Guidelines for Using Aspects in Product Lines

Ján Kohut and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology
Ilkovičova 3, 84216 Bratislava 4, Slovakia
jan.kohut84@gmail.com, vranic@fiit.stuba.sk

Abstract—**Software product lines are a successful approach to software reuse. To a significant extent, the development of product lines is complicated by crosscutting concerns. This is especially true for product line configuration. Aspect-oriented programming can often help to develop product lines more effectively, but this is not always so. This paper shows it is possible to sublimate the expert knowledge about using aspect-oriented programming in product line development in the form of independently applicable guidelines by proposing an approach to expressing such guidelines. A complete guideline entitled Implementing Mandatory Features with no Crosscutting Concerns is presented as an example. The guidelines have been evaluated in a product line case study that confirmed their applicability. The results of this study obtained by the application of a number of metrics are provided and discussed in the paper.**

I. Introduction

Software product lines (most often denoted just as product lines) represent a successful approach to software reuse. Instead of striving for general component reusability, each product lines targets a specific domain. Reusability is achieved by separating common features from variable ones and projecting this along the design and implementation lines. Ideally, a specific application is developed by configuring the product line, i.e. by selecting variable features to be included. However, often some additional code is needed arising from the previously unforeseen requirements. Nevertheless, the gain from reuse usually overweighs this inconvenience. Moreover, the new artifacts may be incorporated into the product line, too.

Development of product lines is a complicated process. As in any software development, this is in part due to crosscutting concerns. This is most obvious in the product line configuration in which features that represent crosscutting concerns cannot be easily incorporated nor taken out once they have been incorporated.

Aspect-oriented programming enables the modularization of crosscutting concerns, so it may be very useful in product lines, especially in feature configuration. However, aspect-oriented programming cannot be considered as the best solution always. It is probably impossible to formulate strict rules that will tell when to apply aspect-oriented programming and when not to due to hard context dependencies that can never be fully formalized. However, we believe it is possible to make some recommendations on the use of aspect-oriented programming in product lines in the form of guidelines that will sublimate expert knowledge in this field.

The rest of the paper is organized as follows. Section II presents the form of guidelines and guidelines that have been identified. Section III describes in detail one of the guidelines. Section IV describes the evaluation of guidelines. Section V discusses related work. Section VI closes the paper with conclusions and further work.

II. Guidelines

When product line developers are faced with a decision to apply aspect-oriented programming or not for a particular feature or set of features, they could search advice in the experience of others reported in literature, ranging in presentation form from peer reviewed scientific work to informal contributions to discussion forums. This is a highly complicated task both in finding relevant sources and identifying relevant parts in these sources once they have been found.

It would be helpful to have this knowledge in a compact and easily accessible form. To avoid burring the efforts to help developers by another forced step-by-step application of a "complete" method of using aspect-oriented programming in product lines that would never be capable of covering all possible contexts, it logically follows this expert knowledge should be partitioned into individually applicable chunks each of which would provide some kind of recommendation accompanied by both advantages and disadvantages following from its application.

According to what we said so far, the guidlines we are seeking for would be much like patterns. Indeed, we find the pattern form of description quite suitable. One of the popular pattern forms is Coplien's form consisting of these six parts [9]:

- Short name
- Name
- Context
- Problem
- Forces
- Solution
- Example
- Discussion

This form contains all the parts relevant to the application of a guideline.

The short name provides a unique and descriptive name of the guideline. The name provides a one sentence guideline description. The context describes the situation in which it is convenient to consider using the guideline. The problem defines problems that can occur if the guideline is not been applied. The forces provides the reasons why the current state is not satisfactory. The solution describes the process of resolving this situation. Example contains an example implementation or its part. In the discussion, the solution advantages and disadvantages are pointed out.

It is possible to define a guideline at various levels of abstraction of a particular system. Also, guidelines are not strict rules, but just recommendations because the problem of product line feature configuration is very complex. Each guideline must be consistent with other guidelines. It is not necessary to avoid difference in recommendations at all costs, but the guideline should accommodate to them and eventually provide some possibilities of combining even the differing guidelines.

When applying a guideline, it is necessary to consider how the product line is being developed (from scratch or incrementally, and evolutionary or revolutionary [6]), what experience software company has with aspect-oriented programming development, and to what extent it is ready to go into the risk of developing software with a new approach.

Here are the guidelines that we identified:

- **Implementing Features of Refactored Legacy Applications**: *The aspects are unsuitable for implementing features of refactored legacy applications.* Feature refactoring of legacy applications is a difficult problem because such applications do not imply that their design was amenable to feature extensibility [12].
- **Implementing Mandatory Features with no Crosscutting Concerns**: *Do not use aspects in mandatory features if there are no crosscutting concerns.* Aspects flatten inherent object-oriented structure of collaboration, obscure the intent of the programmer, and the result is a program that is difficult to read [1].
- **Code Reduction in Homogenous Crosscutting Concerns**: *The aspects are suitable for reduction of replicated code in homogenous crosscutting concerns.* Aspects reduce replicated code in code with homogenous crosscutting concerns.
- **Transforming a Mandatory Feature into Alternative Features**: *Do not use aspects in transforming a mandatory feature into alternative features.* By transforming a mandatory feature into two or more alternative features AspectJ adds and changes more components and lines of code compared to non-aspect-oriented approaches because all aspects rely on the join points provided by the core [11].
- **Implementing Features which Share no Code**: *Use aspects in implementing features which share no code and which have crosscutting concerns.* Aspect-oriented solution provides low cost and superior stability in terms of tangling and scattering over components [11].

III. An Example: Reduction of Replicated Code

This section presents the guideline *Implementing Mandatory Features with no Crosscutting Concerns* as an example. This guideline is about suitability of using aspects for reduction of replicated code with homogenous crosscutting concerns. Homogenous crosscutting concerns address multiple join points with a single piece of advice. Heterogeneous crosscutting concerns, in contrast, address multiple join points each with a different piece of advice [1]. Homogeneity lies in a consistent application of the same or very similar policy in multiple places [14]. It has been reported that aspects reduced replicated code in homogenous crosscutting concern implementation [1].

Further sections follow the guideline form structure as proposed in Section II (apart of the guideline short name and name).

A. Context

Often, the same concern—i.e. its code—is repeated across different modules in a product line.

B. Problem

It is hard to maintain and keep consistent the code scattered accross the application.

C. Forces

We would like to modularize the crosscutting concerns and represent each one by a single feature for ease of maintenance and configurabilty, but the conventional approach does not provide mechanisms for this.

D. Solution

Using aspects in homogenous crosscutting concerns will enable to factor them out into separate and easily configurable modules. Each such concern would be implemented by its own advice.

E. Example

Examples of homogenous crosscutting concerns include logging, tracing, and exception handling. The following code shows an example of a crosscutting concern:

```
public aspect Homogenous {
   pointcut accessAuthorization(): call(...);
   pointcut accessDemilitarizedZone(): call(...);
   pointcut accessCommunicationInterfaces(): call(...);
   pointcut accessApplication(): call(...);

   before(): accessAuthorization() ||
      accessDemilitariziedZone() ||
      accessCommunicationInterfaces() ||
      accessApplication() {
      System.out.println("Accessing " + thisJoinPoint);
   }
}
```

The presented aspect affects a set of otherwise unrelated methods using one coherent advice.

F. Discussion

Homogenous crosscutting concerns address multiple join points by a single advice. The conventional approach require repeating almost identical pieces of code in every affected method which results in code replication and problems connected with it [2].

On the other hand, spects influence classes by the code that is inside of their inter-type declarations and advices and no longer a physical part of the affected class. This can lead to misunderstanding object-oriented architecture and structure of classes. Subsequently, the program may become more difficult to read and understand.

IV. Evaluation

Evaluation of the approach has been performed on the Java Email Server application.[1] Java Email Server is a Java implementation of the SMTP and POP3 e-mail server. The original application has 4500 line of code, 22 classes, and several configuration files. To evaluate the approach proposed in this paper, a product line has been created out of this application.

A. Feature Model

Figure 1 shows the feature model of the product line developed out of Java Email Server.

We use basic Czarnecki–Eisenecker feature modeling notation [10]. The main part of the feature model is the feature diagram. Its root node represents a concept of the application as one of the possible applications in a given product line.

The rest of the nodes are features each of which represents some functionality. Features are organized hierarchically and the inclusion of a subfeature requires the inclusion of its parent feature.

Some features are common for all product line configurations, i.e. they are *mandatory*. Mandatory features are denoted by filled circle ended edge. In Java Email Server product line, mandatory features represent core modules for processing passwords, users, e-mails, POP3 and SMTP protocols, and logging tool Log4J.

The rest of the features are variable. Optional features—that may, but don't have to be included—are denoted by an empty circle ended edge.

Some features are exclusive with respect to some other features. Such features are alternative, which is graphically denoted by an arc. The configuration parameters file format is an example. In our application, there are two such formats: ConfigFileManager stands for property files, while ConfigXMLManager stands for XML files.

Note that the mandatory features whose parent is a variable features are mandatory only if the parent feature is included.

We express constraints between non-adjacent features using natural language [10]. For this, first-order predicate logic [22] or OCL could be employed, too, as a widely accepted and powerful notation for such uses (but even of wider applicability, e.g. instead of object algebras [18]). If kept consistent, the notations for expressing additional constraints could be used interchangebly, for it is not uncommon for a language to have multiple notations for one language [19].

[1]http://www.ericdaugherty.com/java/mailserver/

B. Metrics Applied

Various kinds of metrics have been employed to evaluate the application of the proposed approach to the Java Email Server product line. One group of metrics targets the scope of changes:

- Lines of Code (LOC)—expresses the size of modules; it is important for comparing the size of modules and cost of implementation
- Number of Affected Classes (NAC)—expresses the number of classes affected by the implementation of a feature

Another group of metrics that have been applied are various metrics targeting the quality of code suitable both for object-oriented and aspect-oriented programming [7], [8], [21] (module denotes a class or aspect):

- Weighted Operations in Module (WOM)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Crosscutting Degree of an Aspect (CDA)
- Coupling of Method Call (CMC)
- Coupling of Field Access (CFA)
- Coupling between Modules (CBM)
- Response for a Module (RFM)
- Lack of Cohesion in Operations (LCO)

The last group of metrics that has been applied measures package dependencies [15], [21]:

- Number of Types (NOT)
- Abstractness (A)
- Afferent Couplings (Ca)
- Efferent Couplings (Ce)
- Modified Efferent Couplings (Ce)
- Instability (I)

C. Performing Evaluation

Two independent product line implementations were created out of the original Java Email Server application. In the aspect-oriented implementation, the features were configured using the AspectJ language. Every feature was implemented in its own file. Features are weaved into the product line through compilation.

The object-oriented implementation is configured by a configuration file being loaded on the application start. The features are asserted during program execution by an if statement as shown in the following code fragment:

```
if(OOConfig.getInstance().isInitializeLog4JEnabled()) {
    InitializeLogging.initializeLogging(directory);
}
```

Further sections explain the evaluation of each guideline we have identified.

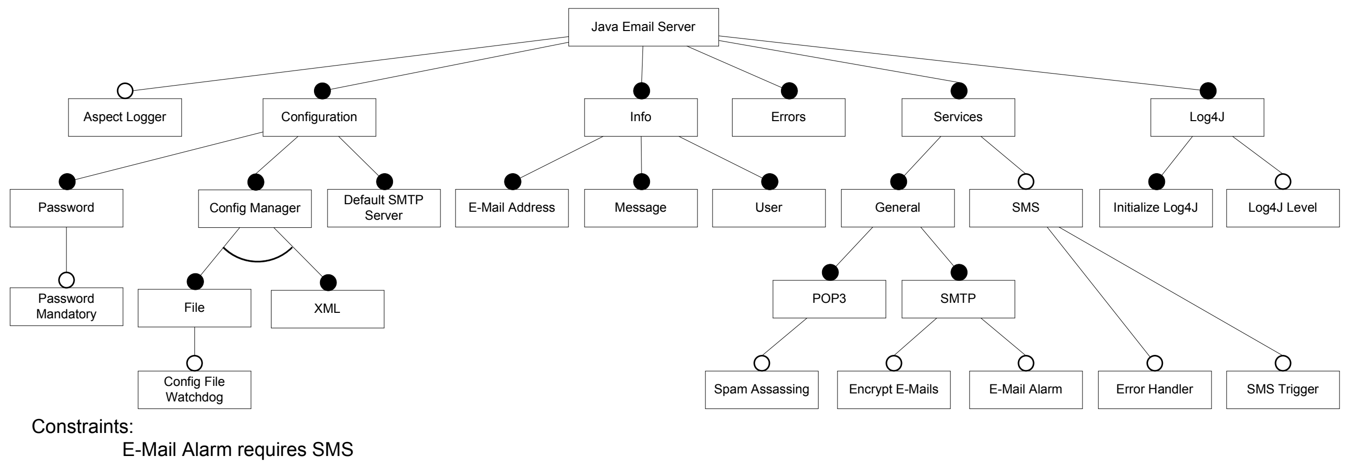

Figure 1. Java Email Server product line feature model.

Table II
NUMBERS OF CALLING LOGGING FUNCTIONS.

Method	Number of calls
log.isDebugEnable()	27
log.debug()	47
log.isInfoEnable()	10
log.info	26

Table III
SIZE OF CHANGES FOR ADDING ALTERNATIVE FEATURE CONFIGXMLMANAGER INTO EXISTING PROGRAM INFRASTRUCTURE.

	AO change	OO change
Class/Aspects	Creating new aspect	Change class code
LOC	11	3

D. Implementing Mandatory Features with no Crosscutting Concerns

Using aspects to implement mandatory features without crosscutting concerns hides before programmer the object-oriented structure of program. The number of LOC used to implement the Password Mandatory feature is similar (11 and 9 LOC). Table I shows some characteristics of the Password Mandatory feature.

Using aspects to implement mandatory features without crosscutting concerns hides before programmer the object-oriented structure of program. The number of LOC used to implement the PasswordMandatoryFeature application feature is similar (11 and 9 LOC). In Table I are shown some characteristics of PasswordMandatoryFeature.

E. Code Reduction in Homogenous Crosscutting Concerns

Homogenous crosscutting concerns are one of the most appropriate areas for the using aspects. Example of these homogenous crosscutting concerns in Java Email Server are calls to the Log4J logging framework. In Java Email Server, the same or similar pieces of code appear frequently. Table II shows the number of calls to logging methods.

This code snippet repeated throuhout the product line shows how homogenous crosscutting concerns are being solved without aspects:

```
if (log.isDebugEnabled()) {
    log.debug("Loading SMTP Message " + messageFile.getName());
}
```

This is the same code if we apply aspects to modularize this corsscutting concern:

```
log.debug("Loading SMTP Message " + messageFile.getName());
```

The LoggingLevel aspect reduced LOC saving 27 (potentially 47 times) calls of the isDebugEnabled() method. Similarly, moving the isInfoEnable() method into the aspect reduced calls to this method for 10 times (potentially 26 times).

F. Transforming a Mandatory Feature into Alternative Features

We implemented a change of a mandatory feature into two or more alternative features in both versions of the Java Email Server product line. The change was realized by object-oriented refactoring. Refactoring was required to enable later use of the features. After this change, it became simple to create and configure alternative features with aspects. Both aspect-oriented and object-oriented approach add comparable number of LOC when used for adding new alternative feature (see Table III).

G. Implementing Features of Refactored Legacy Applications

Adding new features by object-oriented approach intorduced new crosscutting concerns. On the other hand, as expected, aspects did not cause the code tangling and crosscutting, because aspects do not add any code into original code in the first place. Aspect-oriented approach enabled to add new features easily.

The size of code has been growing, but the code showed similar level of coupling (see Tables IV and V). The metrics have been evaluated on the code without implemented homogenous crosscutting concerns (LoggingLevel and ErrorAlarm features) for the reason of showing other characteristics

Table I
METRICS USED ON PASSWORDMANDATORY FEATURE.

	LOC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
OO	45	3	0	0	0	1	1	0	0	5	0
AO	35	2	0	0	0	0	0	0	0	2	0

Table IV
METRICS EVALUATION FOR THE ORIGINAL JAVA EMAIL SERVER VERSION (ORIG.), OBJECT-ORIENTED PRODUCT LINE (OO) AND ASPECT-ORIENTED PRODUCT LINE (AO).

	LOC	WOM	DIT	NOC	CFA	CMC	CBM	CDA	CAE	RFM	LCO
Orig.	124.73	8.55	0.36	0.05	0.14	3.55	3.68	0.00	0.00	15.77	54.27
OO	96.21	6.68	0.32	0.06	0.09	3.15	3.24	0.00	0.00	12.68	37.26
AO	88.17	6.00	0.28	0.06	0.08	2.64	2.72	1.00	1.00	11.92	33.22

Table V
PACKAGE METRICS EVALUATION.

	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
Orig.	3.14	0.06	2.14	4.14	0.45	0.48	7.14	4.14	0.61	0.33
OO	3.78	0.04	2.44	4.67	0.40	0.56	6.78	4.67	0.53	0.43
AO	4.00	0.05	3.00	4.22	0.43	0.53	7.33	4.22	0.56	0.40

as benefits of using aspects on homogenous crosscutting concerns.

H. Implementing Features which Share no Code

Figure 2 shows a comparison of method call coupling in the object-oriented and aspect-oriented version of the product line per class (class names not shown). The aspect-oriented version shows lower value coupling of method call for most classes which brings us to the conclusion that this approach supports better modularity and reusability of classes.

V. RELATED WORK

There is a significant body of work with respect to applying aspect-oriented programming in product lines. The target of this paper is not to find the best approach for the configuration of product lines, but create and evaluate guidelines for using aspects in product lines.

It is suitable to implement homogenous crosscutting concern using aspects, but using aspects to solve heterogeneous crosscutting concerns does not bring such positive results. This is confirmed by several studies [1], [14].

Figueiredo et al. [11] represent results of a quantitative study that evolves two product lines to assess various facets of design stability of aspect-oriented implementations. Various metrics have been applied in the study, but the authors find the application of the metrics not specifically created for aspect-oriented programming to be disputable.

Kästner et al. represent a case study on refactoring of the Berkley DB system into a product line [13]. They documented the cases in which AspectJ is unsuitable for implementing features of refactored legacy applications. The results presented here are broader in their context as our study covered not only legacy application refactoring, but also an incremental development of a product line.

An approach to software evolution based on aspect-oriented change realization can also lead to establishing product-lines [3], [17]. Changes can be implemented using aspect-oriented programming even if the source code of the code to be changed is not available [4], [5].

VI. CONCLUSIONS AND FURTHER WORK

An approach to expressing independently applicable guidelines that sublimate the expert knowledge about using aspect-oriented programming in product line development has been proposed in this paper. An example of a complete guideline has been presented.

To evaluate applicability of the guidelines that have been identified, a case study has been performed. An aspect-oriented version of the product line has been developed using the guidelines. An object-oriented version of the product line has been developed, too, in order to enable the evaluation of the guideline applicability. For this, several metrics suitable both for aspect-oriented and object-oriented programs have been applied confirming the applicability of guidelines.

One of the obvious directions of further work is the identification of other guidelines. However, it is also important to evaluate guidelines with respect to other aspect-oriented languages. For example, it would be interesting to expand our study to other domains like service-oriented architecture [16], for which we have available suitable application developed in Java [20].

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Sven Apel and Don Batory. When to use features and aspects?: A case study. In *Proc. of 5th International Conference on Generative Programming and Component Engineering, GPCE 2006*, pages 59–68, Portland, Oregon, USA, 2006. ACM.

[2] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: aspects and features in concert. In *Proc. of 28th International Conference on Software Engineering, ICSE 2006*, pages 122–131, Shanghai, China, 2006. ACM Press.

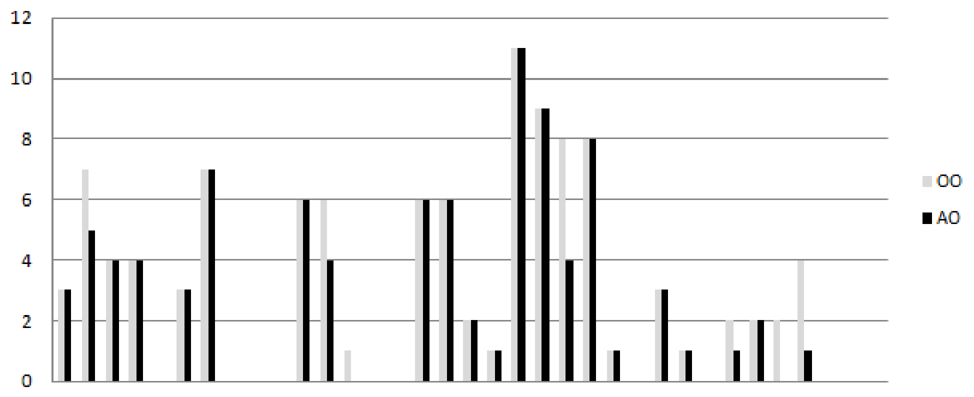

Figure 2. Method call coupling in the object-oriented (OO) and aspect-oriented (AO) product line implementation.

[3] 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.

[4] Ilona Bluemke and Konrad Billewicz. Aspect modification of an EAR application. In *Proc. of International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering, CIS^{2}E 08*, Krakow, Poland, December 2008. Springer. To appear.

[5] Ilona Bluemke and Konrad Billewicz. Aspects in the maintenance of complied program. In *Proc. of 5th International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008*, pages 253–260, Szklarska Poręba, Poland, June 2008. IEEE.

[6] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[7] Mariano Ceccato and Paolo Tonella. Measuring the effects of software aspectization. In *Proc. of 1st Workshop on Aspect Reverse Engineering, WARE 2004*, Delft, The Netherlands, 2004.

[8] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1996.

[9] James O. Coplien. Software patterns. http://hillside.net/patterns/definition.html.

[10] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programing: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[11] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proc. of 30th international Conference on Software Engineering, ICSE 2008*, pages 261–270, Leipzig, Germany, 2008. ACM.

[12] Martin Fowler. Writing software patterns. http://martinfowler.com/articles/writingPatterns.html.

[13] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *Proc. of 11th International Software Product Line Conference, SPLC 2007*, pages 223–232, Kyoto, Japan, 2007. IEEE Computer Society.

[14] Axel Anders Kvale, Jingyue Li, and Reidar Conradi. A case study on building COTS-based system using aspect-oriented programming. In *2005 ACM Symposium on Applied Computing*, pages 1491–1497, Santa Fe, New Mexico, USA, 2005. ACM.

[15] Robert Martin. Object oriented design quality metrics: An analysis of dependencies. *ROAD*, 2(3), September–October 1995.

[16] Pavol Mederly, Marián Lekavý, Marek Závodský, and Pavol Návrat. Messaging-based enterprise integration solutions using ai planning. In *Proc. of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009*, Krakow, Poland, October 2009. To appear.

[17] Radoslav Menkyna and Valentino Vranić. Aspect-oriented change realization based on multi-paradigm design with feature modeling. In *Proc. of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009*, Krakow, Poland, October 2009. To appear.

[18] Matúš Navarčik and Ivan Polášek. Object model notation. In *Proc. of 8th International Conference on Information Systems Implementation and Modelling, ISIM 2005*, Rožnov pod Radhoštěm, Czech Republic, 2005.

[19] Jaroslav Porubän and Peter Václavík. Generating software language parser from domain classes. In *Proc. of International Scientific Conference on Computer Science and Engineering, CSE 2008*, pages 133–140, Stará Lesná, Slovakia, September 2008.

[20] Viera Rozinajová, Marek Braun, Pavol Návrat, and Mária Bieliková. Bridging the gap between service-oriented and object-oriented approach in information systems development. In D. Avison, G. M. Kasper, B. Pernici, I. Ramos, and D. Roode, editors, *Proc. of IFIP 20th World Computer Congress, TC 8, Information Systems*, Milano, Italy, September 2008. Springer Boston.

[21] Michal Stochmialek. aopmetrics project home. http://aopmetrics.tigris.org/.

[22] Valentino Vranić. Reconciling feature modeling: A feature modeling metamodel. In Matias Weske and Peter Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS 3263, pages 122–137, Erfurt, Germany, September 2004. Springer.