Aspect Oriented Programming: Views and Facts

P. R. Sarode **R. N. Jugele**

Abstract- **The term "Aspect-Oriented Programming" (AOP) came into existence at the Xerox Palo Alto Research Center (PARC). AOP was based on an extensive body of prior work, but somehow the existing terminology wasn't appropriate for describing what AOP does. The new programming technology beginning to devise to change the world! This paper gives how AOP came into existence; AOP – the ideas, the technologies and the name – came. History is just marginally interesting if one doesn't make the effort to learn from it and apply that knowledge in things that are still to come. AOP didn't "change the world" but it had an impact in research communities and in programming at-large. There are valuable lessons to be learned from the emergence of AOP, and an analysis of those is the ultimate goal of this article.**

I. INTRODUCTION

Giving an historical perspective of a technology involves stating facts as much as it involves describing the technical context in which the technology emerged and understanding the dynamics of the group of people who created it. Study focuses on the time period when AOP and, AspectJ emerged. A lot had happened before and a lot has happened since then. AOP emerging in 1995 that followed many of the types of questions like "what is AOP?" Is it a programming language? Macros in disguise? A design methodology? A clever pre-processor? Meta-programming? How is this different from X. But frequently questions were "what are aspects?"

II. FORMULATION OF THE PROBLEM

The increasing complexity of today's software applications and innovative technology make it necessary for programs to incorporate and deal with an ever greater variety of special computing concerns such as concurrency, distribution, real-time constraints, location control, persistence and failure recovery. Underlying all of these special purpose concerns is the basic concern responsible for the fundamental computational algorithm and the basic functionality. Special purpose concerns exist to either fulfill special requirements of the application (real-time, persistence, distribution) or manage and optimize the basic computational algorithm (location control, concurrency).

Typical approaches to integrate an additional concern have been to extend a given programming language by providing a few new programming language constructs that address the concern. An example of such an extension is the Distributed Real-time Object Language DROL [42] an extension of C++ with the capability of describing distributed real-time systems.

Even though the concerns may be separated conceptually and incorporated correctly, commingling them in the code brings about a number of problems:

- Programming intertwined code is hard and complex since all concerns have to be dealt with at the same time and at the same level.
- The extended programming language provides no adequate abstraction of concerns at the implementation level.
- Intertwined code is hard to understand because of the above lack of abstraction.
- Commingled code is hard to maintain and modify because the concerns are strongly coupled.
- Specific to object-oriented systems, the intertwined code gives rise to inheritance anomalies [1][30][44] due to the strong coupling of the different concerns.

It becomes impossible to redefine a method implementation or the commingled special concern in a subclass without redefining both. Many researchers have recognized the above problems for single concerns in their specific area of expertise and have started to address them. Many devised techniques for separating individual concerns [1][2][10][42][32][23].

III. ANALYSIS OF THE PROBLEM AND SPECIALIZED SOLUTIONS

For software concerns, we distinguished two different levels of separation:

Conceptual level: At the conceptual level, the separation of concerns needs to address three issues. 1) Provide a sufficient abstraction for each concern as an individual concept. 2) Ensure that the individual concepts are primitive, in the sense that they address the natural concerns in the mind of the programmer.

Implementation level: At the implementation level, the separation of concerns needs to provide an adequate organization that isolates the concerns from each other. The goal at this level is to separate the blocks of code which address the different concerns, and provide for a loose coupling of them.

The concerns identified at the conceptual level are mapped into the implementation level using a programming language. Separation of concerns at the conceptual level is generally considered a primary means to manage complexity in all engineering disciplines. However, few programming languages allow these abstractions to actually be separately programmed. The resulting code organization is monolithic, intertwining statements for different purposes.

In programming all concerns in one monolithic program block increases complexity considerably and unnecessarily. By abstracting concerns out and separating them, programming individual concerns becomes substantially less complex, and

code can be effectively reused. Like this some approaches that had been suggested in the literature Demeter [24]. Set of papers that focused on the separation of concern from the basic algorithmic concern. Table 1 gives an overview of this survey; this study also added more references than originals.

Technique → **Concern ↓**	*Meta-level Programming*	*Adaptive Programming*	*Composition Filters*	*Others*
Class organization		Lieberherr et al. 1994		
Process synchronization	Watanabe and Yonezawa 1990	Lopes and Lieberherr 1994	Aksit, Wakita et al. 1994	Frølund and Agha 1993 Reghizzi and Paratesi 1991
Location control	Okamura and Ishikawa 1994			Zeidler and Gerteis 1992 Takashio and Tokoro 1992
Real-time constraints			Aksit et al. 1994	Barbacci and Wing 1986
Others				Liskov and Schifler 1983 Jacobson 1986 Magee et al. 1989

Table 1. Approaches for separating certain concerns from the functionality of the programs.

IV. SEPARATION TECHNIQUES

This report presents a distinction between separation of concerns at the conceptual level and at the implementation level. The former may exist without the latter, and that was pretty much the state-of-the-art in 1994. There were some programming techniques that looked promising for achieving the separation at the implementation level. Table 1 shows the techniques we identified at the time. Following are the highlights of this analysis.

4.1 Meta-Level Programming

Meta-level programming is a well-know paradigm that has been documented in several publications[38][44][14][32]. A reflective system incorporates structures for representing itself. The basic constructs of the programming language, such as classes or object invocation, are described at the meta-level and can be extended or redefined by meta-programming. Each object is associated with a metaobject through a meta-link. The metaobject is responsible for the semantics of operations on the base object.

Meta-level programming support the separation of concerns at the implementation level by trapping message sends and message receives to objects, metaobjects have the opportunity to perform work on behalf of the special purpose concerns. They can check for synchronization constraints, assure real-time specifications, migrate parameters between machines, write logs, and so forth. This allows the base-level algorithms to be written without the special purpose concerns, which in turn can be programmed in the metaobjects. Also, by having structural reflection (meta-knowledge about the relations between classes), meta-level programming can achieve separation between algorithms and data organization.

4.2 Adaptive Programming

The work described in [20] presents adaptive software, a programming model based on code patterns. The relations between the data structures of the application is described by graphs (called class dictionary graphs) to which the patterns apply. A pattern compiler takes a set of patterns and a class dictionary graph and produces an object-oriented program. Code patterns are classified in different categories, each one capturing abstractions in programming:

Propagation patterns define operations (algorithms) on the data. Propagation patterns identify sub graphs of classes that interact for a specific operation.

Transportation patterns abstract the concept of parameterization. They are used within propagation patterns in order to carry parameters in and out along the sub graphs.

Synchronization patterns define synchronization schemes between the objects in concurrent applications. Their purpose is to control the processes' access to the execution of the operations.

Adaptive programming support the separation of concerns at the implementation level each pattern category addresses a different concern. The patterns that define a program can be viewed as the basic software components that interact with each other in a very loose manner through name resolution. Each pattern is quasi independent of both the other patterns and the data organization with the effect that changes in the class organization don't necessarily imply updates in the operations and modifications of the algorithms don't necessarily imply changes in the synchronization scheme.

4.3 Composition Filters

The composition filter model is an extension of the conventional object-oriented model through the addition of object composition filters. For a detailed description of the model and its various applications take a reference from [1][2][4] Filters are first class objects and thus are instances of filter classes. The purpose of filters is to manage and affect message sends and receives. In particular, a filter specifies conditions for message acceptance or rejection and determines the appropriate resulting action. Filters are programmable on a per class basis. The system makes sure that a message is processed by the filters before the corresponding method is executed: once a message is received, it has to pass through a set of input filters and before a message is sent and pass through a set of output filters. Composition filters support the separation of concerns at the implementation level: Separation of concerns is achieved by defining a filter class for each concern. For example, in [2] a real-time filter RealTime was proposed to affect the real-time aspects of incoming messages. RealTime filters have access to a time object that is carried with every message and which specifies the earliest starting time and a deadline for the message. Each filter class is responsible for handling all aspects of its associated concern.

The filter mechanism gives programmers a chance to trap both message receives and sends, and to perform certain actions before the code of the method is actually executed. The resulting code is thus nicely separated into the special purpose concern (in the filter) and basic concern (in the method).

The above techniques are the fact that they provide some mechanism to intercept message sends and receives. Metaobject protocols perform the interception at the meta-level through computational reflection and reification of messages. Composition filters trap messages through the built-in filter mechanism. In both cases, interception was done at run-time. Adaptive programming achieves message interception at compile time. The AOP compiler detects when a method needs to be extended with code for special purpose concerns and inserts that code directly, i.e. a preprocessor. An important aspect of meta-level programming is that the separation of concerns is not imposed by the model. Rather, meta-level programming facilitates the separation of concerns by providing the reflective information about the constructs of the language itself. Programming the special purpose concerns at the meta-level is a strategy that may or may not be followed by the programmers. This is contrary to filters and Adaptive programming, which provide specific language constructs to achieve the separation of concerns. A consequence of this fact is that in both the filters approach and the code patterns approach a new language construct is necessary for each new concern to be dealt with, while in the meta-level programming it is not so.
The important thing about this paper is to point out how the search for better expression mechanisms that focused on certain software development concerns were, in fact, driving a large number of research efforts at the time. This research was being driven by some common goal, and it was important to understand what that was.

V. THE BIRTH OF AOP @ PARC

The term "Aspect-Oriented Programming" (AOP) came into existence at the Xerox Palo Alto Research Center (PARC). There are many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in "tangled" code that is excessively difficult to develop and maintain. [14][15][16][17] present an analysis of why certain design decisions have been so difficult to clearly capture in actual code, call the properties these decisions address *aspects* and shows that, the reason they have been hard to capture is that they *cross-cut* the system's basic functionality and present the basis for a new programming technique, called AOP. It makes possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code. Software is not formally defined, but many good properties of software have been identified among which the crucial ones are understandability, maintainability, reusability and evolvability. These properties are tightly linked to the issue of modularization, which can in essence be seen as the possibility to cleanly encapsulate concerns of the software in separate modules. A module is basically a work assignment, whose role is to localize and hide design decisions. This principle is known as the Separation Of Concerns (SOC) principle. Aspect-Orientation deals with the SOC, the concerns that cross-cut the modularity of traditional programming mechanisms and it aims at reduction of code and to provide higher cohesion.
Separation of concerns (SoC) is the most important aspect in software engineering. It refers to the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal or purpose. Concerns are classified as primary concerns and cross-cutting concerns. Primary concerns are related to core functionality and lie within problem domain and can be easily implemented in traditional programming paradigm like Object Oriented Programming (OOP). OOP implements primary/core concerns in classes. On the other hand cross-cutting concerns for example security, synchronization, error and exception handling, scheduling and optimization, the code implementing these cross-cutting concerns is scattered or woven with the other code in different classes which contradicts to fundamental principle of OOP i.e. SOC. OOP languages have serious limitations in modularizing adequately crosscutting concerns in a program from that point Kiczales Paper presents a statement that AOP build on the basis of OOP technology.

VI. CONCLUSION

Historical approach of AOP is really benefitial for novice users of AOP handlers. AOP is a relatively new programming paradigm and it builds on the basis of OOP technology. It deals with those concerns that cross-cut the modularity of traditional programming mechanisms and it aims at reduction of code and to provide higher cohesion. As with any new technology AOP provides some benefits and securities to the data than other programming languages.

REFERENCES

[1] Aksit M., Bergmans L., and Vural S. 1992. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In O. Lehrman Madsen, editor, European Conference on Object-Oriented Programming (ECOOP), pages 372:396, Utrecht, The Netherlands, June/July 1992. Springer Verlag, Lecture Notes in Computer Science. Vol. 615.

[2] Aksit M., Bosch J., van der Sterren W., and Bergmans L. 1994 Real-Time Specification Inheritance Anomalies and Real-Time Filters. In Mario Tokoro and Remo Pareschi, editors, European Conference onObject-Oriented Programming (ECOOP), pages 386:407, Bologna, Italy, July 1994. Springer Verlag, Lecture Notes in Computer Science. Vol. 821.

[3] Aksit M., Wakita K., Bosch J., Bergmans L., and Yonezawa A. 1994. Abstracting Object Interactions using Composition-Filters. In M. Guerraoui, O. Nierstrasz, and M. Riveill, editors, Object-Based Distributed Processing. Springer Verlag, Lecture Notes in Computer Science, 1994. Ballard, B. and Biemann, A. 1979. Programming in

Natural Language: NLC as a Prototype. Proc. ACM/CSC-ER Annual Conference, 228-237.

[4] Bergmans L. 1994. Composing Concurrent Objects. PhD thesis, University of Twente, Enschede, The Netherlands, July 1994.

[5] Barbacci M. and Wing J. 1986. Specifying Functional and Timing Behavior for Real-Time Applications. Technical Report CMU/SEI-86-TR-4 ADA178769, Software Engineering Institute (Carnegie Mellon University), 1986.

[6] Frølund S. and Agha G. 1993. A Language Framework for Multi-Object Coordination. In Oscar M. Nierstrasz, editor, European Conference onObject-Oriented Programming (ECOOP), pages 346-360, Kaiserslautern, Germany, July 1993. Springer Verlag, Lecture Notes in Computer Science. Vol. 707.

[7] Gamma E., Helm R., Johnson R., and Vlissides J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Reading, MA, October 1994. ISBN 0-201-63361-2.

[8] Grosso W. 2002. Aspect-Oriented Programming and AspectJ. In Dr. Dobbs Journal. August 2002. http://www.ddj.com/articles/2002/0208

[9] Harrison W. and Ossher H. 1993. Subject-oriented programming (a critique of pure objects). In proc. Object-Oriented Programming Systems Languages and Applications (OOPSLA), pp.411-428. 1993.

[10] Honda Y. and Tokoro M. 1992. Soft Real-Time Programming through Reflection. In International Workshop on Reflection and Meta-Level Architecture, pages 12:23, Tama-City, Tokyo, Japan, November 1992.

[11] Hürsch W. and Lopes C.V. 1995. Separation of Concerns. Northeastern University, College of Computer Science Technical Report NU-CCS-95-03. February 1995.

[12] Irwin, J., Loingtier, J.-M., Gilbert, J.R., Kiczales, G., Lamping, J., Mendhekar, A. and Shpeisman, T. 1997. Aspect-oriented programming of sparse matrix code. Scientific Computing in Object-Oriented Parallel Environments. First International Conference, ISCOPE 97.Proceedings. Springer-Verlag, 1997. p.249-56.

[13] Jacobson I. 1986. Language Support for Changeable Large -Real Time Systems. In Proc.Conference on Object-Oriented Programming Systems, Tools and Applications (OOPSLA'86).ACM Press. pp. 377-384.

[14] Kiczales G., des Rivieres J., and Bobrow D.G. 1991. The Art of the Metaobject Protocol. TheMIT Press, Cambridge, Massachusetts, 1991. ISBN 0-262-11158-6 (hc.).

[15] Kiczales G. and Andreas Paepcke. 1995. Open Implementations and Metaobject Protocols.Tutorial slides and notes. http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf

[16] Kiczales, G. 1996. Beyond the black box: open implementation. IEEE Software, vol.13, (no.1),IEEE, Jan. 1996.

[17] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M. and Irwin J. Aspect-Oriented Programming. 1997. In Proc. 11th European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag LNCS 1241. June 1997.

[18] Laddad R. 2002. I want my AOP! In Java World magazine. January, March and April 2002.

[19] Lesiecki N. 2002. Test flexibility with AspectJ and mock objects. In Java Technology Zone for IBM's Developer Works. May 2002.

[20] Lieberherr K.J., Silva-Lepe I., and Xiao C. 1994. Adaptive object-oriented programming using graph-based customization. Communications of the ACM, 37(5):94:101, May 1994.

[21] Lieberherr K.J., Orleans D. and Ovlinger J. 2001. Aspect-Oriented Programming with Adaptive Methods. In Communications of the ACM 44(10). October 2001.

[22] Liskov B. and Scheifler R. 1983. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM Transactions on Programming Languages and Systems, July 1983.

[23] Lopes C.V. and Lieberherr K.J. 1994. Abstracting Process-to-Process Relations in concurrent Object-Oriented Applications. In Mario Tokoro and Remo Pareschi, editors, European Conference onObject-Oriented Programming (ECOOP), pages 81:99, Bologna, Italy, July 1994.

[24] Lopes C. 1996. Adaptive Parameter Passing. In Proc. International Symposium on Object Technologies for Advanced Software (ISOTAS'96). Springer-Verlag LNCS n.1049. Japan, 1996.

[25] Lopes C. 1998. D: A Language Framework for Distributed Programming. PhD Thesis, College of Computer Science, Northeastern University.

[26] Lopes C. and Kiczales G. 1998. Recent Developments in AspectJ. In Proc. Aspect-Oriented Programming Workshop at ECOOP'98. Workshop Reader, Springer-Verlag LNCS 1543. July 1998.

[27] Maes P. 1987. Concepts and Experiments in Computational Re ection. In Norman Meyrowitz, editor, Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), pages 147{155, Orlando, Florida, October 1987. ACM Press. Special Issue of SIGPLAN Notices, Vol.22, No.12.

[28] Magee J., Kramer J., and Sloman M. 1989. Constructing Distributed Systems in CONIC. IEEE Transactions on Software Engineering, 15(6):663:675, June 1989.

[29] Mahoney J.V. 1995. Functional Visual Routines. Xerox Palo Alto Research Center Technical Report SPL95-069, July 1995.

[30] Matsuoka S. and Yonezawa A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, Research Directions in Concurrent Object-Oriented Programming, chapter 1, pages 107:150. The MIT Press, Cambridge, Massachusetts, 1993.

[31] Mendhekar A., Kiczales G. and Lamping J. 1997. RG: A Case-Study for Aspect-Oriented Programming. Xerox Palo Alto Research Center Technical Report SPL97-009 P9710044. February 1997.

[32] Okamura H. and Ishikawa Y. 1994. Object Location Control Using Meta-level Programming. In Mario Tokoro and Remo Pareschi, editors, European Conference on Object-Oriented Programming (ECOOP), pages 299:319, Bologna, Italy, July 1994. Springer Verlag, Lecture Notes in Computer Science. Vol. 821.

[33] Orleans D. and Lieberherr K.J. 2001. DJ: Dynamic Adaptive Programming in Java. In Proc. Reflection 2001.

[34] Springer-Verlag. Price D., Riloff E., Zachary J. and Harvey B. 2000. NaturalJava: A Natural Language Interface for Programming in Java. Proc. ACM Intelligent User Interfaces Conference.

[35] Reghizzi C. S. and de Paratesi G.G. 1991. Definition of Reusable Concurrent Software Components. In Pierre America, editor, European Conference on Object-Oriented Programming (ECOOP), pages 148:166, Geneva, Switzerland, July 1991. Springer Verlag, Lecture Notes in Computer Science. Vol. 512.

[36] Sammet, J. 1966. The Use of English as a Programming Language. Comm. ACM, 9(3), 228-230.

[37] Silva-Lepe I., Hursch W., and Sullivan G. 1994. A Report on Demeter/C++. C++ Report, 6(2):24:30, February 1994.

[38] Smith B.C. 1984. Reflection and Semantics in Lisp. In ACM Symposium on Principles of Programming Languages, pages 23:35, Salt Lake City, UT, January 1984. ACM Press.

[39] Sousa P., Sequeira M., Ferreira P., Zúquete A., Lopes C., Pereira J., Guedes P. and Alves Marques J. 1993. Distribution and Persistence in the IK Platform: Overview and Evaluation. In Usenix Computing Systems Journal 6(4), Fall 1993.

[40] Spurlin V. 2002. Aspect-Oriented Programming with Sun ONE Studio. In Sun ONE Studio Developer Resource page. October 2002. http://forte.sun.com/ffj/articles/aspectJ.html

[41] Steele G. 1990. Common Lisp: The Language. Second Edition. Digital Press.

[42] Takashio K. and Tokoro M. 1992. DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems. In Andreas Paepcke, editor, Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA), pages 276:294, Vancouver, Canada, October 1992. ACM Press.

[43] Walker R.J., Baniassad E.L.A., Murphy G.C. An initial assessment of aspect-oriented programming. 1999. Proceedings of the 21st International Conference on Software Engineering (ICSE '99), Los Angeles, CA, USA, 16-22 May 1999. ACM, 1999. p.120-30.

[44] Watanabe T. and Yonezawa A. 1990. Reflection in an Object-Oriented Concurrent Language. In Akinori Yonezawa, editor, ABCL: An Object-Oriented Concurrent System, chapter 3, pages 45- 70. The MIT Press, Cambridge, Massachusetts, 1990. ISBN 0-262-24029-7.

[45] Winkler D., Kamins S. and DeVoto J. 1994. Hypertalk 2.2: The Book. Random House.

[46] Zeidler C. and Gerteis W. 1992. Distribution: Another Milestone of Application Management Issues. In G. Heeg, B. Magnusson, and B. Meyer, editors, Technology of Object-Oriented Languages and Systems (TOOLS Europe), pages 87-99, Dortmund, Germany, March 1992.