RC22685 (W0212-147) December 30, 2002
Computer Science

IBM Research Report

Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition

William H. Harrison, Harold L. Ossher, Peri L. Tarr
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition

William Harrison
IBM T. J. Watson Research
P.O. Box 704
Yorktown Heights, NY 10598
1-914-784-7339
harrisn@watson.ibm.com

Harold Ossher
IBM T. J. Watson Research
P.O. Box 704
Yorktown Heights, NY 10598
1-914-784-7975
ossher@watson.ibm.com

Peri Tarr
IBM T. J. Watson Research
P.O. Box 704
Yorktown Heights, NY 10598
1-914-784-7278
tarr@watson.ibm.com

ABSTRACT
Composition as an approach to software construction has been of interest since at least the beginning of work on Module Interconnection Languages in the late 1970's. Most recently, research in aspect-oriented software development (AOSD) has exploited composition approaches that provide more flexible extension, adaptation and integration of components. A critical issue in composition is *symmetry* versus *asymmetry.* Most AOSD approaches have used an asymmetric paradigm, in which "aspects" are composed (woven) into components that implement a "base" model; aspects and components are different, and component-component, aspect-aspect, and class-class composition are not supported. A symmetric paradigm, on the other hand, makes no distinction between components and aspects, and does not mandate a distinguished base model. The choice of symmetric versus asymmetric paradigm greatly affects the set of good software engineering properties promoted, and the set of software engineering activities facilitated. This paper analyzes the ramifications of the use of symmetric and asymmetric paradigms.

General Terms
Composition, integration and evolution

Keywords
Software composition, separation of concerns, aspect-oriented software development, symmetric composition, asymmetric composition, modularization, evolution, reuse, software integration, component-based software engineering

1. INTRODUCTION
The appropriate use of the compositional paradigm has been demonstrated to offer a number of potential benefits, including improved separation of concerns (modularization), reduced impact of change, increased adaptability and evolvability, especially in directions that were not fully anticipated, better alignment of artifacts, and hence traceability, and improved reuse and integration.

As with any paradigm, different approaches to composition enhance or reduce different software engineering benefits, and no single approach is good for everything. It is therefore critical for software engineers to understand the key differences among classes of approaches and the cost-benefit tradeoffs those differences entail for their particular tasks, goals and contexts. Given the expanding interest in AOSD, it is timely and important to examine the tradeoffs involved in the variety of different composition approaches in this area.

As a step in this direction, this paper identifies what may be the most critical differentiator between the composition approaches: the question of symmetry versus asymmetry. The composition approaches used in viewpoints [16], subject-oriented programming [8], multi-dimensional separation of concerns (MDSOC) [21] and dynamic view connectors [9] employ a *symmetric* paradigm, in which all components are treated as first-class, co-equal building-blocks of identical structure, and in which no component's object model is more basic than any other's. Most other approaches in the AOSD community, such as AspectJ [11] [12], use an *asymmetric* paradigm, in which "aspects" are composed (woven) into "components" that implement a base model. Aspects and components are different in structure, and component-component composition is not supported; aspect-aspect composition is not universally supported. Some approaches, such as adaptive programming [13], have evolved from asymmetric to symmetric, and others, like composition filters [1] [3], are in between. We present a critical evaluation of symmetric versus asymmetric paradigms when used in some key software engineering scenarios, based on how they promote or hinder achieving some key software engineering properties in software produced by those paradigms.

It is not the goal of this paper to be comprehensive—doing so would require several papers and much additional research and evaluation—nor to promote any particular approach to composition. Our primary purpose is to raise a critical issue for researchers in, and consumers of, compositional software engineering, and to analyze its ramifications. Similarly, the parameters selected for evaluation are also not intended to be comprehensive, but rather, to reflect a set of key issues that are of importance to software engineers, and we leave the identification and analysis of other issues to future work.

Section 2 describes symmetric versus asymmetric compositional paradigms, identifying three separate kinds of symmetry. Section 3 discusses two concrete examples, to illustrate some of the differences and tradeoffs between symmetrical and asymmetrical approaches. Section 4 then analyzes the tradeoffs in the contexts of some common AOSD scenarios. Section 5 summarizes the conclusions of the analysis and Section 6 discusses related work.

2. SYMMETRICALLY AND ASYMMETRICALLY ORGANIZED PARADIGMS
A composition paradigm involves three kinds of entities: composable elements, join points and composition relationships. The *composable elements* are the things being composed, such as concerns, components or aspects. *Join points* to refer to those

points within the composable elements at which composition can occur, such as functions in an interface or members of a class [11][17]. *Composition relationships* (e.g., *rules* [8] or *advice declarations* [11]) specify the details of how composition is to occur at join points.

The three kinds of entities lead naturally to three kinds of symmetry, discussed in the rest of this section.

2.1 Element Symmetry

Paradigms with element symmetry employ a single kind of composable element. For example, subjects [8] and hyperslices [21] are composable elements that consist of declaratively complete class hierarchies; their definitions stand alone.

Paradigms with element asymmetry use two (or more) kinds of composable elements. As introduced in [12], the term "component" denotes elements whose descriptions stand alone, while the term "aspect" denotes elements that are described in relation to other components or aspects. Such paradigms support component-aspect composition, but not component-component composition. A subsidiary division is based on whether or not they support aspect-aspect composition. However, if aspect-aspect composition is fully supported, it is always possible to start with a null component and build systems entirely by composition of aspects. Since this is equivalent to symmetric composition, we assume henceforth that aspect-aspect composition is not permitted in paradigms with element asymmetry.

In this paper, we will use the AOP terminology of "component" and "aspect" and use the term "concern" as the inclusive term for either. Symmetric composition will be said to consist just of components.

2.2 Join-Point Symmetry

Join points occur within concerns.[1] Individual join points themselves may be intrinsically symmetric or asymmetric. A symmetric join point is a point in an artifact or execution, such as a method definition or call, at which composition can occur. Such composition combines the points in some way, as described by the composition relationships, such as to call or execute multiple methods. Most of the work in this field, and this paper, confine themselves to method granularity, taking this loosely to include regions that could be extracted to form methods. Joining at arbitrary points embedded in a context within other method code cannot be supported symmetrically.

An asymmetric join point occurs when a piece of an artifact, such as a method body, contains one or more explicit points at which references to (e.g., invocations of) "the other concerns" occur, or when the expression of the concern is tangled with others and cannot be extracted, and thus, can only have elements composed with them in their own context. Examples of the former are Lisp's "call-next-method" [10] and AspectJ's "proceed" [11]; an example of the latter is a Java exception handler. In compositional engineering, it provides an alternative to exposing structure within methods and allowing join points at arbitrary points within code.

2.3 Relationship Symmetry

There are two symmetry issues concerning relationships: scope and placement. In paradigms with both relationship and element asymmetry, relationships are placed only in aspects, and not in components. The relationships have a binary scope, referring implicitly to the aspect containing them and explicitly to the component with which it is being composed. In paradigms with relationship symmetry, the relationships may be placed anywhere, but they have a scope that ranges across all the concerns being composed. This applies whether there is element symmetry or not. It is also possible to have element symmetry and yet have relationship asymmetry, in which some components contain relationships specifying how they are to be composed with other components. From the point of view of this analysis, we consider the addition of such relationships to the components to render them aspects unless they are simply regarded as "suggestions" appropriate when used in particular circumstances.

Relationship asymmetry therefore has the limitation that relationships can be only binary, relating an aspect to the component into which it is to be composed (or perhaps, as a shorthand, to multiple components into which it is to be composed uniformly). A relaxation of this definition allows relationships also to refer to other aspects, to control the interaction of multiple aspects composed with the same component. This "partial" relationship symmetry has asymmetric placement, but symmetric scope with respect to aspects.

3. REAL-WORLD EXAMPLES

Before giving a detailed analysis of the benefits and drawbacks of symmetric versus asymmetric organizations, we discuss two examples that illustrate some of them. Both examples involve separating features, one an extension development scenario and the other a parallel/cooperative development scenario.

3.1 Refactoring to Separate Features

The experiment on separating features in source code by Murphy, et. al. [15] provides an interesting illustration of the respective benefits and disadvantages of different approaches. In their first example, they discuss a method called RETokenStart.match in the gnu.regexp package, in which code for several special kinds of matching (multiline, "not beginning of line" and anchored) and basic matching are tangled within the method. They discuss different ways of separating these special matching concerns from one another and from basic matching. One solution they present involves refactoring the method to put the different kinds of matching into separate methods within the RETokenStart class, and then using Hyper/J to compose the matching methods. A second solution they present involves refactoring the method to put the special kinds of matching in separate aspects, and then using AspectJ to weave the aspects into the base method.

In their analysis, they point out that their Hyper/J solution reduces cohesion of the base structure (by introducing additional methods that are not even directly called, but are needed for composition) and achieves limited separation (because the methods for the special matches remain in the RETokenStart class)[2], but promotes

[1] In AspectJ [11], join points are points in the execution of the component, and aspects do not have join points as such, but rather specify advice code to be executed at join points. This is equivalent to saying that the advice code blocks are implicit join points, and the rest of the advice declarations specify how they are to be composed with join points in the component. We take the latter view, because it facilitates uniform discussion of symmetric and asymmetric approaches.

[2] This point is subject to debate over whether a different separation approach should have been used.

locality (for the same reason). Their AspectJ solution, on the other hand, maintains cohesion of base and achieves full separation, at the expense of locality (because the methods for the special matches are in separate aspects). When asked why a Hyper/J design with greater separation, similar to the AspectJ design, was not used, Murphy responded that she wanted to maintain the locality. Clearly, different developers will have different relative priorities for obtaining the cohesion, separation and locality properties, and it is not generally possible to obtain all of them simultaneously, which strongly suggests that developers must be in control over the choice.

3.2 Cooperative Development of Features

SAGE is a research prototype developed for performing translation of messages between different formats [22]. Without going into a description of SAGE itself, we can draw on experience with its development as a small real example of the use of composition in cooperative development. SAGE's design is expressed as a common information model for the representation of declarative information about messages, and several features that share this model to provide the message translation system's function: dictionary management, context management, message definition parsers, UI's, and code generation. A symmetric methodology for using UML was employed for SAGE's design, in which more than one concern (common model or design feature) could describe the same element (entity or association). Although various researchers in the group created UML models when developing new concerns, all felt free to add to or change the models used in other concerns when that was appropriate, and there were ongoing discussions about how best to model concepts. In effect, the concerns were preserved as logical constructs rather than having each concern "belong" to one or another team member. The UML models for the concerns were converted to an equivalent Java implementation with a model-composition/Java-generation tool called Tengger [23]. Each concern had a package of Java classes where the operations defined in the UML model are implemented. The Java classes in the concerns are all composed using Hyper/J [18] to produce the final SAGE system.

We retrospectively analyzed the SAGE design to see to what extent it could fit an asymmetric model. The common information model is a natural base component, and we expected that each feature might be represented as an aspect of it. To explore this, we first had to remove the inherently symmetric approach of multiple concerns describing the same entity by conceptually declaring each entity to "belong" to a single concern, and be imported into other concerns as needed. We then examined concern-concern interactions, and found five ways in which entities defined in a concern are augmented by other concerns that import them. From least to greatest "impact" they are:

1. the feature adds subclasses to imported entities (noted as < in Table 1),
2. the feature adds associations from some of its entities to imported entities (R),
3. the feature adds methods or attributes to imported entitites (M),
4. the feature generalizes (adds superclasses) to imported entities (>), which often implies (3), or
5. the feature adds inheritance relationships between imported entities (I).

The actual interactions in the SAGE design are summarized in Table 1. The interactions below the shaded diagonal represent cases where features later in the table import elements from concerns earlier in the table. If the portion of the table above the diagonal were empty, we would have a sequence of concerns, each building upon prior ones. This would fit an asymmetric organization in which each feature is an aspect, and the aspects are applied to the common model. However, even though the features have been ordered so as to place as many interactions as possible below the diagonal, the upper right portion is not empty. This reflects the fact that there are symmetries inherent in SAGE's design.

Table 1. Concern Interactions in SAGE

depends on: / dependent:	Common model	Contexts feature	Dictionary feature	Surfaces feature	UI features	Generation feature
Common model			R			
Contexts feature	>, M, R		>, M			
Dictionary feature	>, M, R	>, M, R				
Surfaces feature			I, R, <			
UI features	>, M, R		<			
Generation feature	>, M			M		

The simplest symmetry is denoted by the R in the first row: while the dictionary feature is heavily defined in terms of entities that belong to the common model, the common model defines an association between one of its own entities (DenseRange) and one that belongs to the dictionary feature (NamedValueDictionary). This symmetry could be removed: merely asserting that NamedValueDictionary belongs to the common model instead of the dictionary removes the "R" from the matrix. But this argument is facile; it hides the fact that in an asymmetric model, the component should be an "element whose descriptions stand alone," as discussed in Section 2.1. We would either have a component that cannot stand alone, but must be augmented by an aspect to be functionally complete, or have to move a good deal of the dictionary feature into the common model, defeating the separation.

The other symmetry arises from the fact that, while the dictionary feature refers to and generalizes an entity defined in the context feature (BusinessContext), the context feature generalizes an entity defined in the dictionary feature (MessageSet). Having a feature generalize entities that belong to other features is a common phenomenon that often arises from one of two causes. The first cause is that the feature defines an association from one of its entities that may link to any one of several entities. This requires forming a type that characterizes the operations actually expected to be common to those entities in this union, so that they can be called on any entity reached by traversing the association. For example, the function and behavior of the context feature had no need for BusinessContexts to have names. But the design of the dictionary feature called for a dictionary of BusinessContexts, requiring them to be a specialization of its entity called IdentifiableEntity, which provides names. The second cause is

that the feature has a view that simply demands a new generalization. For example, the Code Generation feature considers several unrelated entities in the common model to all be CodeGenParts, with the corresponding operations and state needed for that behavior. It is harder here to argue that the assignment of entities to features is in error. Neither aspect is really closer to the base component than the other, and, in fact, the order of the two rows/columns would need to be switched to yield the actual historical order of their development. A paradigm with relationship asymmetry does not cover this case.

In general, all symmetries can, with effort, be converted into asymmetries by imagining or declaring that some elements of an aspect belonged in the component. However, the issue is not whether the transformation is possible *post-facto*, but whether it clarifies the design, and whether the desired separation can be maintained as more and more material is moved into the component.

The "asymmetric" interactions in this example, below the diagonal, nonetheless reveal an interesting point about relationship symmetry. The interactions in the first column are aspect-component interactions. All the others are aspect-aspect interactions, which are essential in situations like this where one aspect can build on another. The fact that most rows have entries in the first column as well another column shows that it is necessary for composition relationships to refer to join points in the component and in other aspects together (or, perhaps, to join points that result after weaving other aspects into a component). At least a degree of relationship symmetry is therefore needed even in these cases.

4. EVALUATION

In this section, we compare symmetric and asymmetric composition paradigms against a set of evaluation criteria. These criteria fall roughly into 5 categories: Creation, Understanding, Separation, Reuse and Scaling. In the following sections, we use the first three of these criteria to evaluate the suitability of asymmetrically and symmetrically organized paradigms in three common AOSD usage scenarios: extension, parallel/cooperative development, and integration. We chose these scenarios to be illustrative and broadly representative, but they are *not* intended to be (and cannot be) exhaustive. Analysis of reuse and scaling is largely independent of usage scenario, so these are discussed later. The results of the analysis are summarized in Table 2.

4.1 Creation and Extension Scenarios

4.1.1 Creation (Writing or Extraction)

Concern creation takes place either directly, *ab-initio* or by reuse and modification, or by extracting a concern from an existing body of software.

4.1.1.1 Creation of Co-Evolving Concerns

Asymmetric paradigms actively promote the representation and encapsulation of crosscutting concerns that are to evolve, and be reused as a unit, with the underlying component representation (e.g., class hierarchy). The intent here is that the component dictates the primary structure, and the aspects fit into it, rather than having a structure of their own. This allows convenient separation of fragments that make up crosscutting or subsidiary concerns without the complications of different concerns dictating their own structures. Symmetric paradigms have to address these complications by being able to reconcile the different structures of concerns being composed.

In extension scenarios, the simplicity of expression of both asymmetric join points and asymmetric composition relationships may also be an advantage. In addition, placing the relationships at an aspect's join point increases the cohesion of the aspect as it is written.

The simplicity enforced by the asymmetric paradigm is thus an important advantage in this scenario. It is important to note, however, that suitable tooling or syntactic sugar can provide asymmetric usage features on a symmetric paradigm; a symmetric paradigm thus does not require that all *uses* of it be symmetric. When there is natural asymmetry, as in extension scenarios, it can be represented and exploited even in symmetric paradigms.

4.1.1.2 Concern Creation for Reuse

With known approaches to software engineering, there is an intrinsic trade-off between initial development cost and evolution cost. One common example of this trade-off is illustrated by the development of closed components vs. the development of OO frameworks. A framework developer framework attempts to foresee the behaviors that would be modified to extend the function of the framework and to encapsulate them appropriately. In effect, the developer is identifying anticipated join points or "open points." This anticipation raises the cost of initial development, but reduces the cost of development of extended components. From a non-reuse standpoint they seem to be "unnatural" methods[3].

The need for reuse reduces the effectiveness of using an asymmetrically-organized paradigm. Element asymmetry means that the elements that are aspects cannot themselves be augmented by other aspects. This implies that the designers may need to embed functionality into the base component to make it available for subsequent reuse. Relationship asymmetry is especially inimical to reuse, because relationships in aspects stating how those aspects are to be composed preclude their being composed differently as new, reuse contexts arise.

The same phenomenon takes place with concern extraction. Extraction to produce a delta from a known base naturally produces an aspect. But the aspect no longer forms a reusable element until it is actually composed into the base. And even then, additional effort is needed to convert the aspect into a useful concern that anticipates interesting extensions.

4.1.1.3 Creation for Comprehension

A concern is sometimes created for purposes of comprehension or documentation, to help developers understand and comprehend the concern without having to follow irrelevant material reflecting other concerns. For these purposes, the simplicity of expression and cohesion advantages shown by asymmetrically-organized paradigms for creating one-time use software come to the fore again.

4.1.2 Understanding

Understanding is the crucial enabler for software maintenance and evolution. In working with software-by-composition, we can separately treat the understanding of separate concerns from the understanding of their joint behavior. Murphy, et. al. [15] use the two measures of *cohesion* and *locality* to characterize

[3] As described in the "Comparison" subsection of section 4.2 in the Murphy paper [15].

understandability[4]. Cohesion is the likelihood that two pieces of information appearing together are actually relevant to one another, while locality refers to the likelihood that two pieces of information relevant to one another will actually appear together.

4.1.2.1 Separate Understanding

Paradigms with relational asymmetry bundle the join points and relationship specifications of an aspect together. This increases the locality of the aspect considered separately while reducing its cohesion somewhat when taken as a separate element. Using a symmetric paradigm's capability to move the specification of how the extension fits into the base elsewhere can raise the cohesion of an extension. Doing this requires that the extension not refer directly to the base; instead, interfaces required by the extension and structural relationships that are important to it must be part of the extension. This promotes locality, by increasing the extent to which the information needed to understand the extension is present within the extension. Locality might not be perfect, however, because an understanding of patterns of interaction within the base might be necessary to full understanding of the extension. This depends on the nature of the extension, including the extent to which it itself forms a significant body of logic, versus a set of small additions to the base's functionality.

If an extension's join points are not intrinsically asymmetric, locality of separate understanding is also reduced by using an asymmetrically-organized paradigm. Use of an asymmetric join point when not necessary forces the reader to try to understand why the execution of "the other" needs to occur where it does, when the choice was actually random.

4.1.2.2 Joint Understanding

While asymmetry decreases the cohesion of an extension's design when considered separately, it increases the cohesion of the joint design of the extension as it relates to the base component because the relationship specifications are a more crucial part of the joint design.

If an extension's join points are intrinsically asymmetric, both symmetrically and asymmetrically organized paradigms have the same cohesion and locality characteristics for the joint design. This is because the invocation of "the other" forces a nested composition relationship and the placement is evident in the code for the extension.

4.1.3 Concern Separation [decoupling]

Separation of concerns is, ultimately, the *raison d'être* for AOSD. In the abstract, relationship or join-point asymmetry implies an incomplete separation: the aspect is specified in terms of the base and its expression therefore depends on the lower-level decisions made for the base. In fact, the extension scenarios share this basic asymmetry, as an extension is generally created with respect to a base. In practice, even with relationship symmetry, complete separation is usually possible only in cases where there is no interdependence.

In addition to its impact on understandability, separation can have an impact on concern analyzability, testing, and evolution.

4.1.3.1 Analyzability

OO software development trades analyzability for increased reusability, and compositional software development does so as well. For purposes of analyzing a component, exactly the same issues apply, but one of the usual points-of-entry used by analysis approaches becomes less useful. An extension can alter the analysis characteristics of a class in the same ways that a subclass can, and can even access and alter variables that are private to the class. So just as an analysis might need to be conservative because an object may be subclassed, it must also be conservative because it may be composed. One point-of-entry often used in program analysis is creation of object instances, because at that point, the actual class of the object being created is known. But modern programming practice suggests the use of a factory pattern for the creation of objects, and this results in the loss of those points of entry. The same is true of composition. The class being instantiated is not limited to the characteristics it has in the base, but may have additional characteristics acquired in the composition.

But while analysis of a component is still useful in AOSD, analysis of an aspect is generally unworkable. The aspect code is fragmentary and partial, and the interconnecting relationships and logic are missing. The analyzer does not know which parts of an aspect will be pulled together by the base's inheritance hierarchy or which other parts will be executed since the calls to them are not direct within the aspect but are side-effects of calls into the base. In this sense, a symmetrically organized paradigm is technically at an advantage. But this technical advantage may be less than it first appears, being actualized only for large extensions. For very small extensions, the fact that all of the interesting flow connections are missing places analysis of symmetrically organized concerns on the same footing as asymmetric ones. But when the concern is large and its join points few, the analysis of an extension concern in a paradigm with relationship symmetry can yield as much useful information as the analysis of a base.

Less conservative estimates of possible interaction are possible if the analysis could rule out hidden "cross-talk" between the concerns, as is being done in [20]. Developing guidelines and exploiting their enforcement in analysis tools is still a relatively unexplored area of AOSD.

4.1.3.2 Testing

Testing encompasses both the derivation of test cases for a body of software, and their execution and evaluation. The derivation of test cases usually relies on the external specifications or on the code body. Test case generation from external specifications should be insensitive to the composition paradigm. Test case generation from the code body generally depends on analysis of the code, whose characteristics were discussed earlier. Test cases generated from pre-composition code reduce the delay in constructing a test suite and the burden of doing so, although they may be overly redundant because of the weakness of the program analysis. Test execution is another matter. While the current state of the art is too weak to consider the execution of pre-composition tests to be adequate, this does not imply that pre-composition testing is valueless. Like unit testing, it can reduce the cost and burden of the later integration tests. The relative merits of symmetrically and asymmetrically organized paradigms for testing are similar in direction to those for analysis, although smaller in magnitude.

4.1.3.3 Impact of Change

As the base component of a system evolves, its evolution may have an impact on the extensions that have been added to it. The

[4] Their third measure, *separation*, is treated separately because it has an impact on many issues as well as on understanding.

most common forms of evolution are the creation of new classes and subclasses, the refactoring of class hierarchies, the creation of new methods or the refactoring of methods. Even though the advent of AOSD can be expected to reduce the need for evolution of components by allowing new functionality to be built as extensions, base changes will still take place.

These sorts of changes to a base most visibly affect the composition relationships. With asymmetric relationships, a change to the base must be reflected in all of the aspects that modify it, while symmetric relationships lie outside the elements themselves, making them potentially more accessible and easier to change. More importantly, symmetric relationships, by nature, have to deal with connecting join points in different concerns, and thus with such issues as reconciling different class hierarchies and interfaces. These capabilities facilitate handling an evolving base. Asymmetric relationships specify attachment to join points in a base component, leaving the base to dictate the structure. They therefore usually do not deal with hierarchy and interface mismatches, and so are not well equipped to deal with an evolving base. In cases where the base and extension have been separately acquired, source code may be unavailable, making the upgrade unworkable in paradigms with relationship asymmetry.

Less visibly, but of potentially greater impact, is that the use of asymmetric elements encourages exploitation of the base's structure. If for example, an aspect defines a class containing several methods that go together into extensions of a base, but also defines what is, conceptually, a subclass with specialized behavior, it depends on the subclassing relationships in the base to carry out its own inheritance. As a result, if the base evolves so that the methods are no longer inherited into the expected subclasses, inappropriate behavior takes place.

It is possible to separately evolve an aspect on top of an unchanging base. In this situation, there is advantage to packaging the composition relationships with an aspect. A change can be incorporated simply by installing a new extension. This advantage can be realized equally well within a symmetrically organized paradigm that allows composition relations for various bases to be packaged with the concerns implementing the extensions.

4.2 Parallel/Cooperative Development

4.2.1 Creation (Writing or Extraction)

The use of AOSD to facilitate parallel and/or cooperative development tends to move away from circumstances where extensions all cluster around a common pre-specified base. As the number of mutually composable elements rises, multipart composition relationships are needed. Coping with the fact that different concerns may be based on slightly different versions of each other is facilitated by removing the composition relationships from being intimately intertwined with concern artifacts.

Software written for flexible composition has many of the characteristics of software being developed for reuse. Developers put some advance effort into thinking about the major functional divisions of their algorithms and the points at which other components might be expected to be joined.

4.2.2 Understanding

4.2.2.1 Separate Understanding

The same analysis applies here as in Section 4.1.2.1.

4.2.2.2 Joint Understanding

In discussing the extension scenarios, we observed that an asymmetrically organized paradigm increases the cohesion of the joint design. For the parallel/cooperative scenarios, however, the joint understanding is not just with the single base because there may be a cascade or a set of related code bodies to examine. Under that circumstance, it becomes more important to have a simple way of identifying and gathering the composed code together. So, for this class of scenarios, symmetrically organized paradigms provide for better understanding of the joint behavior.

4.2.3 Separation [decoupling]

As discussed in Section 4.1.3, extension scenarios have a natural asymmetry – the extension vs. the base. In our original formulation of subject-oriented software design [8], we looked toward a more complete separation to suit cooperative development and integration scenarios as well.

In MDSOC, a symmetric paradigm, each concern provides a complete, even if partially implemented, design of the classes and the methods involved in its coding and in its interaction with other concerns [21]. Each concern is designed without reference to a base, and each is completely separated from the others. This enables the developers working with each part of an application to focus on a hierarchy of abstraction appropriate for it, without the clutter of inheritance structures needed for other parts.

Not all concerns separate so clearly, which is why the space of scenarios really is a spectrum. But developers of software for composition are mindful of trying to avoid tangling many concerns in one class or method and of exposing the proper join points for cooperative use. As with framework design, what could have been written as in-line code in a method is exposed as a separate method called where needed, to expose it as a join point.

Separation of concerns demands that developers of a concern avoid dependence on the structure of other concerns. For example, awareness that there are two different calls to a class's method from within another concern is a structural dependence that should be avoided. Wrapping another method with an "around" also presumes the knowledge that the other method doesn't need to be around this one instead. So in evaluating the symmetric vs. asymmetric paradigms, we must recognize that although it is possible to use asymmetric constructs in a symmetric organization, it is desirable to avoid them and that in this central group of scenarios we should evaluate them from the point of view of their ability to achieve further separation.

4.2.3.1 Impact of Change

The advantages discussed for a symmetric paradigm's management of impact of change become even more compelling in these scenarios. Where previously the centralization of composition relations helped point to potentially related parts of extension aspects, the cooperative scenarios exploit concern-concern interactions like those in the SAGE example discussed above much more heavily and the composition relationships help manage the relatively independently evolution of the concerns. The case for evolution of extensions loses its separateness and is evaluated in the same way.

4.3 Integration Scenarios

The use of AOSD to facilitate integration reflects the need to integrate separately developed software. It is the far point in the spectrum of composition scenarios and cannot be handled at all by an asymmetrically organized paradigm because each of the

products is a full component – there are no aspects. Tight integration is seldom easy, but loose integration is often possible by using some of the method calls in each component to trigger code in a glue component that then invokes necessary operations in the other. But even this scenario – two components and a glue concern, requires a symmetrically organized composition paradigm. It is not possible to view either of the original components as modifying the other.

4.4 Scenario Independent Issues

4.4.1 Reuse

While the merits of the two paradigms have been discussed with respect of the creation of software for reuse, their suitability for the actual reuse of software concerns bears additional comment. Is a reusable concern to be built as a component, or as an aspect? Building it as a component would mean that two such could not be combined in an asymmetric paradigm, but building it as an aspect would mean building it to apply only to a specific base, or, in the case of relationship symmetry, to a broader class of bases, but not to other aspects. The fact that neither choice is suitable indicates that only a symmetrically organized paradigm is suitable as a vehicle for promoting a reusable components industry, and relationship symmetry is essential to any kind of reuse.

4.4.2 Scaling

This same issue of suitability as a component model for large-scale use and reuse shows up when looking at the scalability of the paradigms. In software construction by composition, it must be possible to take several parts and compose them to produce a new part. With asymmetric models, one cannot talk about a part in general, only about either the single base component or about an aspect. If, for simplicity, we abandoned the idea of a base component and focused on the composition of aspects to produce aspects, we would have a uniform part model. But it would be one in which the universe of parts is partitioned according to the base component to which they are applied. And if we then resolved that problem by edict that there be a common null base the result would be a paradigm that was only vacuously asymmetric – it would be a symmetric paradigm in the end.

On the other hand, within the context of the extension scenario, paradigms with relationship asymmetry have an advantage: since their composition relationships are simpler, not having to be concerned about structural mismatches, the composition is simpler and scales better.

Table 2 summarizes the evaluation presented above. Each cell indicates whether the evaluation suggested better handling by an asymmetrically organized paradigm (A), a symmetrically organized one (S), or neither (-). Three evaluations are shown for each case: for element symmetry, for join-point symmetry, and for relationship symmetry.

Table 2. Element/Join-Point/Relationship Advantages Summarized

advantage in / advantage for	Creation/ Extension	Parallel/ Cooperative	Integration
Creation		SSS	SSS
Co-evolving concerns	AAA		
For Re-use	SS-S		
For Comprehension	-AA		
Understanding			
Separately			SSS
Cohesion	-ssSS		
Locality	aaaAAA	-SS	
Jointly		AAA	
Cohesion	AAA		
Locality	---	S-S	
Separation [de-coupling]		SSS	
Analyzability	S--s--		SSS
Testing	S--s--	S--	
Base Evolution	S--SS	S--	
Extension Evolution	---	SSS	
Reuse	SSS		
Scaling	SSSAAA		
		SSS	

5. CONCLUSIONS AND FUTURE WORK

The use of any compositional paradigm has a significant impact on a developer's ability to achieve various desirable software engineering properties. It is critical for both developers and AOSD formalism developers to understand how key differences among AOSD approaches affect particular a approach's ability to promote or hinder the attainment of these properties for particular tasks, goals, or contexts.

The issue of symmetry vs. asymmetry in AOSD approaches may be the most critical differentiator among compositional approaches. Analysis of the use of asymmetrically and symmetrically organized paradigms for software composition across a wide spectrum of usage scenarios indicates that:

1. Asymmetric paradigms are superior for co-evolving crosscutting or subsidiary concerns.
2. Both paradigms have some advantages in simple extension scenarios, but symmetric paradigms are at least as good as asymmetric paradigms in those scenarios.
3. The relative advantages of symmetrically organized paradigms increase with the independence of the development efforts in multi-extension and multi-component software, until, in the limiting case of software integration, only symmetrically organized paradigms are suitable.
4. Only symmetrically organized paradigms are suitable as the basis of a reusable component model for software construction by composition.

While symmetry vs. asymmetry is a critical issue, it is not the only one (see, for example, [7]). For the future, other important issues must also be identified and analyzed. Only when these issues are understood will developers be able to leverage AOSD to its fullest.

6. RELATED WORK

Workshops on "Advanced Separation of Concerns" at OOPSLA '00 and ECOOP '01 [4] included discussion groups on the design space of aspect-oriented systems. Symmetry, and related issues

such as composability, closure and location of composition specifications were discussed. Aksit et. al. discuss six concerns in separation of concerns approaches [2]. They include *composability*, and *closure*, which is the property that composed entities must also be composable elements so that they can be involved in further compositions. There is to our knowledge, however, no other detailed analysis of symmetry issues in the context of composition. On the other hand, there are a number of approaches to AOSD with different symmetry properties. Our work on subject-oriented programming and multi-dimensional separation of concerns, both symmetric approaches, was mentioned during the course of the paper. We discuss a few others briefly here.

Composition filters [1] is an approach in which sequences of filters can be associated with objects. All messages to and from such objects pass through the filters in sequence. Each filter has the opportunity to act upon or modify each message, throw an error condition, or pass it through. Filters can implement new methods not directly supported by the objects, and can introduce their own state. This approach exhibits element asymmetry, since filters are different from objects and can only be attached to objects. The only composition relationships are separate specifications (or calls to the runtime system) to attach filters to objects in the appropriate sequences. Since these relationships can refer to the any of the filters and objects being composed in a uniform way and are external to all of them, they are symmetric, though the definition of filter-filter interaction is limited to sequencing. Composition filters therefore allow flexible attachment of reusable filters to arbitrary objects, adding or modifying behavior, but they cannot directly be used to integrate separate components (class hierarchies or domain models).

The work on adaptive programming [13] shows an interesting progression from asymmetric to symmetric. Earlier work allowed propagation patterns containing routing specification and handler code to be associated with class graphs, thereby injecting routing and handling code into the classes. The first version of this approach exhibited both element and relationship asymmetry. More recent work on adaptive plug-and-play components [14] has propagation patterns and a skeletal class graph they refer to packaged into components, which can then be composed with one another and/or instantiated relative to a full class graph.

AspectJ [11] makes explicit the distinction between components (classes) and aspects, and therefore exhibits element asymmetry. To date, aspects cannot be woven with aspects. The element asymmetry renders AspectJ suitable for extension scenarios, but unsuitable for supporting multiple views or domain models or for integration. Earliest versions of AspectJ exhibited relationship asymmetry: advice specifications in aspects listed the join points to which the advice was to apply. When multiple pieces of advice within aspects are woven into the same join point, precedence rules determine how they should be ordered. These rules now include a "dominates" declaration, in which one aspect can be explicitly specified to dominate another. This gives AspectJ a limited capability to specify aspect-aspect interactions, but still places the specification in one or other of the aspects rather than moving in the direction of providing symmetric relationships. The use of abstract pointcuts allows separation of pointcut details from advice code. The degree of relationship symmetry provided by these approaches, when they are used, helps to mitigate some of the disadvantages discussed earlier, making it possible for later aspects to reuse earlier ones to some extent, and for generic aspects to be woven into different components, but is not sufficient to provide full control of the weaving of multiple aspects into the same components.

Dynamic view connectors [9] are, essentially, composition relationships, possibly containing glue code, which allow integration of tools using custom views into an object repository system. The composable elements are objects in the repository, and the results of composition (or translation) are virtual objects that can be used by tools. This approach exhibits both element and relationship symmetry, which enable it to support materialization and integration of multiple views.

Mixin layers [19] represent an interesting mix of symmetric and asymmetric properties. The composable elements are mixin layers, which are like parameterized templates. They are essentially uniform in structure (element symmetry), except that one can distinguish layers without parameters as being base components. Join points are the parameters, and are asymmetric, but the composition relationships are instantiation expressions, which are symmetric. The element and relationship symmetry allow layers to be composed in various combinations and orders, and to be reused widely.

Viewpoints [16] represent a symmetric compositional approach to requirements engineering. Modules, called viewpoints, encapsulate developers' views of both the requirements-building process and the pieces of the requirements artifact being developed. Different viewpoints may describe the same requirements artifacts in different notations, and they may create conflicting definitions for given requirements. Each viewpoint is co-equal (element symmetry), and the specification of relationships among viewpoints is also symmetric (relationship symmetry). One characteristic of the Viewpoints approach that differentiates it from other compositional approaches is that different viewpoints are not physically combined as part of a composition process. Instead, viewpoints are integrated by coordination, and synchronization points replace join points.

7. REFERENCES

[1] M.Aksit, L.Bergmans, S.Vural. "An object-oriented language-database integration model: The composition filters approach." In Proceedings ECOOP'92

[2] M. Aksit, B. Tekinerdogan and L. Bergmans, "The Six Concerns for Separation of Concerns." Position paper at the ECOOP '01 workshop on Advanced Separation of Concerns, http://trese.cs.utwente.nl/workshops/ecoop01asoc/.

[3] Lodewijk Bergmans and Mehmet Aksit, "Composing Crosscutting Concerns Using Composition Filters." CACM 44(10), October 2001, pages 51–57.

[4] J. Brichau, M. Glandrup, S. Clarke and L. Bergmans, "Advanced Separation of Concerns." Workshop report in *ECOOP '01 Workshop Reader*, Springer Verlag, 2001.

[5] F. DeRemer and H.H. Kron. "Programming-in-the-Large versus Programming-in-the-Small", IEEE Transactions on Software Engineering, SE-2(2):80-86, June 1976.

[6] D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and

Knowledge Engineering, Volume 1, World Scientific Publishing Co., 1993.

[7] W. Harrison, H. Ossher, Member-Group Relationships Among Objects, Workshop on Foundations of Aspect-Oriented Languages at International Conference on Aspect-Oriented Software Development, 2002.

[8] W. Harrison and H. Ossher. "Subject-oriented programming (a critique of pure objects)." In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), September 1993.

[9] S. Herrmann and M. Mezini, PIROL: A case study of multidimensional separation of concerns in software engineering environments. In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), October, 2000, 188–207.

[10] S. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

[11] G. Kiczales, E.Hilsdale, J. Hugunin, Mik Kersten, J. Palm, W. Griswold, *"An Overview of AspectJ."* In Proceedings ECOOP'01, Springer-Verlag, June 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, *"Aspect-Oriented Programming."* In Proceedings ECOOP'97, Springer-Verlag, June 1997.

[13] K. Lieberherr. *"Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns."* PWS Publishing Company, 1996.

[14] M. Mezini and K. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development." In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), October 1998.

[15] G. Murphy, A. Lai, R. Walker and M. Robillard, "Separating Features in Source Code: An Exploratory Study." In Proceedings ICSE '01, May, 2001.

[16] B. Nuseibeh, J. Kramer and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications." IEEE Transactions on Software Engineering 20(10), October 1994, 760—773.

[17] H. Ossher and P. Tarr, "Operation-level composition: A case in (join) point." In *ECOOP '98 Workshop Reader*, 406–409, July 1998. Springer Verlag. LNCS 1543.

[18] H. Ossher and P. Tarr, "Using Multidimensional Separation of Concerns to (Re)shape Evolving Software." Communications of the ACM, October 2001.

[19] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers." In Proceedings of the 12th European Conference on Object-Oriented Programming, (ECOOP '98), July 1998.

[20] G. Snelting and F. Tip, Semantics-based composition of class hierarchies, In Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002), (Malaga, Spain, June 10-14, 2002), pp. 562-584.

[21] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "*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.

[22] ? , http://www.research.ibm.com/messagecentral/

[23] ? , http://www.research.ibm.com/Tengger/