Member-Group Relationships Among Objects

William Harrison, Harold Ossher
IBM T. J. Watson Research
P.O. Box 704
Yorktown Heights, NY 10598
{harrisn, ossher}@watson.ibm.com

ABSTRACT

Aspect-Oriented Software is a broad term, encompassing several different views on the nature of the aspects and the relationships between aspects and objects. Attaching aspects to objects is one way of forming a group. While there are many useful patterns of interaction, e.g. strategies [2], decorators, and the like, we focus on groups in which the group delegates to members to obtain behavior and the members may either perform their own behavior or delegate to the group. Using issues of behavior, this paper explores and classifies the relationships between objects and groups of objects in which they may participate as a first step in laying a foundation for unifying these different views as special cases of a common framework.

Keywords

Aspect-oriented software development, delegation, composition, method combination.

1. INTRODUCTION

Different mechanisms in the AOSD space emphasize different means of separating and integrating concerns. For example, Hyper/J [9] focuses on composing class hierarchies, which in turn involves synthesizing composed classes and their methods from input classes. AspectJ [5] focuses attaching *advice* and *aspects* to *join points* and objects. The composition filters approach [1] focuses on attaching *filters* to objects to filter method calls and returns. These approaches all allow attachment of additional behavior to objects and/or combination of objects to form single objects with combined behavior.

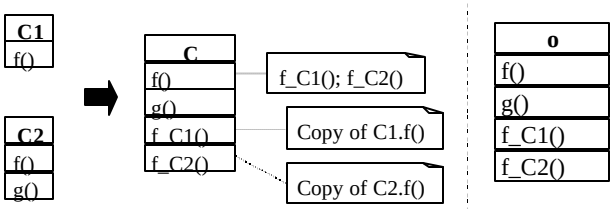

Figure 1. Hyper/J Class Composition and Example Instance

A simple example of class composition is shown in **Figure 1**. Input classes C1 and C2 are composed to form class C. All instantiations of classes C1 and C2 are changed to instantiations of C, so that, at runtime, only C instances exist, like *o* in the figure.

Earlier work on tool integration, including event broadcasting [11], cooperative call [8], and *mediators* [12], on the other hand, was concerned with tying together sets of separate objects (or other modules, but we confine ourselves to objects in this paper). Mediators, for example, provide for *implicit invocation*, so that events in an object can trigger a mediator, which can then trigger actions in other objects. AOSD approaches like those mentioned above do provide implicit invocation but do not address the implicit binding together of behaviors of several base objects performed by tool integration mechanisms.

Object-oriented programmers often split the implementation of functionality across several objects, relying on them to cooperate in carefully-designed ways to achieve the desired objective [4]. A common approach is *delegation*, in which part of the behavior of an object is specified in and delegated to other objects (or classes), such as *strategy* objects [2]. In conversations about Hyper/J, several developers have told us that they would prefer a model where composition produced object collaborations rather than single, composed objects.

Separating functionality into separate objects also provides more dynamic flexibility. Provided great care is exercised to coordinate the activity among all affected objects, it is possible to dynamically add objects to or remove them from a group, thereby adding or removing functionality, or replacing implementations. This kind of dynamism has always been important in some contexts, such as long-running telephone switches, and is becoming ever more important in the context of web-based applications.

Serious problems can arise, however, when the functionality that conceptually belongs in a single object is split across multiple objects in a group. These include *object schizophrenia* and *broken delegation* [14], and are due to the fact that the separate objects making up the group have their own, separate identities; even if they cooperate, they don't truly behave like a single object unless great care is taken, and the breakages are subtle. This paper analyzes these issues, and discusses various ways of handling identity and the relationships between objects and their groups, and their implications.

The first section analyzes a number of the major factors that characterize the ways in which an object's behavior can be related to a group of which it is part and then applies these factors to enumerating the potential kinds of relationships between objects and their groups. We winnow the enumeration by analyzing conflicts and usages that can lead to difficulties. The section concludes with a discussion of synergy and conflict in the relationships a single object may bear to multiple groups. The second section builds on and re-applies the factor analysis and winnowing process to

classes instead of instances of objects. The third section discusses the behavior of composition operations, responsible for aggregating objects into groups in order to realize composed behavior, when dealing with potential conflicts in multiple relationships. The fourth section discusses some options for implementing groups of objects and the composition operations that perform them, and analyzes the implications of some implementation decisions.

2. INSTANCE RELATIONSHIPS

2.1 Groups of Primitive Objects

Assume that, in concept, Java™ objects are either *primitive objects*, with fields and method bodies written by developers, or *group objects*, representing collections of objects that have been composed together. Group objects are created by composition operations, and do nothing but call methods of primitive objects (or, perhaps, of other groups) as determined by the compositions. Assume also that the group exposes the interfaces of all its member objects, so that methods of the primitive objects in a group can also be called on the group object itself. The group will generally delegate such operations to the appropriate primitive object(s), including other objects providing advice, filters, or additional implementations. The group thus serves as a kind of method combination dispatcher, determining how the composed behavior of each method call is to be realized in terms of the primitive methods supplied by the group members.

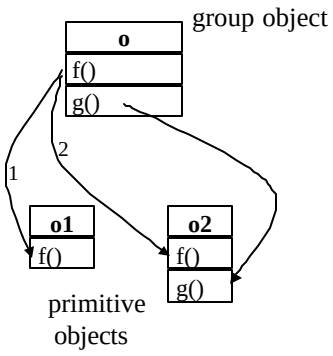

Figure 2. A Simple Group

Figure 2 shows a group of objects that realizes the same composition as illustrated in **Figure 1**. The entire group corresponds to the single, combined instance, *o*, in **Figure 1**. In this case, however, separate instances o1 and o2 of the original, uncomposed classes C1 and C2 do exist at runtime, as well as the group object, *o*, that ties them together.

Within the bodies of primitive object methods, *either of two* identities might be used, called *this* and *self*.[1] Many factors enter the analysis, but when they are different, *this* refers to the primitive and *self* to the group. Calls directed to *this* are thus directed to the primitive object itself, and do not invoke composed behavior, whereas calls to *self* are directed to the caller's group, and do invoke composed behavior.

[1] These terms are not ideal, because they are typically used to mean the same concept (object self-reference), though in different languages. Here we are using them to denote different concepts within the same model. We are in the process of trying to come up with better terms.

2.2 Factors in Describing Relationships

Trying to remain independent of the way the behaviors are actually implemented, we now explore and categorize the kinds of relationships among primitive and group objects to lay the groundwork for systematic support.

Leaving aside, until Section 2.4, situations in which groups act as members of larger groups, each kind of relationship between a primitive object and a group can be operationally characterized by several effects. The following table lists the relationships along with the effects ascribed to each. Explanation of columns:

Identity	Assuming that Java's reference equality semantics are appropriately extended, comparison of the identity of a primitive object and the identity of its group object can yield "equal" or "not equal".
Primitive-to-group	When a primitive object calls a method on a primitive object, the primitive object can cause group behavior rather than use its own method implementation. Three alternatives can be listed (see Figure 3):
	no — The primitive does not cause group behavior, but performs its primitive behavior instead.
	identical — The primitive yields to common group behavior (that which results when the method is called on the group object)
	variant — The primitive causes group behavior different from the common group behavior (such as including its own behavior in addition to the common group behavior).
Group-to-primitive	**When a method is called on a group, the group uses behavior defined by various primitive objects of the group. Three alternatives can be listed for how the group uses the primitive's behavior (see** Figure 4):
	no — The primitive's behavior is not included in the group behavior.
	self=primitive — Group behavior includes the primitive's behavior, but in interpreting the primitive's behavior, references to itself as *self*, are not to be interpreted

as if referring to the group, but as references to the primitive.

self= group — Group behavior includes the primitive's behavior and in interpreting the primitive's behavior, references to itself as *self* are to be interpreted as references to the group rather than as references to the primitive. (But see the automanipulation alternatives, next).

Auto-manipulation — When *self*=group, there are a number of ways in which the behavior in a Java method may refer to the group by using *self*. The developer might explicitly refer to *self*, if this is permitted. (It would be, in effect, a Java language extension whose normal Java semantics might be innocuous.) Without a language extension, reference to the group can arise by reinterpreting the Java "*this*". Three cases can be listed (see Figure 5):

this= primitive — Explicit and implicit uses of *this* refer to the primitive.

this= group — Explicit and implicit uses of *this* use the value of *self*, and in this case, *self*=group.

mixed — Although there are hundreds of different mixed variations, corresponding to the different ways in which *this* appears in Java the most frequent suggestion is to make manifest uses of *this* refer to the primitive while other uses use the value of *self*. The only reason that this variation is particularly interesting is that, accomplished in-spite-of any general policy, it can be forced by a developer who copies the bodies of *final* methods into places where they are manifestly invoked on *this*.

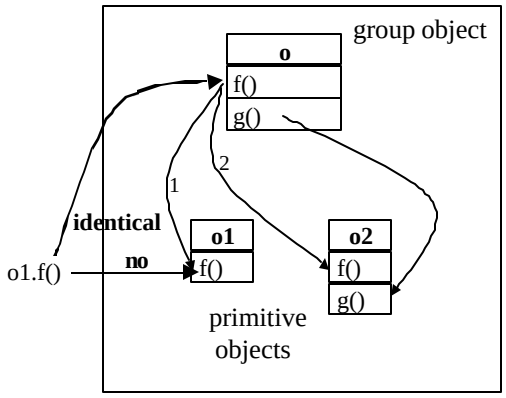

Figure 3. Primitive-to-Group Options "no" and "identical"

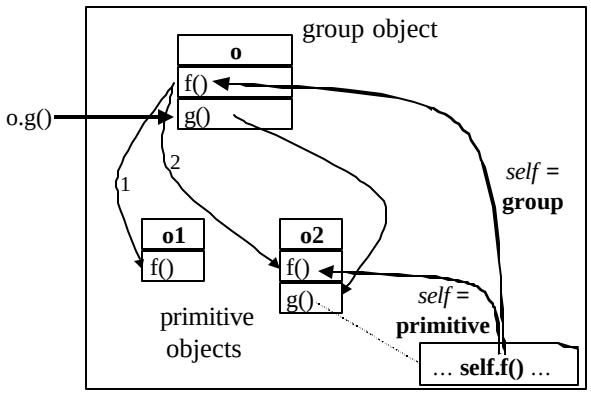

Figure 4. Group-to-Primitive Options "self = primitive" and "self = group"

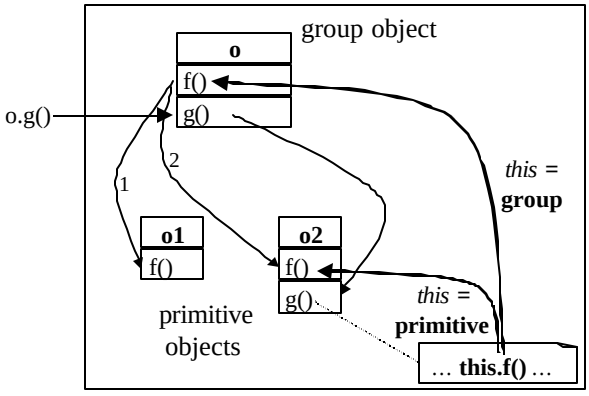

Figure 5. Automanipulation Options "this = primitive" and "this = group"

2.3 Relationships Induced By the Factors

We can make some general observations that reduce the resulting number of enumerable forms to 7, with what we believe to be less controversial rules first:

- Group-to-primitive forms of "no" or "*self*=primitive" render the automanipulation choice irrelevant, eliminating 24 of the 54 enumerable relationships.
- When the primitive-to-group behavior form is "identical", the automanipulation forms of "*this*=primitive" and "*this*=group" are equivalent. This rules out one of the remaining enumerable relationships.
- One of the most useful operational definitions of identity is that two objects have the same identity iff performing an operation on one of them always has the same result as performing the operation on the other. Of the remaining 29 enumerable relationships, this "identity rule" rules out the 10 in which "identity" disagrees with "primitive-to-group".
- Mixed "automanipulation" can be made only with respect to the coding of the body of an object's methods. This is an invasive, coding-dependent process that is probably better carried out by a developer making explicit use of *self* and adopting the *"this*=primitive" form of automanipulation. On the grounds of this fragility, we believe that manifest and other mixed automanipulation forms should be avoided and that the relationships should be deprecated. Of the 19 remaining enumerable relationships, "'mixed' deprecated" rules out 4.
- The variant form of primitive-to-group interaction can lead to a rather confusing collection of behaviors in which each member of the group has different behavior. We are left to wonder why this construct should be regarded as a group at all. An alternative would be to treat each varied behavior as a group of its own, with which the members can be associated on an "identical" footing. Of the 15 remaining enumerable relationships, "'variant' deprecated" encompasses 8.
- We have reservations about relationship 6 (maverick). The claim is that the object has the group's identity and has group behavior when called from primitive objects, but when called from the group its self-calls are not given group behavior. But we do not see a contradiction or a reasonable rule of meaning that prohibits the relationship.

Name for object's relationship to group	Identity	Primitive-to-group	Group-to-primitive	Automanipulation
1. Stand-alone	unequal	no	no	*(this=self=primitive)*
2. Associate	unequal	no	*self*=primitive	*(this=self=primitive)*
3. Aspect	unequal	no	*self*=group	*this*=primitive
4. Affiliate	equal		group	*this*=group
("mixed" deprecated)				mixed
("variant" deprecated)	unequal	variant	—	—
(Violates identity rule)	unequal	identical	—	—
equivalent to 5	equal	identical	*self*=group	*this*=primitive
5. Facet				*this*=group
("mixed" deprecated)				mixed
6. Maverick	equal	identical	*self*=primitive	*(this=self=primitive)*
7. Router	equal	identical	no	*(not used)*
("variant" deprecated)	equal	variant	—	—
(Violates identity rule)	equal	no	—	—

We could, of course, rule out any of these relationships for implementation convenience.

2.4 Objects in Multiple Relationships

We can also examine the question of what relationships an object can have simultaneously to two groups.

	1	2	3	4	5	6	7	Notes
1		Y	Y	Y	N	N	N	Stand-alone doesn't pass control to a group when called from outside
2		Y	Y	Y	Y	Y	Y	Associate, Aspect, Affiliate can coexist with being stand-alone or with being in a group.
3			Y	Y	Y	Y	Y	
4				Y	Y	Y	Y	
5					N	N	N	Can delegate to at most one group when called from outside
6						N	N	
7							N	

2.5 Higher-Order (Group-Group) Relationships

Allowing groups to be members of other groups introduces no new situations. For the nonce, call the group with groups as members a "supergroup", although we intend to observe that it is no different from any other group. Since all real method function lies in primitives, a supergroup never need use a group as an intermediary. With appropriate group-group communication to facilitate plan-sharing, a supergroup can directly employ the primitives. And with respect to primitive-to-group delegation, only the supergroup, with the complete plan, need be the delegation target.

There are two basic circumstances. First, if the member has identity unequal to the group identity, there is no primitive-to-group delegation to be accounted for. Second, if it has equal identity then the identities of all the primitives and groups contained must also be equal, since equality is transitive. A method call from the "outside" is delegated to the supergroup and from there to the primitive objects. Intermediate groups become routers. There is, after all, still only a single *this* and a single *self*.

3. CLASS RELATIONSHIPS

Not all fields and methods of a class belong to instances, and the above classification can apply independently to the static behavior, and their corresponding meaning must be phrased in terms of classes rather than instances:

- We are fortunate that Java provides no way to compare classes for identity[2]. Fortunate, because the fact that Java does not support class equality[3] of differently named classes, which causes great trouble for some Java-generation tools, means that we do not need to eliminate facets, mavericks, and routers as class relationships. But eliminating the "identity" column causes no coalescence of relationships because the "group-to-primitive" column preserves its distinctions.
- The variation of forms for automanipulation refers to the interpretation of *this* in method bodies. Static methods have no *this*, but the analogous meaning for classes applies to how calls on static methods defined by the class itself are handled. The "*this*=primitive" form is interpreted easily as leaving the calls to own class, which are always manifest, as calls to its own class. Likewise, the "*this*=group" form is interpreted by making them refer to the class appropriate to the self= form in use. This can be done by rewriting a copy of the body appropriate to each group from which the static method is called.
- The group-to-primitive forms for "*self*=" must also be reinterpreted without reference to a particular instance. This can be performed, as suggested above, by selecting the appropriate rewriting.

An important case of static behavior is creation. The Java *new* operation (not the constructors that become involved later, during initialization) is equivalent to a static method in the class being instantiated. Creation of an instance of a class may or may not be delegated to a group, which may then call for creation of its parts, including the original.

In general, the class composition form and the instance composition form can be independently selected, so there are actually 7^2=49 kinds of relationships. Of these, perhaps only 13 are of importance, those in which the class composition relation and the instance composition relationships are the same, and those in which the class composition relationship is "stand-alone". We will distinguish these two by prefixing the relationship name with "full-" or "partial-"[4]. When omitted, "partial-" is assumed.

The same constraints on multiple relationships among classes apply as those for instances, and for the same reason. But, for both instances and classes, these constraints can be interpreted either as prohibitions or as reinterpretations of composition operations.

4. COMPOSITION OPERATORS

Groups are created and modified by composition operators. Composition can be described in terms of two operators: ***compose***(relationship,details,group-class-name,object-class-name) and ***compose***(relationship,details,group,object). Both of these operators can produce Java class definitions, and the latter may produce objects and changes to objects as well.

In the discussion of "Objects in Multiple Relationships", certain relationship combinations were noted to be impermissible. That is, however, a static statement. There are two possible ways in which ***compose*** operators could respond to specifying an impermissible combination: the combination could be rejected, or the object could be cloned and the operation performed with respect to its clone. We call these two variant operators: ***compose-two-way*** and ***compose-one-way***.

4.1 Instance Composition and Temporal Instability of Identity – Cloning

The impermissible relationship combinations all arise from incompatible handling of primitive-to-group. And if variant primitive-to-group is forbidden, this is equivalent to the requirement on identity.

So performing a ***compose*** operation for an impermissible combination of instances runs afoul of the conventional idea that identity is an unchanging characteristic[5]. What difficulties can arise from permitting temporal instability in identity? A concrete example occurs if a standalone becomes a facet, router, or maverick of a group, or an object in one of those relationships becomes standalone. Comparisons of its identity with that of the group yield different values after the composition from what they yield before. But facts about the result of this identity test may be presumed externally, and already taken into account in a way that becomes meaningless. This phenomenon is one instance of what we have called "object schizophrenia". A common example of

[2] We are ignoring the library support for reflection. While reflection introduces objects that represent classes, methods, etc., the object is not the class, but only a representative of the class in the current execution. Two different class objects can represent the same class at the same time on two different machines and inequality of class objects is not the same as inequality of classes.

[3] A Java class is either a subclass of, a superclass of, or unrelated to any differently named Java class. Though useful, cycles are not permitted in inheritance graphs.

[4] except in the case of "stand-alone", where they are the same

[5] In fact, there are languages, like Irish, that have two entirely different verb forms for the "changeable is" and the "unchangeable is".

object schizophrenia arises in forming data structures representing sets of objects: no matter how many times an object is added to a set, it is present only once. But what if two objects are added and then they become facets of the same group? The presumed and proven invariant governing the set becomes violated after-the-fact.

However, if it can be assured the group contains at most one facet, maverick or router whose identity it takes, and that no prior remain to other of its facets, mavericks or routers, object-schizophrenia will not arise. Defining ***compose-two-way*** to throw an error after performing the composition is one way of permitting the composition to go ahead but requiring programmers to think about whether they can prove that the identity has not escaped in writing the *catch*. Another, more convenient, solution is to use ***compose-one-way***. The clone it creates is a new object without outstanding uses of its identity.

4.2 Class Composition and Cloning

The same conflicts, with the same potential solutions, arise for class composition as for instance composition, although from different grounds. Classes can always be referenced since their names are available to past and future Java programs with the proper access rights. This means that, except through careful program analysis, developers cannot assure that the exception arising from ***compose-two-way*** can be ignored. Note that this does not mean an object can not be a facet of two groups, only that the two groups must be merged into one larger group so that they are also identical.

5. IMPLEMENTATION NOTES

5.1 Class Composition

Multiple rewriting of a static method has significant cost. The cases in which additional rewritings are needed are noted by shading below in a collapsed version of the table above[6]. If unimplemented, only the 10 relationships: stand-alone, full-associate, associate, aspect[7], affiliate, facet[8], full-maverick, maverick, full-router, and router become available.

Object's relationship to group	Identity	Primitive-to-group	Group-to-primitive	Automanipulation
1. Stand-alone	unequal	no	no	*(this=self=primitive)*
2. Associate	unequal	no	*self*=primitive	*(this=self=primitive)*
3. Aspect	unequal	no	*self*=group	*this*=primitive
4. Affiliate	equal		group	*this*=group
5. Facet	equal	identical	*self*=group	*this*=group
6. Maverick	equal	identical	*self*=primitive	*(this=self=primitive)*
7. Router	equal	identical	no	*(not used)*

[6] "Aspect" with *self*=group, *this*=primitive is implementable without additional rewritings, unless explicit uses of "selfClass" occur in the body. But then, what's the point; it is the same as associate.

[7] AspectJ's "aspect" [4]

[8] full-facet is implemented by Hyper/J [5]

5.2 Instance Composition

Discussion of instance composition presumed that it is possible to treat the call of a method (whether in a group or in a primitive object) from a primitive instance from its call from a group. Discussion of instance composition also presumed that method calls to the group can be distinguished from calls to the primitive objects that are members. The simplest ways of making these distinctions are use two objects, to use two methods, or both. With two methods on two objects, all of the relationships presented can be supported, but without them, some choices are lost.

5.2.1 Instance Composition with a Single Method on Two Objects

The only way of distinguishing calls to an object from a group from calls to the object from outside primitives without coining an additional method is to prevent calls from the outside, managing to substitute the group's identity except in calls from the group. Only in the case of a stand-alone object can the primitive's identity be used outside, and that is because it is never invoked as a group member at all. This voids the columns dealing with identity and primitive-to-group forms, eliminates routers, and renders affiliates and mavericks redundant.

Object's relationship to group	Group-to-primitive	Automanipulation
1. Stand-alone	no	*(this=self=primitive)*
2. Associate ≡ 6. Maverick	*self*=primitive	*(this=self=primitive)*
3. Aspect	*self*=group	*this*=primitive
4. Affiliate		*this*=group
5. Facet ≡ 4. Affiliate		*this*=group
6. Maverick	*self*=primitive	*(this=self=primitive)*
7. Router	no	*(not used)*

5.2.2 Instance Composition with Two Methods on a Single Object

One way of distinguishing calls to objects from a group from calls to the object from outside the group is to use two sets of methods. Using a single object for both the group and its primitives rules out cases in which the identity test should yield "unequal", except in the case of stand-alone objects, which are not part of a group in any case. Despite the fact that coalescing the group object with the member cannot always be employed, it can be used to reduce overheads for facets, mavericks and routers.

Object's relationship to group	Identity	Primitive-to-group	Group-to-primitive	Automanipulation
1. Stand-alone	unequal	no	no	*(this=self=primitive)*
2. Associate	unequal	no	*self*= primitive	*(this=self=primitive)*
3. Aspect	unequal	no	*self*=group	*this*= primitive
4. Affiliate				*this*=group
5. Facet	equal	identical	*self*= group	*this*=group
6. Maverick	equal	identical	*self*= primitive	*(this=self=primitive)*
7. Router	equal	identical	no	*(not used)*

6. RELATED WORK

6.1 Composition Filters

The concept of wrappers and, in particular, wrappers for objects, has long application in software development. Composition filters [1] extend the object-wrapper concept to a group-wrapper. The group embodies dispatch strategies based on its *state* – a set of predicates about the objects in the group. In the classification given above, composition filters are groups. The filtered objects are aspects or full aspects. With composition filters, group behavior is obtained only by directing messages to the group. It has the *compose-two-way* variant of the instance composition operator.

6.2 Subject-Oriented Programming

Subject-Oriented programming [3] introduced the notion that objects in a group can have the same identity and that creation of an instance of one of the member classes causes creation of the group. The member is a class facet, although the creation need not be delegated to all members. As discussed above, this implies that the group should handle the messages directed to its members. In SOP, the subjects are all full facets.

6.3 Objects in Groups

Doug Lea has written a survey on objects in groups [7] recapping prior work. He also presents an alternative delivery model relying on *channels* rather than on object identity to describe the target for the message. The introduction of channels does not change the basic form for the analysis presented above, but does allow for many more mixed or intermediate cases in the analysis.

6.4 Aspect-Oriented Programming

Aspect-Oriented Programming [6] retained the concept that creation of an instance of one of the member classes causes creation of the group. But it does so for only one of the member classes, called the base. Other member classes are treated like members of composition filters. In AOP, the base and the aspects have different relationships to the group. The base is a full facet but the aspects are full aspects. It has the *compose-two-way* class composition operator. AspectJ [5] provides a realization of AOP in which the group and the facet are coalesced into a single object.

6.5 Hyper/J

Hyper/J [9] is a realization of MDSOC [13], an evolution from SOP. It has both the *compose-two-way* and the *compose-one-way* variants of the class composition operator. In Hyper/J, the group and all the facets are coalesced into a single object

6.6 Compound References

Ostermann and Mezini [10] identified a number of separate composition properties, subsets of which are usually bundled together to form composition mechanisms like inheritance and delegation. They showed that use of more powerfully interpreted references, called *compound references*, allows flexible combination of these properties and provides important semantic options not traditionally available. While shifting discussion of dispatch from objects to generalized references provides an important alternative to group formation, this paper deals with solutions within the more traditional view of object identity and reference.

7. REFERENCES

[1] Aksit M., Bergmans L., Vural S. An object-oriented language-database integration model: the composition-filters approach. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 1992

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*. Addison-Wesley, 1995

[3] Harrison, W., and Ossher, H. Subject-Oriented Programming - A Critique of Pure Objects. In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*. September 1993

[4] Helm, R., Holland, I., and Gangopadhyay, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, (Vancouver), ACM, October 1990

[5] Kiczales, G. Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*. Finland, 1997

[6] Kiczales, G. Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J-M. Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Finland, 1997. Invited presentation.

[7] Lea D., *Objects in Groups*, December, 1993, http://gee.cs.oswego.edu/dl/groups/groups.html

[8] Ossher, H. and Harrison, W., "Support for Change in RPDE^3", Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments, pp. 218-228, Irvine CA, December 1990

[9] Ossher, H. and Tarr, P. "Multi-dimensional separation of concerns and the Hyperspace approach." In *Software Architectures and Component Technology* (M. Aksit, ed.), 293–323, Kluwer, 2002.

[10] Ostermann, K., Mezini, M. Object-Oriented Composition Untangled. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications*. October 2001.

[11] Reiss, S., "Connecting Tools Using Message Passing in the Field Environment", IEEE Software, pp. 57-66, July 1990.

[12] Sullivan, K.J. and Notkin, D., ``Reconciling Environment Integration and Software Evolution," *ACM Transactions on Software Engineering and Methodology* 1(3), pp. 229-268, July, 1992.

[13] Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M., "*N* degrees of separation: Multi-dimensional separation of concerns." In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, 107–119, IEEE, May 1999.

[14] Web site, "Subject-Oriented Programming and Design Patterns," http://www.research.ibm.com/sop/sopcpats.htm