattern redesign. Section 3 demonstrates the applicability of pure AO Factory Method design pattern for designing an AO framework. Section 4 analyses related works. Section 5 presents a discussion. Finally, Section 6 concludes the paper.

II. REDESIGNING FACTORY METHOD PATTERN FOR ASPECTS

In this paper, we use terms “redesign” and “pure AO design pattern” in order to distinguish the technique from other design pattern transformation techniques proposed in [6], [8], [9]. By redesign, we mean that design patterns must be reworked from the perspective of its design problem and solution, not by performing simple refactorings or other transformations. Transformation techniques proposed in [8],

 ISBN: 978-1-61208-158-8

[6], [9] search alternative solutions for the same design problem. Our redesign technique redefines design problem for aspects and searches for a design solution using aspects only. More detailed comparison of these techniques could be found in Section IV. The proposed redesign technique is concentrated on two paradigms only, namely OOP and AOP. Such paired paradigm use generates new types of design structures that involve concepts and relations from both paradigms. Moreover, it forces AO paradigm language implementations, such as AspectJ, to inherit elements of a larger scale base paradigm, on which it is built up. Resulted AspectJ language implementation still includes other small scale paradigm elements that are introduced by AOP [20]. This results in complex structures that are problematic to be developed.

The main idea is that some design problems may be stated as common to both paradigms and others as specific with regard to paradigms analyzed (i.e. OOP and AOP). In this case the solution of design pattern that solves these design problems could be performed on both paradigms involving specific paradigm constructs only.

A. Redesign technique

The redesign is based on statement alleging that when OO design pattern can be implemented in AspectJ by using AO constructs only, it can be considered as a pattern that solves similar design problem. It seems that in such case both OO and AO patterns solve the same design problem, but their applicability differs. Thus, the problem in some sense is also different: the OO pattern solves a design problem for objects, whereas the AO pattern solves it for aspects.

Redesign is based on the similarities between aspects and classes, despite the fact that they are different concepts:

- Aspects, similarly as classes, can define data members and behaviors for crosscutting concerns [14]. They can also be defined as abstract entities, or implement Java interfaces.
- Aspects can be used as collaborative entities and build inheritance hierarchies in similar way as it is done with classes.

However, one of the main differences is the fact that aspects cannot be directly instantiated. There is a possibility to use several instances of one aspect in a program by declaring an aspect per object or per control flow in a program. In such case the instantiation still differs from the one that is done with classes. For this reason we refer to aspects as singletons. Redesign technique involves 3 main steps (see Figure 1):

- If a GoF 23 pattern can be implemented using singletons only, it is regarded as a candidate to be a design pattern for rewriting to AspectJ.
- All classes in the candidate pattern should be replaced by aspects.
- The candidate pattern should be analyzed in order to discover and remove irrelevant data members and methods.

These steps are generalized from the original. More detailed technique description can be found in [19].

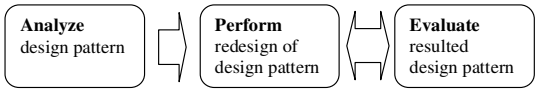

Figure 2. Redesign technique

B. Redesign description

In the case of Factory Method design pattern we are dealing with, it may seem that the AO solution has no sense, because Factory Method belongs to creational pattern category and is highly related to creation of objects. In the AO paradigm we in most cases deal with the singletons only and in fact the creation of aspects cannot be managed directly by other aspects. However, it does not mean that the redesign technique can not be performed on Factory Method design pattern. The creation of aspects can be replaced by passing a reference to already created aspect. In order to do this we can use *AspectOf* method instead of constructor method. *AspectOf* corresponds to an analogue *InstanceOf* that is used for referencing singletons. We will demonstrate that AO solution of Factory Method can be redesigned using

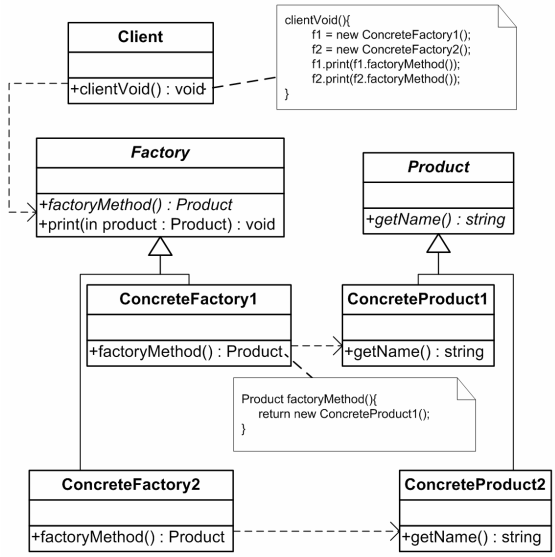

Figure 1. Factory Method design pattern (OO solution)

proposed technique.

The first step is to perform analysis of the pattern to inspect if it can be regarded as candidate for rewriting. The Factory Method design pattern defines an abstract method that can be overridden by subclasses for creating objects that belong to different classes [4]. There are several other variations of the pattern (e.g. the parameterized factory method), but in this particular case we use the general one. The main elements of the general case of Factory Method (see Figure 2) design pattern are:

- *Factory*, an abstract class that contains abstract operation *factoryMethod*, which is overridden by its subclasses,

 ISBN: 978-1-61208-158-8

- *ConcreteFactory1* and *ConcreteFactory2*, concrete *Factory* classes overriding *factoryMethod*, which creates and returns the object of *ConcreteProduct1* or *ConcreteProduct2* respectively.
- *Product*, an abstract class that contains the abstract operation *getName* and defines the interface of *Product* type objects,
- *ConcreteProduct1* and *ConcreteProduct2*, concrete *Product* classes that implement the *getName* operation using some concrete method, and
- *Client*, the class that invokes the *factoryMethod* of the *Factory* object.

There is no critical reason indicating that Factory Method design pattern can not be implemented using singletons only. Abstract classes can be replaced by abstract aspects, subclasses by specializing aspects. The constructors of *ConcreteProduct1* and *ConcreteProduct2* can be replaced by *AspectOf*. All other operations remain the same as in classes.

When it is decided that the Factory Method is a candidate for redesigning, the second step can be performed in Figure 3. The resulted AO Factory Method solution helps to get a

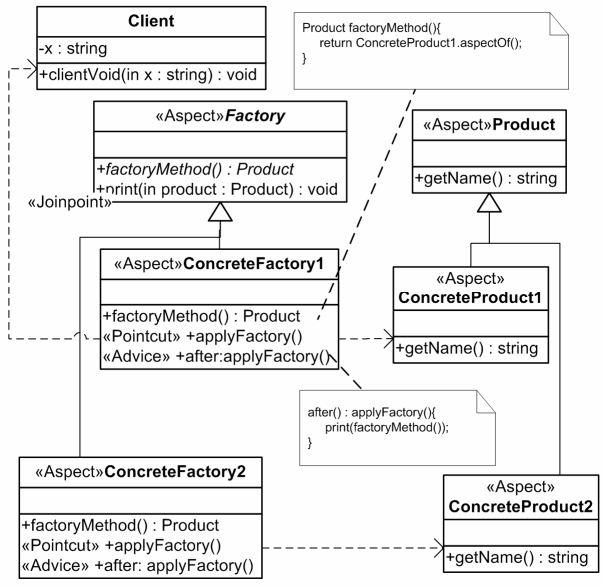

Figure 3. Application of the AO Factory Method design pattern

reference to the necessary aspect defined by specialized *Factory* aspect. This is an analogous solution to that of OO Factory Method design pattern. The main difference is that instances of aspects are created only once and each time we execute *factoryMethod* particular *Product* instance is passed as an argument.

The last step of evaluation of resulted pattern involves possible refactorings to enhance the resulted design and to test its applicability. The main variation of the pattern can be performed by changing or adding pointcuts and advice. The current model includes pointcuts and advice in subaspects of *Factory* aspect and in this way it is defined when *factoryMethod* operation is invoked. Another place for defining pointcuts and advice could be subaspects of *Product* aspect. More comprehensive designs of pattern behavior could be resulted by predefining some pointcus or advice in abstract aspects. The important difference of AO design pattern comparing to its OO analogue is that the developer is limited with a number of predefined subaspects it can use at the same time (except of above mentioned cases per object or per control flow aspects). However, it does not change the principal behavior of this design pattern and demonstrates that AO design pattern preserves all essential elements of the OO pattern.

An example of the application of the AO Factory Method pattern is analyzed in the following part of the paper. In this example, we are dealing with the complex logging concern in a simulation domain framework.

III. APPLICATION OF PURE AO FACTORY METHOD

SimJ simulation framework is used as experimental application providing necessary context for implementing AO Factory Method design pattern. The main research interest is concentrated on logging concern, which has a crosscutting issues that need to be eliminated and the feature of logging needs to be made customizable. SimJ is a simulation framework used for developing simulation applications based on discrete events.

The logging concern in a framework suffers from crosscutting. Pieces of the code belonging to it are scattered and tangled through the remaining part of a framework. The complexity of a logging functionality of this framework makes it a sufficient candidate to apply the AO Factory Method design pattern presented in Figure 3. The framework has several different kinds of things to be logged and must remain customizable in a concrete specialization of a framework. The current version of the framework allows customizing logging. However, it is handled beyond the bounds of logging concern individually by every entity that needs to be logged. The main purpose of application of AOP is to exclude all pieces of code related to logging concern and combine them in aspects. Although the design of these aspects is not an ordinary task to complete, design pattern could be applied to handle it.

The AO Factory Method design pattern was introduced to deal with the following issues: different logging behavior for resources and several kinds of events was necessary and the complexity of triggering of this behavior required its separation. Different behavior of logging was modeled using product hierarchy in Factory Method pattern. The triggering structure of logging behavior was designed using hierarchy of factories Figure 3. The resulted implementation of logging concern is presented in Figure 4. The UML diagram contains complete design that includes two additional instances of Template Method (design pattern is usually used in composition with factories). The stereotype “Hook” is used to denote customizable framework methods in aspects.

Consequently, several advantages can be noticed: all the logging functionality and related code is localized in one place and the customization of the logging concern can be carried out separately from the rest of the hot spots. This also

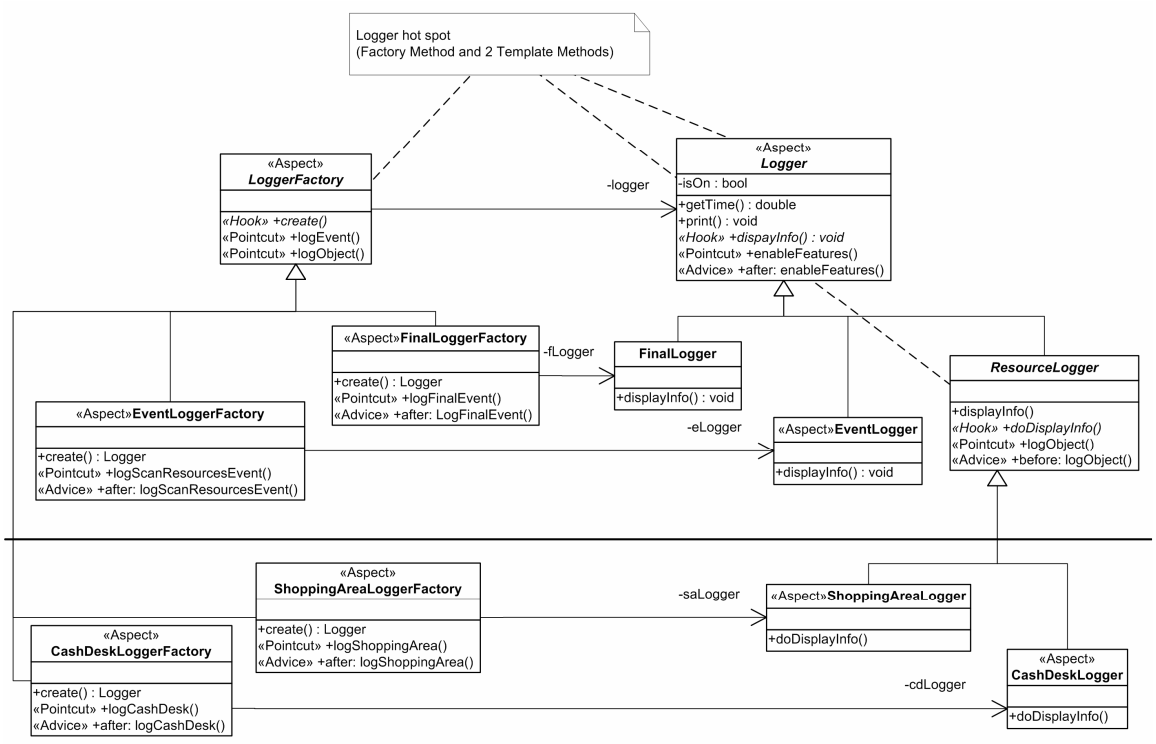

Figure 4. Factory Method desgin pattern (AO solution)

means that maintenance and unplug ability features of the logging were increased. This implementation allows flexible customization so that logging of events and resources can be done separately and the joinpoints triggering logging behaviors can be customized independently. A high number of aspects can be considered as a shortcoming. This is probably related to the complexity of the logging concern behavior.

IV. RELATED WORKS

OO design patterns can be redefined for the AO paradigm in several different ways. Implementation of the OO design pattern in Java, can be directly replaced by the analogous code written in AspectJ [6], [8] a native AO solution can be introduced to the same problems that are addressed by the OO design pattern [9] or pattern solving AOP specific design problems can be elaborated [2], [7], [14], [15]. There is no concrete technique describing how to discover patterns solving AOP specific design problems. However, two different design pattern transformation techniques can be distinguished and compared to the one analyzed in this paper:

a) The authors of [6] and [8] use very similar pattern transformation technique. They introduce AOP constructs to deal with the problems related to crosscutting in the pattern solution. The design problem solved by the pattern does not change and the main idea of the solution remains the same.

b) Authors of [9] use slightly different technique. The main idea is that design solution still must deal with the same problem. However, aspects are used to search for an alternative solution, different than the main idea provided by original design pattern. Both of these techniques provide solutions to the same design problems. Such solutions are alternatives and can be compared.

c) Our technique, presented in [19], is slightly different, because we redefine a design problem for aspects. Considering the similarities between AOP and OOP paradigms we say that a similar design problem that occurs when designing objects can also occur while designing aspects. In such a case we can use the same design idea that has solved the design problem for objects, but this time only aspects are used. In result, the design pattern achieved using this technique is not an alternative solution to the same design problem. It is more like a new AOP design pattern solving a similar to OOP design problem in a similar to OOP way.

A number of quantitative evaluations have attempted to measure the effectiveness of the implementation [5], [8] and [3]. As design patterns can be composed in many different ways and crosscut each other, most of these quantitative assessments agree that aspectization of patterns can significantly improve OO implementations. However, in some cases the results depend on the patterns involved, design complexity, and other circumstances as discussed in [3]. The main problems commonly reduced by the use of aspects are related to code scattering and code tangling. So, it

is reasonable to expect that implementations in AO languages will at least partly solve these problems.

A framework that is used to provide some contextual evidence for AO Factory Method application is considered as the software framework described in [10]. It states that application framework is a reusable, „semi-complete" application that can be specialized to produce custom applications. The application of pure AO design patterns produces a new kind of application framework that we refer to an AO application framework. Similar AO framework design, where aspects were used as glue code for gluing framework core and its application was presented in [17]. A more comprehensive and a more related to this paper AO framework design, that includes the use of customizable aspects, is presented by the following researches [1], [18], including more complex design structures that involve some idioms of AspectJ in [13].

V. DISCUSSION

There are several debatable issues that we would like to discuss. The main of them is the use of aspects as collaborative entities. The designs that include abstract aspect hierarchies hold references and invoke calls to other aspects help to create reusable and flexible implementation structures. These are the main features used to create collaborations of classes in OOP. However, such structures also increase the tangling of the implementation code, which is an issue that AOP used to deal with. It is not always clear, what the constraints of collaborations in aspects are and when a threat of creating too complex designs of aspects appears. We assume that collaborations of aspects are beneficial unless they overstep the boundaries of related concerns.

The Singleton nature of aspects is the second issue. Though, aspects in AspectJ are by default singletons, in special cases aspects can be also instantiated per object or per control flow. From this perspective it is still questionable whether aspects should be treated as singletons or not. Direct instantiation of aspects in AspectJ language is forbidden. Aspects can be globally referenced using static method *aspectOf* and it is not quite compatible with the direct creation. Another problem is that if it were allowed to create several instances of the same aspect at a time, the behavior advised by aspects might repeat several times or act in other unexpected ways. As a result, there may be difficulties related to aspect instantiation control. This is the main reason why we suggest following the Singleton nature of aspects and treating per object and per control flow aspects as special cases of singletons.

VI. CONCLUSIONS AND FUTURE WORK

The paper demonstrated that design patterns solving similar design problems in both, AOP and OOP paradigms, could be used to deal with crosscutting and to design customizable aspects in frameworks. The investigated case of Factory Method design pattern shows that even creational design patterns can be applied for this purpose. It promotes the elimination of crosscutting behavior and localization of scattered implementations. Moreover, this crosscutting behavior can be designed as a reusable hot spot in a framework and customized in a framework application. The purpose of Factory Method design pattern in AOP is slightly changed comparing to OOP. Instead of creating factories it only passes reference to the necessary aspect. In some cases the use of the pure AO design patterns only may be insufficient. They should be used in compositions with available design patterns from other categories of AO design. It is reasonable to expect that compositions with patterns for designing pointcuts and advice could increase the applicability of existing ones or even create new AO design patterns.

Further investigations of pure AO design pattern applications to design programs are necessary. The investigation towards other patterns solving similar design problems in other paradigms is also intended.

ACKNOWLEDGMENT

The authors wish to thank Software Engineering Research Group headed by Prof. Jacques Pasquier for providing SimJ framework for experimental application. Personal thanks to Prof. Jacques Pasquier, Dr. Patrik Fuhrer and Minh Tuan Nguyen for inspiring and initial guiding of related research.

REFERENCES

[1] P. Arpaia, M.L. Bernardi, G. Di Lucca, V. Inglese, and G. Spiezia, "Aspect Oriented-based Software Synchronization in Automatic Measurement Systems", In *Proceedings of Instrumentation and Measurement Technology Conference*, IMTC 2008, IEEE, pp. 1718 – 1721, 12-15 May 2008.

[2] M. Bynens and W. Joosen, "Towards a Pattern Language for Aspect-Based Design", In *Proceedings of the 1st workshop on Linking aspect technology and evolution* (PLATE '09), Charlottesville, Virginia, USA date:March 2 - 6, 2009. ACM, pp. 13-15.

[3] N. Cacho, E. Figueiredo, C. Sant´Anna, A. Garcia, T. Batista, and C. Lucena, "Aspect-oriented Composition of Design Patterns: a Quantitative Assessment", *Monografias em Ciência da Computação*, vol. 5, no. 34. Pontifícia Universidade Católica do Rio de Janeiro, Brasil, 2005.

[4] E. Gamma, R. Helm, R. Johnson, and J.Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[5] A.Garcia, C.Sant'Anna, E. Figueiredo, and U. Kulesza, "Modularizing Design Patterns with Aspects: A Quantitative Study", In: *Proceedings of the International Conference on Aspect-Oriented Software Development* (AOSD'05), Chicago, USA, 14-18 March 2005. ACM Press, pp. 3-14.

[6] O. Hachani and D. Bardou, "Using Aspect-Oriented Programming for Design Patterns Implementation", In *Proceedings of 8th International Conference on OOIS 2002*, Position paper at the Workshop on Reuse in Object-Oriented Information Systems Design, Montpellier, France - Sept. 2-5 2002.

[7] S. Hanenberg and A. Schmidmeier, "Idioms for building software frameworks in AspectJ", In *Y. Coady, E. Eide, D. H. Lorenz (Eds.) Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (ACP4IS), College of Computer and Information Science, Boston, Massachusetts, 2003, pp. 55-60.

[8] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ", In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems,*

 ISBN: 978-1-61208-158-8

Languages, and Applications (OOPSLA '02), ACM Press, 2002, pp. 161-173.

[9] R.Hirschfeld, R. Lämmel, and M. Wagner, "Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration", In *Proceedings of the 3rd German Workshop on Aspect-Oriented Software Development* (AOSD-GI 2003), 2003, pp. 25–32.

[10] R. E. Johnson and B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June/July 1988, vol. 1, no. 2, pp. 22-35.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect oriented programming", In *Proceedings of European Conference on Object Oriented Programming, ECOOP*, 1997, vol. 1241, pp. 220–242.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ", *Communication of the ACM*, October 2001, vol. 44, no. 10, pp. 59–65.

[13] U. Kulesza, V. Alves, A. Garcia, C. J. P. de Lucena, and P. Borba, "Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming", In *Proceedings of Intl Conference on Software Reuse (ICSR)*, Torino, Italy, pp. 231-245, 2006.

[14] R. Laddad, *AspectJ in Action: practical aspect-oriented programming*, Manning Publications Co, 2003.

[15] R. Miles, *AspectJ Cookbook*, O'Reilly Media, 2004.

[16] Charles C. Ragin, ""Casing" and the process of social inquiry", In *Charles C. Ragin and Howard S. Becker (eds), What is a Case? Exploring the Foundations of Social Inquiry*, Cambridge: Cambridge University Press, 1992, pp. 217–26.

[17] A. Rausch, B. Rumpe, and L. Hoogendoorn, "Aspect-Oriented Framework Modeling", In *Proceedings of the 4th AOSD Modeling with UML Workshop*, UML Conference 2003, October 2003.

[18] A. L. Santos, A. Lopes, and K. Koskimies, "Framework specialization aspects", In *Proceedings of AOSD '07 the 6th international conference on Aspect-oriented software development*, ACM New York, NY, USA 2007, pp. 14 - 24.

[19] Ž. Vaira and A. Čaplinskas, „Paradigm-independent design problems, GoF 23 design patterns and aspect design", *Informatica*, Institute of Mathematics and Informatics, Vilnius, in press.

[20] V. Vranić, "AspectJ Paradigm Model: A Basis for Multi-Paradigm Design for AspectJ", In *Jan Bosch, editor, Proc. of the Third International Conference on Generative and Component-Based Software Engineering* (GCSE 2001), LNCS 2186, Erfurt, Germany, September 2001, pp. 48-57, Springer.

 ISBN: 978-1-61208-158-8