

Roles, Subjects and Aspects: How do they relate?

Daniel BARDOU

L.I.R.M.M.

161 rue Ada

34392 MONTPELLIER CEDEX 5

FRANCE

email: bardou@lirmm.fr

phone: +33 4 67 41 85 86

fax: +33 4 67 41 85 00

Abstract

The need for comparison between AOP and related approaches has been retained as a key issue in [MLTK97]. We have noticed strong similarities between AOP and some object-oriented approaches including some notion of viewpoint. We report them in this paper and we discuss how this brief comparison may be used for further discussion of the other key issues that were retained in [MLTK97].

1 Introduction

The notion of aspect has been mainly introduced in [KLM⁺97] where Aspect-Oriented Programming was also briefly compared to subjective programming as two different yet complementary programming style. On the other hand the need for further comparison between AOP and subject-oriented programming has been recognized as a key issue in [MLTK97].

We consider subject-oriented programming to be a subset of works achieved around the notion of viewpoint in many research domains related to object technology including knowledge representation, database management, programming and design. This notion appears in various approaches under various terms such as subjectivity, subjects, roles, perspectives, views, viewpoints, coreference, multiple representation, and even aspects. In order to find out how our previous work on delegation and split objects [BD96] was related to those approaches, we compared our own notion of viewpoint to those of more than 15 other approaches [Bar98]. This comparison study pointed out there are indeed a large diversity of similar yet different viewpoint notions. The aim of this paper is to concentrate on some of those notions that we consider to be closely related to AOP as introduced in [KLM⁺97] which we hadn't explicitly considered yet.

The rest of this paper is organized as follows. Section 2 give short presentations of some approaches including a viewpoint notion that will serve as a basis for further comparison with Aspect-Oriented Programming in Section 3. Section 4 consider other key issues in aspect orientation in the light of this new comparison.

2 Some works including the viewpoint notion

We summarize in this section the presentation of some works achieved in object-oriented technology, which deals with some notion of viewpoint. The list of presented works is not exhaustive and we rather concentrate on the works we consider to be most related to AOP. The presentations themselves also are limited to the characteristics we considered relevant to the comparison. A more complete overview of works related to the viewpoint notion can be found in [Bar98].

2.1 Role modeling

Role modeling [AR92] is an object-oriented design technique which emphasizes separation of concerns between different aspects at possibly different level of details in the design of more or less independent aspects of the overall design. A *role model* is a design unit which describes the way several roles should interact in order to achieve some particular task. A *role* is itself a description of the needs and responsibilities an object must satisfy so that it can be used in the execution of the task. A *role* is thus a particular point of view on an entity, that of its participation in the application of a task. Not only one role model can be defined for each different task which is carried out by the system, but several role models can describe the same task at different levels of detail.

A significant operation on role models is role model synthesis which consists in putting together several roles to be played by the same entity in order to establish connection between several role models. This regrouping of roles is called projection of roles and results in a new aggregate role, which gathers the requirements expressed in the original roles.

Synthesis of role models may be used in particular to find out the implementation of an application in an object-oriented language. By successive synthesis of role models describing its different aspects, one can obtain a role model describing application as a whole. Role projections then help in identifying requirements for the objects to be implemented. Role models can also be used for reusing existing objects, if it can be established that their behavior is conform to the roles having to be assumed by the same entity in the application.

Role models might be only used at the design level or be retained at the implementation level, but this requires more specific techniques than role projection described here. [VN96a, VN96b] propose role components as such an implementation technique using C++ templates. The key principle is to implement a role by a template class with a parameter for each other entity interplaying in the role model. Roles might then be composed in various ways at compile-time.

2.2 Activities and roles

Works by KRISTENSEN have considered both *transverse activities* [Kri93, KM96] and *roles* [KØ96, ØK95]. A *transverse activity* is an activity the execution of which involves several objects in an object-oriented modeling, they are non object-centric actions. KRISTENSEN's proposal is to consider an activity as a single entity, defined by a set of *participants* (the different objects involved in the activity execution) and a *directive* which describes how the participants collaborate. Activities might be specialized and aggregated. A sub-activity has a more specific directive than the activity it specializes, and at least the same participants. The directive of an activity might be decomposed in part-activities and actions of participants. Activities might also have properties which wouldn't exist independently in part-activities or participants.

An object might be involved as a participant in several different activities (of either different or the same kind), and may then be considered from the point of view of a definite activity. The different activities of which an object is a participant can change during its lifetime. The notion of *role* supports those dynamic changes. Roles are described as role-classes, to be role for some (regular) class or another role-class. Role-classes describes additive state and behavior for the instances of the classes they are role for. Specialization and aggregation of roles is possible [ØK95]. An object might have several roles at the same time. Activities can be defined as relation-classes having role-classes as their domains [KM96]. Different roles of an object then denote its involvement as a participant in different activities.

2.3 CROME

CROME [VCD97] is an object-oriented design framework introducing the notions of *functional planes* and *contexts*. At the design level, the system is orthogonally subdivided into *contexts* and objects. A *context* is the description of the system which is relevant to one of its definite function. A context involves several (possibly all) objects in the system, and an object might be involved in several (possibly all) contexts.

The framework is organized on top of what is called the *base plane*. The *base plane* defines classes and inheritance and composition relationships between them. The *base plane* determines the basic description of the objects, i.e. the one which is relevant to any of the contexts. A *functional plane* holds together the additional description relevant to a definite context. For each class specifically involved in the context, the *functional plane* might complete the basic

description of its instances either in terms of variables and methods. Local classes might also be added to *functional planes* in order to describe objects which are local to some context.

CROME has been implemented in ROME, a language supporting multiple representation of objects and the notion of viewpoint [Car89, CG90], allowing to retain *functional planes* at the implementation level. Another way to keep track of *contexts* throughout the implementation is based on the extension of the *categories* of SMALLTALK [Van97].

2.4 Subject-Oriented Programming

Subject-Oriented Programming is introduced in [HO93] where a *subject* is defined as “a collection of state and behavior specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool.” In the programming language, a subject is a class inheritance hierarchy, where as in other traditional class-based languages, each class defines instance variable names and methods for its instances. A class may appear in different subjects and define there different instance variable names and methods. A subject is an abstraction and can be instantiated (possibly more than once) to obtain a *subject activation* holding actual data. The notion of object identity is what relate different subject activations together: an unique object may appear in several subject activations (even with different states and behaviors) but it has the same unique identity in all of them.

A subject activation corresponds to the implementation of an application. Subject activations can be composed to obtain larger applications. This operation is called subject composition and is defined at the subjects level with respect to *subject composition rules* which determine precisely how the composed subject activations interacts. Several subject composition rules might be considered including *merge* and *override* [OH92, OKH⁺95]. The former allows to merge several subjects into a global class hierarchy, whereas the latter allows to extend existing applications in a non destructive manner.

2.5 Split objects

We consider the delegation mechanism of prototype-based languages as a natural way to express viewpoints. We stated in [BD96] that a correct interpretation for delegation links (also called *parent links*) was to consider objects connected by them as the representation of the same entity of the application domain. We call such a representation a split representation. Each object in the delegation hierarchy of a split representation denotes a different viewpoint on the represented entity, and the delegation hierarchy denotes a specialization relationship between these viewpoints.

Split representations nevertheless raise an important problem related to object identity. Since delegation is often used for other purposes (including inheritance in most prototype-based languages) than split representation, and also since split representations are not first-class entities, they can't be properly identified and located in large delegation graphs. To get rid of this problem, we proposed *split objects* [BD96] as objects described by a collection of *pieces* organized in a delegation hierarchy. Each piece holds a part of the description of both the state and the behavior of the split object. Each piece denotes a different viewpoint from which the split object can be considered (by message sending), and the delegation hierarchy denotes a specialization hierarchy on these viewpoints. Pieces do not have an object status, whereas split objects do, so that the object identity problem is not raised.

Split objects have been compared to other related approaches including a certain notion of viewpoint [Bar98]. We noticed in this comparison strong similarities between the notion of role appearing in many other works (including some of the ones considered in this paper), and more specifically that they could be used for the implementation of designs based on role models (see § 2.1). Requirements of a role can be described by a piece and the detail of the different roles an entity has to assume in different role models can be retained at the implementation level if this entity is represented by a split object. So far role models cannot be better supported than by piece naming (this implies a different unique name has to be given to each role model in the system). We are currently considering different possible solutions for sharing between split objects, some proposals either in a prototype-based language or in a class-based language can be found in [Bar98]. We believe one or more of those solutions may provide some way to explicitly represent role models (and not only role details).

2.6 Us

The idea to use delegation as a mechanism for viewpoints can also be found in Us [SU96] “a *subjective version of SELF*” [SU95]. The notion of *layer* is introduced in Us in order to support multiple representation of objects: each layer holds a part (eventually empty) of the whole description of an object in the system. Layers are organized in a delegation hierarchy where a layer considered together with all its ascendant layers corresponds to a *perspective* on the system form which messages can be sent. A perspective is here considered as a global point of view on the system in which objects are organized in a delegation hierarchy similar to the one of SELF. There are thus two kinds of delegation links (and two interconnected delegation hierarchies): the *parent* link between objects and the *parent layer* link between layers.

Message sending in Us is handled with respect to the so called “*perspective-receiver symmetry principle*” which corresponds to the authors’ opinion on subjectivity. Indeed they make an analogy between the evolution of programming languages and a world with one dimension (procedural languages) evolving to two dimensions (object-oriented languages) and then to three dimensions (subjective object-oriented languages). Procedure invocation might be considered in a procedural language as message sending in a world where there is only one possible receiver. In an object-oriented language there can be several possible receiver for the same message selector and both have to be taken into account to determine which code is executed. Subjective object-oriented programming amount to consider a perspective (or viewpoint) in addition to the receiver and the selector to correctly handle a message. Sending messages either to the same perspective, or to the same receiver, allows the programmer to respectively consider either all the objects from a definite point of view on the whole system, or a definite object from all the points of view on the system. There even exist an implicit receiver in Us (called `self`) and an implicit perspective (called `here`), so that messages can be sent without specifying either the receiver, the perspective or both of them.

3 Comparison with Aspect-Oriented Programming

In this section we compare AOP with other works presented in the previous section. We have found strong similarities that we point out and briefly discuss here. We won’t consider all differences in details and will refer to some of them in the next section. We first consider cross-cutting as a common feature which will serve as the basis for the comparison.

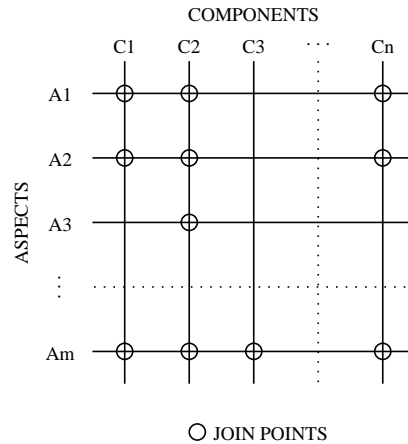


Figure 1: Cross-cutting between components and aspects in AOP.

Cross-cutting Cross-cutting (between *components* and *aspects* is a key feature of AOP. This cross-cutting is represented in Fig. 1 where n components are depicted by vertical bars, and m aspects are depicted by horizontal bars and where circles represent *joint points* where aspects interact with components. The same aspect may interact with all components (e.g. A_m) or only some of them (e.g. A_1), and vice versa (e.g. C_2 and C_1). We believe this cross-cutting to be the main characteristic of AOP: aspects wouldn’t be called aspects if they didn’t cross-cut components.

ASPECT-ORIENTED PROGRAMMING	Role modeling	Activities & roles	CROME	Subject-Oriented Programming	Split objects	Us
ASPECTS	role models	activities	contexts	subjects	viewpoints	perspectives
COMPONENTS	objects	objects	objects	objects	objects	objects
JOIN POINTS	roles	roles	part of description in a functional plane	part of description in a subject	pieces	part of description in a layer

Table 1: Cross-cutting in the different approaches

Similar cross-cutting can be found in all approaches presented in the previous section with other terms being used in place of components. We consider this common point as a basis for our comparison and a mapping between similar terms as summarized in Tab. 1. We discuss this mapping further in the following subsections.

Components and component support All the approaches we have considered in previous section are object-oriented. Hence it seems straightforward and natural to choose objects as components and object-oriented programming languages serve then as component language. However subjects for example might have been chosen as components in subject-oriented programming (and objects would then become aspects) and it is true the choice of components might be discussed. This can be explained by the cross-cutting: if aspects and components define two orthogonal dimensions, one might choose any of them prior to the other. This is what is expressed by the *perspective-receiver symmetry principle* of Us, and what has also been reported in [MLTK97]. However we believe symmetry is never completely achieved and as it is the case in AOP where the main emphasis is on aspects, there is always some dimension which takes precedence on the other.

Aspects and aspect support We have considered as aspects what cross-cut objects in all approaches. What is provided by each approach to support aspects varies from an approach to another. It might be a design notation (role models, activities & roles), specific support for description (activity classes, functional planes, subjects, layers). The way aspects might be organized together (defined in term of each others, e.g. specialized or composed) depends on this support. Aspects might also be retained at the implementation level or not, and represented by explicit entities in the system or not. In this latter case, we assume join points have to be explicitly represented in order to not step on the code tangling problem discussed in [KLM⁺97].

Join points and weaving Because of the cross-cutting we believe the intersection between components and aspects to be as much important as components and aspects themselves. Join points should be easily located in systems, either because there exists explicit constructs to express them (roles, role components of [VN96a], pieces) or because they can be determined from aspect support (part of the description of an object in a functional plane/subject/layer). Join points are directly related to weaving. We believe weaving does not necessarily rely on weavers and is always a process achieved once or several times as it is presented in [KLM⁺97]. Such processes are role models synthesis and subject composition for example. We believe the rules and the principles according to which weaving is achieved to be the more important feature of weaving. For example, specification of roles as domains for activities [KM96], subject composition rules, or even sharing specified by delegation links (in split objects and Us) is also weaving. An important notion which is implicitly implied in weaving is the notion of identity: roles are attached to an object with a unique identity, the description of an object might be scattered in several classes and functional planes but it has a unique identity, an object might appear in several subject activation but has a unique identity used as a basis for subject composition, pieces belongs to an object with a unique identity.

4 Discussion

We have noticed cross-cutting to be a common point between approaches presented in § 2 and AOP, and we were able to locate where the notions of components, aspects and join points could be found in those works. We have so far put the emphasis on similarities and have let differences apart. We suggest those differences should be considered as possible benefits for AOP. In order to discuss this we review most of the key issues noticed in [MLTK97] to discuss possible future works.

- **Need for more technical research.** We believe technical research achieved in considered works, more particularly on how roles or similar notions can be represented, and on which mechanism might be used to handle them, might contribute to AOP.
- **Need for AO*.** The scope of the works considered is not limited to programming but includes also design and modeling. Similarities noticed between them might be used throughout the software life-cycle.
- **Need for a theoretical foundation.** Theoretical research has been achieved around the notion of role [KØ96], elements of this theory might be applied to AOP.
- **Separation of concerns.** Two questions were raised in [MLTK97]: what are the concerns to be separated? and how much to should them be separated? Separation of concerns is present in any of the considered works in this paper. We didn't however consider aspects (and concerns) such as memory allocation and/or code optimization [KLM⁺97]. We addressed the second question more directly, in the sense that a common point to most of the considered works is that aspects (or similar notions) might be organized in specialization hierarchies or be only considered aside to each others as in CROME.
- **Is AOP bound to OO?** We have considered only object-oriented approaches in this paper, although AOP has clearly been introduced in a larger scope. With respect to this position, this paper could be considered as a direct contribution to some non empty branch of AOP which we might term AOP (Aspect- and Object-Oriented Programming) and as a possible indirect contribution to AOP in its general sense.
- **Other.** This paper's comparison might also prove useful with respect to the domain of aspects, aspects description, weaving, specifying join points, visual representations of AOP (notations for activities and roles [KM96]; the implementation of CROME in Smalltalk includes a context-oriented browser [Van97]).

References

- [AR92] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, pages 133–152, Utrecht, The Netherlands, June 1992. Springer-Verlag.
- [Bar98] Daniel Bardou. *Étude des langages à prototypes, du mécanisme de délégation, et de son rapport à la notion de point de vue*. Thèse de doctorat en informatique, Université Montpellier 2, April 1998.
- [BD96] Daniel Bardou and Christophe Dony. Split Objects: a Disciplined Use of Delegation within Objects. In *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 122–137, San Jose, California, USA, October 1996. Published as ACM SIGPLAN Notices 31(10).
- [Car89] Bernard Carré. *Méthodologie orientée objet pour la représentation des connaissances, concepts de points de vue, de représentation multiple et évolutive d'objet*. Thèse d'informatique, Université des Sciences et Techniques de Lille, January 1989.
- [CG90] Bernard Carré and Jean-Marc Geib. The Point of View Notion for Multiple Inheritance. In Norman Meyrowitz, editor, *Proceedings of the 5th Conference on Object-Oriented Programming Systems, Languages, and Applications / 4th European Conference on Object-Oriented Programming (OOPSLA/ECOOP'90)*, pages 312–321, Ottawa, Canada, October 1990. Published as ACM SIGPLAN Notices 25(10).
- [HO93] William H. Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In Andreas Paepcke, editor, *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, Washington, DC, USA, October 1993. Published as ACM SIGPLAN Notices 28(10).
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.
- [KM96] Bent Bruun Kristensen and Daniel C. M. May. Activities: Abstractions for Collective Behavior. In Pierre Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 472–501, Linz, Austria, July 1996. Springer.
- [KØ96] Bent Bruun Kristensen and Kasper Østerbye. Roles: Conceptual Abstraction Theory & Practical Language Issues. *Theory And Practice of Object Systems (TAPOS)*, 2(3):143–160, 1996. Special Issue on Subjectivity in Object-Oriented Systems.
- [Kri93] Bent Bruun Kristensen. Transverse Activities: Abstractions in Object-Oriented Programming. In S. Nishio and Akinori Yonezawa, editors, *1st JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *Lecture Notes in Computer Science*, pages 279 – 296, Kanazawa, Japan, November 1993. Springer.

- [MLTK97] Kim Mens, Cristina Lopes, Bedir Tekinerdogan, and Gregor Kiczales. Aspect-Oriented Programming Workshop Report. In Jan Bosch and Stuart E. Mitchell, editors, *Object-Oriented Technology — ECOOP'97 Workshops Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 481–494. Springer, June 1997.
- [OH92] Harold Ossher and William H. Harrison. Combination of Inheritance Hierarchies. In Andreas Paepcke, editor, *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, pages 25–40, Vancouver, British Columbia, Canada, October 1992. Published as ACM SIGPLAN Notices 27(10).
- [ØK95] Kasper Østerbye and Bent Bruun Kristensen. Roles. Technical Report R-95-2006, Department of Mathematics and Computer Science, Aalborg University, Denmark, March 1995.
- [OKH⁺95] Harold Ossher, Matthew Kaplan, William H. Harrison, Alexander Katz, and Vincent Kruskal. Subject-Oriented Composition Rules. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pages 235–250, Austin, Texas, USA, October 1995. Published as ACM SIGPLAN Notices 30(10).
- [SU95] Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 303–330, Århus, Denmark, August 1995. Springer-Verlag.
- [SU96] Randall B. Smith and David Ungar. A Simple and Unifying Approach to Subjective Objects. *Theory And Practice of Object Systems (TAPOS)*, 2(3):161–178, 1996. Special Issue on Subjectivity in Object-Oriented Systems.
- [Van97] Gilles Vanwormhoudt. Programmation par contextes en Smalltalk. *L'objet*, 3(4):429–444, December 1997. numéro spécial Smalltalk.
- [VCD97] Gilles Vanwormhoudt, Bernard Carré, and Laurent Debrauwer. Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk. In Roland Ducournau and Serge Garlatti, editors, *Actes de la conférence Langages et Modèles à Objets (LMO'97)*, pages 223–239, Roscoff, France, October 1997. Hermès.
- [VN96a] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *2nd JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, page ??, Kanazawa, Japan, March 1996. Springer-Verlag.
- [VN96b] Michael VanHilst and David Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 359–369, San Jose, California, USA, October 1996. Published as ACM SIGPLAN Notices 31(10).