Volume 5, Issue 3, March 2015 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Comparative Study of Component – Oriented and Aspect – Oriented Programming

Parul Rajpal[*]
Research Scholar
Chandigarh University, India

Prof. Amanpreet Kaur
Associate Professor
Chandigarh University, India

Abstract: Programming is an activity of constructing computer programs that are a sequence of instructions describing how to perform certain tasks with computers. Programming can be classified or described in terms of the major techniques, concepts or facility used for constructing computer programs. For all modern electronic computers, however, certain programming languages have to be used to program the computers. If a specific technique, say X, is being used to program computers, we will say this is X-oriented programming. So, this paper compares two advanced programming methods i.e., component-oriented programming and aspect-oriented programming. The purpose of this comparative analysis is to give a succinct and clear review of these two programming methods.

Keywords: aspect, component, programming, object-oriented, interface-based programming

I. INTRODUCTION

Programming efficiency has generally been a problem in the development of scientific applications. [1] [2] Application developers must juggle a number of concerns, which include the correct implementation of algorithms, high performance requirements, and use of large data sets. The need to analyze the outputadds other programming aspects, such as the use of graphics routines. The ever changing architecture designs of computers further complicates the scenario. Also, target problems are transitioning from a narrow focus (i.e., the study of an isolated physical phenomenon) to multi-discipline applications where different element applications must interact to correctly capture the target problem [3] [4]. And, the need to share part or all of an application is increasing.

Clearly, the development of modular, well-organized applications that are portable is an important goal of most application developers. The use of library packages has traditionally been the means of supporting such a programming requirement. But, many applications still end up with the different types of programming aspects coded in an intertwined style which results in applications that are hard to decipher and to manage. The intertwining is done sometimes to improve the performance of applications, but also is done because traditional programming systems do not provide or encourage alternatives.

Humans are better able to understand and manage complex systems if they can be described as a set of individual concepts along with a set of coupling procedures. Object-oriented and component-based systems provide some support, but they still leave difficult implementation details to the programmer. A programming approach, referred to as aspect-oriented programming (AOP) is particularly focused on the programming of cross-cutting aspects of applications. [5] [6] An AOP-based system should encourage and make straightforward the coding of different programming aspects. The system should then provide support for weaving those aspects to generate a correct and efficient program. This can be done both during compile and execution phases.

This paper reviews and compared component-oriented programming and aspect-oriented programming. The comparison includes a description of the approaches, their characteristics, advantages and disadvantages. We hope this will help enable the generation of a conceptual mind-map and can be used to guide the activities of designing.

The remainder of this paper is organized as follows: Section 2 discuss the basics of Component-oriented programming. Section 3 discusses the basics of Aspect-oriented programming. Section 4 gives a comparative analytical discussion based on severalmetrics that are important to researchers and developers. Finally, Section 5 discusses the overall conclusion to the paper.

II. COMPONENT-ORIENTED PROGRAMMING (COP)

Component-oriented programming (COP) enables programs to be constructed from prebuilt software components, which are reusable, self-contained blocks of computer code. These components have to follow certain predefined standards including interface, connections, versioning, and deployment.

Components come in all shapes and sizes, ranging from small application components that can be traded online through component brokerages to the so-called large grain components that contain extensive functionality and include a company’s business logic. In principle, every component is reusable independent of context, that is to say, it should be ready to use whatever from wherever.[7]

COP is to develop software by assembling components. COP emphasizes interfaces and composition. In this sense, we could say that COP is an interface-based programming. Clients in COP do not need any knowledge of how a component implements its interfaces. As long as interfaces remain unchanged, clients are not affected by changes in interface implementations.[7]

A. Component definition

The word "component" has been around in computer industry for a long time. As a matter of fact, the concept of component itself had been living with us even before computers were invented. So, a component is a reusable, self-contained piece of software with well specified interface that is independent of any application. The very important point which has to be kept in mind, while developing a component is the reusability aspect, regardless of whether or not an organization can identify what the future requirements of the component will be.

A component may be thought of as an independent module that provides information of what it does and how to use it through a public interface, while hiding its inner workings. The interface identifies the component, its behaviors and interaction mechanisms.

B. Component architecture and interactions

Components are defined by their "requires" (inputs) and "provides" (outputs) interfaces that are related by the function. These components are capable of providing a service to other client components, while they themselves could be clients as illustrated in Fig 1. Each Interface specifies a one-way flow of dependencies from one module (which provides a service) to a client. Each flow implements usable services to the clients.

However, every related pair of module and client are codependent, which is, a client depends on the related module to receive a service in a particular method, and the module is dependent on the client to access and utilize the said services[8].

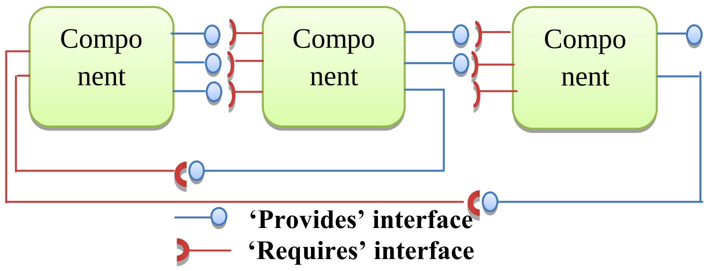

Fig. 1 Component interfaces

1. This type of interface specifies the services that the component provides.
    a. This is the API of the component.
    b. Specifies the method that user can call.
    c. Denoted with a circle at the end of the line from the component.

2. This interface shows the services that should be provided in the system by other components.
    a. The component will not work if the service is unavailable.
    b. It does not compromise that the component will be independent or deployable.
    c. Denoted with a semi-circle at the end of a line to the component.

A component can also interact with other software entities, either component-based or traditional ones, via its interfaces. This interaction is possible with the whole system mounted on a component infrastructure as shown in Fig. 2.

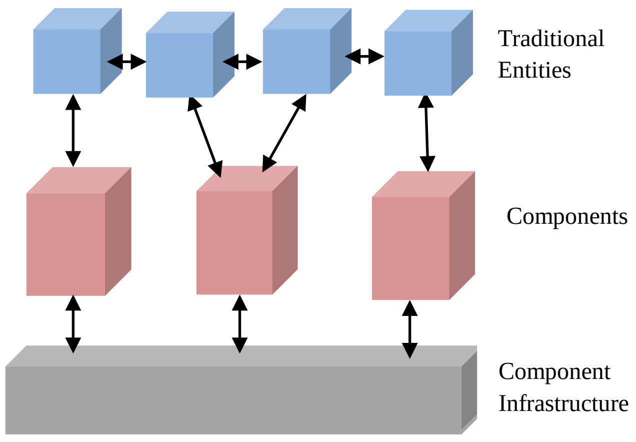

Fig. 2 Component Interactions

C. Component characteristics

The characteristics of a software component are listed below in a tabular form in Tab. 1:

Table 1 Component characteristics [9]

Component characteristics	**Description**
Standardized	Component conforms to some standardized model which may define interfaces, metadata, documentation, composition and deployment.
Independent	Components can be deployed without using other particular components. When a computer needs a service, it uses the "requires" interface explicitly.
Composable	All external interactions must take place through interfaces. Also a component must provide external access to information about itself, such as its methods and attributes.
Deployable	Component has to be able to operate as a standalone entity on the model platform. The component is binary and does not need to be compiled before being deployed.
Documented	Components have to be fully documented for the users to decide if they are satisfactory to their needs. All component interfaces should be syntactically and semantically explicit.
Testable	The interface should conform to specific standards. In this way the test approaches can also be standardized.

D. COP Advantages

The advantages of using components are rather more fascinating than using objects. Recognized by domain experts, some of these advantages are [8, 10]:

a. Better markets
b. Confirmable license fees
c. Encapsulation shielding or creating transparency
d. Faster software delivery
e. Software/hardware independence
f. High functionality
g. Immediate availability and early payback
h. In-house software development
i. Lower development and maintenance costs
j. Lower risks of development
k. Performance predictability improvement
l. Reduced time-to-market
m. Reusability
n. Substitutability
o. Tracking technology trends
p. Upgrades regularly anticipate organization's needs.

E. COP Disadvantages

The disadvantages of COP approach that users should take into consideration are:

a. Constraints on functionality and efficiency
b. Dependence on vendor
c. Difficult to synchronize multiple-vendor upgrades
d. Difficult scalability of the system
e. High functionality compromises performance and usability
f. Integration not always negligible due to some incompatibilities among vendors
g. Less control over the growth and evolution of the system
h. Less control over maintenance and upgrades
i. Licensing delays
j. Needed compromises in requirements
k. Possible recurring maintenance fees
l. Reliability of the system is often insufficient
m. Up-front licensing fees.

III. ASPECT-ORIENTED PROGRAMMING(AOP)

It has been more than a decade since the aspect-oriented programming(AOP) paradigm was introduced by Kiczales et al. [11]. AOP was presented as an alternative approach to Object-Oriented Programming (OOP) for better modularization and separation of concerns (SoC), especially for those concerns that cut across a system's functionality and hence, can result in redundant, scattered and tangled code. This relatively new paradigm has attracted a lot of interest from researchers and practitioners recently. A wide variety of AOP languages and tools have been developed. It has been argued that AOP and its related techniques can have a positive impact on the overall software development process and improve software quality [12].

The core idea of AOP, separation of concerns, has been around for many years with different names. Earlier approaches such as adaptive programming [13], subject-oriented programming [14] and composition filters [15] shared the same idea. However, the model of AOP (as implemented in the AspectJ programming language) proposed by Kiczales et al. [16] proved to be a simpler extension to the popular OOP language Java. It is now well supported by the Eclipse project and many different plugins have been developed. The remainder of this section briefly discusses different features of AOP in languages such as AspectJ.

In AOP languages, crosscutting concerns are encapsulated in an aspect, a class-like construct. This encapsulation is sometimes colloquially termed as aspectization. A single aspect can contribute to the behavior of a number of methods or objects through implicit invocation of additional behavior, which is composed at specific points of interest in the execution of a program. These points of interest are called join points. A point cut is a language construct which defines a join point in the code. The additional behavior can execute before, after or around join point, and is defined in an advice.

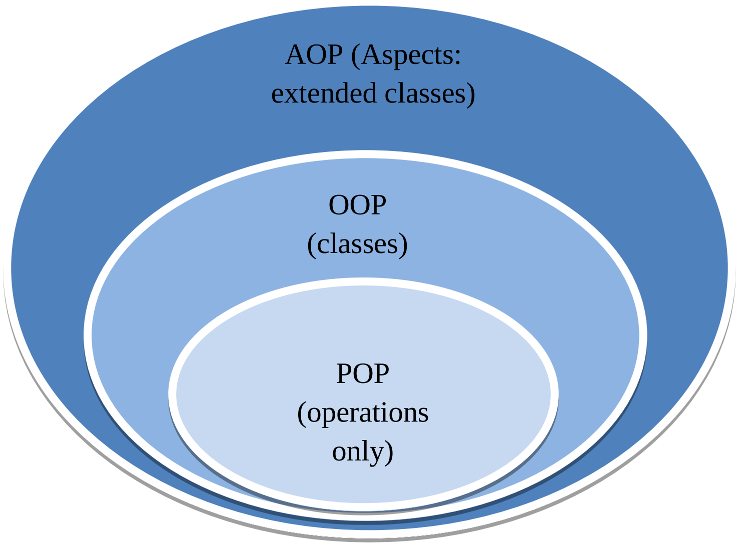

Fig. 3 The relationship between POP, OOP and AOP

Programs written with aspects can be composed and compiled with the base code. Aspect-oriented code is either transformed into the base code language where it becomes indistinguishable for the interpreter, or modifies the interpreter/environment to understand aspect-oriented code. Since it is difficult to change a programming's runtime environment, a special program transformation process called weaving is used. The weaving converts aspect-oriented code into object-oriented code with the aspects integrated into the code.Fig 3 shows the central concepts in each of the three programming approaches and how they are related to each other.

A. Aspect definition

Nowadays, there are many large scale, complex, distributed systems, all of them dealing with many concerns, such as security, auditing, logging, consistency, synchronization, error handling, timing constraints, user interface and etc. These concerns that affect each other are called crosscutting concerns and due to their interrelationships cannot be well modularized using Object Oriented languages. The original idea of AOP is to modularize these crosscutting concerns. Two main problems caused by crosscutting concerns are:

- Code scattering (one concern in many modules): This is when a concern exists in many places and leads to repeated code in the program.
- Code tangling (one module with many concerns): This is when a unit of decomposition carries out multiple tasks making it difficult to see what it is exactly doing.

The basis for AOP technique was first presented by XEROX Corp [17] who found several programming and software developmental process problems where both procedural and object-oriented programming techniques were weak to sufficiently and clearly capture many of the major design decisions that the program must implement. This was the reason for those design decisions to be implemented in a scattered way throughout the code, resulting in tangled code, which was excessively tedious to develop and maintain. Hence, AOP assisted by including the appropriate isolation followed by composition and reuse of the specified aspect code in order to express the programs involving such aspects with clarity.

"AOP is an approach to software development that combines generative and component–based development" [9]. Main concerns in a program are identified and implemented as aspects. They are then weaved into the appropriate places in the program by a weaver which is the compiler (or interpreter) of an Aspect language. Fig. 4illustrates the conversion of an Object Oriented program to Aspect Oriented and how a concern is modularized.

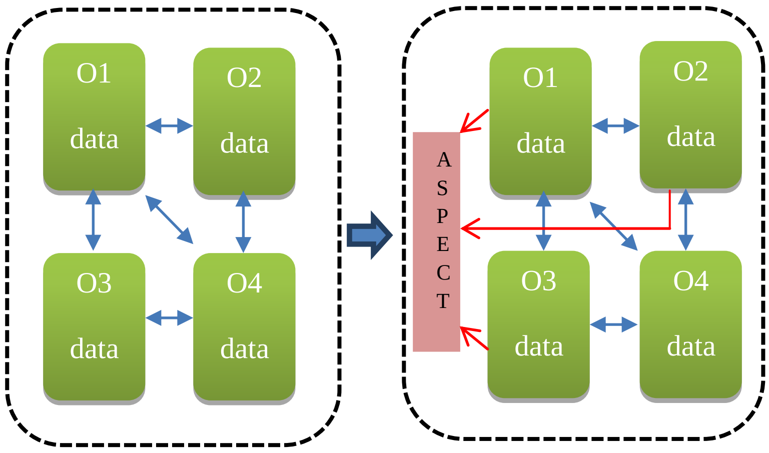

Fig. 4 Modularizing system concerns

B. AOP characteristics

As a young methodology for software development, AOP has the following mentioned characteristics. The ones that are considered as advantages or disadvantages to this approach are listed in the following two sections.

a. Aspects have a pre-definition of where to be included in a software system.
b. AOP allows the designers/implementers/users to understand each element of the system by knowing only its concern and without the need to understand other elements. Therefore, any required changes are localized to specific elements [9].
c. Quantification allows the programmer to code the desired unitary statements separately, having effect on many non-local parts of the system In other words, Aspects can crosscut any number of components, simultaneously.
d. Obliviousness is that the abovementioned quantifications are applicable to any place in the system, not necessarily prepared for them as enhancements. Therefore, there is no need for the components to be aware of those aspects crosscutting them and also there is no need for extra effort from programmers to implement any functionality [18].
e. Aspect-Based dichotomy is the distinction between components and aspects [18]. This dichotomy, along with separation of concerns and modularity, results in:

i. Decomposition of the system into components and Aspects.
ii. Modularization of crosscutting concerns by Aspects.
iii. Modularization of non-crosscutting concerns by components.
iv. Explicit representation of Aspects, apart from other aspects and components.

C. AOP advantages

According to Kiczales [5] the OOP and POP have many programming problems that did not allow these approaches toclearly capture some design elements which are important for software implementation. Therefore, AOP presented itself as a promising approach and as a solution for conventional programming approaches problems. However, solutions provided by AOP do not necessarily come in terms of lower compilation time or less memory usage. Rather, according to Laddad [12], using AOP for implementing software systems will certainly enhance software quality in many ways including:

a. Clear responsibilities for individual modules: AOP offers better modularization, by gathering the code that deals with the same aspect in one module avoiding the redundancy of crosscutting concerns. This also leads to a better programming development process because each developer could use his/her expertise with the module he/she knows better.
b. Consistent implementation: Unlike traditional implementations of crosscutting concerns, which are conspicuous in their inconsistency, AOP provides consistent implementation by having each aspect handled once.
c. Improved reusability: AOP isolates core concerns from the crosscutting ones, enabling more mixing and matching, and therefore improving the overall reusability in both modules. In contrast, traditional methods do not have this kind of separation between concerns.
d. Improved skill transfer: The concepts of AOP are reusable and transferable. Therefore, developers training time and cost will be minimized even if they need to learn more than one language. This is because core concerns and design patterns are universal. However, this is not the situation in other frameworks, where developers have to learn from the beginning each time, wasting considerable time and money on training.
e. System-wide policy enforcement: AOP allows programmers to enforce a variety of contracts and provide guidance in following “best” practices by creating reusable aspects.
f. Logging-fortified quality assurance: The disability of replicating a bug is one of the major disappointments for traditional methods’ developers, because it is such a ponderous process and thus barely used. On the other hand, AOP enables quality-assurance persons to attach the bug paper with its log, easing the reproduction of the behavior by the developer.

g. Better simulation of the real world through virtual mock objects: Software quality testing is enhanced in AOP application by using mock objects. Some scenarios often are not tested because of their complexity that requires an effort to simulate faults such as a network failure. AOP makes the difficult and cumbersome testing process easier without the need to compromise the core design for testability.
h. Nonintrusive what-if analysis: Dissimilar to non-AOP approaches, AOP does not waste time and space by checking whether functionality is needed by running what-if analysis every time before changing the system behavior.

D. AOP disadvantages

According to Laddad [12] there are two common oppositions to AOP, the first being that it makes the debugging process much harder. The second opposition is the fact that crosscutting modules implementation requires understanding the core module implementation details and vice versa. This is not the case in the OOP approach, though, where understanding is only required of the exposed abstraction between two classes. Moreover, Luca and Depsi [19] have discussed the challenges that AOP faces as a new programming approach in the following points:

a. Lack of expertise: The community members of AOP are approximately only 2000 programmers worldwide, and only 10-15% of them are experienced enough to use AOP in an OOP environment.
b. Concerns: Although AOP came to provide and to deliver a better separation of concerns (SoC), in reality, when a system reaches a certain degree of complexity, such separation is very hard to achieve, if not impossible.
c. Standardization: AOP introduced new dimensions and standards to programming. This, in general, creates complexity and possible resistance, but it was also the case when the OOP was introduced after the POP, which indicates that this is a normal scenario.
d. Sequential top-down code reading is impossible. This makes Aspect-Oriented programs difficult for humans to understand [9].

IV. COMPARISONS

Now we will compare the two development approaches from the human elements involved in the lifecycle of each, in terms of the following:

a. Requirements capture
b. System design specification
c. Programming, testing & debugging
d. System (program) integration
e. Documentation
f. Program maintenance

Tab. 3 merely summarizes the result of the metric comparison between the four approaches. As can be seen from this table, there is a huge number of metric attributes or parameters involved in software development life cycles. We use the following terminology and grading scheme in compiling the table:

a. The grades ‘Very Easy’, ‘Easy’, ‘Medium’, ‘Hard’ and ‘Very Hard’ refer to metrics related to required human effort, both physically and mentally. Not in all cases all of these grades appear but their ranking remains the same as stated.
b. The grades ‘Very Low’, ‘Low’, ‘Medium’, ‘High’and ‘Very High’ refer to metrics that indicate the quality, quantity or possibility.
c. The grades ‘Yes’ and ‘No’ refer to metrics that indicate a possibility, occurrence or existence.
d. The grade ‘N/A’ indicates ‘Not Applicable’.

Tab. 2 is the key to the terms used in Tab. 3, giving the value of the weights in a percentile format.

Table 2 Weights used for keys in tab.3

Benefits (Terms used)	**Percentile points**	**Risks (Terms used)**
Very Easy, Very High	100	Very Easy, Very Low
Easy, High	80	Easy, Low
Medium	60	Medium
Hard, Low	40	Hard, High
Very Hard, Very Low	20	Very Hard, Very High
Yes	10	Yes
No	5	No
N/A	0	N/A

Table 3 Comparison of software development methods

Approaches	**COP**	**AOP**
Metrics benefits		
Requirements inspection & capture	Hard	Hard
Project management	Hard	Medium
Functionality	High	Medium
Design quality	High	High
In-house development	Yes	No
Development speed	High	Medium
Schedule-ability	High	High
Documentation	Yes	Yes
Testing (Basic)	Easy	Easy
Testing (Integrated)	Easy	Hard
Assessability	Easy	Hard
Maintainability	High	Medium
Learning	Medium	Hard
Self-contained	Yes	Yes
Structurability (Control flow)	High	High
Structurability (Data)	High	High
Encapsulation	Yes	Yes
Modularity	High	High
Reusability	Very High	Medium
Substitutability	High	High
Predictability	Yes	Yes
Reliability	Medium	Medium
Efficiency	Medium	High
Integrity	High	Medium
Commonality	High	High
Standard ability	Yes	Yes
Interoperability	High	High
Portability	High	High
Scalability	Low	Medium
Self-descriptiveness	High	High
Expressiveness	Medium	High
Language independency	No	No
Augment-ability	High	High
Flexibility	High	High
Composability	Yes	Yes
Satisfaction	High	High
Software independency	Yes	Yes
Hardware independency	Yes	Yes
Profitability	High	High
Risks		
Development costs	Low	Low
Maintenance costs	Low	Medium
Complexity	Low	High
Skill sets	High	Very High
Risks	Low	Low
Up-front licensing fees	Medium	High

V. CONCLUSION

The execution environments for scientific applications have evolved significantly over the years. Vector and parallel architectures have provided significantly faster computations. Cluster computers have reduced the cost of high-performance architectures. However, the software development environments have not kept pace. Object-oriented and component-based languages have not been widely adopted. Distributed computing on local area networks and Grids is only being used by a most number of applications.

Clearly, there is a need for development environments that support the efficient creation of applications that use modern execution systems. This has been the goal of a continuing research effort over the last several years. The previous focus has been on using component-based ideas to develop a programming model and associated framework to support such a development approach. Aspect-oriented concepts are applied to support the reduction of intertwined code related to different programming concerns; mixing I/O with a numerical computation is one example.

We can conclude by stating that "one size does not fit all". Obviously, each of the methods discussed has its adherents as a function of the differing attributes of each approach. With the ever-growing need for faster development of applications and other systems software, there is the need to reduce the required programming skills by easing the programming paradigms.

REFERENCES

[1] F. Brooks, “No silver bullet: Essence and accidents of software engineering,”*Computer,* p. 12, April 1994.

[2] T. Sterling, P. Messina, and J. Pool, “Findings of the second Pasadena workshop on system software and tools for high performance computing environments,” Center of Excellence in Space Data and Information Sciences, Goddard Space Flight Center, Greenbelt, MD, Tech. Rep. Report TR-95-162, 1995.

[3] A. Salas and J. Townsend, “Framework requirements for mdo application development,” in *7th AIAA/USAF/NASA/ISSM Symposium onMultidisciplinary Analysis and Optimization, St. Louis, MO*, no. AIAA Paper 98-4740, September 1998.

[4] R. Weston, J. Townsend, T. Eidson, and R. Gates, “A distributed computing environment for multidisciplinary design,” in *5thAIAA/NASA/USAF/ISSMOSymposium on Multiple Disciplinary Analysis and Optimization, Panama City, Fl*, September 1994.

[5] G. Kiczales and e. a. J. Lamping, “Aspect-oriented programming,” in *Proceedings of the European Conference on Object-Oriented Programming (OOPSLA), Finland*. Springer-Verlag, June 1997.

[6] R. Filman and D. Friedman, “Aspect-oriented programming is quantification and obliviousness,” in *Workshop on Advanced Separation ofConcerns, OOPSLA 2000, Minneapolis*, October 2000.

[7] Andy Ju An Wang and Kai Qian, “ Component-Oriented Programming,” 2005.

[8] Bachmann, F. et al. Volume II: Technical Concepts ofComponent-Based Software Engineering, SoftwareEngineering Institute, Carnegie Mellon UniversityCMU/SEI-2000-TR-008, 2000.

[9] Sommerville, I. Software Engineering, 8 ed, 2007.

[10] Boehm B. Abts, C. COTS Integration: Plug and Pray? IEEE Computer, vol. 32, January 1999 pp. 135-138.

[11] G. Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.V. Lopes, J.-M.Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP’97), Springer, 1997, pp. 220–242.

[12] R. Laddad, Aspect-oriented programming will improve quality, IEEE Software 20 (2003) 90–91.

[13] K. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, USA, 1995.

[14] W. Harrison, H. Ossher, Subject-oriented programming: a critique of pure objects, in: Proceedings of 8th International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’93), ACM, 1993, pp. 411–428.

[15] L.M.J. Bergmans, M. Aks!it, J. Bosch, Composition filters: extended expressiveness for OOPLs, in: Proceedings of OOPSLA Workshop on Object- Oriented Programming Languages: the Next Generation, 1992.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of Aspectj, in: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP’01), Springer, 2001, pp. 327–353.

[17] Kiczales, G. et al. Aspect-oriented programming, ed 1997, pp. 220-242. doi: 10.1007/BFb0053381

[18] Chavez C. v. F. G. d. Lucena, C. J. P.ATheory of Aspects for Aspect-Oriented Software Development 7th Brazilian Symposium on Software Engineering, 2003. doi:10.1.1.94.2670

[19] L. Luca, I. Despi, “Aspect Oriented Programming Challenges,” AnaleSeriaInformatica, 2005.vol. 2, no. 1, pp. 65-70.