Evaluation of UML Extensions for Aspect Oriented Design

Fernando Asteasuain, Bernardo Contreras,
Elsa Estévez, Pablo R. Fillottrani *
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253 – (8000) Bahía Blanca
Argentina
{fa,bec,ece,prf}@cs.uns.edu.ar

Abstract. There are some concepts, such as synchronization, error checking, distribution, security, and others, that are dissimilar from the basic functionality of a system, and they are not correctly encapsulated when using traditional programming methodologies. Thus, the code corresponding to these concepts result disseminated through the complete system, and therefore the overall software quality decreased. Aspect Oriented Programming (AOP) provides solutions for *code scattering* by promoting the separation of these concepts through mechanisms that allow an adequate abstraction and composition, in order to finally integrate them into a complete system. The unit that aggregates the entire semantic of each of these concepts is defined as *aspect*. The explicit disunion of these concepts enables a more reliable management of software complexity, and results in a final product of better quality. The key principles applied are modularity, and separation of concerns. During the first stages, AOP research was focused to the implementation and codification level. In this paper, we evaluate different UML extensions that include aspects during the design phase. We include examples to illustrate them, and we define evaluation criteria, and compare them using these criteria.
Keywords: Aspect Oriented Programming, extensions to UML, aspect oriented design, design evaluation criteria.

1 Introduction

Aspects [11], basic programming unit of AOP [1, 10], can be identified during the design and implementation phases, although they will effective manifest during the implementation or codification activities [6]. When aspects appeared during the implementation, software engineers add or remove them easily due to the *plug-in plug-out* nature of the elements of this paradigm. In contrast, for those aspects identifiable during the design phase it is not possible to take any action, and we should only wait until the codification phase. This fact promotes the need to develop mechanisms and tools that enable aspect manipulation and specification during the design phase.

* Grupo de Ingeniería de Software - Miembro del IICyTI

Including aspect special specification during the design, simplifies the development of aspect oriented software. First, documentation and learning are facilitated. Considering aspects as design constructors, software engineers can recognize them in a higher level of abstraction and during an early stage of the development process. Aspect designers and software engineers initially involved in AOP can learn and document models in a more intuitive fashion. Second, it promotes aspect reuse. The way in which aspects are documented and further studied, the considerations about how aspects are designed, and how they depend on their associated abstractions, are crucial points for aspect reuse. Finally, it promotes evolutionary development. If aspects are considered during the design phase, it is possible to produce incremental aspect oriented software. Briefly, the considerations and motivations for considering aspects during the design phase are semantically equivalent to those that justify to include the design phase during traditional software development.

Kandé et al. [5] define two main topics that are needed to evaluate an aspect oriented design model. First, the model needs to define clearly which are the elements that crosscut the basic functionality. Without this information, the representation and reasoning about aspects structure and its behavior in the system is more difficult. Second, it needs to provide mechanisms and means to separate the elements that crosscut functionality from basic system components, and also must encapsulate these elements in a special artifact.

The most direct and natural approach is to investigate UML adaptability [7], as a specification language for aspect modeling. UML is the most popular design language in software engineering, and it is considered as a standard. It provides a powerful set of modeling tools for system analysis and design, for the definition of the system architecture, and also for specifying the system behavior. It also provides, a set of mechanisms to extend or adapt the language to a specific domain. This characteristic of extensibility make it more suitable for AO design.

In this paper, we analyze UML for specifying aspect oriented design. In the next section, we discuss if UML, as it is defined, is able to express aspects, and if it is able to satisfy the two topics mentioned by Kandé et al. These concepts are illustrated by an example. In section 3 we evaluate two approaches to manipulate aspects during the design phase, based on different UML extensions. In section 4, we present different design evaluation criteria to compare these approaches. Finally, we present our conclusions and future work.

2 UML for Aspect Oriented Design

Suppose we need to model a system for the following requirements. We consider two classes *Client*, and *Supplier*. The *Supplier* class includes a method *invoice(q:int)* that is used to store the different invoices that the supplier receives from the clients. One of the system requirements stipulates that is needed to save all the information related to the requests that the supplier receives from his clients. This requirement is a concept that crosscut different parts of the system, and its code will be disseminated among several classes. The im-

plementation of this requirement is naturally supported by an aspect oriented development. The *Log* aspect will be responsible for the implementation of this requirement and is shown in Figure 1.

```
Aspect Log {
Add Attribute: file:File in  Supplier class.
Add Method: initLog(name:string) {
                    createFile(file, name)
                    }
                    in Supplier class.
Connection Points:
                    calls_to_Invoice = calls to Supplier.invoice(q:int)
Notices:
          After calls_to_Invoice (C:Client, S:Supplier) {
                    S.file.save(S, received an invoice from:, C);
                    S.file.close();
                    }
              }
```

Fig. 1. *Log* Aspect: saves the requests to a *Supplier*.

The aspect introduces the method *initLog* in the *Supplier* class. This method creates the file where all the information about invoices will be saved. The file is managed through the *file* attribute, which is also introduced by the aspect in the *Supplier* class. The semantic for the *Connection Point* and for the *Notice* defined in the aspect is quite simple: after each request to the method *invoice* of the *Supplier* class, the relevant information is saved on the file. Figure 2 shows the class diagram for the example.

Collaboration diagrams may be used to model the dynamic structure of the system. The diagram shown in Figure 3 shows a behavioral view based on the interactions between an object *c* from *Client* class and an object *s* from *Supplier* class. The object *c* invokes the method *invoice* defined in *s*, shown by the arrow labeled with *invoice()*. One approach to show the aspect semantic, is to apply the technique used to model interceptions to method invocations in [5]: it consists of adding a new class whose instances are interposed between the interacting objects. The application of this technique is shown in Figure 3. An interceptor object *i* is placed between the client and the supplier. This object implements the *Log* aspect, and also has an interface that supports all the methods of the *Supplier* class that can be invoked by the *Client* class, plus a mechanism to resend the invocations intercepted to the *Supplier* class.

UML, as it has been used, admits the structure and the interactions modeling of the aspect oriented system. Although, the resultant models suffer important disadvantages [5]. First, the crosscutting concepts are not modeled correctly. The design of the *Log* aspect results disseminated all over the system because it is

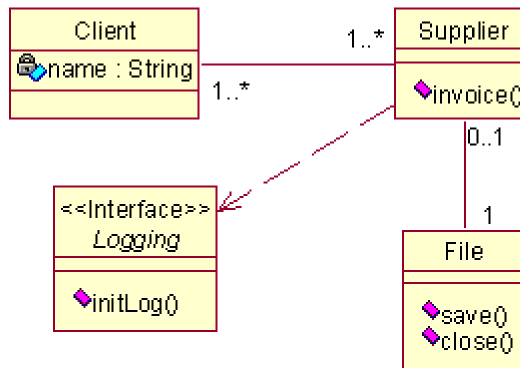

Fig. 2. Static structure of the aspect semantic.

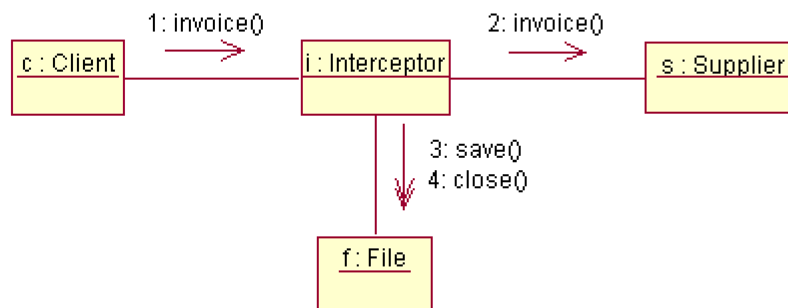

Fig. 3. Dynamic structure of the aspect semantic.

partially modeled with the *Logging* interface and the association between *File* and *Supplier* classes. The dynamic of the system is represented by an object of the *Intercept* class, that interchanges messages with objects of the other classes. Second, the diagrams do not show differences in class modularization and aspect modularization. The specific concepts of AOP, such as *Connection Points*, or *Notices* are not explicitly modeled. Third, the model does not reveal the fact that the interception to the method invocation is done transparently. Finally, the model does not show the property "plug-in plug-out" of the *Log* aspect.

In the following section we evaluate different UML extensions that aim to overcome this situation.

3 UML Extensions for Aspects Managing

There are two alternatives to extend UML in order to manage aspects. One of them is the *General Extension to UML* (GEU). It modifies the meta model

of UML to include at this level the concepts related to the paradigm. Thus, UML will be able to model all type of aspects in any situation and in any context. Hence, aspects will be first class objects because they will be predefined mechanisms of the language, such as classes. Consequently, it will be possible to construct correct and simple models with enough level of abstraction. The disadvantage of this approach is that in order to be so general, perhaps some concepts related to the aspects may be lost during modeling. To solve this problem, a different approach was proposed. It consists on a *Specific Extension to UML* (SEU) that prioritizes the understandability of the problem and aims to reduce complexity. In this approach, design is not done for any aspect, just only for the aspects in the current system. Therefore, a model that reflects more closely the problem is obtained, and this model presents high level of abstraction. The disadvantage is that when changing to another problem, it will not be possible to reuse anything of the previous products, and we must start from the beginning again. In being so specific, if we change the environment, the extensions constructed to one system will not serve to the other, making difficult to reuse.

This discussion is the same that raises when developing an aspect oriented language [2]. Should the development consider aspects in general, or only those specific to the problem? These alternatives are well known in literature as General Purpose Aspect Oriented Languages, and Specific Aspect Oriented Languages. The decision of which one should be selected during the design phase is the same: both alternatives are valid, but depending on the application domain, one of them will be the most convenient.

In the next subsections we present both alternatives.

3.1 General Extension to UML

Suzuki and Yamamoto [4] propose a general approach to express aspects during design. They suggest an extension to UML to support aspects appropriately without violation to the metamodel specification. For this purpose, they add to the meta model new elements for an aspect and the weaver [1, 2] , and an existing element is reused to model the relationship class-aspect.

Aspects Ideally, the aspect should be a constructor derived from the Classifier element, describing its structure and behavioral properties. The Class, Node, Interface, and Component elements are Classifier types. An aspect can have attributes, operations and relationships. The aspect operations are used to express its behavior. The aspect relationships include generalization, dependence, and associations. Figure 4 shows the proposed extension for the aspect constructor.

Class-Aspect Relationship The UML Metamodel defines three primary relationships derived from the metamodel element Relation. These are the associations, the generalizations and the dependencies. Considering that the dependency relationship stipulates that the implementation of one element requires

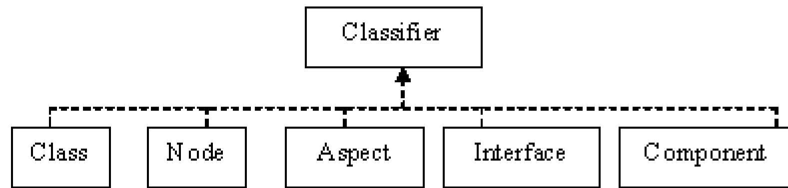

Fig. 4. An aspect as a metamodel element.

the presence of other element or elements, we consider this type of relation the most adequately to model the class-aspect relationship.

The elements derived from the dependency element are: *abstraction*, *bound*, *permission*, and *use*. The class-aspect relationship is a type of abstraction. An abstraction dependency relates two elements that corresponds to the same concept in two different levels of abstraction, or from two different points of view. The UML Metamodel defines four stereotypes for the abstraction dependency: *derivation*, *realization*, *refinement* and *trace*. The most appropriate is the << *realization* >> stereotype. A realization is a relationship between an element of the specification model, and the element of the model that implements it.

Weaver Using a weaver, the aspect code and the class code are combined, generating a *weaver* class. The structure of this class depends on the weaver and on the base language used. To express it in the metamodel, an stereotype << *Weaver Class* >> is introduced to the Class element. It is advisable to use a labeled value to specify the aspect, and the class or classes associated with it.

3.2 Specific Extension to UML

In [3], UML is extended to support the synchronization aspect. Each object is designed in two levels: a *functional level* where the basic functionality of the object is defined, and the *synchronization level*, where the synchronization properties are described. The synchronization policies are not explicitly modeled without UML extensions, consequently they result simply in code written in the body of the methods.

The mechanisms that UML provides for possible extensions are used to extend UML in this direction[7]. These are stereotypes, constraints, and labeled values. The stereotypes provide a mean for elements classification, adding new attributes, relationships, and operations. The constraints enable the specification of new semantic for the model elements. The labeled values permit to associate information to any model element.

A new stereotype << *Synchronization* >> is defined to represent the synchronization policies. All the synchronization semantic is represented by its attributes and methods.

Also, a new type of state diagram is described to model the dynamic structure, such as synchronization conditions, state changes, and each of the event constraints. This new diagram is known as *state diagram for synchronization.* The construction rules are:

- A synchronization state represents a set of methods that can be invoked at any time.
- A transition represents a relationship between two states. It indicates that an object in a state can change to other state when an event occurs. A transition between two synchronization states will happen only when a method is invoked.
- The transition condition evaluates the possibility to execute the transition. Thus, it evaluates the possibility to execute the method.

Finally, the synchronization actions are defined. They are classified in two categories: pre-actions, and post-actions. The pre-actions are the set of actions that must be executed before the method invocation, and the post-actions are the set of actions that must be executed after the method invocation. This information is added to the transition. Figure 5 shows the transition scheme.

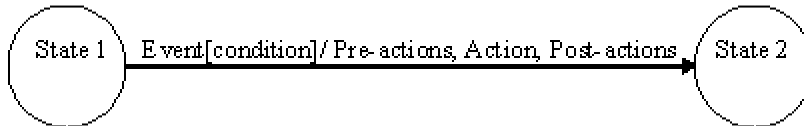

Fig. 5. Transition scheme for an UML extension.

As an example, we consider a chat server. The server provides several methods, two of them are the connection and the disconnection methods. The synchronization policy stipulates that it will be possible to establish a connection if the server is not overloaded. This means that the number of connections already established are less than the maximum bound.

First, we define the *Server_Sync* class with the << *Synchronization* >> stereotype. It implements the synchronization policies for the server methods invocation. In this case, it only involves an attribute *counter*, for the registration of the number of connections, and two methods *increase* and *decrease*, to manipulate the counter. In Figure 6 we show the static structure of the system.

As it was mentioned, a state diagram for synchronization is required to show the dynamic behavior. This diagram is related with the *Server_Sync* class and defines the synchronization policies required. Figure 7 shows such diagram.

We have presented in this section two approaches based on extensions to UML for aspect oriented design. In the next section we define evaluation criteria to compare them.

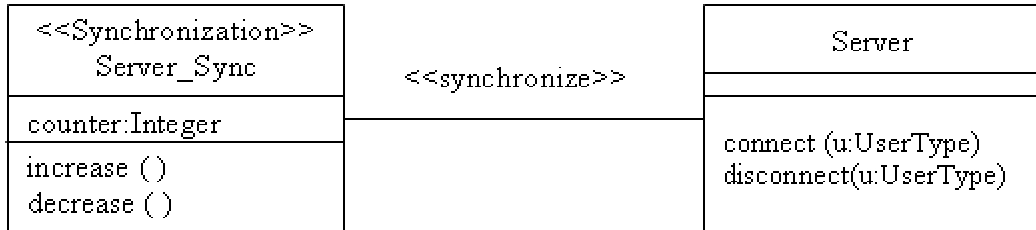

Fig. 6. UML Extension: Synchronization stereotype.

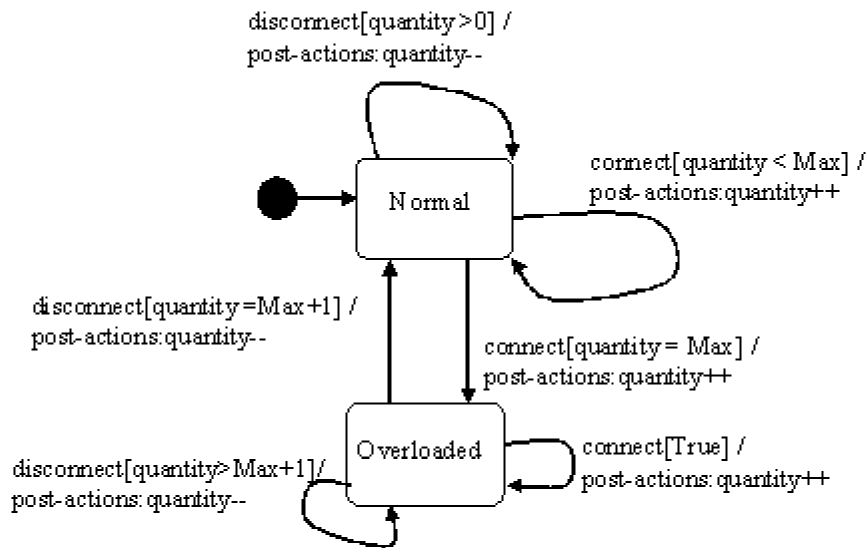

Fig. 7. State Diagram for synchronization of Server class.

4 Evaluation of Both Approaches

To compare both alternatives we follow the criteria defined in [9]. These criteria are used to evaluate how each approach satisfies the considerations mentioned in Section 2. The criteria are:

– *Traceability*: how easy is to follow the requirements in the design models. This criterion is defined to ensure that the design approach has provided the adequate constructors.
– *Propagation of Change*: how the model facilitates the incorporation of future changes in the design.
– *Reuse*: how it promotes design reuse for other systems.

- *Understandability*: how the design facilitates the understanding of the system functionality.
- *User Friendliness*: how the design techniques are easy to learn and use.
- *Separate Development*: how the design techniques enables parallel development of different modules of the system. This criterion promotes a better productivity of the development team.

In the next subsections we evaluate the two approaches presented, based on these criteria.

4.1 Traceability

The SEU approach provides a good traceability as a result of the separation of the static and dynamic behavior. Within the static models, stereotypes and relationships are used, objects, aspects, and also how they are related are clearly specified. Within the dynamic models, new state diagrams are defined, where the actions related to aspects are captured by pre and post-actions related to a transition. These actions enable the modeling of those requirements that crosscut the design.

In the GEU approach, each paradigm concept has its associated constructor. This is a main characteristic because in this way the connection point notion is explicit. This one-to-one relationship between the paradigm concepts and the language constructors is very important because it produces a perfect traceability. This situation is not present in the SEU approach where several concepts, such as the connection points are modeled in an implicit way.

4.2 Propagation of Change

The SEU approach shows a weak point in the dynamic behavior modeling. As all the actions related to aspects are specified within the transitions, a change or modification implies a review to all the state diagrams to check the modification.

In the GEU approach, this problem is not present because of the one-to-one relationship between concepts and constructors. Although, the aspects have an excessive view and control of the internal structure of the objects related to them. Thus, the effects of a change can be rapidly propagated. The job of controlling and restricting this strong influence of the aspect over the object is a responsible concern of a good designer.

4.3 Reuse

In the SEU approach, while the aspect results correctly encapsulated, such encapsulation is mainly based on the current system, presenting all its constraints and characteristics. Mostly, the decisions are taken considering only the current context. Thus, moving the aspect to other system, with other properties, is a complex task.

In the GEU approach, as the aspects and their objects elements are first classes in the model, richer and more expressive models can be produced. Therefore, promoting the alternative to model aspects inheritance, abstract aspects, and other characteristics. As a result, a good reusability of aspect designs is obtained.

4.4 Understandability

In the SEU approach, the understandability is very good working with models with few aspects, since they are based on well known and familiar constructors. As more aspects appear in the model, it results more difficult to identify the relationships between aspects, and between aspects and their objects.

The GEU approach does not provide clear composition rules, and there also exists a strong influence of aspects over objects. These two characteristics reduce the design understandability.

4.5 User Friendliness

In the SEU approach, the extensions to manage aspects are realised with specific UML extension mechanisms. Thus, extended models preserve simplicity. With solid knowledge of UML and its extension mechanisms, it is simple to apply those mechanisms to aspect handling and learning. This approach is quite easy to use, and also is easy to understand the system behavior.

In the GEU approach, as new elements are added to provide a general solution, they require more attention about their behavior, and also the way in which they can be used and composed. Thus, it is necessary a previous learning stage, not trivial, before using them. Therefore, the learning and user friendliness is considerably reduced.

4.6 Separate Development

In the SEU approach, as simplicity is prioritized, the aspects interaction is essentially reduced, enabling a separate development of all the aspects involved in the system.

In the GEU approach, because of a greater interaction between aspects and objects, and also because of the aspect design references to the objects related, separate development is seriously affected.

In the next section we present our final conclusions and future work.

5 Conclusions and Future Work

In this work we have presented the need for expressing aspects in the first stages of the development process, particularly during design. It represents the first main step for the consolidation and maturity of AOP.

First, we have analyzed if UML, the most popular and accepted specification language in software community, is able to express aspects. The conclusion is that it is possible to represent aspect semantics, but the way to express them is not so satisfactory. Thus, the next step is to study different approaches to extend UML. A general and formal alternative is to include the paradigm concepts in the language metamodel. The other alternative, not so rigorous, but also effective, is to extend UML for the current problem, using the mechanisms that UML provides to extend its semantic.

In the first approach, aspects are treated during the design phase in a more formal and general fashion. With this extension, it is possible to express in UML any aspect that appears in the system at any moment. Also, when changing to another system, and having to manage different aspects, the same extensions will serve to model the new system. Although granularity is not very good, there are benefits with respect to documentation, learning, reusability, and formality.

In the second approach, a specific extension to UML was defined to manage the synchronization policy for the functional behavior of the objects of the system. To model the static structure, we create a new stereotype $<<$ *Synchronization* $>>$, and for the dynamic structure a new state transition diagram was added. Thanks to these extensions, the synchronization policies can be designed separately and independently, enabling a greater reuse. The abstraction level was correct, and the concepts related with synchronization resulted clearly encapsulated. As it was mentioned, this extension is usable only for the synchronization aspect. For other aspects, new stereotypes and new state diagrams should be considered.

We compared both extensions based on particular criteria. In table 5 we resume the result of each approach for each of the criteria, based on our analysis. We have used the following scale: Excellent, Very Good, Good, Sufficient, Regular, and Bad.

Criteria	**GEU**	**SEU**
Traceability	Excellent	Sufficient
Propagation of Change	Sufficient	Bad
Understandability	Sufficient	Regular
User Friendliness	Regular	Excellent
Reuse	Good	Bad
Separate Development	Good	Very Good

Table 1. GEU and SEU Evaluation

In a little system, with a controllable number of aspects, namely no more than five plus or minus two, the SEU approach is a direct way, simple and effective. Although, when the system complexity increases, and the number of aspects is important, the simplicity of SEU is a disadvantage for the developer. In this

situations it is revealed that this approach is an ad hoc solution, an approach to solve a particular situation. As it was also mentioned in [8], SEU is a compromise solution, a solution not conceptually robust.

We believe that the main issue behind aspect oriented design is to consolidate the GEU approach, in such a way that it can satisfy all the criteria mentioned. The future work is to study its weakness and to look forward to improve them, specially facilitating its use, making it more adaptable to changes, and more understandable.

As far as we know, the research made about including aspects during the design phase, as those presented in this paper, try to solve a particular problem, in an isolated way. The different approaches are quite different, they do not respect standards, they include different stereotypes, and there is no consensus. The main contribution of this work is an attempt to unify different approaches, and to guide the developer in deciding the aspect oriented design strategy.

Our future work is to study automated tools able to generate code from models containing aspects. Thus, tools that take as input an UML model, then ask all the information required for the elements, and finally generate an application skeleton. Those tools should consider the aspect language, but also the language where the basic functionality will be expressed.

References

1. *Aspect-Oriented Programming.* In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June,1997. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin.
2. *Programacin Orientada a Aspectos: Análisis del Paradigma.* Final Thesis for the Bachellor Degree in Computer Science. Universidad Nacional del Sur. November, 2002. Fernando Asteasuain, Bernardo Ezequiel Contreras.
3. *Introducing Separation of Aspects at Design Time.* In Proceedings of AOP Workshop at ECOOP '00, Cannes, France. June, 2000. José Luis Herrero, Fernando Sánchez, Fabiola Lucio, Miguel Toro.
4. *Extending UML with Aspects: Aspect Support in the Design Phase.* 3er Aspect-Oriented Programming(AOP) Workshop at ECOOP 99. Junichi Suzuki, Yoshikazu Yamamoto.
5. *From AOP to UML- A Bottom-Up Approach.* Software Engineering Laboratory, Switzerland. June, 2001. Mohamed M. Kandé. Jorg Kienzle, Alfred Strohmeir.
6. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* Ph.D. Thesis, Technische Universitat Ilmenau, Germany, 1998. K. Czarnecki.
7. *UML: The Unified Modeling Language.* Addison Wesley, 2000. Grady Booch, James Rumbaugh, Ivar Jacobson, Rational Software Corporation..
8. *A Metamodel for Aspect-Oriented Modeling.* Workshop on Aspect-oriented Modeling with the UML. First Conference of Aspect-oriented Software Development (AOSD 2002), Holland. April, 2002. Christina Chavez, Carlos Lucena.
9. *An Analysis of Design Approaches for Crosscutting Concerns* . Workshop on Identifying, Separating and Verifying Concerns in the Design (AOSD 2002), Holland. April, 2002. R. Chitchyan, I. Sommerville, A. Rashid.

10. *Programacin Orientada a Aspectos: Metodología y Evaluación.* IX Congreso Argentino de Ciencias de la Computación CACIC, Octubre 2003 Fernando Asteasuain, Bernardo Contreras, Elsa Estévez, Pablo R. Fillottrani.
11. *Getting Started with AspectJ* . Communications of the ACM (CACM) Vol. 44 Nro.10, Octubre 2001. Gregor Kickzales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G.Grisnold.

This article was processed using the LaTeX macro package with LLNCS style