Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 551 – 557

ISBN 978-83-60810-14-9
ISSN 1896-7094

Separation of Crosscutting Concerns at the Design Level: an Extension to the UML Metamodel.

Adam Przybyłek
Gdańsk University, Department of Business Informatics,
Piaskowa 9, 81-824 Sopot, Poland
Email: adam@univ.gda.pl

***Abstract*—Aspect-oriented programming (AOP) was proposed as a way of improving the separation of concerns at the implementation level by introducing a new kind of modularization unit - an aspect. Aspects allow programmers to implement crosscutting concerns in a modular and well-localized way. As a result, the well-known phenomena of code tangling and scattering are avoided. After a decade of research, AOP has gained acceptance within both academia and industry. The current challenge is to incorporate aspect-oriented (AO) concepts into the software design phase. Since AOP is built on top of OOP, it seems natural to adapt UML to AO design. In this context the author introduces an extension to the UML metamodel to support aspect-oriented modelling.**

I. Introduction

A. The evolution of the aspect-oriented paradigm

THE TERM "crosscutting concern" describes part of a software system that should belong to a single module, but cannot be modularized because of the limited abstractions of the underlying programming language [20], [27], [33]. When crosscutting concerns are implemented using an object-oriented (OO) language, their code usually spreads over several core concerns [20], [26], [27]. Aspect-oriented programming (AOP) overcomes this problem by introducing a new unit of modularity—an aspect. Aspects allow programmers to avoid the well-known phenomena of code tangling and scattering, which adversely affect the readability, understandability, maintainability and reusability of the software [6], [20], [27], [30].

Programming and modelling languages exist in a relationship of mutual support. A software design co-ordinates well with a programming language when the abstraction mechanisms provided at both levels correspond to each other [26]. Successful adoption of AOP in both academia and industry has led to growing interest in aspect-oriented (AO) techniques for the whole software development lifecycle. Currently, one of the most active topics of research is modelling languages in support of aspect-orientation. Taking into account that (1) UML is considered to be the industry standard for OO system development and that (2) the AO paradigm complements the OO paradigm, it is quite natural to investigate UML as a possibility for the notation for aspect-oriented modelling (AOM) [2]–[4], [7], [18], [25], [28], [32], [34], [37]. Although UML was not designed to provide constructs to describe aspects, its flexible and extensible metamodel enables it to be adapted for domain-specific modelling [4], [23]. Thus in recent years a large number of proposals have been put forward in this area, but none of them has gained common acceptance. This paper is one more step towards closing the gap between AO concepts and UML.

B. The UML extensibility mechanisms

There are two alternative methods of extending UML to incorporate aspects: by elaborating a Meta Object Facility [1] (MOF) metamodel or by constructing a UML profile. A UML profile is a predefined set of stereotypes, tagged values, constraints, and graphical icons which enable a specific domain to be modelled [1], [7], [9], [23], [30], [35]. It was defined to provide a light-weight extension mechanism [23], termed light-weight because it does not define new elements in the metamodel of UML. The intention of profiles is to give a straightforward mechanism for adapting the standard UML metamodel with constructs that are specific to a particular domain [23]. The advantages of choosing the light-weight extension mechanism are that models can be defined by applying a well-known notation and that generic UML tools can be used. On the other hand, the drawbacks are that, since stereotypes are extensions to the existing elements, certain principles of the original elements must be observed, and consequently expressiveness is constrained.

Elaborating an MOF metamodel is referred to as heavy-weight extension and is harder than constructing a profile. It also has far less tool support. However, the metamodel constructed can be as expressive as required. Another drawback of the heavy-weight mechanism is the introduction of interdependency between specific versions of UML and its extensions. If UML changes in any way, its extensions may also have to change.

C. Motivation and goals

In the last few years, research in to AOM has concentrated on providing UML profiles, while less attention has been given to constructing heavy-weight extensions. The common

[1] Meta Object Facility (MOF) is the Object Management Group (OMG) standard, specifying how to define, interchange and extend metamodels.

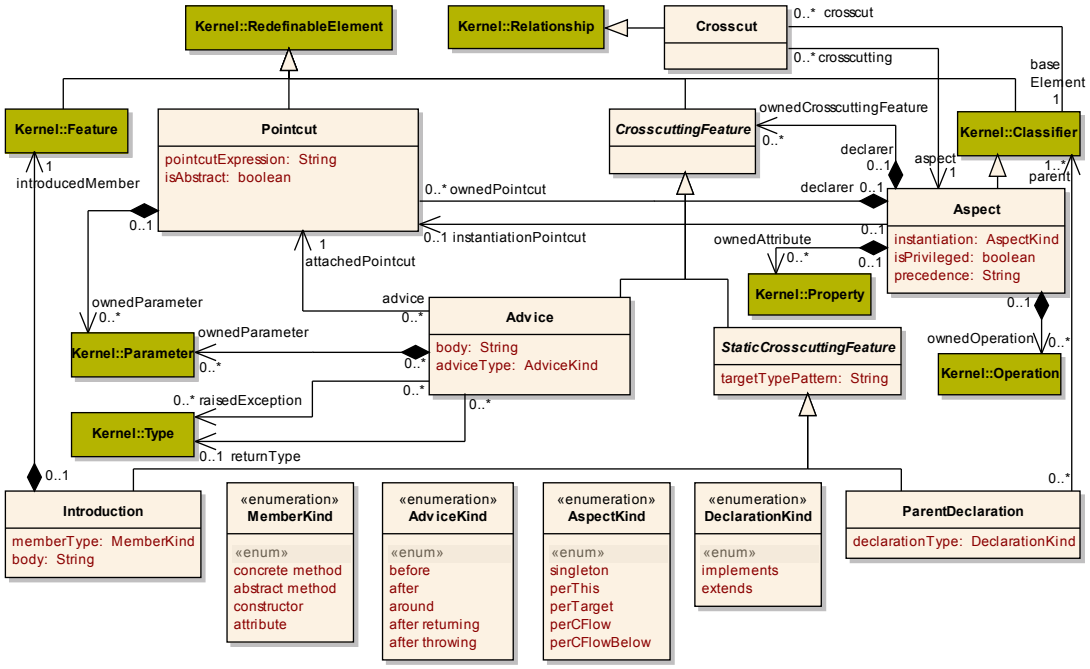

Fig. 1 . The AoUML package

practice [7], [9], [10] – [12], [22], [31], [32], [37] used to be to stereotype the class element as <<aspect>> and the method element as <<advice>>, although an aspect is not a class, nor is an advice a method. While such stereotyping was acceptable until UML 1.5, it can no longer be used; the 2.0 release requires semantic compatibility between a stereotyped element and the corresponding base element. In this context, using light-weight extensions is more an intermediate step in supporting the transition from OO modelling to AOM than a final solution.

The most valuable contributions to AOM have been made by Hachani [13], [14] and Yan [36], who proposed elaborately created and carefully specified metamodels for AspectJ. The main drawback of these extensions is the lack of graphical representation for new modelling elements. Moreover, they contain too much implementation detail and so seem to overwhelm the designer. Hachani's proposal is specified more strictly and in a more formal fashion but now needs updating, because it extends UML 1.4.

The motivation behind this research is to integrate the best practices of the existing AO extensions (particularly [5], [7], [13], [14], [16], [17], [19], [21], [29], [31], [32]) and to define a MOF metamodel that supplements the UML with means to AOM. The metamodel, which is presented in the next section, is based on the AspectJ approach to the AO paradigm. AspectJ has been chosen as the most representative AO programming language because of its mature implementation, industrial-strength tool support and wide popularity. Efforts [1], [8], [13], [28] to create a generic metamodel which could be fitted to each AO implementation have been unsuccessful, because a metamodel of this kind introduces an impedance mismatch between the design constructs and the language constructs. The conceptual differences between aspect implementations such as AspectJ, JAsCo, Spring, AspectWerkz are significant and cannot be captured effectively in a single metamodel. Moreover, generalizing aspects at the design level would be counter-productive at a time when AspectJ is squeezing out other technology at the implementation level.

II. An extension to the UML metamodel

The elaborated extension is described by using a similar style to that of the UML metamodel. As such, the specification uses a combination of notations:

- UML class diagram – to show what constructs exist in the extension and how the constructs are built up in terms of the standard UML constructs;
- OCL – to establish well-formedness rules;
- natural language – to describe the semantic of the meta-classes introduced.

The proposed extension introduces a new package, named AoUML, which contains elements to represent the fundamental AO concepts of aspect, pointcut, advice, introduction, parent declaration and crosscutting dependency (Fig. 1).

The proposal reuses elements from the UML 2.1.2 infrastructure and superstructure specifications by importing the Kernel package. Fig. 2 shows the dependencies between the UML Infrastructure [23], the UML Superstructure [24] and the AoUML package.

A. Aspect meta-class

1) Semantic s

An Aspect is a classifier that encapsulates the behaviour and structure of a crosscutting concern. It can, like a class, realize interfaces, extend classes and declare attributes and

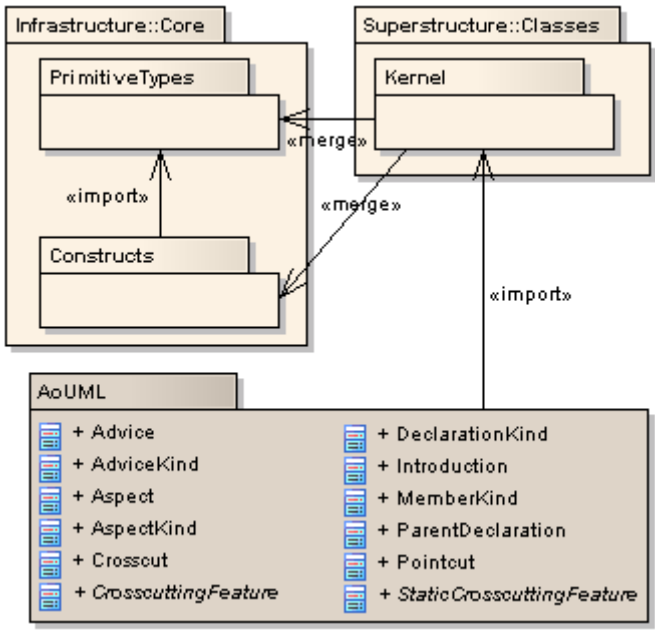

Fig. 2 . Dependencies between packages

operations. In addition, it can extend other aspects and declare advices, introductions and parent declarations.

2) Attributes

isPrivileged – if true, the aspect code is allowed to access private members of target elements as a "friend"; the default is false.

instantiation – specifies how the aspect is instantiated; the default is a singleton.

precedence – declares a precedence relationship between concrete aspects.

3) Associations

ownedPointcut – a set of pointcuts declared within the aspect.

instantiationPointcut – the pointcut which is associated with a per-clause instantiation model.

ownedCrosscuttingFeature – a set of crosscutting features owned by the aspect.

ownedAttribute – a set of attributes owned by the aspect.

ownedOperation – a set of operations owned by the aspect.

4) Notation

The aspect element looks similar to the class but has additional sections for pointcuts and crosscutting features declarations. Fig. 3 provides a graphical representation for an aspect.

B. CrosscuttingFeature meta-class

1) Semantic s

A CrosscuttingFeature is an abstract meta-class, which declares a dynamic (an advice) or static feature to be combined to some target elements.

2) Associations

declarer – the aspect that owns this crosscutting feature.

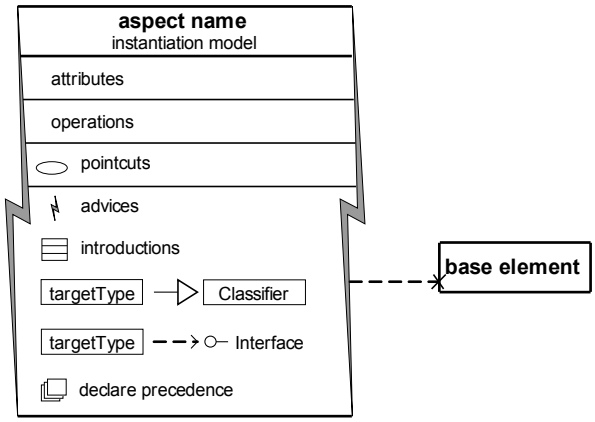

Fig. 3 . Aspect representation

C. StaticCrosscuttingFeature meta-class

1) Semantic s

A StaticCrosscuttingFeature is a crosscutting feature that can be woven with core concerns on the basis of information available before runtime.

2) Attributes

targetTypePattern – a pattern that matches classes, interfaces or aspects which are affected by the crosscutting feature.

D. Introduction meta-class

1) Semantics

An Introduction allows designers to add new attributes or methods to classes, interfaces or aspects.

2) Attributes

memberType – specifies the kind of the inter-type member declaration.

3) Associations

introducedMember – the new member which has to be added to the target type.

E. ParentDeclaration meta-class

1) Semantics

A ParentDeclaration allows designers to add super-types to classes, interfaces or aspects.

2) Attributes

declarationType – specifies the kind of the declaration.

3) Associations

parent – the type implemented or extended by the target type.

F. Advice meta-class

1) Semantic s

An Advice is a dynamic crosscutting feature that affects the behaviour of base classifiers. Each advice has exactly one associated pointcut and specifies the code that executes at each join-point picked out by the pointcut. The advice is able to access values in the execution context of the pointcut.

2) Attributes

adviceType – specifies when the advice code is executed relative to the join-points picked out.

body – the code of the advice.

3) Associations

ownedParameter – an ordered list of parameters to expose the execution context.

attachedPointcut – refers to the pointcut that defines a set of join-points at which the advice code is to be executed.

raisedException – a set of checked exceptions that may be raised during execution of the advice.

returnType – specifies the return result of the operation, if present (the "before" and "after" advice cannot return anything).

4) Constraints [2]

Advice parameters should have a unique name:

self.ownedParameter->forAll (p1, p2 | p1.name = p2.name implies p1=p2).

The before and after advice cannot return anything:

(self.adviceType = #before or self.adviceType = #after) implies (self.ownedParameter->forAll (p | p.kind = #in)).

An advice can have at most one return parameter:

self.ownedParameter->
select (par | par.direction = #return)->size() <= 1.

G. Pointcut meta-class

1) Semantics

A Pointcut is designed to specify a set of join-points and obtain the context surrounding the join-points as well. Join-points are well-defined places in the program flow where the associated advice must be executed. The purpose of declaring a pointcut is to share its pointcut expression in many advices or other pointcuts. A pointcut cannot be overloaded.

2) Attributes

isAbstract—if true, the Pointcut does not provide a complete declaration; the default value is false.

pointcutExpression—if a pointcut is not abstract, it specifies a set of join-points picked out by this pointcut; it has the same form as in AspectJ.

3) Associations

ownedParameter—an ordered list of parameters specifying what data is passed from runtime context to the associated advice.

advice—an advice that executes when the program reaches the join points.

4) Notation

The pointcut signature is as follows:

[visibility-modifier] pointcut name([parameters]): PointcutExpression

H. Crosscut meta-class

1) Semantics

A Crosscut is a directed relationship, from an aspect to one or more base elements, where the additional structure and/or behaviour will be combined.

2) Associations

aspect – the aspect specifying the crosscutting concern affecting the base element.

baseElement – refers to the classifier that is crosscut .

[2] Due to limitations on space, OCL constraints are not included for other elements.

III. An Example

This section illustrates how the presented extension works in practice by modelling the Observer pattern adopted from Hannemann and Kiczales [15]. The participants in the Observer pattern are subjects and observers. The subject is a data structure which changes over time (such as a figure), and the observer (a screen) is an object whose own invariants depend on the state of the subject (Fig. 4).

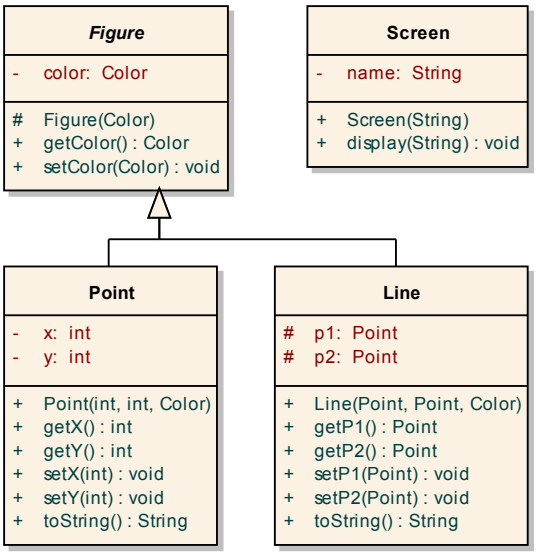

Fig. 4 . A typical scenario for the Observer pattern

The intent ion of the Observer pattern is to define a one-to-many dependency between a subject and multiple observers, so that when the subject changes state, all its observers are notified and updated automatically [15], [26]. The main problem with the OO implementation of this pattern is that it requires modification either to the structure of the classes that play the roles of Subjects and Observers or to the structure of the class hierarchy. It is therefore hard to apply the pattern to an existing design. Hanneman and Kiczales showed how the Observer pattern could effectively be implemented using AOP (Listing 1) [15].

To keep a figure display updated, the ColorObserver and PositionObserver aspects are introduced (Listing 2). Their after advices are triggered whenever a figure should be updated (the subjectChange pointcut is reached).

This paper shows how the Observer pattern could be specified using the AoUML extension. Fig. 5 gives a visual representation of Listing 1 and Listing 2.

IV. Conclusion

The evolution of the AO paradigm is progressing from programming towards the design stage. Modularization of crosscutting concerns at the design phase should provide benefits in two areas: (1) the system model will be consistent with system implementation; (2) the artefacts developed will be more reusable and maintainable.

The contribution of this research is a MOF metamodel that enriches UML with constructs for modelling crosscutting

```
public abstract aspect ObserverProtocol {
  protected interface Subject {};
  protected interface Observer {};
  private WeakHashMap perSubjectObservers;
  protected List getObservers(Subject s) {
    if (perSubjectObservers == null) perSubjectObservers = new WeakHashMap();
    List observers = (List)perSubjectObservers.get(s);
    if ( observers == null ) {
      observers = new LinkedList();
      perSubjectObservers.put(s, observers);
    }
    return observers;
  }
  public void addObserver(Subject s, Observer o) {
    getObservers(s).add(o);
  }
  public void removeObserver(Subject s, Observer o) {
    getObservers(s).remove(o);
  }
  protected abstract void updateObserver(Subject s, Observer o);

  protected abstract pointcut subjectChange(Subject s);
  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() ) updateObserver(s, ((Observer)iter.next()));
  }
}
```

Listing 1. The AO implementation of the Observer pattern

concerns. Although many existing works on AOM either do not fit the UML standard or are not complete, there is some valuable research [7], [13], [14], [21], [36] which has inspired this work. Nevertheless, the presented research offers some advantages over these previous proposals. Firstly, the extension put forward is easier to comprehend for UML users than [14] and [21], while at the same time being powerful enough to express crosscutting concerns precisely.

```
public aspect ColorObserver extends ObserverProtocol {
  protected void updateObserver(Subject s, Observer o) {
    ((Screen)o).display(s + " has changed the color");
  }

  protected pointcut subjectChange(Subject s):
    call(void Figure.setColor(Color)) && target(s);
  declare parents: Point implements Subject;
  declare parents: Line implements Subject;
  declare parents: Screen implements Observer;
}

public aspect PositionObserver extends ObserverProtocol {
  protected void updateObserver(Subject s, Observer o) {
    ((Screen)o).display(s + " has changed the position");
  }

  protected pointcut subjectChange(Subject s): target(s) &&
    !call(void Figure.setColor(Color)) && call(void Figure+.set*(..));
  declare parents: Point implements Subject;
  declare parents: Line implements Subject;
  declare parents: Screen implements Observer;
}
```

Listing 2. Definitions of two concrete observers

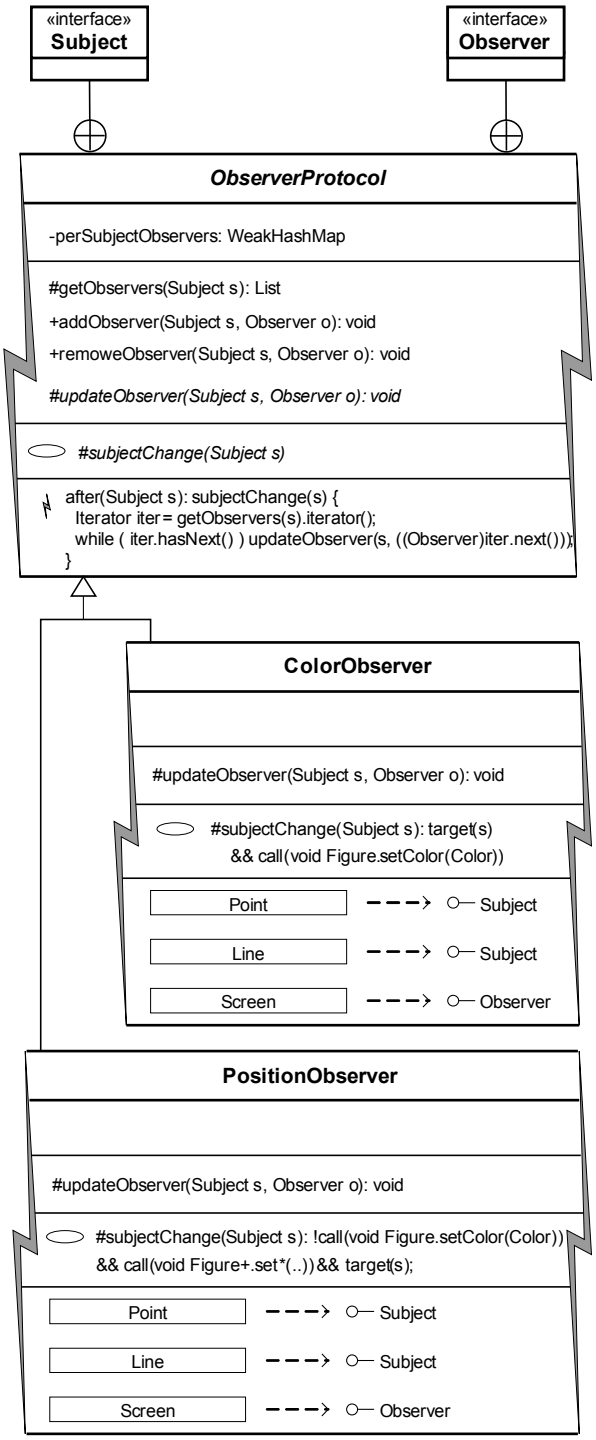

Fig. 5 . The class diagram using the AoUML extension

Moreover, in contrast to [7], [13], [14], [36] the presented metamodel provides dedicated icons for the aspect concepts. Graphical representation improves the understanding of models. Secondly, the proposal allows all aspect-related concepts to be specified in metamodel terms, so that no textual specification or notes are necessary. This means that automatic verification of the created models are simplified. Furthermore, the proposed metamodel does not modify the UML metamodel in any way; it merely adds some metaclasses. This contrasts with proposals that either are based on light-weight extensions [7] or modify the UML metamodel [13]. Thirdly, the extension put forward is adjusted to the newest UML specification (version 2.1.2). The main drawback of the proposal is that it has no support from the available modelling tools.

References

[1] Aldawud, O., Elrad, T., Bader, A.: UML Profile for Aspect-Oriented Software Development. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston (2003)

[2] Barra, E., Genova, G., Llorens, J.: An Approach to Aspect Modelling with UML 2.0. In: 5th Aspect-Oriented Modeling Workshop at UML'04, Lisbon (2004)

[3] Basch, M., Sanchez, A.: Incorporating Aspects into the UML. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston (2003)

[4] Chavez, C., Lucena, C.: A Metamodel for Aspect-Oriented Modeling. In: Proceedings of the AOM with UML workshop at AOSD'02, Enschede (2002)

[5] Clarke, S., Banaissad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley, Upper Saddle River (2005)

[6] Elrad, T., Aldawud, O., and Bader, A.: Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In: 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh (2002)

[7] Evermann, J.: A meta-level specification and profile for AspectJ in UML. In: Journal of Object Technology, vol. 6(7), Special Issue: Aspect-Oriented Modeling, 27-49 (2007)

[8] France, R., Georg, G., Ray, I.: Supporting Multi-Dimensional Separation of Design Concerns. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston (2003)

[9] Fuentes, L., Sanchez, P.: Towards Executable Aspect-Oriented UML Models. In: 10th International Workshop on Aspect-Oriented Modeling at AOSD'07, Vancouver (2007)

[10] Gao, S., Deng, Y., Yu, H., He, X., Beznosov, K., Cooper, K.: Applying Aspect-Orientation in Designing Security Systems: a Case Study. In: 16th International Conference on Software Engineering (SEKE'04), Banff (2004)

[11] Groher, I., Baumgarth, T.: Aspect-Orientation from Design to Code. In: Workshop on Early Aspects at AOSD'03, Lancaster (2004)

[12] Groher, I., Schulze, S.: Generating Aspect Code from UML Models. In: 3rd Workshop on Aspect-Oriented Modeling with UML at AOSD'03, Boston (2003)

[13] Hachani, O.: Aspect/UML: extending UML metamodel for Aspect. Research report, France (2003)

[14] Hachani, O.: AspectJ/UML: extending UML metamodel for AspectJ. Research report, France (2003)

[15] Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In 17th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02), Seattle (2002)

[16] Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley, Upper Saddle River (2005)

[17] Kande, M.M.: A Concern-Oriented Approach to Software Architecture. PhD. Swiss Federal Institute of Technology, Lausanne (2003)

[18] Kande, M.M., Kienzle, J., Strohmeier, A.: From AOP to UML - A Bottom-Up Approach. In: Proceedings of the AOM with UML workshop at AOSD'02, Enschede (2002)

[19] Kande, M.M., Kienzle, J., Strohmeier, A.: From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. Technical Report, Swiss Federal Institute of Technology Lausanne (2002)

[20] Kiczales, G. et.al.: Aspect-Oriented Programming. In: 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS, vol. 1241, pp. 220–242. Springer, New York (1997)

[21] Lions, J.M., Simoneau, D., Pilette, G., Moussa, I.: Extending OpenTool/UML Using Metamodeling: an Aspect Oriented Programming Case Study. In: 2nd Workshop on Aspect-Oriented Modeling with UML at UML'02, Dresden (2002)

[22] Mosconi, M., Charfi, A., Svacina, J.: Applying and Evaluating AOM for Platform Independent Behavioral UML Models. In: 7th International Conference on Aspect-Oriented Software Development (AOSD'08), Brussels (2008)

[23] Object Management Group: OMG UML, Infrastructure, V2.1.2. Document Number: formal/2007-11-04, http://www.omg.org/spec/UML (2007)
[24] Object Management Group: OMG UML, Superstructure, V2.1.2. Document Number: formal/2007-11-02, http://www.omg.org/spec/UML (2007)
[25] Pawlak, R. et.al.: A UML Notation for Aspect-Oriented Software Design. In: Proceedings of the AOM with UML workshop at AOSD'02, Enschede (2002)
[26] Piveta, E.K., Zancanella, L.C.: Observer Pattern using Aspect-Oriented Programming. In: 3rd Latin American Conference on Pattern Languages of Programming, Porto de Galinhas (2003)
[27] Przybylek, A.: Post Object-Oriented Paradigms in Software Development: a Comparative Analysis. In: 1st Workshop on Advances in Programming Languages at IMCSI'07, Wisła (2007)
[28] Reina, A. M., Torres, J., Toro, M.: Towards Developing Generic Solutions with Aspects. In: 5th Aspect-Oriented Modeling Workshop at UML'04, Lisbon (2004)
[29] Sapir, N., Tyszberowicz, S., Yehudai, A.: Extending UML with Aspect Usage Constraints in the Analysis and Design Phases. In: 2nd Workshop on Aspect-Oriented Modeling with UML at UML'02, Dresden (2002)
[30] Schauerhuber, A. et.al.: A Survey on Web Modeling Approaches for Ubiquitous Web Applications. Technical Report, Vienna University of Technology, 2007
[31] Stein, D., Hanenberg, S., Unland, R.: An UML-based Aspect-Oriented Design Notation. In: Proceedings of the AOM with UML Workshop at AOSD'02, Enschede (2002)
[32] Stein, D., Hanenberg, S., Unland, R.: Designing Aspect-Oriented Crosscutting in UML. In: Proceedings of the AOM with UML Workshop at AOSD'02, Enschede (2002)
[33] Störzer, M., Hanneberg, S.: A Classification of Pointcut Language Constructs. In: SPLAT'05 Workshop, Chicago (2005)
[34] Suzuki, J., Yamamotto, Y.: Extending UML with Aspects: Aspect Support in the Design Phase. In: 3rd Aspect-Oriented Programming Workshop at ECOOP'99, Lisbon (1999)
[35] Wrycza, S., Marcinkowski, B., Wyrzykowski, K.: UML 2.0 in Information Systems Modeling. Helion, Warsaw (2005)
[36] Yan, H., Kniesel, G., Cremers, A.: A Meta Model and Modeling Notation for AspectJ. In: 5th Workshop on Aspect-Oriented Modeling at UML'04, Lisbon (2004)
[37] Zakaria, A. A., Hosny, H., Zeid, A.: A UML Extension for Modeling Aspect-Oriented Systems. In: 2nd Workshop on Aspect-Oriented Modeling with UML at UML'02, Dresden (2002)