Introduction to Aspect-Oriented Programming

LÁSZLÓ LENGYEL, TIHAMÉR LEVENDOVSZKY

{lengyel, tihamer}@aut.bme.hu

Reviewed

Key words: aspect-oriented programming (AOP), crosscutting concerns

Aspect-oriented programming is a fortunate extension to the wide-spread object-oriented paradigm. In this paper we present the most important concepts of AOP based on the most widely used AspectJ approach. The problem of crosscutting concerns is introduced, and the facilities provided by AOP are enumerated as possible solutions. The most popular implementations (HyperJ, Composition Filters) are also mentioned briefly.

1. Introduction

Nowadays, the object-oriented programming (OOP) is the dominant paradigm of software engineering. The solutions provided by OOP can be applied to facilitate creating well-structured program as well as the code reuse. That is the reason for its wide adoption and its relative dominance. The concept behind the OO approach is that the program under development consists of autonomous entities, so-called objects, whose functionality is realized by the communications of these objects. This method, according to the experience, supplies a well-structured solution even for complex systems [1,2].

If a complex problem is decomposed into objects, the creation of the autonomous entities is focused along with the encapsulation of the data and the related operations. In this way, however, we have to ignore more important logical aspects of structuring and grouping, such as persistence or debug, which characteristically scattered across the code. This makes the software difficult to comprehend and maintain. These tangled but logically connected code parts that are scattered across different module are called *crosscutting concerns*.

An example for crosscutting concerns can be tracing the execution of a program. Distributed applications frequently write a log file, which helps debugging in case of an application error with collecting all the function calls and exceptions.

In order to write a log file, each class must contain program lines implementing the log functionality, usually scattered, whereas the code parts that perform logging are closely connected: they realize the same function.

As another possible example [3] the UML class diagram of a simple figure editor is illustrated in *Figure 1.* The *FigureElement* has two concrete descendants: the *Point* and the *Line*.

The decomposition into classes seems promising: both classes have a well-defined interface, and the data is encapsulated with the operations performed on them. However, the screen manager must be notified about the movement of each element. That demands that each function performing movements should notify the screen manager. The rectangle *DisplayUpdating* frames the functions that should implement this feature. Similarly, the rectangles *Point* and *Line* frame the functions implementing concerns related to them. It is worth noting that the *DisplayUpdate* square fits into none of the other rectangles in the figure, but it cuts across them.

Extending the OO facilities, Aspect-Oriented Programming (AOP) [4] offers a solution to the problem of crosscutting concerns. AOP divides the program code on the basis of the concerns that they contribute to the operation of the program. Approaching the problem in an AOP way, we can group the concerns into *aspects* implemented separately and independently of each other, and then an *aspect weaver* application joins these separate parts. Weaving is dependent on the particular AOP implementation it can happen either dynamically at run-time or statically at compiling time or after that.

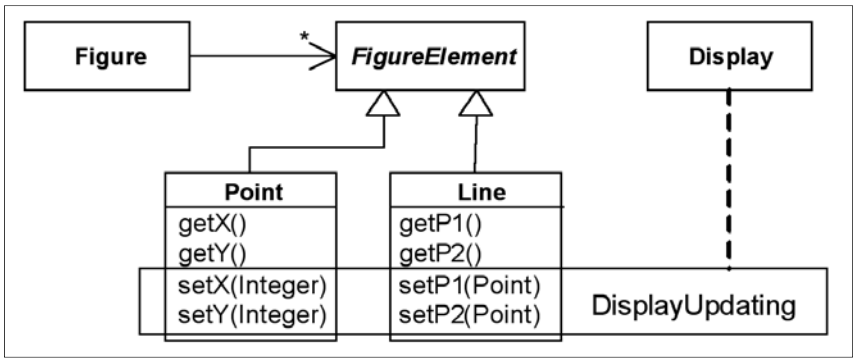


Figure 1. Crosscutting Concerns

If OOP is applied, the implementation of the crosscutting concerns are scattered across the system. However, if AOP mechanisms are used, the concern surrounded by the rectangle *DisplayUpdate* can be implemented using only one aspect. Besides these AOP facilitates thinking in aspects also on the design level along with realizing modularity.

The key point in modularization is that the program parts constituting one unit can be supplied in one physical unit as well. It is a general principle that the cohesion inside a module should be strong, and the modules should be loosely coupled. Using abstraction we can highlight the common traits of different elements. While the abstraction is rather vertical, separating the crosscutting concerns we can achieve structuring our system horizontally.

A programming paradigm or technology is mainly determined by the type of abstraction it uses. The frequent, repetitive code snippets or patterns mean the lack of abstraction facilities. The redundancy is unwanted, because a small change in the design may result that several, not connected module needs to be changed.

2. Crosscutting Concerns

Separating concerns means the ability that we need to identify and highlight those parts of the software which realize a concrete intention or goal. Separation of concerns primarily aims at decomposition of the software into parts that can be treated more easily and comprehensible. A natural question is how to accomplish this decomposition. What are the functions that should belong to a class or an aspect?

It is important to notice that the crosscutting is connected with a specific decomposition, since the crosscutting concerns cannot be separated completely. The basic design rule is to consider the fundamental concerns as a primary abstraction, and to implement them in classes and their extensions with aspects is done afterwards if needed.

Regarding the figure editor example, there are two important design concerns: representing the graphical elements and tracing the movement of them. The classes depicted in *Figure 1* represent the first concern. Each graphical class encompasses its inner data structure, which can be extended with aggregation and inheritance. The second concern, tracing the movement of the elements, should be implemented as separate classes, but the firs concern prevents this, because the functions realizing the movements are part of the graphical classes because of the encapsulation. The system could have been designed around the concern tracing the movements, but in that case the graphical functionality would have crosscut the tracing classes. Which solution is better?

The problem can be solved with the help of the dominant decomposition. The software – similarly to books – is written like text, as the book consists of chapters and paragraphs, the software has modules e.g. classes. The modules constituting the dominant decomposition contain uniform concerns and most of the time *they can be executed separately*. A dominant module cannot contain concern crosscutting several other modules. These are going to be crosscutting concerns.

Typical crosscutting concerns are synchronization, monitoring, buffering, transaction handling or context-sensitive error handling. Crosscutting concerns can be very high-level, e.g. security or the aspects of quality of service (QoS).

3. Aspect-Oriented Programming

The AOP paradigm was created in the mid-90s, and it has become an important area of research related to programming languages, so it is expected to gain more popularity.

It is apt to say that every programming language has a feature since Fortran, that facilitate to separate the concerns with subroutines. Subroutines are still useful constructs, and OOP cannot exist without block structures or structured programming. Similarly, AOP does not replace the technologies in use.

It happens quite often that the concerns cannot be realized by simple procedure calls, because a concern gets mixed with other structural elements, and becomes a fuzzy mess. The other disadvantage of subroutines that the programmers working on the caller component must be aware of the concerns, they must know how to include or use them. Besides the subroutines, AOP offers a call mechanism, where the developers of the caller components do not need to know about the extending concerns, namely, about calling the subroutine.

The two most important principle of AOP are the separation of crosscutting concerns and the modularity [5]. AOP has recognized that the boundary of modular units is rarely the same as the boundary of concerns. The modularization can be solved with certain concerns, but the code implementing the crosscutting concerns are scattered across the program, crosscutting the modular units. The main goal of AOP is to make it possible that the crosscutting concerns can be organized into autonomous modular units, thus it decrease the complexity of the software product (code and/or design), and the reuse, readability and maintenance of the program becomes simpler.

An aspect is a modular unit of implementation; it encapsulates a behavior, which affects several classes. Using AOP we implement the application in an arbitrary OO language (e.g. C++, Java or C#), then we deal with the crosscutting concerns, which means including the aspects. Applying the aspect weaver, finally, the executable application is built via combining the code and the aspects. As a result, the aspect becomes the part of the implementation of several functions, modules, or objects, thus increasing the reuse as well as the maintenance facilities.

The weaving process is depicted in *Figure 2.* (on the next page). It is worth noting that the original code does not need not to know anything about the extending aspect, if one translates it without weaving, the result is the original application. If the aspects are included, the result is the application extended with the functionality of the aspects. It means that the original code need not be changed; the same program is used in both cases.

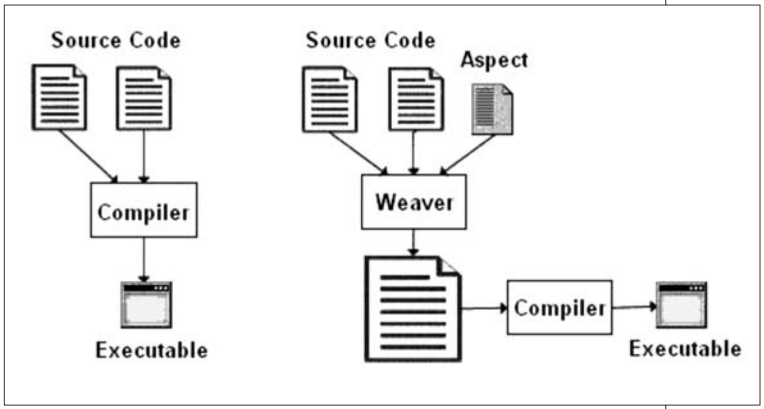


Figure 2. Aspect Weaver

AOP enhances, but does not replace OOP. It offers a different type of decomposition, in addition to classes, it introduces new element of modularization to realize each aspect separately from the classes in a different place. AOP is built on OOP, the objects and functions are not considered to be obsolete, aspects are meant to be used together with them.

The enhancement provided by AOP is that the entry point is declared by the function instead of the caller, which is not aware of calling the function. If the entry point of a function is given in the called part, it is woven into the code (obviously, it compiles to function call on the programming language level, but it is handled by the weaver automatically). Since the called code can be executed independently from its extension, as it has already been mentioned, this is a useful feature.

E.g. for memory paging, operating systems use a dirty bit to register the changes on the memory pages. If there is a change, the modified data must be saved to the storage. Using AOP techniques, handling the dirty bit can be separated from paging: the system can be executed with or without dirty bit handling, and the functions related to dirty bit handling are physically in the same place, instead of being scattered across the code according to the entry points.

Now a question arises, namely, how to provide where we want a piece of code to be called such that it does not appear in the caller part at all. The solution is the *join point.* The join points mean those well-defined places in the program, where the aspect interacts with the other parts of the system.

The join points mean the places of the program text or an execution point, so they can be divided into static or dynamic join points, respectively. Static join points are the first statement of a public function in case of the logging example, which results in a log file where the function call stack can be traced. Dynamic join points are connected with the events of the program execution like a method call (both inside of outside of the function), attribute query, throwing an exception, initializing a class or an object.

Aspect reuse is a fundamental result of AOP. The simple, small aspects facilitate the reuse of individual pieces of code more. Learning from the experiences and collecting aspects we can create aspect libraries. An apt question is how to deal with the large number of aspects and what notation should be used for them. This and similar questions are expected to be researched in the next few years.

A really important but open issue is the semantical correctness of the aspects. In case of component-based systems there has always been a question how to ensure the correct operation of the components. The AO approach offers far richer mechanisms than those provided by interfaces or message-based connections. Each aspect must thoroughly be examined from the point of specification and component test. If we use an aspect, there is no guarantee that we have the correct operation in every place where we reuse it afterwards. A way must be found to describe the operation of the aspects under specific circumstances.

Having presented the AOP, we briefly introduce the three most popular AOP implementations.

AspectJ

The environment AspectJ (www.aspectj.org) is a natural extension to Java: every Java program is an AspectJ program as well. The AspectJ introduces the following new programming language constructs for the AOP definitions [6,7]:

- *aspect:* A new programming unit, which encapsulates the crosscutting concerns. An aspect can contain the definitions of pointcuts, advices and introductions. Similarly to classes, they can have methods and data members.
- *pointcut:* It defines a set of dynamic join points with the help of a logical expression. These are called pointcut descriptors. Pointcuts can be parameterized, the objects of the pointcut environment can be passed to the advice in the parameters.
- *advice:* It is a programming unit similar to methods. Advices are always associated with a pointcut. Their body contains the behavior that should be executed in the join point described by the pointcut.
- *introduction:* Introductions help to define new data members and methods in classes. Here the join point is the class.

In order to decide the runtime order in which the elements of the aspects associated with the same join point are executed, precedence relations have been established within the aspects [8].

Classes and aspects are not on the same level in AspectJ [9,10].

Whereas classes can be regarded as autonomous entities, aspects can be interpreted along with the classes whose crosscutting cede they contain. The aspects can be considered that they contain the modifications of the original program code. Therefore aspects cannot be compiled alone, their reuse is possible on the source level only. The current AspectJ implementation works at compile-time only and the base program code is also necessary [11].

Hyperspaces – HyperJ

The hyperspace approach (www.research.ibm.com/hyperspace/index.htm) is based on the multidimensional separation of concerns. According to this principle there are several concerns of different types in the software, while the currently popular languages and methods facilitate the decomposition driven by only one concern. This phenomenon is called the tyranny of the dominant decomposition. The basis of the dominant decomposition is classes (OO languages), functions (functional languages) and rules (rule-based programming languages).

Hyperspaces facilitate identifying explicitly an arbitrary dimension of the concerns in the arbitrary phase of the development process. The hyperspace model uses the following definitions:

- *hyperspace:* It is for identifying the concerns. Hyperspace means the set of the software building blocks. E.g. classes in an OO environment, a method, or a data member. The hyperspace organizes these elements into a multidimensional matrix. In the imaginary coordinate system of the hyperspace the concerns are the dimensions, and the coordinates are the specific concerns within a dimension. The coordinates of the units in the hyperspace define concern that the given unit describes.
- *hyperslice:* It encapsulates the concerns. The units belonging to the same concerns are placed on the same hyperslice. Defining hyperslices we can encapsulate the units related to the concerns.
- *hypermodule:* It serves as a basis to integrate the concerns. A hypermodule encompasses a set of the hyperspaces to be integrated as well as the integrating relations describing the way of the integration and the relations between the hyperslices.

The tool HyperJ is a Java implementation for multidimensional separation of concerns [12]. The HyperJ performs the integration on the compiled hyperslice packages, not on the source code, thus one can remodularize the already existing applications for reuse. The join points are static, their definitions are contained by the specification of the hypermodule.

Composition Filters

Since in the OO languages the behavior is determined by the messages passed between the objects, a large scale of the behavioral modifications can be achieved by manipulating the incoming and outgoing messages (typically the function calls) of the objects.

In the model of composition filters (http://trese.cs.utwente.nl/composition_filters/) the manipulation and analysis of the messages are performed by filters. The model expresses the crosscutting concerns as the modular and orthogonal extensions of the objects. The modularity is ensured by the well-defined interfaces of the filters, and they inherently independent from the implementation of the objects [13,14]. The filters are orthogonal to each other, because in their specification there is no reference to other filters.

4. AOP and crosscutting constraints

Aspect-Oriented Software Development (AOSD) [4] is a new technology that has introduced the *separation of concerns* (SoC) in software development. The methods of AOSD facilitate the modularization of crosscutting concerns within a system. Aspects may appear in any stage of the software development lifecycle (e.g. requirements, specification, design, implementation etc.). Crosscutting concerns can range from high-level notions of security to low-level notions like caching and from functional requirements such as business rules to non-functional requirements like transactions. AOSD has started at the programming level of the software development life-cycle and the last decade several aspect-oriented programming languages have been introduced.

Aspect-oriented programming eliminates the crosscutting concerns in the programming language level, but the aspect-oriented techniques must be applicable on a higher abstraction level as well to solve this issue. In [15] an aspect oriented approach is introduced for software model containing constraints where the dominant decomposition is based upon the functional hierarchy of a physical system.

The modularization of crosscutting concerns is also useful in model transformation. Model transformation means converting an input model available at the beginning of the transformation process to an output model or to source code. Models can be considered special graphs; simply contain nodes and edges between them. This mathematical background makes possible to treat models as labeled graphs and to apply graph transformation algorithms to models using graph rewriting. Therefore a widely used approach to model transformation applies graph rewriting [16] as the underlying transformation technique, which is a powerful tool with a strong mathematical background.

The atoms of graph transformation are rewriting rules, each rewriting rule consists of a left hand side graph (LHS) and right hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of the LHS in the graph the rule being applied to (host graph), and replacing this subgraph with RHS. Replacing means removing elements that are in the LHS but not in the RHS, and gluing elements that are in the RHS but not in the LHS.

In general, graph rewriting rules parse graphs only by topological concerns, but they are not sophisticated enough to match a graph with a node which has a special property or there is a unique relation between the properties of the parsed nodes.

In case of diagrammatic languages, such as the *Unified Modeling Language* (UML), the exclusive topological parsing is found to be not enough. To define the transformation steps in a more refined way – beyond the topology of the graphs – additional constraints must be specified which ensures the correctness of the attributes among others.

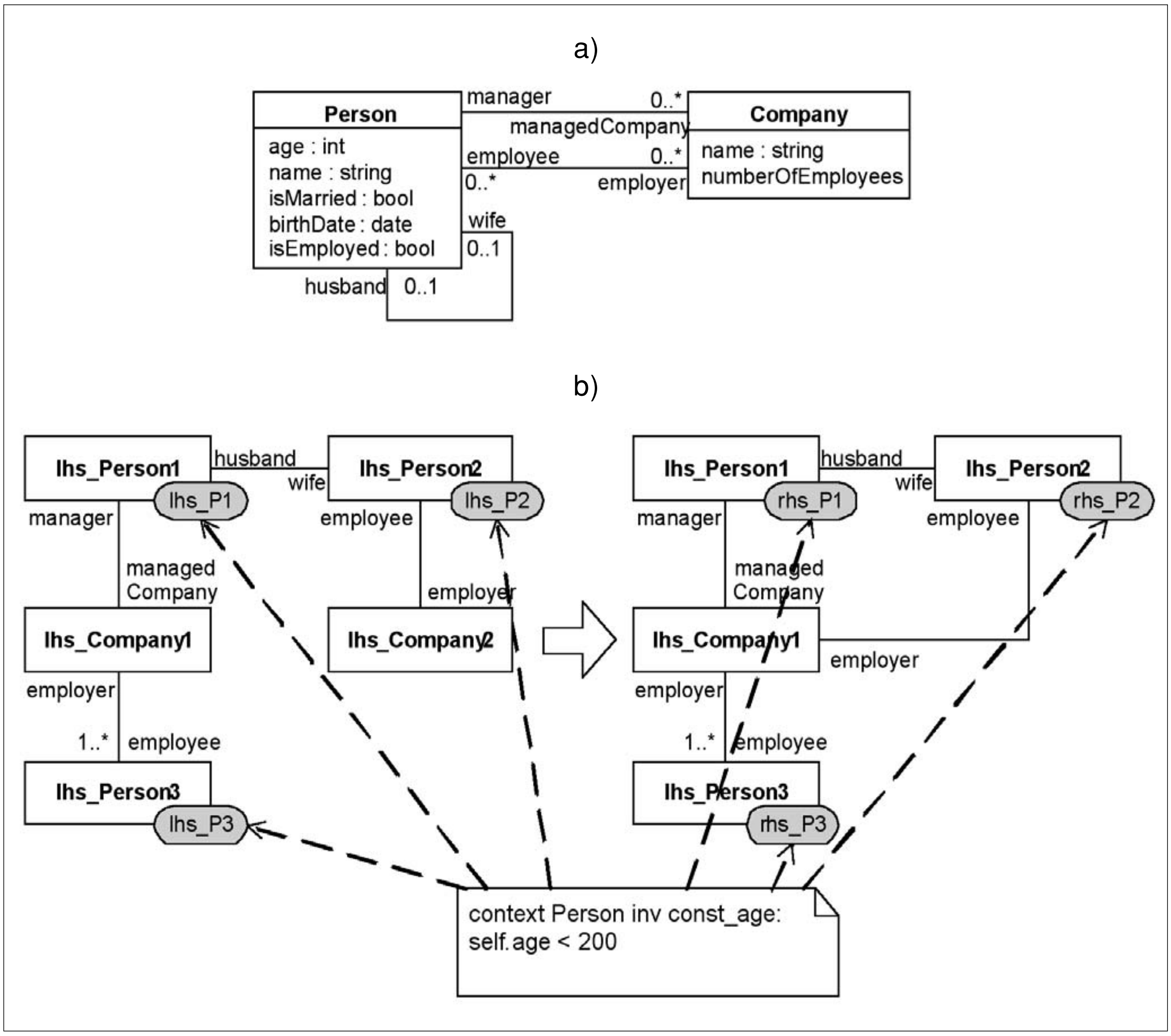
Dealing with constraints provides a solution for the unsolved issues, because topological and attribute transformation methods cannot perform and express the problems, which can be addressed by constraint validation.

The use of the constraints in graph transformation rules and in graph rewriting is found to be useful. Often, the same constraint is repetitiously applied in many different places in a transformation.

E.g. we have a transformation which modifies the properties of *Person* type objects and we would like the transformation to validate that the *age* of a *Person* is always under 200 *(Person.age < 200)*. It is certain that the transformation preserves this property if the constraint is defined for all rewriting rule element whose type is *Person (Figure 3/b)*.

It means that the constraint can appear several times, and therefore the constraint crosscuts the whole transformation, its modification and deletion is not con-

Figure 3. A sample metamodel and a transformation step with a crosscutting constraint
a) Metamodel b) Transformation step



sistent because such an operation has to be performed on all occurrence of the constraint. Besides this often it is difficult to reason about the effects of a complex constraint when it is spread out among the numerous nodes in rewriting rules.

It would be beneficial to describe a common constraint in a modular manner, and to designate the places where it is to be applied. We need a mechanism to separate this concern. Having separated the constraints from the pattern rule nodes, we need a weaver method which facilitates the propagation (linking) of constraints to transformation step elements.

It means that using separation and weaver method we can manage constraints using AO techniques: Constraints can be specified and stored independently of any graph rewriting rule or transformation step node and can be linked to the rewriting rule nodes by the weaver.

To summarize the main idea of the AO constraints, we can say that one can create the constraints and the rewriting rules separately, and with the help of a weaver constraints can be propagated optional time to the rewriting rule nodes contained by the transformation steps. Therefore constraints are similar to the aspects in AOP.

5. Conclusions and future work

AOP is a language-independent construct, a concept above the implementations. In fact, it can remedy the shortcomings of the programming languages (not only OO) with a simple and hierarchical decomposition.

The AOP concepts have been implemented in several programming languages: C, C++, Java, Perl, Python, Ruby, SmallTalk and C#. The research community targets Java the most, thus the most sophisticated tools and environments are available in this language.

In the field of software engineering the long-term issues of the software lifecycle play a more important role nowadays. These include the problems of simplifying the development, maintenance, being able to accommodate to changes or reuse. AOP helps to achieve these goals with ensuring a model more flexible and working on a higher abstraction level than the OO paradigm.

The number of publications is quite large. As a starting point we recommend the special issue of CACM devoted to the topic [17] and the web sites of each implementation.

References

[1] Czarnecki, K. and Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison Wesley, Boston, 2000.
[2] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, 1976.
[3] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, Harold Ossher: Discussing Aspects of AOP, CACM Vol. 44, Issue 10 (October 2001)
[4] Aspect-Oriented Software Development, http://www.aosd.net/
[5] Tzilla Elrad, Robert E. Filman and Ataf Bader: Aspect-oriented Programing, CACM Vol. 44, Issue 10 (October 2001)
[6] Gregor Kiczales, Erik Hilsdale, Jim Hungunin, Mik Kersten, Jeffrey Palm and William G. Griswold: An Overview of AspectJ. J. Lindskov Knudsen (Ed.): Proceedings of the 15th ECOOP, Budapest, 2001, pp.327–353.
[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold: Getting started with AspectJ, CACM Vol. 44, Issue 10 (October 2001)
[8] Kiczales, G., et al.: An overview of AspectJ. In Proceedings of the 15th European Conference on OOP (ECOOP). Springer, 2001.
[9] The AspectJ Programming Guide, www.aspectj.org
[10] Bill Griswold, Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten: Aspect-Oriented Programming with AspectJ, http://www.aspectj.org
[11] The AspectJ Primer, www.aspectj.org/doc/primer.
[12] Peri Tarr, Harold Ossher: Hyper/J User and Installation Manual, http://www.research.ibm.com/hyperspace http://www.math.klte.hu/~espakm/GOF/hires/Pictures/mvc.gif#
[13] Mehmet Aksit, Bedir Tekinerdogan: Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.
[14] Mehmet Aksit, Lodewijk Bergmans: Software evolution problems using composition filters. ECOOP 2001, Budapest.
[15] Jeff Gray, Ted Bapty, Sandeep Neema: Aspectifying Constraints in Model-Integrated Computing, OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Minneapolis, MN, October 2000
[16] Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol. 1., World Scientific, Singapore, 1997.
[17] Communications of the ACM, Vol. 44, Issue 10 (October 2001)