N Degrees of Separation: Multi-Dimensional Separation of Concerns

Peri Tarr
Harold Ossher
William Harrison
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
(914) 784-7278
{tarr,ossher,harrisn}@watson.ibm.com

Stanley M. Sutton, Jr.*
EC Cubed, Inc.
15 River Road, Suite 310
Wilton, CT 06897
(203) 761-3971
ssutton@eccubed.com

ABSTRACT

Done well, *separation of concerns* can provide many software engineering benefits, including reduced complexity, improved reusability, and simpler evolution. The choice of boundaries for separate concerns depends on both requirements on the system and on the kind(s) of *decomposition* and *composition* a given formalism supports. The predominant methodologies and formalisms available, however, support only *orthogonal* separations of concerns, along *single* dimensions of composition and decomposition. These characteristics lead to a number of well-known and difficult problems.

This paper describes a new paradigm for modeling and implementing software artifacts, one that permits separation of *overlapping* concerns along *multiple* dimensions of composition and decomposition. This approach addresses numerous problems throughout the software lifecycle in achieving well-engineered, evolvable, flexible software artifacts and traceability across artifacts.

Keywords

Hypermodules; hyperslices; software decomposition and composition; multi-dimensional separation of concerns

1 INTRODUCTION

The primary goals of software engineering are to improve software quality, to reduce the costs of software production, and to facilitate maintenance and evolution. In pursuit of these goals, software engineers constantly seek development technologies and methodologies that reduce software complexity, improve comprehensibility, promote reuse, and facilitate evolution. These properties, in turn, induce several specific requirements on the formalisms used to develop software artifacts. Reduced complexity and improved comprehensibility require *decomposition* mechanisms to carve software into meaningful and manageable pieces. They also require *composition* mechanisms to put pieces together usefully. Reuse requires the development of large-scale reusable components, low coupling, and powerful, *non-invasive* adaptation and customization capabilities. Ease of evolution depends on low coupling and also requires *traceability* across the software lifecycle, mechanisms for minimizing the impact of changes, and substitutability.

Despite much good research in the software engineering domain, many of the problems that complicate software engineering still remain. Software comprehensibility tends to degrade over time (if, indeed, it is present at all). Many common maintenance and evolution activities result in high-impact, invasive modifications. Artifacts are of limited reusability, or are reusable only with difficulty. Traceability across the various software artifacts is limited, which further complicates evolution.

These somewhat diverse problems are due, in large part, to limitations and unfulfilled requirements related to *separation of concerns* [19]. Our ability to achieve the goals of software engineering depends fundamentally on our ability to keep separate *all* concerns of importance in software systems. All modern software formalisms support separation of concerns to some extent, through mechanisms for decomposition and composition. However, existing formalisms at all lifecycle phases provide only small, restricted sets of decomposition and composition mechanisms, and these typically support only a single, "dominant" dimension of separation at a time. We call this "tyranny of the dominant decomposition."

We believe that achieving the primary goals of software engineering requires support for *simultaneous* separation of *overlapping* concerns in multiple dimensions. We will illustrate how limitations on current mechanisms prevent this and thereby lead directly to the failure to achieve these goals. We propose a model of software artifacts, decomposition, and composition to overcome

*Stanley Sutton performed this work while at the University of Massachusetts at Amherst. He was supported there in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

ICSE '99 Los Angeles CA

these limitations. This model allows for simultaneous, multi-dimensional decomposition and composition. It is *not* a "universal" artifact modeling formalism; rather, it complements existing formalisms, giving developers additional modularization flexibility while continuing to use the formalisms of their choice. Moreover, this model is not particular to any phase of the software lifecycle. The extra flexibility to represent alternative decompositions of artifacts within a development phase also enables us to relate artifacts in multiple ways across phases, and even to *co-structure* artifacts—permit different artifacts, developed during different phases of the software lifecycle, to be structured in such a way that corresponding elements align clearly. We show how this increased flexibility can help to address the problems of software complexity and comprehensibility and difficulties with reuse, facilitate software evolution, and enhance traceability between artifacts, both within and across development phases.

The rest of this paper is organized as follows. Section 2 motivates the need for multiple dimensions of decomposition and rich mechanisms for composition. Section 3 describes our abstract model of software artifacts. It also shows how this model can address many of the issues raised in Section 2. Section 4 describes the issues involved in instantiating the model for particular artifact development formalisms, such as UML [21] or Java [5]. Section 5 describes related work shows how our approach has been partially realized in some existing work. Finally, Section 6 presents some conclusions and future work.

2 MOTIVATION

To illustrate some pervasive and serious problems in software engineering that help motivate our work, we present a running example involving the construction and evolution of a simple software engineering environment (SEE) for programs consisting of expressions. We assume a simplified software development process, consisting of informal requirements specification in natural language, design in UML, and implementation in Java.

The First Go-Round

The initial set of requirements for the SEE are simple:

> The SEE supports the specification of expression programs. It contains a set of tools that share a common representation of expressions. The initial toolset should include: an evaluation capability, which determines the result of evaluating an expression; a display capability, which depicts an expression textually; and a check capability, which checks an expression for syntactic and semantic correctness.

Based on these requirements, we design the system using UML. Figure 1 shows a subset of the design, which

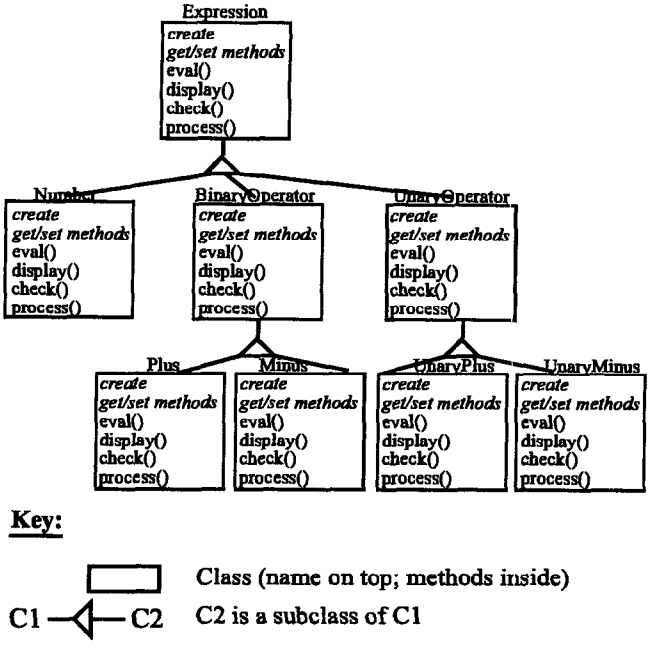

Figure 1: Initial (Partial) Design Artifact for SEE.

represents expressions as abstract syntax trees (ASTs) and defines a class for each kind of AST node. Each class contains accessor and modifier methods, plus methods `eval()`, `display()`, and `check()`, which realize the required tools in a standard, object-oriented manner.

The code that implements this design has a similar structure, except that it separates *interfaces* to AST nodes from *implementation classes*, resulting in two hierarchies instead of one.

This simple example raises some noteworthy issues that occur commonly in software. Despite being representations of the same system, each of the three kinds of artifacts decomposes the system differently. The requirements decompose by tool, or *feature* (e.g., [23]), while the design and code decompose by *object*. The code further separates interface from implementation parts. The difference in decomposition models leads directly to *scattering*—a single requirement affects multiple design and code modules—and *tangling*—material pertaining to multiple requirements is interleaved within a single module. These problems compromise comprehension and evolution, as we will see shortly.

Evolving the SEE: An Environmental Hazard

After using the SEE for some time, clients request some changes in the system:

- Expressions should be optionally persistent.
- Style checking should be supported as well as syntax and semantic checking. It should be possible to check expressions against multiple styles.

Any meaningful combination of checks (e.g., syntax only; syntax plus style(s)) should be permitted.

Unfortunately, these seemingly straightforward enhancements have a significant impact on the design and code. Figure 2 shows the impact on the Java implementation class hierarchy. A simple implementation of persistence requires adding "save" and "retrieve" methods to all AST classes, and inserting additional code into all accessor and modifier methods to retrieve persistent objects upon first access and to flush modifications back to the database. This represents a non-trivial, invasive change to all AST design classes and to all of the interfaces and implementation classes in the code, a serious case of scattering.[1] Code to support retrieval and update of persistent objects becomes tangled with other code in the accessor and modifier methods, impeding comprehensibility and future evolution. Further, the persistence code also has an impact on the new style checkers. If the persistence option is present, the style checkers must include their state information in the persistent representation of expressions. This kind of context-dependent feature is extremely difficult to represent in modern formalisms.

The ability to permit arbitrary combinations of checks is also problematic. It requires special infrastructure support, in both the design and implementation. This infrastructure is not present—it comes at high cost in terms of conceptual complexity and run-time overhead, so it was not included originally as it was not necessary. We choose to address this problem by retrofitting the Visitor design pattern [4], which permits optional combinations of features, into the design and code. Visitor requires us to replace all AST `check()` methods with `accept(Visitor)` methods, and to define a separate Visitor class for each type of check. The modifications to the check feature needed to support this capability are invasive, affecting every module in the design and code, and complicating all the artifacts and their interrelationships. The presence of arbitrary checks further complicates the persistence capability, since the information to be made persistent depends on the particular combination of syntax and/or style checkers. Finally, these modifications significantly impede the future evolution of the artifacts. They introduce a higher degree of coupling between the AST classes and the visitor classes, as evident in Figure 2, and the presence of visitors in the design will necessitate extensive changes to accommodate modifications to the AST hierarchy [4].

The Postmortem

This example demonstrates, in a microcosm, many problems that plague software engineers and suggests why we still fall short of our goals.

Impact of change: The goal of low impact of change requires *additive*, rather than *invasive*, change. Yet conceptually simple changes, like those in the expression SEE, often have widespread and invasive effects, both within the modified artifact and on related pieces of other artifacts. This is primarily because units of change often do not match the units of abstraction and encapsulation within the artifacts. Thus, additive changes in one artifact, like requirements, may not translate to additive changes in other artifacts, like design and code.

Modern extensibility features, such as subclassing and design patterns, help but are not sufficient [16] because they require significant pre-planning. It is not feasible to pre-enable artifacts for all possible extensions, even if it were possible to anticipate them.

Reuse: Despite wide recognition of its benefits, reuse is limited and occurs mostly on code, not requirements or designs. Part of the impediment to large-scale reuse is that larger artifacts entail more design and implementation decisions, which can result in tangling of concerns and coupling of features, reducing reusability. Given large and complex artifacts, plus the weak set of adaptation and customization capabilities available in most formalisms, developers face a significant amount of invasive work to adapt a component for a given context.

Traceability: Different artifacts are written for different purposes and include different levels of abstraction. Thus, they are specified in different formalisms and are often decomposed and structured differently. A case in point is the requirements scattering and tangling problem illustrated earlier. No clear correspondence of abstraction or structure across artifacts exists, in general, to aid traceability. Instead, developers must create connections among related artifacts explicitly (e.g., [9]). These connections are complex, can be invalidated readily, and, most importantly, they do not reduce scattering or tangling. They can help developers assess the impact of a given change, but they cannot localize it or reduce its impact. Developers must therefore make invasive, time-consuming changes to multiple artifacts to propagate the effects of a given change. When time constraints are tight, they often choose to make changes only to code, letting other artifacts become obsolete.

We believe that a major cause of these impact of change, reusability, and traceability problems is the "tyranny of the dominant decomposition." Existing modularization mechanisms typically support only a small set of decompositions, and usually only a single "dominant" one at a time. This dominant decomposition satisfies some important needs, but usually at the expense of others. For example, a decomposition may be chosen to limit the

[1] Subclassing is a non-invasive mechanism for change, but it is not a reasonable option here. It produces combinatorial explosions of classes and still requires invasive changes to any client that creates instances of the original classes.

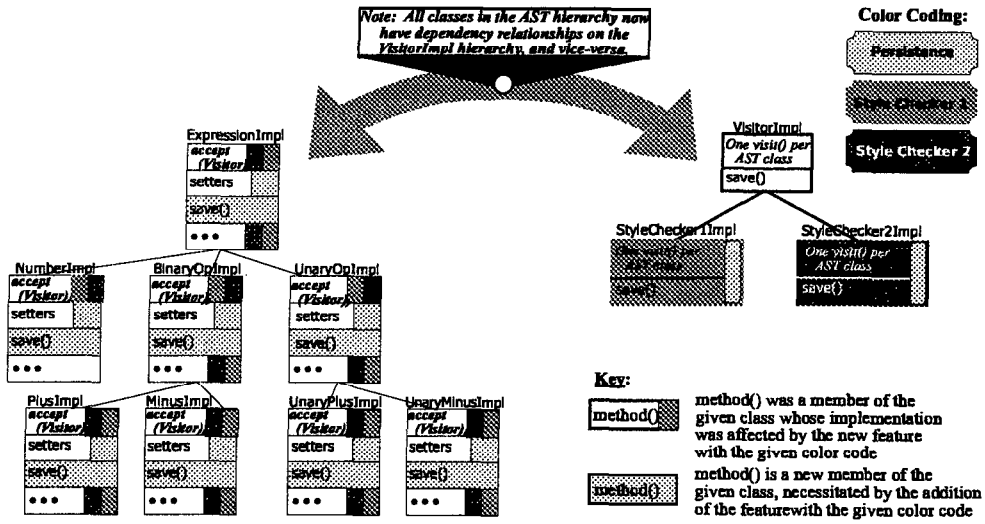

Figure 2: The Java Implementation Classes, Post-Evolution.

impact of some changes, but traceability may thereby be sacrificed (or, indeed, the ability to limit the impact of other changes); or, in a data decomposition designed to match application-domain concepts, code for a feature may be scattered across multiple application modules and tangled with code for other features. To make matters worse, different formalisms typically support different dominant decompositions, reducing traceability across artifacts. Many different kinds of concern are important in a software system, and designating one of them as dominant in each context, at the expense of the others, contibutes significantly to the problems identified above.

Breaking the Tyranny

To achieve the full potential of separation of concerns, we need to break the tyranny of the dominant decomposition. In the example and related discussion, several kinds of concerns were identified:

- *feature:* these include display, basic check, evaluate, persistence, and style check. Features may also be required or optional
- *unit of change:* additions made due to user requests
- *customization:* the additions or changes needed to customize a component for a particular purpose
- *data* or *object:* the classes involved in the system.

If the system could be modularized according to concerns of all these kinds, *simultaneously*, the problems described above would be greatly ameliorated. Traceability would be improved by encapsulating features separately, with clear correspondence between the representation of a particular feature in different artifacts (i.e., co-structuring). Impact of change would be reduced by the ability to encapsulate each unit of change separately. Reuse would be enhanced by the improved traceability, and by separating customization details from the base component, provided composition is rich enough to apply them effectively.

These are just a few of the dimensions of concern along which separation may be desirable. Others include: to match conceptual abstractions; to conform to a given modeling paradigm (object-oriented, functional, etc.) or to take advantage of special-purpose formalisms; to separate "optional" from "required" pieces; to separate variants for different host systems, classes of users, etc.; to permit distribution or parallel processing; to facilitate concurrent or cooperative development; etc. The possibilities are limitless, and vary with context. What is more, different dimensions of concern are seldom orthogonal: they overlap, and can affect one another. A truly flexible approach to modularization must allow any and all that are needed to apply simultaneously, and must be able to handle overlap and interactions among them.

3 MULTI-DIMENSIONAL SEPARATION OF CONCERNS

This section introduces a model of decomposition and composition that we believe satisfies these needs. The model is used in conjunction with developers' artifact formalism(s) of choice, giving developers additional power without requiring changes to the formalisms.

We begin with a model of conventional software, to set the context and introduce some terminology, then describe our model and show how it addresses many of the issues raised earlier.

A Model of Conventional Software

A particular software system is written to address some problem or provide some service within a problem *domain.* To do this, it must model or implement a variety

of *concepts* of importance in that domain. These concepts include objects (e.g., "expression" in the example), functionality (e.g., "evaluation"), and properties (e.g., "persistence"). Concepts derived directly from the domain as well as internal software concepts (e.g., data structures) are both important.

The software system itself consists of a set of *artifacts*, such as requirements specifications, designs, and code. Each artifact consists of descriptive material in some formalism, whose purpose is to model needed concepts in a manner appropriate for that artifact. The formalisms differ for different projects, different phases, and different artifacts, and perhaps even within an artifact. Different artifacts often share the same concepts, with each concept potentially being described in a different way, and with different details, in the different artifacts. For example, the word `expression` in the requirements and the class `Expression` in the design and code all describe the concept "expression" in their rather different ways and at different levels of detail.

It is convenient to think of the descriptive material in each artifact as being made up of *units*. What constitutes a unit depends on the formalism, and perhaps on the context. For example, in object-oriented design formalisms or programming languages, classes are one kind of unit. If one looks below the class level, individual methods may also be considered units. This illustrates the important point that formalisms typically consist of at least some basic elements, which we call *primitive units*, and some grouping construct(s), which we call *compound units* or *modules*.

We treat primitive units as indivisible; our model works with them, but never looks inside them. A single concept is typically modeled by a collection of many units (primitive or compound). Perhaps surprisingly, a single unit often participates in modeling more than one concept. For example, the `eval()` method within the `Plus` class participates in modeling both the "plus expression" concept and the "evaluation" concept.

The purpose of modules is to accomplish *separation of concerns* [19]. Even software systems of moderate size contain so many primitive units that they cannot all be held in one's mind at once. When performing some development task, a developer must be able to focus on those units that are pertinent to that task and ignore all others. To accomplish this, software engineers identify *concerns* of importance, and seek to localize units representing concepts that pertain to each concern into a module. Ideally, one only need look inside a module if one is interested in a given concern. For example, a class is a module containing units (describing methods and instance variables) that model a particular kind of object; all internal details of such objects, such as their representation, are described within the class.

Many kinds of concerns are important during the software lifecycle. These *dimensions of concern* help to organize the space of concepts and units. Common dimensions of concern are data or object (leading to data abstraction) and function (leading to functional decomposition). Others include feature (both functional, such as "evaluation," and cross-cutting, such as "persistence"), role, and configuration. As illustrated by these examples, some dimensions of concern derive from the domain, often aligning with important domain concepts, while others come from system requirements, from the development process, and from internal details of the system itself. In short, there are any number of dimensions of concern that might be of importance for different purposes (e.g., comprehension, traceability, reusability, evolvability, etc.), for different systems, and at different phases of the lifecycle.

Modern artifact formalisms typically allow decomposition (i.e., grouping of units) into modules according to only a single dimension of concern, which we term the *dominant dimension*. The formalism often dictates specifically what the dominant dimension must be. For example, object-oriented formalisms support decomposition based on the object (or data) dimension, while procedural and functional programming languages permit decomposition based on function. Even formalisms that do not impose a specific dominant dimension typically do not support simultaneous decomposition according to multiple dimensions, so the developer ultimately chooses a dominant dimension. In either case, the modular structure of the artifact achieves separation of concerns only along this dominant dimension.

Thus, in our model, a conventional software system is a set of artifacts that model domain concepts in appropriate formalisms. Artifacts contain modules, which contain units. The modular structure reflects decomposition based on one dominant dimension of concern.

Multi-Dimensional Decomposition: Hyperslices

As discussed in Section 2, decomposition according to concerns along a single, dominant dimension is valuable, but usually inadequate. Units pertaining to concerns in other dimensions end up "scattered" across many modules and "tangled" with one another. Separation according to these concerns is, therefore, not achieved. To alleviate this problem, we introduce *hyperslices* as an additional, flexible means of decomposition.

A *hyperslice* is a set of conventional modules, written in any formalism. Hyperslices are intended to encapsulate concerns in dimensions other than the dominant one. The modules within it contain all, and only, those units that pertain to, or address, a given concern. Hyperslices can overlap, in that a given unit may occur, possibly in

different forms, in multiple hyperslices. This supports simultaneous decomposition according to multiple dimensions of concern. A system is written as a collection of hyperslices, thereby separating all the concerns of importance in that system, along as many dimensions as are needed. The hyperslices are composed to form the complete system (discussed below).

The choice of the term "hyperslice" is intended to reflect relationships to both "program slicing" [25] and "hyperplane." Hyperslices are similar to program slices in that both involve cuts through a system that do not align with the standard modules. They differ, however, in that program slices are at the code level only, generally consist specifically of statements that affect particular variables, and are extracted from existing programs by analysis, rather than being used to build systems by composition. Hyperslices are hyperplanes in that they encapsulate concerns that cut across multiple dimensions in a space defined by the dimensions of concern.

To demonstrate the utility of hyperslices, we consider the initial version of the expression SEE described in Section 2. We identified two separate dimensions of concern applicable to the initial design: object (different kinds of expressions) and feature (display, evaluation, and basic checking). Since we used object-oriented formalisms for the design and code, the object dimension was the dominant one, and separation of concerns along that dimension was effective. Separation by feature could not be accomplished, however, leading to scattering and tangling of feature-specific units. We therefore introduce five hyperslices, one to encapsulate each of these concerns (features), as shown in Figure 3. One hyperslice encapsulates the basic ("kernel") expression AST capabilities (node creation, accessor, and modifier methods), modularized using UML classes in the design and Java classes and interfaces in the code. The other hyperslices encapsulate, respectively, the display, evaluation, and syntax and semantic checking features. Note that these hyperslices also contain many of the same class modules as found in the kernel hyperslice (i.e., their concerns *overlap*), but the modules in these hyperslices contain only those units that pertain to the particular concern they encapsulate. Thus, e.g., the display hyperslice defines `display()` methods and instance variables (units) in AST node classes (modules), while the evaluation hyperslice defines `eval()` methods and instance variables.

Note that hyperslices have been introduced without requiring the definition of new artifact formalisms. We deliberately do not modify the artifact formalisms themselves, preferring instead to allow developers to use their familiar formalisms throughout the lifecycle. The modules within a hyperslice are standard modules in the desired formalism, except that they contain only those units pertinent to the hyperslice's concern. That means that these modules might not satisfy all of the completeness constraints that the formalism normally requires. For example, the implementation code in the display hyperslice might refer to accessor methods that it does not define, on the expectation that the kernel hyperslice will provide them. This is not legal in Java, which requires modules to define any methods they use. It is fine in our model, however, because hyperslices are eventually composed together to form a "complete" hyperslice that must satisfy all of the formalism's constraints.

The definition of hyperslice above is sufficiently broad that it is possible, for any concern, to form a hyperslice consisting of exactly those units pertaining to that concern. For example, hyperslices can correspond to features, to units of change, or to specific customizations or components. If this approach is followed for all concerns of interest in a system, there is likely to be a good deal of overlap: the same unit, or different units describing the same concept, might be involved in multiple concerns. We saw this in the expression example—each of the hyperslices includes expression concepts in the form of class modules, but it defines those concepts in a way that is appropriate to its task. Overlap is acceptable; indeed, it is responsible for much of the power of this approach. Composition must be able to resolve the overlap, as discussed later.

This great flexibility raises the question of how developers should choose hyperslices for decomposing a given system, and whether the freedom is likely to lead to error and abuse. Simple uses, such as for major features or units of change, provide great benefit with little difficulty. Formulation of guidelines for more complex use of hyperslices is an issue for future research. Even with outstanding guidelines, however, use of hyperslices, like any other modularization mechanism, requires good judgement. If key structural decisions turn out to be incorrect because of design error or dramatic changes to requirements, system restructuring may be necessary, as with conventional technology. The support for simultaneous separation of concerns along multiple dimensions, however, opens the possibility of introducing new dimensions and ignoring obsolete ones, without dismantling the system. This, too, needs further research.

Composing Hyperslices Using Hypermodules

Hyperslices provide a flexible means of decomposing artifacts. To be useful, however, it must be possible to compose them to produce complete and consistent artifacts in unchanged artifact formalisms of choice.

A *hypermodule* is a set of hyperslices, together with a *composition rule* that specifies how the hyperslices must be composed to form a single, new hyperslice that synthesizes and integrates their units. Because of this com-

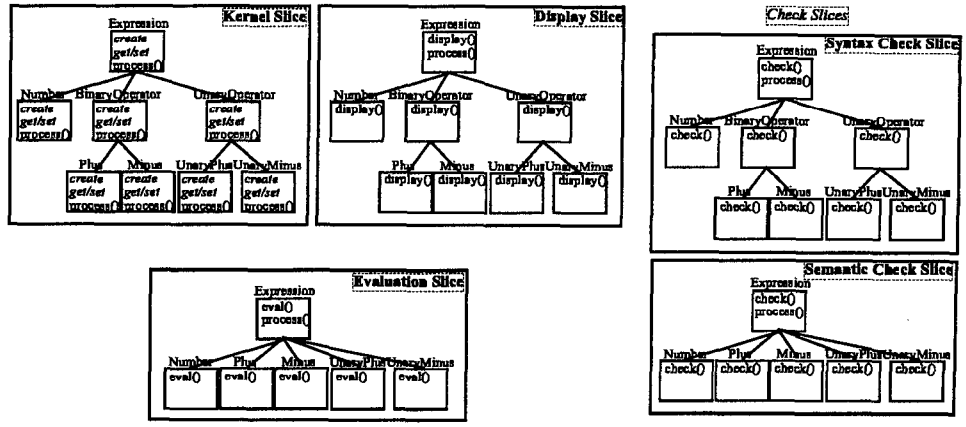

Figure 3: Defining the SEE with Hyperslices.

position property, a hypermodule is appropriate wherever a hyperslice may be used. Hypermodules can thus be nested. An entire artifact can be modeled as a hypermodule; the artifact consists of all the modules in the composed hyperslice and must satisfy whatever consistency and completeness constraints are required by the artifact formalism. The system as a whole—all of its artifacts—can also be modeled as a hypermodule, whose composition rule describes the relationships between the artifacts. The simplification of these relationships, made possible by hyperslices, and their reification in the composition rule, is a key advantage of this model.

Figure 4 shows a hypermodule consisting of the hyperslices from Figure 3. The composition rule must indicate which units in the hyperslices describe the same concepts, and how those units must be integrated. In this case, it asserts that classes in different hyperslices with the same name model the same concept and should be "merged" into a new, composed class with the same name and combined details. When the composition rule is applied, the resulting hyperslice contains exactly the modules shown in Figure 1. Notice that the syntax and semantic checking hyperslices can be grouped optionally into a "check" hypermodule that is nested within the SEE hypermodule. The result of (optionally) composing the syntax and semantic checking hyperslices within the "check" hypermodule is a check hyperslice, which can then be composed with the other SEE hyperslices. The ability to nest hypermodules in this manner promotes abstraction and encapsulation.

Details of composition vary greatly depending on the formalism in which units are written, and on which of the formalism's constructs are treated as units and modules. These are details that are specified as part of an *instantiation* of this model (described in detail in Section 4), which represents a mapping between a particular formalism and the concepts embodied within the model. They are also dependent on the details of the particular units involved, and can vary from straightforward to highly complex. Nonetheless, some general properties are worth discussing.

Composition is based on commonality of concepts across units: different units describing the same concept (usually, though not necessarily, differently) are composed into a single unit describing that concept more fully. This process involves three steps: *matching* units in different hyperslices that describe the same concept, *reconciliation* of differences in these descriptions, and *integration* of the units to produce a unified whole. Clearly, composition cannot be a fully automatic process. It is the task of the composition rule in the hypermodule to specify the details of composition.

One approach to composition rules, suggested by our work on subject-oriented programming [7, 17], is for the rule to be a combination of a concise, general rule, and detailed, specific rules that specify exceptions to the general rule or handle cases that it cannot handle. The general rule essentially names an automatic approach to apply as a starting point or default, such as matching by unit name (i.e., the name denotes the concept). General rules can be applied to an entire composition, or selectively to portions of it; different automatic approaches can thus be applied to different areas of a composition. Only in cases where no automatic rule suffices are detailed rules needed, in which the developer says explicitly exactly what to do. Detailed rules can handle such issues as matching units with different names that do describe the same concept, not matching units with the same names that do not describe the same concept, and reconciling different module structures, such as matching units nested at different depths in different hyperslices that nonetheless describe the same concept. The degree of mismatch in module structure and abstraction level that can be handled effectively is an issue for

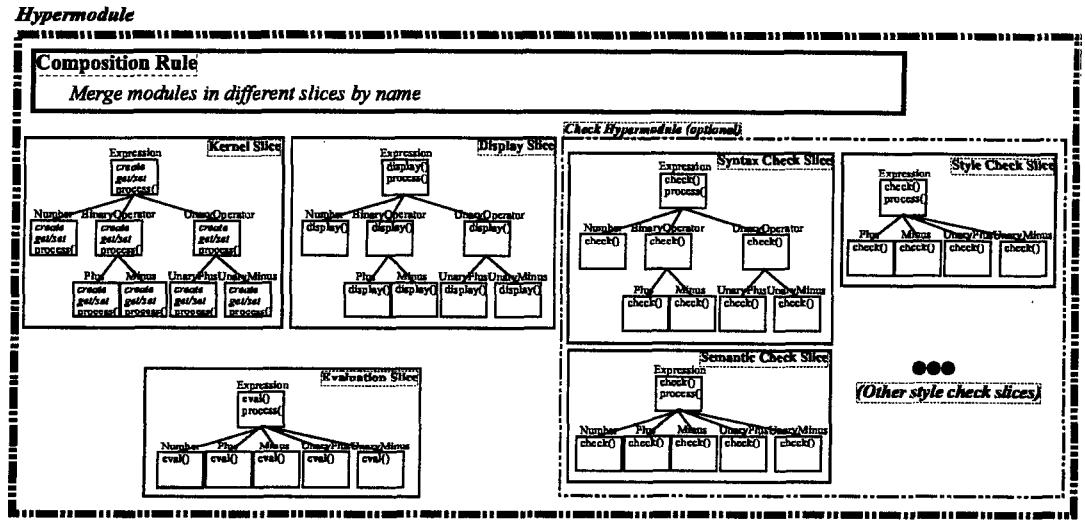

Figure 4: Composing Hyperslices using Hypermodules.

future research, as is determining how much mismatch occurs in practice in composed hyperslices.

An alternative is to split the composition rule across the hyperslices, allowing each hyperslice itself to specify how it is to be composed. If the rule in a hyperslice can refer to other hyperslices, this increases coupling and reduces reusability of hyperslices; if it cannot, this limits the flexibility with which overlap can be handled. Putting the composition rule a level higher, in the hypermodule, allows both flexible overlap and enhanced reuse.

In this model, therefore, developers write each artifact as a hypermodule. For each concern of importance that cannot be encapsulated effectively using the artifact formalism, they write a hyperslice that consists of modules in the artifact formalism. They also write a composition rule that specifies how these hyperslices are to be composed into a set of legal modules that make up the artifact. They also write an enclosing hypermodule that contains all the artifacts and whose composition rule specifies the relationships between them.

Using the Model

We have already begun to see how this artifact model can help to address some of the software lifecycle problems identified in Section 2. We now explore its impact on these problems in more detail, by revisiting the expression SEE example. We apply the same software development and evolution process, but this time, we use the proposed artifact model. We then evaluate how well the resulting artifacts address the problems presented earlier.

Revised First Go-Round

As described in Section 3, Figure 3 shows a somewhat different decomposition of the design and code artifacts than that produced during the initial design and coding process (depicted in Figure 1). The model has allowed us to separate the major non-object concerns identified during requirements-gathering: the "kernel," which encapsulates basic functionality pertaining to expressions, and display, evaluation, and checking features. Each of these concerns is encapsulated in a hyperslice. Since we chose to decompose the check feature further, we represent it as a nested hypermodule, which includes two subhyperslices, one each for the syntax and semantic checkers.

This decomposition has some significant benefits. First, hyperslices permit decomposition along multiple dimensions—in this case, object and feature—even within object-oriented formalisms that generally support only the object dimension. Second, the improved separation of concerns eliminates the scattering and tangling problems we saw earlier, by keeping units pertaining to separate requirements and features separate. A key benefit is that we have achieved encapsulation of coherent concerns *across the lifecycle.* This improves traceability and can significantly simplify the interrelationships among different artifacts that are traditionally so difficult to maintain. This approach also improves reusability considerably. For example, the entire expression AST concept, from requirements all the way to code, has been defined in a context-independent manner and can be reused readily, since the context-specific pieces are encapsulated in other hyperslices.

The use of composition to assemble hyperslices into the final SEE provides some substantial benefits as well. Observe that because composition of hyperslices is always optional, we have managed, just by separating the concerns, to ensure that we will later be able to "mix-

and-match" syntax and style checking. We can also create versions of the SEE that contain different combinations of checking, evaluation, and display features—an ability we did not have in the original SEE. Notice also that we have a choice over how we define our hypermodules. We could, for example, define three hypermodules: one each that includes all hyperslices pertaining to a particular *artifact*. This allows us to compose the full requirements specification, design, and code artifacts. But we could also choose to define one hypermodule per *concern*—e.g., an "expression" hypermodule, which contains the requirements, design, and code hyperslices that encapsulate the "kernel," a "display" hypermodule that encapsulates all artifact hyperslices pertaining to display, etc. Both kinds of composition are valid and are useful for different purposes; the former permits the creation of the final artifacts, while the latter facilitates reuse of concerns and permits certain forms of inter-artifact completeness and consistency checking. As noted earlier, developers may need to decompose *or* compose differently for different reasons. This model permits them to do just that.

SEE Evolution: Saving the Environment

Clients eventually requested support for optional persistence of expressions and for multiple forms of style checking and the ability to "mix-and-match" types of checks. Persistence is a new concern; it represents both a new feature and a unit of change. As such, its addition is not supported well by object-oriented separation of concerns, as we saw in Section 2. This time, we choose to model persistence as an independent concern (hyperslice), which both encapsulates it and provides us the opportunity to use ASTs with or without persistence. Adding style checkers is trivial—the checking hyperslice already separates syntax and semantic checking, so we need only define the style checkers as hyperslices and compose any set of them together with the syntax and/or semantic check hyperslices. Notice that these new capabilities do not require any modifications to existing hyperslices or artifacts—they can be encapsulated as separate concerns and composed with the existing artifacts.

Postmortem Revisited

We now revisit the set of software engineering problems discussed in Section 2.

Impact of change: Much of the reason for high impact of change is the mismatch between the units of change and the units of abstraction and encapsulation within artifacts. With our model, however, units of change can be separated and encapsulated like any other concern. This can, in many common cases, significantly reduce or eliminate the impact of change.

Reuse: As noted above, this model may significantly improve reuse of *all* artifacts. It permits the separation of generally useful capabilities from special-purpose ones, and it provides composition as a very powerful, non-invasive customization and adaptation mechanism. Thus, it is simpler to create reusable components and to pick up and tailor a component to a particular need.

Traceability: The ability to identify, encapsulate, and co-structure similar concerns across different artifacts greatly facilitates traceability and propagation of change across the lifecycle.

While the appropriate use of the model can directly result in the benefits we have described (and many we have not), it is not a panacea for bad design, bad code, or poor modularization. Further, overseparation of concerns is as bad as underseparation—it leads to large numbers of hyperslices with complex interrelationships, and may actually reduce comprehension and increase complexity. Nonetheless, we believe the model is a valuable tool with potentially high benefit, if used properly.

4 INSTANTIATION

To use this artifact model, one must instantiate it for particular artifact development formalisms. Instantiation entails determining which notational constructs map to units and modules, deciding how to represent hyperslices, and providing support for composition of hyperslices. The mapping to units is especially important, as it significantly affects how well the hypermodules will achieve various software engineering goals and properties. This section briefly describes some of the issues involved. A fuller discussion appears in [18].

Mapping to Units and Modules

Units: Choosing "units" from the set of artifact formalism constructs requires an instantiator to decide the level of *granularity* at which it is appropriate, in the given formalism, to separate and integrate concepts. We illustrate this by example, using the Java language. Java defines both *declarator* constructs (e.g., packages, interfaces, classes, methods) and *statements*. Some subset of these constructs must be treated as units. A decision in favor of fine granularity might include all declarators and statements as units. This potentially provides the flexibility to compose any pieces of Java source, but it has all of the concomitant problems of determining how to match and reconcile different statements and of trying to analyze the properties of the result. Using a coarser level of granularity might result in treating only a subset of declarators (e.g., classes and their members) as units, which simplifies composition and understanding of the composed result, at the cost of generality.

The selection of units has significant ramifications for some important software engineering properties of artifacts [18], including effects on evolution and modular development. If the set of units includes entities that

are typically "hidden," such as method implementation code, composition rules and their results become sensitive to "hidden" changes. Modular development relies on important properties of individual modules being preserved by composition. If composition can occur at too fine-grained a level, such properties might not be preserved, and must be re-examined afresh in the context of each composition.

Data and *functionality* are fundamental and ubiquitous concepts in software. They are frequently the concepts that are described by artifacts, and the concepts that span hyperslices and artifacts. Formalisms generally have constructs for declaring or defining them. For example, UML has boxes representing classes, and entries within class boxes representing instance variable and method declarations. Java has classes, interfaces, instance variable declarations and methods. We believe that constructs related to data and functionality are excellent candidates for units, and hypothesize that they might, in general, be the best choices.

Modules: The selection of formalism constructs to map to modules is somewhat simpler than the choice of units. Essentially, it requires examining the particular modularization constructs the formalism provides in light of the set of units chosen. For example, suppose we choose Java methods, instance variable declarations, classes, and interfaces as units. Instance variable declarations and methods are grouped together into classes and interfaces, which in turn are grouped together into packages. We would therefore choose to map Java classes, interfaces, and packages to modules in our model. An obvious choice for UML is to map classes and package diagrams to modules.

Representation of Hyperslices

Hyperslices are sets of modules. They need not occur explicitly in any given artifact formalisms, though some formalisms may provide a construct to which it is convenient to map hyperslices. For example, C++'s namespace construct, which represents arbitrary collections of program units, Java's package construct, which represents collections of classes and interfaces, and UML's package diagram, which represents collections of packages and classes, may be used to model hyperslices. For formalisms that do not have such constructs, it is necessary either to enhance them or to provide a separate hyperslice-specification mechanism, such as named lists of modules.

Support for Composition

To provide support for composing hyperslices, it is necessary to define a means for specifying composition rules—a language, an interactive tool, or both—and to build a *compositor* that is able to apply the rules to hyperslices. Composition by hand is conceptually possible, but totally unrealistic for actual development.

Providing this support is a large job. That is a powerful reason to make mapping decisions based on formalism, not on content, to avoid the need for project-specific compositors. Compositors specialized to understand particular semantic dimensions may be useful in some circumstances, however, as demonstrated by recent work on aspect-oriented programming [10].

5 RELATED WORK

We discuss two categories of related work: approaches that can (loosely, perhaps) be considered instantiations of our model for particular types of artifacts, and different approaches to similar problems.

Subject-oriented programming [7, 17] partially realizes our model for object-oriented code artifacts. The units are classes, methods and instance variables. Systems are built as compositions of *subjects*—hyperslices—each of which is a class hierarchy modeling its domain from a particular point of view. We have built composition support for C++ and CORBA IDL, prototype support for Smalltalk, and are currently building support for Java. Composition rules, specified textually for C++ and through an interactive user interface for the other systems, provide considerable matching and reconciliation flexibility, and the support is a framework allowing addition of new matchers and reconcilers. We have several small, running examples that demonstrate the value of decomposition into subjects. We are also currently exploring the manifestation of subjects and composition rules in UML, to allow co-structuring of subject-oriented designs and code.

Aspect-Oriented Programming (AOP) [10] expands on the concepts of subject-oriented programming by identifying and illustrating several useful, *non-functional* concerns to be separated, such as concurrency properties, distribution properties, persistence and other "emergent entities" [11]. Initial work used different aspect languages (e.g., [12]) to represent different aspects. This is appealing, since a programming language is not necessarily the best formalism for expressing non-functional requirements, but it results in a need for special-purpose compositors (called *weavers*). More recent work is aimed at providing a general-purpose weaver for hyperslices written in Java [11].

AOP distinguishes the notion of "core classes," which encapsulate a system's functional requirements, from "aspects," which encapsulate non-functional, cross-cutting requirements. Aspects are written with respect to core classes and are essentially orthogonal to one another. Relative to our model, each aspect is a hyperslice, and a set of aspects together with the core classes approximate a hypermodule. The core classes are distinguished; all aspects refer to them, and there-

fore share the same view of the overall class structure. The hypermodule does not have a central composition rule. Instead, each aspect contains its part of the rule, specifying how that aspect is to be *woven* into the base classes. This makes the approach subject to the disadvantages discussed in Section 3, particularly that handling of overlapping concerns (i.e., interaction among aspects) is perforce done in a standard, default manner by the weaver.

Holland discusses the building of systems using compositions of *contracts* [8]. Each contract specifies a set of participant objects and their interactions, expressed as *obligations*. Its primary intent is to encapsulate these particular interactions and obligations so that they are clearly separated from other interactions involving the same objects. A single object can participate in multiple contracts, in which case it must satisfy all their obligations. Holland describes a variety of combination rules for contracts. A contract corresponds to a hyperslice in our model, cutting across classes that describe objects. The combination rules provide some alternative means of combining specifications in different contracts that apply to the same participant.

Similarly, role models (e.g., in OORAM/OORAS [1]) are essentially hyperslices. Each model describes particular roles played by objects, and how those roles interact. Role models must be composed, usually manually, to produce object definitions that satisfy all needed roles. VanHilst and Notkin propose an approach to implementing roles with templates [24]. Each template defines a role, and instantiation expressions create classes that satisfy all required roles. Collections of related templates, such as those defining similar or interacting roles for objects, constitute hyperslices in our model, and instantiation expressions are composition rules.

Adaptive programming is another approach to providing modules other than classes within object-oriented systems. A *class graph* describes some classes and their relationships, from a particular point of view. Class graphs do not contain code; instead, code is written in separate *propagation patterns*. Propagation patterns can be used with any collection of concrete classes that conform to the class graph against which they were defined. Adaptive programs are transformed into standard object-oriented programs by the Demeter tools [6]. With respect to this generated program, each propagation pattern is a hyperslice, since it contains method code that cuts across classes. The composition is performed by the Demeter tool, with matching being based on specifications of class graph conformance. Propagation patterns do not overlap, however—each defines its own method—so reconciliation is not an issue. In a recent paper [13], collaboration-based decomposition is discussed, of which contracts are an example. Collaborations are hyperslices, cutting across classes.

Catalysis [3] facilitates building reusable design frameworks in UML. It incorporates a simple notion of composition based on the union of design models. It therefore represents an instantiation of our model for UML. Catalysis' matching and reconciliation rules are fairly simple, which limits the dimensions along which design models can be decomposed and composed, but makes reasoning about properties of the composed design in terms of its component design models more tractable.

The *Viewpoints* project [15] is an 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. The Viewpoints system defines mechanisms (based on theorem proving) for identifying and helping developers cope with inconsistency.

The Viewpoints approach shares a number of points in common with ours but also has corresponding differences. Both approaches are predicated on the belief that not all concerns can be modularized orthogonally, and that it must be possible to view systems as potentially overlapping pieces. Another similarity is a concern with resolving semantic differences between different aspects or elements of a system (views or hyperslices). Viewpoints emphasizes the detection and characterization of inconsistencies while deferring their resolution (reconciliation) to the encompassing requirements process. We have focused on the activity of composing concerns after they have been separated, including identifying and, especially, reconciling inconsistencies according to a composition rule. Finally, we are primarily concerned with how artifacts are constructed, while the Viewpoints approach is primarily concerned with how they are viewed.

Some of the problems addressed by our approach can be tackled differently. Attempts have been made to address the problem of traceability with environment support for capturing and maintaining the relationships among artifacts (e.g., [9]). The disparate structures of the artifacts make this a particularly tough problem.

The problem of limiting the impact of change has been addressed by various architectures and mechanisms, like implicit invocation [14], mediators [22], event-based integration [20], and design patterns [4]. These are all valuable, but they suffer from the drawback that the kinds of changes they permit—the open points—must be anticipated. Retrofitting any of these mechanisms where not originally planned requires invasive change.

A great deal of work has been done to promote

reuse, and other researchers and developers have recognized the importance of large-component reuse (e.g., [2]). Effective reuse requires powerful adaptation and customization mechanisms, but current customization technology is usually restricted to interface adaptation using some sort of adapter or transformation layer, or to substituting alternative modules at predetermined points, such as in object-oriented frameworks. Interesting recent work builds on adaptive programming to support "adaptive plug-and-play components" [13].

6 CONCLUSIONS AND FUTURE WORK

A number of important problems in software engineering have resisted general solution, including problems related to software understanding, maintenance, evolution, and reuse. We believe that these problems share a common cause: failure of modern artifact formalisms to satisfy the *separation of concerns* requirement adequately. Numerous reasons exist to separate and integrate software artifacts, and these reasons may result in different artifact structures. Moreover, many concerns may be relevant simultaneously, and the entire set of concerns may evolve over time. Despite this observation, artifact formalisms include weak decomposition and composition mechanisms that permit only a small, "dominant" set of concerns to be separated. This leads directly to our inability to achieve many of the goals of software engineering as a discipline.

Our model of multi-dimensional software decomposition helps to overcome these limitations. It permits encapsulation of particular concerns in a software system, both *within* and *across* artifacts, and it allows kinds of separation of concerns that may not be separable in artifact formalisms, such as units of change, features, and overlapping concerns. This improves traceability across the lifecycle. The model also provides a powerful composition mechanism that facilitates integration, adaptation, and "plug-and-play." In so doing, it promotes reuse, improves comprehension, and eases maintenance and evolution. Thus, the approach addresses some fundamental limitations in software engineering. For these reasons, we believe that support for multi-dimensional decomposition and composition represents a key to advances along a broad front of software engineering challenges.

This work is clearly at an early stage, largely unproven yet. Still, a considerable body of experience and related research now exists to support the claim that multi-dimensional separation of concerns is one of the key software engineering issues today. The model presented is just a starting point. It must be refined, stretched and modified, and it must be instantiated for a variety of formalisms to explore issues that arise for different methodologies and at different phases of the software lifecycle. These instantiations must be used for real development, to evaluate them and create new development methods that exploit their strengths; to explore issues in intra- and inter-artifact matching and reconciliation; and to explore the impact of multi-dimensional separation of concerns on areas like development methodology, software process, analysis, testing, reverse engineering, reengineering, and software architecture.

ACKNOWLEDGEMENTS

Joyce Vann, Mark Wegman, and the reviewers provided valuable feedback on earlier versions of this paper. Siobhán Clarke produced the SEE design in UML.

REFERENCES

[1] E. P. Andersen and T. Reenskaug. System design by composing structures of interacting objects. In O. L. Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, pages 133–152, Utrecht, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science, no. 615.

[2] B. W. Boehm and W. L. Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference 1992*, pages 63–82, April 1992.

[3] D. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[5] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[6] D. R. Group. Online material on adaptive programming, demeter/java, and APPCs. http://www.ccs.neu.edu/research/demeter/, 1998.

[7] 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*, pages 411–428, Washington, D.C., September 1993. ACM.

[8] I. M. Holland. Specifying reusable components using contracts. In O. L. Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, pages 287–308, Utrecht, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science, no. 615.

[9] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE5)*, pages 169–180, December 1992.

[10] G. Kiczales. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997. Invited presentation.

[11] G. Kiczales and C. V. Lopes. Aspect-oriented programming tutorial notes, July 1998. (From ECOOP '98.).

[12] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center, February 1997.

[13] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98*, 1998.

[14] D. Notkin, D. Garlan, W. G. Griswold, and K. Sullivan. Adding Implicit Invocagtion to Languages: Three Approaches. In S. Nishio and A. Yonesawa, editors, *Object Technologies for Advanced Software: Proceedings of the First JSSST International Symposium, Janazawa, Japan*, pages 489–510. Springer-Verlag, November 1993.

[15] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *Transactions on Software Engineering*, 20(10):760–773, Oct 1994.

[16] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, Santa Clara, CA, July 1994. IBM.

[17] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *TAPOS*, 2(3):179–202, 1996.

[18] H. Ossher and P. Tarr. Operation-level composition: A case in (join) point. In *Proceedings of the Third Workshop on Aspect-Oriented Programming*, 1998.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[20] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. To appear.

[22] K. J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems*. PhD thesis, University of Washington, Aug 1994.

[23] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. Feature engineering. In *Proceedings of the 9th International Workshop on Software Specification and Design*, pages 162–164, April 1998.

[24] M. VanHilst and D. Notkin. Using roles components to implement collaboration-based designs. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 359–369, San Jose, California, October 1996. ACM.

[25] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.