STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Number 1, 2014

ON UML BASED NOTATIONS FOR ASPECTS

GRIGORETA S. COJOCAR AND ADRIANA M. GURAN

Abstract. Separation of concerns was always an important factor in designing easily maintainable and evolvable software systems. However, practice has shown that it is not easy to clearly separate all the concerns from a software system in analysis, design and implementation phases. Using just one programming paradigm most concerns can be clearly separated, but there are still a few concerns whose design and implementation is entangled with other concerns. New programming paradigms that try to complement the existing ones by providing new ways for a better separation of concerns have been developed. Aspect oriented paradigm is one of the new paradigms. Although the number of languages supporting aspect oriented programming has increased, there is no generally accepted notation for aspects in UML. In this paper we provide an analysis of the existing UML-based notation for aspects in the class diagram using a set of comparison criteria.

1. Introduction

Separation of concerns [20] was always an important factor in designing easily maintainable and evolvable software systems. However, practice has shown that it is not easy to clearly separate all the concerns from a software system. Using just one programming paradigm most concerns can be clearly separated, but there are still a few concerns whose design and implementation is entangled with other concerns. New programming paradigms that extend the existing ones in order to better separate the concerns that are still entangled have been developed. Among these we mention the aspect oriented paradigm (AOP) that usually extends the object-oriented paradigm [18].

There are already a number of aspect oriented languages as AspectJ [4] and AspectC++ [3] that provide new language constructs for implementing the crosscutting concerns. At this moment there is no generally accepted design

Received by the editors: May 5, 2014.

2010 *Mathematics Subject Classification.* 68N19, 68N99.

1998 *CR Categories and Descriptors.* D.2.2[**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering*; D.1.m [**Software**]: Programming Techniques – *Miscellaneous*.

Key words and phrases. aspect oriented paradigm, design, UML class diagram.

notation that supports the design of aspect oriented systems. The need for a design notation is justified by the following advantages:

- it would ease the development of aspect oriented systems;
- it would make the understanding of an aspect oriented system much easier (by using a graphical representation);
- it would serve as a basis for assessing the impact of crosscutting concerns on their base classes (core classes);
- it would allow the adaptation and reuse of existing design constructs.

The Unified Modeling Language (UML) is a graphical language for visualizing, constructing, specifying and documenting the artefacts of a software system. The goal of UML is to provide tools for analysis, design, and implementation of software based systems to all parties involved: developers, system architects, etc [7, 22, 28]. One of the main advantages of using UML is that it has defined a set of modeling concepts that are now generally accepted, and it also contains visual representation of the defined concepts that are easy to understand and interpret by humans.

The concepts defined by UML can be used to represent the static structure (such as classes, components, node artifacts) or the behavior (such as activities, interactions, state machines) of the software system [27]. The concepts can be used to build different kind of diagrams (i.e., class diagrams for the static structure and sequence diagrams for the behavior).

Even though UML contains a very large set of concepts, it does not include all the concepts that may appear during the development of different kinds of software systems, mainly because some of the concepts are specific to a certain application domain. That is why, the language also contains extension mechanisms that allow us to add new modeling elements, modify the specification of the existing ones or change their semantics [7].

The extension mechanisms provided by UML are stereotypes, tagged values, constraints, and profiles [27]:

- A *stereotype* extends the vocabulary of the UML, by allowing the creation of new kinds of modeling elements that are derived from existing ones, but that are specific to a particular domain or problem. The information content of the stereotyped element is the same as the existing model element [7, 22].
- A *tagged value* extends the properties of a UML stereotype by allowing the creation of new information in the stereotype's specification [7]. It is a name-value pair that denotes a characteristic of the corresponding stereotype.
- A *constraint* extends the semantics of a UML modeling element allowing addition of new rules or changing the existing ones [7]. The

constraints may be written as free-form text or using the Object Constraint Language (OCL) [29] for a more precise specification of the semantics.

- A *profile* is a UML model with a set of predefined stereotypes, tagged-values, constraints and base classes. It also selects a subset containing only those modeling concepts that are needed and should be used for a particular application area. This mechanism is not considered a first-class extension mechanism, as it does not allow the addition or modification of existing elements. The intention of profiles is to give a simple and easy to use mechanism for adapting an existing set of modeling concepts with constructs that are specific to a particular domain, platform, or method [7, 22].

In UML a *classifier* is a mechanism that describes structural and behavioral features. Classifiers include classes, associations, interfaces, datatypes, components, nodes, use cases, etc. The classifiers that are usually represented in a class diagram are classes and interfaces.

The main contribution of this paper consists in using a new set of comparison criteria for analyzing UML based notations for the aspects displayed in an UML class diagram. The considered criteria are useful in deciding what notation to use for representing aspects as they show the expressiveness of the notation, how easy will be to learn and adopt the new notation.

The paper is structured as follows. In Section 2 we present the new concepts introduced by the aspect oriented paradigm. The considered notations are briefly described in Section 3. An analysis of the described proposals is given in Section 4. Section 5 presents related work. Some conclusions and future research directions are given in Section 6.

2. AOP Concepts

The aspect oriented paradigm introduces new concepts: *join point*, *pointcut*, *advice*, *aspect* and *introduction* for the design and implementation of crosscutting concerns, and *weaving* for building the final software system. Aspect oriented programming can be used only for crosscutting concerns, the core concerns are still designed and implemented using the base programming paradigm, that usually is object-oriented programming, but it can be any other programming paradigm.

2.1. Join point.

A *join point* is a well-defined point in the execution of a program. Any software systems can be seen as a sequence of execution points like: assignments, conditional statements (if, switch), loop statements (for, while, do-while or repeat), function/method calls, function/methods executions, etc. regardless of the programming paradigm used for developing the system. Aspect oriented programming only uses some of these points, called join points, in order to add new behavior.

The types of execution points that can be used for developing crosscutting concerns depend on the aspect oriented language. AspectJ [4] offers the greatest variety of execution points, like: method calls, method executions, object instantiations, constructor executions, field references and handler executions, etc., while others allow only a small subset. In Spring AOP [23] a join point always represents a method execution.

2.2. **Pointcut.** The execution of a software system consists of many join points. However, not all of them are necessary for the design and implementation of crosscutting concerns. A *pointcut* selects join points, and exposes some of the values in the execution context of those join points. AspectJ [4] and AspectC++ [3] allow pointcuts to be defined as abstract in order to be able to define reusable aspects and aspect libraries.

2.3. **Advice.** A pointcut allows selecting join points from the software system, however they do not change its behavior. An *advice* defines crosscutting behavior and it is defined in terms of pointcuts. The code of an advice runs at every join point selected by its pointcut. There are different options as to when the code of the advice is executed relatively to the corresponding join point(s). The aspect oriented languages developed so far allow three types: before, after and around the join point.

- *Before*: the advice code is executed before the selected join point. This type of advice does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- *After*: the advice code is executed after the selected join point. There can be three situations, depending on the execution of the join point:
  - *After returning*: the advice code is executed only if the join point execution completes normally.
  - *After throwing*: the advice code is executed only if the join point execution ends by throwing an exception.
  - *After (finally)*: the advice code is executed regardless of the means by which the selected join point exits (normal or exceptional return). This type of advice is usually called *after finally*, because of its similarity with the `finally` clause of the `try-catch` block from programming languages.
- *Around*: the advice code surrounds the selected join point. It can perform custom behavior before and after the selected join point. It can also decide whether the selected join point should still be executed

or not (called *proceeding*), or it may cause multiple executions of the selected join point.

2.4. **Introduction.** It is sometimes necessary to modify the static structure of a type (by adding new members - attributes/methods or by modifying its inheritance hierarchy) in order to design and implement a crosscutting concern. Even though advices add new behavior to existing types, they do not modify their static structure. An *introduction* allows developers to extend the static structure of existing types. New methods and/or attributes can be added, or the type inheritance hierarchy can be modified (by adding new interfaces or by modifying the base type of the existing type).

2.5. **Aspect.** An *aspect* is a new kind of type specified by the aspect oriented paradigm that is used to implement one crosscutting concern in a modular way. An aspect is similar to a class, it can contain attributes and methods declarations but it also encapsulates pointcuts, advice and introductions. Because of their similarity with classes, an aspect declaration is often almost the same as a new class declaration, where the `class` keyword is replaced with the `aspect` keyword. In some aspect oriented languages aspects (i.e., AspectJ, AspectC++) can inherit from other classes, implement some interfaces or even inherit from other aspects. However some constraints must be followed when inheriting from another aspect. For example, in AspectJ an aspect can inherit only from an abstract aspect.

2.6. **Weaving.** When the aspect oriented paradigm is used for developing software systems, the core concerns are developed independently of the crosscutting concerns. However, in the end, they still have to be put together in order to obtain the final executing system. *Weaving* is the process that produces the final systems, and the *weaver* is the tool used to produce it.

The weaver takes some representation of the core concerns (source code or binaries), some representation of the crosscutting concerns (source code or binaries) and produces the output, which is often a binary representation.

Some aspect oriented languages allow the weaving process to take place at different times. For example, AspectJ allows three different times: compile-time, post-compile time and load-time. Other languages allow only one possibility which is either compile-time or run-time.

The approach used for weaving depends on the aspect oriented language: AspectJ uses byte-code modification, Spring AOP uses dynamic proxies, while AspectC++ uses source code preprocessing.

Not all the concepts introduced by the aspect oriented paradigm can and should be represented in a class diagram. The join points from a software system do not provide any relevant information about the static structure of the system. However, the visual representation of other concepts can provide useful informtion to the developers. For example, some consider that one of the main difficulties when using the aspect oriented paradigm is that the control flow of the system will be difficult to follow and understand since not all the relevant data about a piece of code can be seen at that code. Some additional information may exist in the aspect that affect that part of code. That is why we consider that adding aspects to the class diagram may ease the understanding of the overall static structure of a software system. The information that should be represented in a class diagram is:

- The aspects that are used for building the system, and their type (abstract or concrete). The internal static structure of an aspect is important as it will show, besides the normal fields and methods, the defined pointcuts together with the collected context, the type of the pointcut (abstract or concret), and the defined advice and their type (before, after, around).
- If and how they change the static structure of other existing elements from the class diagram (classes, interfaces). It should represent the type whose static structure will be modified either by introducing new members (fields, methods, constructors, etc.) or by modifying the inheritance hierarchy of the type (adding a base class or implementing interfaces).
- Relationships with other elements from the class diagram. We consider important the following relationships:
  - Aspect-aspect: An aspect may inherit from another aspect, or an aspect may have precedence over another aspect during weaving.
  - Aspect-interface: An aspect may implement one or more interfaces.
  - Aspect-class: An aspect may inherit (or extend) from a class, it may modify the static structure of an existing class, or it may modify the behavior of a class through one or more advices.

3. Existing UML-based Notations for Aspects

During the last fifteen years, many attempts to identify an appropriate notation of aspect oriented design have been made. Most approaches focused on introducing new modeling elements for the concepts defined by AOP: aspect, advice, pointcut and the relevant relationships, while other approaches (like the one proposed by Herrero et al. [14]) focused on a particular crosscutting concern and tried to introduce special notations for the elements needed to design that crosscutting concern. In this study we have included only the approaches that tried to represented the new concepts introduced by AOP.

Suzuki and Yamamoto were the first to extend the UML with concepts for the design of aspect-oriented programs [25]. The aspect is a stereotype derived from the Classifier element. Aspect can have attributes, operations and relationships. The operations corresponding to an introduction or an advice are represented using the *weave* stereotype followed by the *advice* or *introduce* constraint. The signature of a weave operation also shows which elements (e.g. classes, methods and variables) are affected by the aspect. Relationships of an aspect include generalization, association and dependency. The relationship between an aspect and the classes that the aspect affects is a stereotype of abstraction dependency defined in the UML, called *realize*. They also introduced the stereotype *woven class* into the Class element in order to represent a woven class that should specify the source class and the aspect used to generate it using a tagged value. They do not propose new notations for concepts like pointcut or advice. The proposed model was used to design the Observer design pattern.

Aldawud et al. [2] have proposed an UML profile for aspect oriented modeling. The profile contains an *aspect* stereotype of the Class classifier for representing an aspect, and a *control* stereotype of the Association element for representing relationships, however what kind of relations can be represented using this stereotype is left as a further research question. No notations are provided for join points, pointcuts or advices. They also proposed the use of complex statecharts for modeling aspects behavior [1]. No example of using the profile for designing a crosscutting concern is given.

Kande et al. [17, 16] have first studied the suitability of using UML for representing aspect oriented designs of software system, and then proposed new notations for aspect oriented concepts. The aspect is a stereotype of the Classifier, and a new stereotype of UML collaboration and connection points have been proposed in order to represent the relationships between aspects and other classifiers and pointcuts. Advices and introductions are each represented in their special compartment of the aspect stereotype, and pointcuts are represented as connection points. The proposed design model was used to represent the *Logging* crosscutting concern.

Zakaria et al. [30] have proposed another UML extension for designing aspect oriented systems. The proposal is a more detailed version of the one made by Aldawud et al. in [2]. An aspect is represented as a stereotype of the Class classifier, a pointcut is also modeled as a stereotype of the Class, and advices are represented as stereotypes of the Operation element. They define different stereotypes for each type of advice: before, after, after returning, after throwing and around. For aspect-class relationships they proposed the use of the UML association model element tagged with different values for different kind of associations (control, track, validate, etc), and for aspect-aspect relationships they either used the UML generalization relationship or the *dominates* stereotype of the Association element. The proposed extension was used to model MoveTracking crosscutting concern.

Pawlak et al. [21] introduced the notion of groups in order to represent objects that are not necessarily having the same class or superclass, but that could be the result of a pointcut selection criteria. An aspect is represented as an aspect-class stereotyped with *aspect*, and it may contain special methods called aspect-methods. These aspect-methods are stereotyped with *before*, *after*, *around*, corresponding to the advice type. To represent pointcuts, they use pointcut relations from aspect-class to another class or group. The pointcut relations are oriented associations stereotyped with *pointcut* and the roles have special semantics. The notation was used to design Authentication and Session aspects.

Stein et al. [24] have proposed an aspect-oriented design model that extends UML using the standard extension mechanism, called AODM. They started from the concepts introduced by AspectJ [4] and they tried to propose the most suited modeling element for each concept. An aspect is a stereotype of the Class classifier (*aspect*), pointcuts and advices are represented as operations of special stereotypes (*pointcut* or *advice*), join points are represented as links (some of the links may be stereotyped, depending on the join point kind), and introductions are represented as special stereotype of collaboration template. They also tried to model the weaving mechanism using a use case stereotype. For aspect-class or aspect-aspect relations they mostly used the already defined relations for the Class classifier with some additional constraints. They also introduced a new relation, called *crosscut*, in order to describe the other modeling elements (classes, interfaces or aspects) that are affected by the associated aspect. The proposed design model was used to design the Observer design pattern [11].

Jacobson and Ng [15] in their book *Aspect-Oriented Software Development with Use Cases* also proposed the extension of UML in order to graphically represent aspects. An aspect is a stereotyped classifier, named *aspect*, with two compartments: one for pointcuts and one for class extensions. Aspects have a class extensions compartment to overlay class extensions onto existing class. With AOP the overlay can be achieved with introductions or advices. Each class extension contains a subset of features (attributes, operations and relationships) that are used to represent which class and which operation is extended by the aspect (using introduction or advice). The pointcuts compartment contains the pointcuts defined by the aspect. The authors also introduced the notion of parametrized pointcuts (marked with $<$ and $>$) that can be used to display an operation extension at multiple join points.

Basch and Sanchez [6] have taken another approach in representing aspects in class diagrams. They have considered the *package* UML element as the most appropriate to be used for aspects. These packages are stereotyped with the *aspect* stereotype. Join points are represented as circles with a cross inside. Inside the aspect package, a class diagram is used to show which other component classes are crosscut by the corresponding aspect. The aspect package should also include interaction diagrams to display the behavior of the aspect. No example of using the proposed approach to design a crosscutting concern is given.

Zhang has also considered using the *package* UML element to represent aspects [31]. The aspect contains two compartments: one for pointcuts and one for advices. The pointcut and the advice are also represented as stereotyped packages. The proposed model was used to design the Logging crosscutting concern.

Some researchers, like Han et al. [13] or Chavez and Lucena [8] have taken the approach of first defining meta-model for AspectJ, and then, based on this metamodel, to define graphical notations for the concepts introduced by AOP.

4. Analysis of UML Based Notations for Aspects

In this section we compare the previously described UML-based notations for aspects from different points of view: the UML representation used for displaying the new concepts introduced by AOP in the class diagram (aspect, advice, pointcut, join point, introduction), the concepts that were represented, the kind of relationships considered between elements from the class diagram (as discussed in Section 2) and statically vizualizing the affected parts of the system.

4.1. **AOP Concepts Represented.**

All the proposals described in Section 3 have considered a way of representing the aspect concept into the class diagram. Most of them also considered representing the advice [17, 21, 24, 25, 30, 31] and the pointcut [15, 17, 21, 24, 31]. Very few proposals considered representing introductions [15, 24, 25] and even fewer considered representing join points [6, 24].

4.2. **UML Notations for AOP Concepts.**

Most of the approaches described in Section 3 represent the aspect starting from the *Class* classifier enhanced with the *aspect* stereotype [2, 21, 24, 30]. Others have considered using the *Classifier* as a base for aspect [15, 17, 25]. Only a few proposals considered using a non classifier as a starting element, namely the package with two compartments for pointcuts and advices [6, 31].

For the rest of the concepts introduced by AOP, there is very little consensus related to the appropriateness of a chosen representation. In the following, we will present the proposed representation for the other concepts:

- *Join point* - There are only two proposals for representing them, consisting in links [24] and a notation in the form of a circle with a cross inside [6].
- *Pointcut* - There is very little overlapping between the proposals for representing pointcuts. In [17] and [24] pointcuts are represented using the *pointcut* stereotype, while Pawlak et al. [21] propose a representation based on an association from an aspect class towards a classifier stereotype with *pointcut.* Zakaria et al. [30] model a pointcut based on the *Class* classifier and by providing a link to its aspect by a *has pointcut* association. Zhang sees it as a stereotype package [31]. Jacobson and Ng [15] proposed representing pointcuts as operations in their own compartment of the aspect stereotype.
- *Advice* - Stein et al.[24] and Zakaria et al.[30] model an advice as an UML operation of a stereotyped named *advice*; while Suzuki and Yamamoto [25] provide a suggestion of using a constraint for the corresponding weave and Kande et al. [17] suggest to represent it as a compartment in the new *aspect* classifier.
- *Introductions* - There is only one approach, proposed by Stein et al. [24], in explicitly modeling introductions that uses parameterized templates. The class extensions compartment introduced by Jacobson et Ng [15] in their proposal can also be used to represent introductions, however from their representation it is not very clear how the extensions will be realized (introduction or advice).

4.3. **Relationships.** Only a few proposals have taken into consideration the relationships that the aspects can have with the other elements from the class diagram (classes, interfaces). Aldawud et al. [2] proposed to use the *control* relationship to represent which other classes the aspect code controls. Suzuki and Yamamoto [25] proposed the usage of the already existing *realize* relationship to represent an aspect and the classes that the aspect affects. Zakaria et al. [30] proposed to use one of the newly introduced *control, track, report, customize, validate, save, handleerror, handleexception* relationships to represent the relation between an aspect and a class. Each aspect should have at least one association with a class in order to affect the system. They also introduced the *dominates* relationship between two aspects in order to represent the precedence between those aspects. Kande et al. [17] introduced the *binding* relationship to specify what class of objects an instance of the aspect can be bound to. Stein et al. [24] introduced the *crosscut* relationship between an aspect and a class to specify that the aspect affects the class. The crosscut relationship also implies that the aspect requires the presence of the class in order to behave as expected.

4.4. **Vizualising Affected Parts.** Very few proposals considered explicilty vizualizing the parts that will be affected by introducing one or more aspects, however for some proposals the structure of the aspect notation or the relationships introduced can be used for determining the parts of the software system that are affected by the aspects presence. Jacobson and Ng proposal [15] display the affected parts in the *Class extensions* compartment that contains all the classes from the system that will be affected by the aspect (either statically by introductions or dynamically through advice). The aspect notation proposed by Kande et al. [17] contains compartments that display introductions, meaning that those classes will be affected by the aspect. Also, the binding relationship introduced by them show other affected parts (through dynamic crosscutting). The aspect-class relationships described in Section 4.3 can also be considered as relationships that show the affected parts of the software system.

The analysis shows that most approaches have considered representing only a subset of the new concepts introduced by AOP. However in order to properly communicate and describe an aspect oriented design all the concepts should be possible to represent. The results of the analysis also show there is still a need for further investigation in order to represent and include aspect oriented concepts into the UML class diagram of a software system that can be easily adopted by software developers.

5. Related Work

Chitchyan et al. [9] analyzed four proposed approaches that provided support for crosscutting concern at design level: Composition Patterns [10], Aspect-Oriented Component Engineering [12], Hyperspaces [26] and Suzuki and Yamamoto's approach [25]. They defined a set of "good design" criteria that is used for their analysis. The set contains traceability, change propagation, reusability, comprehensibility, flexibility, ease of learning and use, and parallel development.

Asteasuain et al. [5] analyzed the UML suitability for specifying and visualizing aspect oriented design. They considered two types of UML extensions that can be used: a general one that extends the UML with specific concepts from AOP, and a specific one that extends UML only with a few concepts from AOP that are used for the design and implementation of a particular crosscutting concern. From this perspective they compared the extension proposed by Suzuki and Yamamoto [25] considered as a general one, with the extension proposed by Herrero et al. [14] for the synchronization crosscutting concern, considered as a specific one. The criteria used for comparison are the same with the ones introduced and used by Chitchyan et al. [9] in their analysis.

Lovavio et al. [19] described in their article different notations that were proposed for Aspect Oriented Software Development (AOSD). They first identified the notions used for AOSD (i.e., crosscutting concern, aspect, advice, join point, weaving, relationships, etc.) and then they presented the UML-based notations proposed until that time. However, their focus was not on analyzing and comparing these notations but on using UML for including and modeling aspects early in the software system development life-cycle (not only in the implementation phase).

Our study is different from the above mentioned studies in that it focuses on comparing the way the static structure of an aspect oriented software system can be modeled using UML. Our approach focuses only on the inclusion of aspects in the class diagram, on what AOP concepts can be represented in this diagram and how, and on their relationships with other elements that are displayed in the diagram (classes and interfaces). The first two studies consider other comparison criteria like the ease of tracing crosscutting requirements to design and even to code or how easily the design can be reused for other software systems. The study made by Lovavio et al. also discusses the way the AOP concepts can be represented in UML, but their focus is not mainly on the class diagram. They are interested more in the way UML can be used for identifying and managing crosscutting concerns during requirements and analysis phases.

6. Conclusions and Further Work

We have described in this paper nine existing proposals to introduce aspects into the UML class diagrams of a software system. We have also provided an analysis of the described proposals using different criteria: the aspect oriented concepts represented, the UML notation proposed, the relationships introduced, and if they allow vizualing the affected parts of the software system. Even though there already exist many proposals, none of them is generally accepted nor used in the literature to describe the static structure of an aspect oriented software system. The reasons for this lack of acceptance may be that some of them are incomplete, some of them have as starting point the AspectJ language meaning that some of the features introduced are particular to AspectJ but not to all aspect oriented languages, or the notations proposed are difficult to use for different crosscutting concerns. As described in Section 3, the examples used for presenting the notation are usually small (the Observer pattern, Logging, MoveTracking, Authentication and Session).

Further work should be done in the following directions:

- To propose another notation for representing aspects and their relationships in UML class diagram, starting from the good points of the existing notations.
- To apply the new notation for designing different crosscutting concerns.
- To facilitate the use of the proposed notation by developing a profile or an add-in/plug-in package for an existing CASE tool.

References

[1] Omar Aldawud, Atef Bader, C. Constantinos, and Tzilla Elrad. Modeling intra object aspectual behavior. In *Automating Object-Oriented Software Development Methods Workshop at ECOOP 2001*, pages 1–6, 2001.

[2] Omar Aldawud, Tzilla Elrad, and Atef Bader. A UML Profile for Aspect Oriented Modeling. In *Aspect Oriented Programming Workshop at OOPSLA 2001*, pages 1–6, 2001.

[3] AspectC++ Homepage. http://www.aspectc.org/.

[4] AspectJ Project. http://eclipse.org/aspectj/.

[5] Fernando Asteasuain, Bernardo Contreras, Elsa Estevez, and Pablo R. Fillottrani. Evaluation of uml extensions for aspect oriented design. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*. Madrid, Spain, 2004.

[6] Mark Basch and Arturo Sanchez. Incorporating aspects into the uml. In *Proceedings of the Aspect Oriented Modeling Workshop at AOSD*, 2003.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley Professional, 2005.

[8] Christina Chaves and Carlos Lucena. A metamodel for aspect oriented modeling. In *Workshop on Aspect-Oriented Modeling with UML (held in conjunction with the 1st Aspect Oriented Software Development Conference AOSD*, 2002.

[9] Ruzanna Chitchyan, Ian Sommerville, and Awais Rashid. An analysis of design approaches for crosscutting concerns. In *Workshop on Aspect-Oriented Design (held in conjunction with the 1st Aspect Oriented Software Development Conference AOSD*, 2002.

[10] Siobhán Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 5–14. IEEE Computer Society, 2001.

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

[12] John Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6):713–734, Dec 2000.

[13] Yan Han, Gnter Kniesel, and Armin B. Cremers. Towards visual aspectj by a meta model and modeling notation. In *Proceedings of 6th Aspect Oriented Modeling Workshop at AOSD*, 2005.

[14] J.L. Herrero, F. Sanchez, F. Lucio, and M. Torro. Introducing Separation of Aspects at Design Time. In *Aspect-Oriented Programming (AOP) Workshop at ECOOP 2000.* 2000.

[15] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases.* Addison Wesley, 2004.

[16] Mohamed M. Kande, Jorg Kienzle, and Alfred Strohmeier. From AOP to UML- A Bottom-Up Approach. In *Proceedings of the 1st International Workshop on Aspect-Oriented Modeling with UML*. Enschede, The Netherlands, 2002.

[17] Mohamed M. Kande, Jorg Kienzle, and Alfred Strohmeier. From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. Technical report, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2002.

[18] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume LNCS 1241, pages 220–242. Springer-Verlag, 1997.

[19] Francisca Losavio, Alfredo Matteo, and Patricia Morantes. UML Extensions for Aspect Oriented Software Development. *Journal of Object Technology*, 8(5):85–104, 2009.

[20] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[21] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A uml notation for aspect-oriented software design. In *Proceedings of the Aspect Oriented Modeling with UML workshop at AOSD (2002*, 2002.

[22] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[23] Aspect Oriented Programming with Spring. http://docs.spring.io/spring/docs/4.0.3.RELEASE/spring-framework-reference/htmlsingle/#aop.

[24] Dominik Stein, Stefan Hanenberg, and Rainer Unland. An UML-based Aspect-Oriented Design Notation for AspectJ. In *AOSD 2002*, pages 1–7, 2002.

[25] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Aspect-Oriented Programming (AOP) Workshop at ECOOP'99*, pages 14–18. 1999.

[26] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, May 1999.

[27] UML 2.4.1 Superstructure. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[28] Unified Modeling Language(UML). http://www.uml.org/.

[29] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] Aida Atef Zakaria, Hoda Hosny, and Amir Zeid. A uml extension for modeling aspect-oriented systems. In *Second International workshop on Aspect-Oriented Modeling with UML at UML 2002.* 2002.

[31] Gefei Zhang. Towards aspect-oriented class diagrams. In *Proceedings of 12th Asia-Pacific Software Engineering Conference*, pages 763–768, 2005.

Babeş-Bolyai University, Department of Computer Science, 1, M. Kogălniceanu street, 400084 Cluj-Napoca, Romania

E-mail address: `grigo@cs.ubbcluj.ro`

E-mail address: `adriana@cs.ubbcluj.ro`