Chapter #

Multi-Dimensional Separation of Concerns and The Hyperspace Approach

Harold Ossher and Peri Tarr
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{ossher,tarr}@watson.ibm.com

Key words: Separation of concerns, software decomposition and composition, modularization, evolution, traceability, limited impact of change.

Abstract: Separation of concerns is at the core of software engineering, and has been for decades. This has led to the invention of many interesting, and effective, modularization approaches. Yet many of the problems it is supposed to alleviate are still with us, including dangerous and expensive invasive change, and obstacles to reuse and component integration. A key reason is that one needs different decompositions according to different concerns at different times, but most languages and modularization approaches support only one “dominant” kind of modularization (e.g., by class in object-oriented languages). Once a system has been decomposed, extensive refactoring and reengineering are needed to remodularize it.

Multi-dimensional separation of concerns allows simultaneous separation according to multiple, arbitrary kinds (dimensions) of concerns, with *on-demand remodularization*. Concerns can overlap and interact. This paper discusses multi-dimensional separation of concerns in general, our particular approach to providing it, called *hyperspaces*, and our support for hyperspaces in Java™, called Hyper/J™.

1. INTRODUCTION

Separation of concerns [par72] is at the core of software engineering. In its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Many different *kinds* of concerns may be relevant to different developers in different roles, or at different stages of the software lifecycle. For example, the prevalent kind of concern in object-oriented programming is *data* or *class;* each concern in this dimension is a data type defined and encapsulated by a class. *Features* [tur98], like printing, persistence, and display capabilities, are also common concerns, as are *aspects* [kic97], like concurrency control and distribution, *roles* [and92], *viewpoints* [nus94], variants, and configurations. We refer to a *kind* of concern, like class or feature, as a *dimension* of concern. Separation of concerns involves decomposition of software according to one or more dimensions of concern. Achieving a "clean" separation of concerns has been hypothesized to reduce software complexity and improve comprehensibility; promote traceability within and across artifacts and throughout the software lifecycle; limit the impact of change, facilitating evolution and non-invasive adaptation and customization; facilitate reuse; and simplify component integration.

These goals, laudable and important as they are, have not yet been achieved in practice. This is primarily because the set of relevant concerns varies over time and is context-sensitive—different development activities, stages of the software lifecycle, developers, and roles often involve concerns of dramatically different kinds and, hence, multiple dimensions. Separation along one dimension of concern may promote some goals and activities, while impeding others; thus, *any* criterion for decomposition and integration will be appropriate for some contexts and requirements, but not for all. For example, the data decomposition in object-oriented systems greatly facilitates evolution of data structure details, because they are encapsulated within single (or a few closely related) classes, but it impedes addition or evolution of features, because they typically include methods and instance variables in multiple classes. Further, multiple dimensions of concern may be relevant *simultaneously*, and they may overlap and interact, as features and classes do. Thus, modularization according to different dimensions of concern is needed for different purposes: sometimes by class, sometimes by feature, sometimes by viewpoint, aspect, role, or other criterion.

These considerations imply that developers must be able to identify, encapsulate, modularize, and manipulate multiple dimensions of concern simultaneously, and to introduce new concerns and dimensions at any point

during the software lifecycle, without suffering the effects of invasive modification and rearchitecture. Modern languages and methodologies, however, suffer from a problem we have termed the "tyranny of the dominant decomposition" [tar99]: they permit the separation and encapsulation of only one kind of concern at a time. Examples of tyrant decompositions are classes (in object-oriented languages), functions (in functional languages), and rules (in rule-based systems). It is, therefore, impossible to encapsulate and manipulate, for example, features in the object-oriented paradigm, or objects in rule-based systems. Thus, it is impossible to obtain the benefits of different decomposition dimensions throughout the software lifecycle. Developers of an artifact are forced to commit to one, dominant dimension early in the development of that artifact, and changing this decision can have catastrophic consequences for the existing artifact. What is more, artifact languages often constrain the choice of dominant dimension (e.g., it must be *class* in object-oriented software), and different artifacts, such as requirements and design documents, might therefore be forced to use different decompositions, obscuring the relationships between them. We believe the tyranny of the dominant decomposition is the single most significant cause of the failure, to date, to achieve many of the expected benefits of separation of concerns.

We use the term *multi-dimensional separation of concerns* to denote separation of concerns involving:

- Multiple, arbitrary dimensions of concern.
- Separation along these dimensions *simultaneously*.
- The ability to handle new concerns, and new dimensions of concern, *dynamically,* as they arise throughout the software lifecycle.
- Overlapping and interacting concerns; it is appealing to think of many concerns as independent or "orthogonal," but they rarely are in practice.

Full support for multi-dimensional separation of concerns opens the door to *on-demand remodularization,* allowing a developer to choose at any time the best modularization, based on any or all of the concerns, for the development task at hand.

Multi-dimensional separation of concerns represents a set of very ambitious goals. They apply irrespective of software development language or paradigm. No existing mechanism fully satisfies them, and much research remains to be done in pursuit of these goals. We believe that it is necessary to achieve them in order to overcome the problems associated with the tyranny of the dominant decomposition.

The remainder of this paper is organized as follows. Section 2 motivates the need for multi-dimensional separation of concerns by exploring an evolutionary scenario for a simple software system. Section 3 describes multi-dimensional separation of concerns. Section 4 introduces our approach, *hy-*

hyperspaces, and Section 5 describes Hyper/J™, our tool support for Java™, and illustrates its use on the evolutionary scenario. Section 6 discusses related work, and Section 7 conclusions and future work.

2. BACKGROUND AND MOTIVATION

To illustrate some of the serious and ubiquitous problems in software engineering that motivate our work, we begin by describing a running example involving the construction and evolution of a simple software engineering environment (SEE), first introduced in [tar99]. The SEE facilitates the development of fairly simple programs that consist solely of expressions. Expression programs constructed using the SEE are represented using abstract syntax trees (ASTs), as illustrated in *Figure 1*. This environment has a straightforward and commonly used architecture, also shown in *Figure 1*, in which a collection of tools operates on a shared data structure—the AST. Though the example is, of necessity, small and simple, it is typical of a broad class of real systems that involve multiple tools or applications manipulating wholly or partially shared domain models.

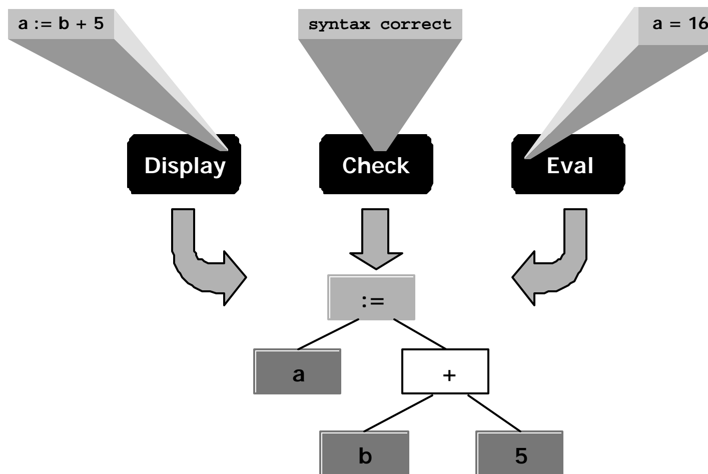

Figure 1. Tools and Shared AST in the Expression SEE.

2.1 The Scenario: A Matter of Concern in an SEE

The running example involves the initial creation of the SEE, and a series of evolutionary changes to it. We assume a simplified initial software development process, consisting of informal requirements specification in natural language, design in UML [rum98], and implementation in Java™ [gos96]. The initial requirements specification is straightforward:

> The SEE supports the creation and manipulation of expression programs. It contains a set of tools that share a common representation of expressions. The set of tools should include the following:
>
> – **Evaluation tool**: Determines the result of evaluating an expression and displays it.
>
> – **Display tool**: Depicts an expression program textually to a default display device.
>
> – **Check tool**: Checks an expression program for syntactic and semantic correctness.

A straightforward partial UML design for the SEE is shown *Figure 3*. This design uses a standard, object-oriented approach, in which a class is defined to represent each kind of expression AST node. Each class contains constructor, accessor and modifier methods, plus methods eval(), display(), and check(), which realize the required tools in a standard, object-oriented manner. The code is structured similarly.

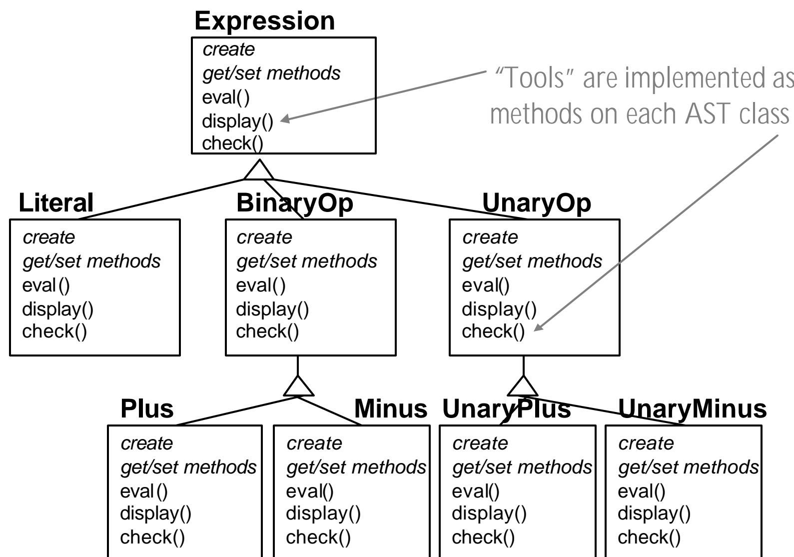

Figure 3. Partial UML Design for the Expression SEE

Even this simple example demonstrates several different kinds (dimensions) of concerns. These include:

Classes (or Objects): Each of the classes in the design and code represents one class concern.

Features: Particularly from the statement of requirements, we can decompose the software into four coherent features: the "kernel" AST, which includes the actual representation of expressions independently of any of the SEE tools; the display feature; the check feature; and the evaluation feature. Note that each feature includes the corresponding requirement specification, design elements, code, and test cases, since these all pertain to addressing that feature concern in the system.

Artifacts: Traditionally, different stages of the software lifecycle have produced different kinds of software artifacts. Some common ones are requirements specific ations, designs, code, and test plans.

As noted earlier, we refer to these different kinds of concerns as *dimensions of concern*. Informally, a dimension of concern is simply an approach to decomposing, organizing, and structuring software according to concerns of a particular kind. Note that, despite the clear presence of these different dimensions of concern, only a subset of them can be identified and encapsulated explicitly in the languages used in this example: artifacts, features within the requirements artifact, objects within the design and code artifacts.

After using the resulting SEE, the clients request some changes:

– It should be possible to have versions of the SEE that include subsets of the tools and capabilities.
– It should be possible to impose, optionally, checks for conformance to one or more programming styles.
– It should be possible to log, selectively, the execution of the SEE.

This set of modifications suggests the following set of concerns:

– **Configurations**: The first new requirement—to permit different variants of the SEE with different tool configurations—is essentially a request to be able to "mix and match" tools in the SEE. Thus, we can think of the SEE as representing a *family* of software [par76], where each member of the family contains some combination of tools.
– **Feature**: Style checking is a new concern in the feature dimension.
– **Logging**: Logging is not the same kind of "feature" as the SEE tools—it is not a coherent tool itself, and it may (optionally) affect some or all of the features during any execution of the SEE.
– **Design patterns**: While the initial version of the software was simple enough not to require any design patterns [gam94], some of the new requirements present opportunities to benefit from the extra flexibility that design patterns offer. For example, the logging capability could be mod-

eled readily using Observer. From the perspective of comprehensibility, it may be beneficial to look at software in terms of the design patterns from which it is architected [kel99].

Even without deep analysis, it is clear that making the changes to satisfy these rather simple requirements is by no means a simple matter with standard object-oriented technology. Allowing selection of features and addition of optional style-checking requires substantial reengineering, probably to introduce design patterns, like Visitor, that provide the needed flexibility. Support for logging requires invasive changes to every method to be logged, such as to perform the logging directly or to participate in Observer design patterns. More detailed analysis of a similar example appeared in [tar99].

2.2 The Tyranny of the Dominant Decomposition

The primary reason for these difficulties is that the new requirements deal with concerns that are not encapsulated in the original SEE. We can draw some important conclusions from even this simple scenario.

Different decompositions have different properties, both good and bad: Each dimension of concern promotes different subsets of the key software engineering properties noted earlier and, perhaps most importantly, facilitates different kinds of change. For example, objects are units of data abstraction; as such, they aid in isolating users from the details of data representation and implementation, thereby localizing the effects of representation changes. On the other hand, by-class decomposition results in two negative phenomena with respect to features: *scattering* and *tangling*. Features are *scattered* across multiple classes—e.g., each class in the AST design and implementation has its own display() method, the key method in the Display feature. The display method within a class is not isolated: it co-exists—is *tangled*—with methods supporting other features within the same class. Scattering and tangling complicate understanding of how particular functions or features are realized in the software. Further, any change to a function or feature, like Display, has high impact, because scattering implies that it entails modifications to some or all of the classes that define display() methods, and tangling implies that some of the modifications might inadvertently affect other features. Each of the dimensions of concern has both positive and negative software engineering characteristics; there is no "best" dimension for all purposes, which is one reason why different artifact languages are used for different purposes.

Different dimensions are useful for different reasons, at different times: This example demonstrates clearly two crucial points. First, the set of dimensions of concern, and the set of concerns within those dimensions,

vary over time. Note, for example, that the design patterns, configurations, and logging dimensions were not relevant to the initial software—they only became relevant with the introduction of new requirements. Second, the fact that each dimension of concern provides only a subset of desirable software engineering benefits means that developers will find different dimensions to be more or less useful, depending on the developer, his/her role, the stage of development, and the particular goals he/she wishes to accomplish. Thus, for example, adding the new style-check feature would be simple and additive if the software were decomposed by feature, but because features could not be encapsulated, the feature had to be retrofitted into the object dimension. These concerns do not match—they *cut across* [tar93, kic97] each other—so the modification is conceptually difficult, invasive, high-impact, and costly.

Anticipation causes ulcers: Deeply ingrained within software engineering is the notion of anticipating and designing for the "most likely" kinds of changes, towards the goal of limiting the impact of future evolution. Thus, for example, one could argue that a good developer would have anticipated the need for new features, like style checking, and might have designed the software with Visitors from the start, which would have facilitated the introduction of style checking. This is true, and we believe in anticipating and planning for changes whenever possible. Anticipation is not, however, a panacea for evolution. It clearly is not possible to anticipate all major evolutionary directions. Further, even if it were possible, building in evolutionary flexibility always comes at a price: it increases development cost, increases software complexity, reduces performance, or often all of the above. This observation holds for identifying dimensions of concern as well—even if it were possible to identify and encapsulate all possible dimensions, the resulting software would probably be even more complex and unwieldy than it is when only a small number of "dominant" dimensions are encapsulated.

Multiple dimensions, simultaneously: Artifact formalisms (such as programming languages, design notations like UML, etc.) in general provide only one (or a small number of) means of decomposing software—that is, they support only one dimension of concern. In fact, different artifacts for the same software may be written in languages with different "dominant" dimensions, leading to conceptual mismatches between artifacts and to poor traceability, which further complicates evolution. For example, requirements are often specified by function or by feature, as they were in this example—this is how the customers who specify the requirements think of the software—while object-oriented designs and code are decomposed using classes. Thus, developers must continuously translate between different expressions of the same concepts across different formalisms. Unless an artifact language specifically supports a given dimension, it is not possible for developers to identify, separate, and encapsulate concerns along that dimen-

sion in that particular artifact. And, as we have seen, if some kinds of concerns cannot be identified, encapsulated, and manipulated as first-class entities, the software engineering benefits that they might provide cannot be achieved. The "tyranny of the dominant decomposition" becomes oppressive whenever the concerns a developer has, at some point during the lifecycle, do not match any of the ones that have been, or can be, encapsulated. Its symptoms are the kinds of scattering, tangling, and cascaded, high-impact changes that this scenario demonstrates.

3. BREAKING THE TYRANNY: MULTI-DIMENSIONAL SEPARATION OF CONCERNS

The observations in the previous section lead to some important requirements on separation of concerns mechanisms. We use the term *multi-dimensional separation of concerns* to denote separation of concerns mechanisms that satisfy these requirements:

It is necessary for developers to be able to identify and encapsulate *any* kinds, or dimensions, of concern, *simultaneously*. Further, all dimensions must be created equal—there must not be "tyrant" dimensions that preclude decomposition along other dimensions.

Developers must be able to identify new concerns, and new dimensions of concern, *incrementally*, at any time during the course of the software lifecycle. For example, it must be possible to identify only some dimensions, or some of the concerns in a given dimension when the dimension is first introduced, and then identify or introduce others as they are needed, without having to rearchitect the software or make invasive modifications.

Developers must not be required to know about, or pay attention to, any concerns, or dimensions of concern, that do not affect their particular activities. One key purpose of separation of concerns is to reduce the amount of complexity a developer must deal with. Forcing all developers to know about all concerns would, instead, increase this complexity.

It must be possible to represent and manage *overlapping* and *interacting* concerns. As noted earlier, while independent or "orthogonal" concerns have particularly pleasing properties, overlapping and interacting concerns are at least as common in the real world. In representing such concerns, it must also be possible to identify the points of interaction and maintain the appropriate relationships across these concerns as they evolve.

Separation of concerns is clearly of limited use if the concerns that have been separated cannot be integrated together; as Jackson notes, "having divided to conquer, we must reunite to rule" [jac90]. Thus, any separation of

concerns mechanism must also include powerful *integration* mechanisms, to permit the integration of separate concerns.

An important additional goal, though not, in our opinion, a defining characteristic of multi-dimensional separation of concerns, is the ability to impose new decompositions on existing software (i.e., decompose it into concerns along a new dimension), without explicit refactoring, reengineering, or other invasive change. We call this capability *on-demand remodularization*. It allows a developer to choose, at any time, the best modularization for the development task at hand, without perturbing existing ones. In addition to reducing impact of change substantially, this feature opens the door to non-invasive system refactoring and reengineering.

There are potentially many ways to achieve multi-dimensional separation of concerns. As will be discussed in Section 6, there are a variety of modern mechanisms that break the tyranny to at least some extent. The rest of this paper describes our approach, called *hyperspaces*. The goals listed above are extremely challenging, however, and much research remains, for us and for others, before they are fully achieved.

4. THE HYPERSPACE APPROACH

We have developed a particular approach to multi-dimensional separation of concerns that we refer to as *hyperspaces*. Hyperspaces permit the explicit *identification* of any concerns of importance, *encapsulation* of those concerns, identification and management of *relationships* among those concerns, and *integration* of concerns. Many of the decisions we made in defining hyperspaces are aimed at achieving limited impact of change and simplified evolution. We deliberately left some detailed decisions open, to allow for variations with different tradeoffs. We describe hyperspaces in this section, and illustrate their use on the expression SEE example in the next section, in the context of Hyper/J™, a tool that supports hyperspaces for Java™.

4.1 Concern Space of Units

We begin by introducing some terminology that applies to separation of concerns approaches in general. Software consists of *artifacts*, which comprise descriptive material in suitable languages. A *unit* is a syntactic construct in such a language. A unit might be, for example, a declaration, statement, state chart, class, interface, requirement specification, or any other coherent entity that can be described in a given language. We distinguish *primitive* units, which are treated as atomic, from *compound units*, which group units together. Thus, for example, a method, instance variable, or per-

formance requirement might be treated as a primitive unit, while a class, package, or collaboration diagram might be treated as a compound unit.

A *concern space* encompasses all units in some body of software, such as a set of software systems or component libraries, or a product family. For example, a concern space for the expression SEE contains all of the software artifacts described in Section 2 for both the initial system and the extensions.

The job of a concern space is to organize the units in the body of software so as to *separate* all important concerns, to describe various kinds of interrelationships among concerns, and to indicate how software components and systems can be built and integrated from the units that address these concerns. We identify three distinct components to "separation" of concerns: *identification, encapsulation*, and *integration*. Identification is the process selecting concerns and populating them with the units that pertain to them.[1] Thus, for example, we can identify the "display feature" concern in the expression SEE as comprising the display requirement and all display() methods in the UML design diagrams and the Java™ code. Identification is useful, but to fully realize the benefits of separation of concerns, the concerns must also be *encapsulated* such that they can be manipulated as first-class entities. A Java™ class is an example of an encapsulated concern. The display feature is not an encapsulated concern in Java™, however, as its units are scattered across many Java™ classes. Once concerns have been encapsulated, it must be possible to *integrate* them to create software that addresses multiple concerns. In standard Java™, classes are integrated simply by loading them; a combination of import specifications and the class path determines their relationships. Concerns other than classes and interfaces cannot be integrated in standard Java™.

4.2 Identification of Concerns: The Concern Matrix

A *hyperspace* is a concern space specially structured to support our approach to multi-dimensional separation of concerns. Its first distinguishing characteristic is that its units are organized in a multi-dimensional matrix. Each axis represents a dimension of concern, and each point on an axis a concern in that dimension. This makes explicit all the dimensions of interest, the concerns that belong to each dimension, and which concerns are affected by which units. The coordinates of a unit indicate all the concerns it affects;

[1] Note that concern identification can be done either top-down or bottom-up, depending on the stage of the software lifecycle. During design activities, concerns may be selected first, and then units may be developed based on the concerns that were selected. During system evolution, units may already exist when new concerns are identified. In this case, the identification process determines which existing units address the new concerns.

the structure clarifies that each unit affects exactly one concern in each dimension. Each dimension can thus be viewed as a partition of the set of units: a particular software decomposition. Any single concern within some dimension defines a hyperplane that contains all the units affecting that concern. The matrix structure permits uniform treatment of all kinds of concerns, and it allows developers to navigate or slice through the matrix according to any desired concerns. We believe that the concerns within a dimension, though disjoint, need not be unrelated, and we expect some concern structure (e.g., hierarchies) within dimensions to be valuable [oss88, kim99]. This remains an issue for future research.

Some dimensions of concern naturally partition the concern space. For example, if every unit in a system addresses exactly one feature, then the Feature dimension naturally partitions the units. However, some units in a system may not pertain to any "feature" at all, such as an error-reporting routine in the SEE. To handle this situation, each dimension in a hyperspace has a specially-designated "none" concern, containing units that are not of interest at all from the perspective of that dimension.

4.2.1 Units

Hyperspaces can be used to organize and manipulate units written in any language(s), though, of course, tool support is often language-specific. To date, we have worked only with units at the granularity of declarations (e.g., methods, functions, classes, UML diagrams) rather than lower-level constructs, such as statements or expressions. We believe, however, that hyperspaces can be extended to handle finer-grained units in a disciplined fashion; this remains an issue for future research.

4.2.2 Concern Specifications

Concern specifications in hyperspaces serve to identify the dimensions and their concerns, and to specify the coordinates of each unit within the matrix. A simple approach is to use a *concern mapping* consisting of specifications of the form

```
x: dimension.concern
```

where *x* is the name of a unit or a collection of units (e.g., a directory or package), or a pattern representing many units or collections of units. Examples of concern mappings for Java™ units are given in Section 5.

In general, concern specifications can be more complex, and can specify the "meaning" of each dimension and concern formally or informally. There are two styles of specification. *Extensional specifications* explicitly enumerate the units in each concern. *Intensional specifications* specify properties of

concerns and units that can be used to determine whether a given unit pertains to a concern. Intensional specifications have the advantage of conveying intent more explicitly, and of being able to accommodate changes to the underlying set of units without manual intervention.

4.3 Encapsulation of Concerns: Hyperslices

The concern matrix identifies concerns and organizes units according to dimensions and concerns. It allows many useful sets of units to be identified based on the concerns they affect, such as all units pertaining to a single concern, or to all of several concerns (areas of overlap), or to one concern but not another. However, the matrix does not, in itself, support encapsulation of concerns: the sets of units cannot simply be treated as modules, without additional mechanism. In hyperspaces, that additional mechanism is *hyperslices*: sets of units that are *declaratively complete*.

Units are typically related in a variety of ways; for example, one function unit may *invoke* another, or it may *define* or *use* a variable declaration unit. When these kinds of interrelationships exist between units in different concerns, high coupling results. To decouple them, hyperslices are defined to be *declaratively complete*: they must *declare* everything to which they refer. For example, a hyperslice must, at minimum, include a declaration for every function that any of its members invokes, and for any variable its members use. The hyperslice need not provide a full *definition* for these declarations—e.g., it may declare a function without providing an implementation. Thus, declarations can be abstract, specifying (partially or fully, formally or informally) the properties upon which the hyperslice relies.

Declarative completeness is important because it eliminates coupling between hyperslices. Instead of one hyperslice referring to another, thereby depending upon the other specific hyperslice, each hyperslice states what it needs by means of the abstract declarations, thereby remaining self-contained. It does, however, require someone to provide full definitions of the abstractly-declared entities to be fully complete, but *any* appropriate hyperslice(s) can provide these, as will be shown in Section 4.5. This approach therefore fosters flexible configuration and reuse of hyperslices, and is crucial to achieving limited impact of change.

For example, suppose a Display hyperslice contains a unit, Plus.display(), which uses a Plus.getOperand() accessor function, defined in a Kernel hyperslice. To make Display declaratively complete, it must be augmented with its own declaration of Plus.getOperand() (without necessarily implementing it). Plus.display() must then refer to this local declaration, instead of to the accessor function in the Kernel. This eliminates the coupling between

Display and Kernel, in favor of the assertion that the new, abstract declaration must eventually be "bound" to a unit in *some* hyperslice that provides a suitable implementation.

Any set of units can be fashioned into a valid hyperslice by *declaration completion*: providing abstract declarations for everything referenced but not declared within the set. To some extent, this process can be performed automatically, using straightforward (though language-specific) analysis. Automatic declaration completion can determine what declarations are needed, and can create valid declarations. Semantic information associated with declarations—formal or informal specifications—is another matter however, and probably requires human intervention. Specifications on declarations, and the extent to which they can be determined automatically by analysis during declaration completion, remain issues for future research.

Since any set of units can become a hyperslice through declaration completion, arbitrary concerns can be encapsulated using hyperslices. Thus, whatever limitations the underlying artifact language(s) has, and whatever the concern, it is always possible to synthesize a hyperslice that contains just those units pertaining to the concern (plus some abstract declarations).

4.4 Relationships among Concerns

Units, concerns and hyperslices do not exist in isolation; they can be interrelated in a number of different ways. For example, the "display feature" and the "expression class" are related in that they *overlap*—they share some of the same units, as the display() method is part of both concerns—so a change to one concern may affect the other. As another example, we might choose to integrate "syntax check" and "style check" hyperslices to produce a "check" feature that performs both syntax and style checks. In this case, these two hyperslices would be related by one or more *integration* relationships that indicate how they are to be combined.

We can identify two distinct classes of relationships: *context-insensitive* and *context-sensitive*. "Overlap" is an example of a context-insensitive relationship—the "display feature" and "expression class" are always related this way, as long as they share units in common. Integration relationships exemplify context-sensitive relationships—the "syntax check" and "style check" concerns only have this relationship if they are being integrated in some context (e.g., to create a check tool), but the relationship is not inherent in their definition. Other common kinds of concern relationships are "generalizes," "subsumes," and "precludes." Hyperspaces permit the identification and representation of both context-insensitive and context-sensitive relationships, and their use in analysis (e.g., impact of change) and integration.

4.5 Integration of Concerns: Hypermodules

Hyperslices are building blocks; they can be integrated to form larger building blocks and, eventually, complete systems. For example, to create a working SEE containing the Display hyperslice discussed above, Display must be integrated with some other hyperslice that provides a unit that can be bound to the new, abstract declaration of Plus.getOperand(), to provide an implementation. We refer to this kind of "binding" relationship between units as *correspondence*. Correspondence is a context-sensitive relationship. It occurs within the context of the integration of a particular software component or system—the same declaration unit may be associated, for example, with different implementation units in different systems. In a hyperspace, this integration context is a *hypermodule*.

A *hypermodule* comprises a set of hyperslices being integrated and a set of *integration relationships*, which specify how the hyperslices relate to one another, and how they should be integrated. Correspondence is an important integration relationship, indicating which specific units within the different hyperslices are to be integrated with one another. However, additional details are often needed to specify just how the integration is to occur. For example, if two methods correspond, should one override the other in the integrated system, or are they both to be executed? If both, in what order, and how should the return value be computed? If the types of their parameters do not match, what transformations are needed to reconcile them? In the example above, it is sufficient to integrate the corresponding declarations of Plus.getOperand() in Display and Kernel, which results in the Kernel implementation being called by Plus.display() at run time.

Conceptually, and often in practice through use of a *compositor* tool, the integration specified by integration relationships can actually be performed to produce a set of integrated units. This set will be declaratively complete, and is therefore a hyperslice. A hypermodule can therefore be thought of as a composite hyperslice, produced by integrating a number of subsidiary hyperslices. This implies that hypermodules can be nested, allowing large systems to be built by successive integration.

Declarative completeness, correspondence, and even the more detailed integration relationships, represent fairly loose forms of binding, which promotes evolvability. Since hyperslices do not depend on each other directly, software artifacts are subject to a *completeness constraint* in which each declaration unit in a system must correspond to compatible definition(s) or implementation(s) in some hyperslice(s). Replacing a definition or implementation is non-invasive on hyperslices; it merely requires the redefinition of integration relationships. Correspondence thus provides great flexibil-

ity and directly supports substitutability, including mix-and-match and plug-and-play. Completeness constraints can be imposed as needed (e.g., on code, to ensure that it can run), but they are not necessary when a hypermodule represents a building block (e.g., a reusable component or framework), whose remaining needs can be satisfied through future integration.

Different types of correspondences can occur, beyond the association between declarations and definitions. For example, a requirements unit may correspond to one or more design units that satisfy it; or a unit implementing the eval() function may correspond to a unit that encapsulates code to check for a previously cached result before evaluating. Correspondence and other relationships are deliberately left abstract in this model, as their details depend on many factors, including the language(s) in which units are written, which constructs are treated as primitive and compound units, the extent of environment support for correspondence, etc. Our intent is to provide an abstract model within which multiple semantics for correspondence and multiple realizations of hyperspaces can be specified. Correspondence relationships in Hyper/J™, described in Section 5, extend the *composition rules* from our earlier work on subject-oriented programming [oss96].

Clearly, the issue of whether corresponding units are "compatible" (e.g., whether an implementation unit satisfies a declaration unit's requirements, or whether a design unit satisfies a requirement) involves both syntactic and semantic issues. How to characterize and check for such compatibility remains an issue for future research. Even once resolved, however, we expect checking to be semi-automatic in general; ultimately, software engineers must understand enough about corresponding units to determine whether or not they are compatible and how best to integrate them.

Hypermodules can be used to encapsulate many kinds of software artifacts, components, and fragments thereof, and to integrate them in different ways. For example, an entire artifact, like a requirements specification, a design, or code, can be modeled as a hypermodule. A software system as a whole is also a hypermodule, subject to the completeness constraint. A system hypermodule might consist of a hyperslice for each artifact, with composition relationships describing how the artifacts interrelate; they might, for example, indicate how particular design and code units elaborate given requirements units. Alternatively, it might consist of a subsidiary hypermodule for each feature, with integration relationships specifying how the features interact. Each feature hypermodule, in turn, consists of a hyperslice for each artifact, with integration relationships as above.

5. HYPER/J™: HYPERSPACES FOR JAVA™

We have implemented a tool, called Hyper/J™ [hyp99], which supports hyperspaces. It currently supports one language for defining units—Java™—and we have begun looking at incorporating UML also [cla99]. In this section, we describe Hyper/J™ and illustrate its use by describing its application to the development and evolution of the expression SEE example. We conclude this section by summarizing how Hyper/J™, and the model of hyperspaces it embodies, overcome the problems described in Section 2 to achieve the software engineering goals noted in Section 1.

5.1 The Tool

Hyper/J™ permits the identification, encapsulation and integration (though composition) of multiple dimensions of concern, and it realizes the model of hyperspaces presented in Section 4. It takes as input a *project specification*, which identifies the units (Java™ code) in a given hyperspace; a *concern mapping*, which describes how the units are organized in the concern matrix; and a *hypermodule specification*, which describes hypermodules and controls composition. We will describe these in more detail below. Hyper/J™ can be used at all stages of the software lifecycle, for initial development as well as for extension or evolution of software initially developed with it or without it.

Hyper/J™ includes visual, WYSIWYG support for specifying and integrating concerns. Though its support is still preliminary at present, Hyper/J™ allows developers to identify and manipulate concerns, to focus in on particular dimensions of concern, and to create hypermodules by trial-and-error integration of concerns. A developer starts creating a hypermodule by choosing a set of concerns and an overall default relationship among those concerns (such as "mergeByName," described below). Hyper/J™ creates valid hyperslices for the concerns by automatic declaration completion, and composes them based on the specified relationship(s). The resulting composed hyperslice is displayed. If it is not as desired, the developer can specify new relationships, and examine the new result. S/he can also modify the original concerns; Hyper/J™ will automatically recompute the relationships (deactivating, but not deleting, any that no longer apply) and create a new composed hyperslice that, in many cases, is either correct or close to what is desired. This interaction continues until the composed hyperslice meets the developer's requirements.

The development of Hyper/J™ was influenced by some important design goals, intended to foster easy, *incremental* adoption. First, we did not want

to require developers to adopt new programming languages, or to use special-purpose compilers or virtual machines. We therefore implemented Hyper/J™ to work on and generate standard Java™ class files. All the support for multi-dimensional separation of concerns occurs *outside* the artifact language, Java™. Second, we wanted Hyper/J™ to provide useful benefits when applied to standard Java™ programs, and additional benefits when applied to programs written with Hyper/J™ in mind. It is therefore able to identify, encapsulate and integrate concerns from standard Java™ programs, without requiring special coding conventions or packaging.

5.2 Developing with (and without) Hyper/J™: The Expression SEE in Hyperspace

To convey a sense of the different ways in which developers can leverage Hyper/J's capabilities throughout the software development lifecycle, we present here part of the development process of the expression SEE. Only small illustrations of code are shown here; the full, runnable code for the SEE example is available at the hyperspace web site [hyp99].

5.2.1 Initial Development, without Hyper/J™

To illustrate incremental adoption of Hyper/J™, we assume that the initial SEE was developed using standard object-oriented design and implementation techniques, without Hyper/J™, to produce the design and code shown in *Figure 3* (Section 2.1). Feature concerns are not identified or encapsulated within this code.

5.2.2 Mix-and-Match of Features (and Developing Product Lines) with Hyper/J™

The first change in the requirements entailed permitting the creation of different versions of the expression SEE, each with different subsets of features. Hyper/J™ can help here in two ways. First, it provides on-demand remodularization—the ability to identify and encapsulate new dimensions of concern at any time, without invasive changes. Thus, developers can introduce the needed feature concerns using Hyper/J™, and then manipulate those concerns as first-class entities. Second, Hyper/J's composition capability permits the *selective* integration of concerns, and hence creation of variants of the expression SEE that integrate different subsets of the available features, as needed, non-invasively.

To use Hyper/J™ to accomplish this task, a developer performs the following steps:

Create a project specification: Developers create hyperspaces initially by specifying a set of Java™ class files that contain the code units that will populate the hyperspace. This is analogous to creating a project or a repository in an integrated development environment. In the Hyper/J™ GUI, developers can choose to add class files at any time using a browser; when using the batch system, developers write a *project specification,* such as:

```
hyperspace Expression_SEE_Hyperspace
   class com.ibm.hyperJ.ExpressionSEE.*;
   file c:\u\smith\com\ibm\hyperJ\util\Set.class
```

Project specifications can contain wildcards, as shown above, and can use either Java™ fully qualified class names or path names of files.

Hyper/J™ automatically creates one dimension—the *Class File* dimension—and it creates one concern in that dimension for each class file it loads. The contents of those concerns are the units (interfaces, classes, methods, and member variables) in the corresponding class files.

Create concern mappings: To achieve the mix-and-match of features that is desired, the developer must first encapsulate the features as first-class concerns. S/he does this by creating a new dimension—the Feature dimension—and describing how existing units in the hyperspace address concerns in that dimension. To do so, s/he specifies *concern mappings,* such as:

```
package com.ibm.hyperJ.ExpressionSEE: Feature.Kernel
operation display:                    Feature.Display
operation check:                      Feature.Check
operation eval:                       Feature.Eval
```

The first mapping indicates that, by default, all of the units contained within the Java™ package com.ibm.hyperJ.ExpressionSEE address the Kernel concern in the Feature dimension. Since the Feature dimension does not yet exist, Hyper/J™ will create it (and the Kernel concern) upon processing this concern mapping. The other three mappings indicate that any methods named "display," "check," or "eval" address the Display, Check, or Eval features, respectively. These later concern mappings override the first one, whenever they apply. This illustrates an approach employed throughout Hyper/J™: specification of a general rule followed by exceptions, to clarify and shorten specifications.

The concern matrix now contains two dimensions: Class File and Feature. Each unit addresses exactly one concern in every dimension. Thus, for example, the method Expression.display() addresses the concern com.ibm.hyperJ.ExpressionSEE.Expression in the Class File dimension, and the Display concern in the Feature dimension.

Create hypermodules: Once the feature concerns have been identified, the developer can create versions of the SEE that contain different sets of features by defining *hypermodules*. A *hypermodule specification* comprises:

– A set of hyperslices, specified in terms of concerns identified in the concern matrix. Examples include individual concerns, and unions, intersections or complements of concerns. Hyper/J™ performs declaration completion automatically to create valid hyperslices.
– Integration relationships among the hyperslices and their units. General rules for determining relationships can be followed by exceptions.

For example, the following hypermodule specification defines a version of the SEE that contains the Kernel, Display, and Check capabilities:

```
hypermodule SEE_With_Display_And_Check
   hyperslices: Feature.Kernel, Feature.Display,
                Feature.Check
   relationships:  mergeByName
```

In this hypermodule, the Kernel, Display, and Check concerns are related by a "mergeByName" integration relationship. The "ByName" indicates that units in the different concerns are considered to correspond if they have the same names (and signatures, where appropriate). The "merge" indicates that corresponding entities are to be combined so as to include all their details; for example, all members in corresponding classes are brought together in the composed class. This integration relationship, and many of the others in Hyper/J™, are based on composition rules defined for subject-oriented programming [oss96].

The hyperslice that results from composing these concerns contains all the AST classes, but with just Kernel, Display and Check functionality in each. In particular, no eval() methods are present.

When the developer is satisfied with the composed hyperslice, s/he can ask to have code generated. Hyper/J™ generates Java™ class files and composed pseudo-source files for any composed hyperslice. The class files can then be executed (or otherwise used) like any other Java™ classes, and the pseudo-source files can be used as "source" in debugging sessions. In the example, the composed program represents an executable version of the expression SEE that contains the Kernel, Display, and Check functionality only. Other versions of the environment can be created similarly, by defining new hypermodule specifications and using composition. Note that creation of hypermodules is entirely non-invasive, unlike the retrofitting of design patterns usually needed to achieve a similar result.

This part of the scenario has demonstrated the utility of Hyper/J's on-demand remodularization and integration capabilities on *existing* code. Notice that the feature concerns did not have to be identified or separated during initial development to permit them to be encapsulated. Also notice

that each of the concerns is itself a reusable component that can be integrated in different contexts with different other concerns—none of them is coupled with any other. These properties imply powerful support for development and configuration of variations within product lines or families.

5.2.3 The Addition of Style Checking

The expression SEE clients eventually requested an enhancement that permits optional style checking of expression programs. Hyper/J™ allows the new feature to be developed separately from the existing features, and non-invasively. That is, developers can write the code for the new feature as a new, separate Java™ package (or packages), which we call a *hyperslice package*, because it is deliberately written to encapsulate a concern. They will then be able to integrate this package with other concerns as needed. This adds tremendous flexibility to the code architectures that developers can select, and to the range of software development processes they can use.

Figure 5 helps to illustrate this point. It depicts the code in the new package that realizes the style checking feature. Notice that the package contains *solely* the code needed to implement the style checking feature (plus abstract declarations, not shown, for anything "foreign" that is used, such as accessor methods from Kernel). Its class structure is similar to that of the original system (*Figure 3*), but not identical, because style checking only affects some of the Expression classes. This is an important feature of multi-dimensional separation of concerns using Hyper/J™: that different concerns can have different perspectives on, or views of, the domain model under development. These different views can later be reconciled by specifying the appropriate relationships between the concerns.

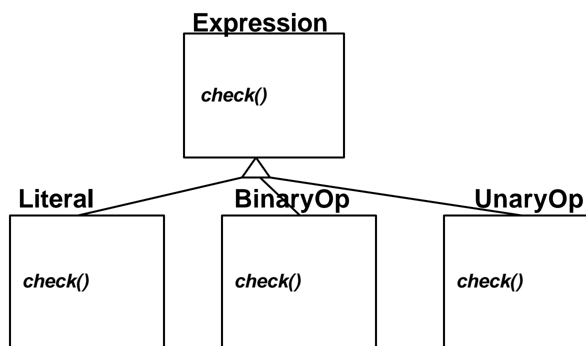

Figure 5. The Style Checking Hyperslice Package

Once the code in *Figure 5* is complete and has been compiled with any Java™ compiler, the developer can add the resulting class files to the existing hyperspace. This results in the automatic addition of new concerns in the Class File dimension—one for each of the class files in the style checker package. The developer can then define an additional, simple concern mapping to create a Style Checker feature concern:

```
package com.ibm.hyperJ.StyleChecker.* : Feature.StyleChecker
```

With the style checking feature now identified as a concern, the developer can create variants of the expression SEE that contain style checking or not, as desired, in much the same way as s/he can mix-and-match the other features, described earlier.

The addition of style checking has demonstrated an important feature of Hyper/J™. As shown in Section 5.2.2, developers need not use Hyper/J™ during initial development—they can use it after development—but if they choose to use it during initial development of some part of the system, they can achieve separation of concerns, and code architectures, that would be difficult or impossible to achieve using standard object-oriented techniques. The extra flexibility does not derive from the use of new languages or paradigms—the style checker, for example, was written as a standard package in Java™—but, instead, from the integration (composition) features of Hyper/J™. It has many important advantages and uses, including:

– The ability to treat hyperslice packages as reusable components. When hyperslice packages are used in new contexts, the composition relationships (possibly referring to special-purpose glue code) can include any adaptation that might be necessary (white-box reuse).
– The ability to structure code and design along the same lines as requirements [cla99], thereby enhancing traceability, by encapsulating the code that realizes a particular requirement in one or more hyperslice packages.

5.2.4 Retrofitting a Design Pattern for Logging

The final change we will explore is the addition of optional logging (or debug tracing) throughout the expression SEE. This modification entails making some or all methods in various classes or features print log messages upon method entry and exit.

Clearly, the logging capability is not specific to the expression SEE—it makes no reference to any expression classes or methods, and the same logging capability could be used in multiple contexts. Thus, the developers determined that they already had a pre-existing, generic, reusable logging component that they could use to satisfy the new end-user requirement. Their reusable component library contains an implementation of the Observer design pattern, along with a particular instantiation of that pattern to

implement logging, as shown in *Figure 7*. In this case, the developers use Hyper/J™ to retrofit the logging capability, which is already encapsulated in a separate hyperslice package (the reusable library), by integrating it into the SEE. Hyper/J™ permits them to make this change additively. The developer simply loads the existing component class files into the hyperspace with the rest of the system, and specifies an appropriate concern mapping to create a new Feature concern, which s/he calls Logging.

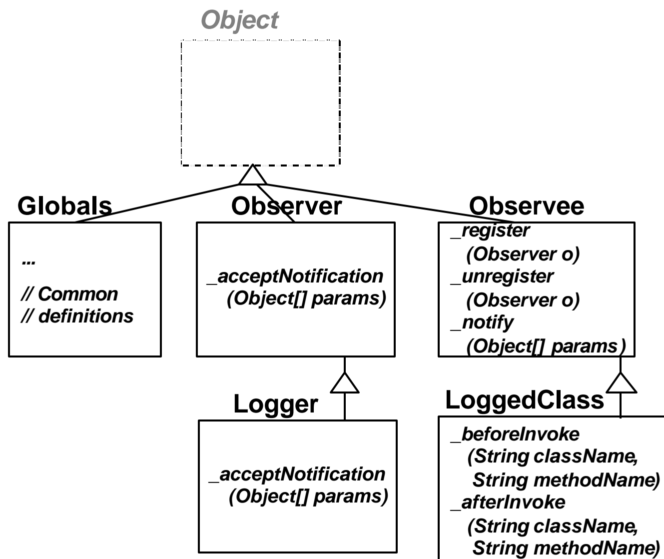

Figure 7. The Logging Hyperslice Package

To instrument methods in the existing code, the developer defines a hypermodule to integrate the Logging feature with any concerns that s/he wants to be logged. For example, to create a version of the expression SEE that contains the Kernel, Evaluation, and Syntax Check features, with these features logged, s/he might define the following hypermodule:

```
hypermodule SEE_With_Logged_Eval_And_Check
   hyperslices: Feature.Kernel, Feature.Check,
                Feature.Eval, Feature.Logging
   relationships:
      mergeByName;
      merge Feature.Logging.LoggedClass with *;
      bracket ~_* with
         _logEntry(ClassName, MethodName)
         _logExit(ClassName, MethodName)
```

The "merge" relationship expands on the "mergeByName" relationship; it indicates that the LoggedClass unit in the Logging concern from the Feature dimension is to be merged with *all* other class units in the other hyperslices, even though their names differ. Thus, for example, LoggedClass will be merged with the Expression class in Feature.Kernel. Only the same types of units are merged, classes in this case. The "bracket" relationship indicates that each method whose name does not begin with an underscore should be *bracketed* by the methods _logEntry and _logExit. Thus, for example, each display() method in the composed hyperslice will call _logEntry upon entry and _logExit before exit. The parameters passed to these bracketing methods will be the names of the class and method, enabling the logger to identify the method called. The bracket relationship is very useful for circumstances, like this one, where developers need to add behavior to the beginning and/or end of methods.

Notice that this development scenario entailed the integration of generic, reusable components—the Observer design pattern and logging—into an existing system that had not been designed to use them. This is a common problem for developers, and it occurs in many forms, at all stages of software development—for example, integrating a commercial-off-the-shelf database or library component into software during initial development, or retrofitting a design pattern or other component into the software during the course of evolution. Hyper/J™ facilitates a wide range of such integration activities. The same mechanisms can be used both for integration and customization, as this example shows.

We note that the multi-dimensional approach permits integration and customization using *any* concerns, in any dimensions. Thus, for example, while the developers chose to add logging to a subset of *features*, they could equally well have decided to add it to a subset of *classes*, or to some mix of features and classes. The only difference is in the set of hyperslices specified in the hypermodule. This ability to treat all concerns as equal provides developers the ability to focus their attention on precisely the part of a system that they care about to accomplish their goals.

5.3 Evaluation: Achieving the "ilities"

The development and evolution scenarios just described demonstrate some of the ways in which developers can use Hyper/J™ to facilitate a broad range of common software engineering activities throughout the software lifecycle. We conclude here by evaluating Hyper/J™ briefly with respect to how well it helps developers to achieve the desirable "ility" properties described in Section 1.

Comprehensibility: By their structure, hyperspaces enable software engineers to focus in on any dimensions and concerns of importance that are represented in the hyperspace—they need only examine the hyperplane containing the concern. This facilitates comprehensibility. Further, the ability to define new concerns on demand permits developers to identify concerns on an ongoing basis, as they arise during the course of the software lifecycle. This reduces the initial design and development burden on developers by not forcing them to identify *all* concerns up front that might possibly be important at some point in the software lifecycle. It also improves comprehensibility by permitting developers to identify concerns only when they actually do become important, rather than imposing the cognitive burden of identifying and separating all potentially useful concerns in advance.

The ability to separate, simultaneously, all concerns of importance enables developers to focus on the interactions among concerns and to identify (and encapsulate) new concerns. Hyperspaces make explicit the ways in which concerns affect one another, based on how they intersect and on the interrelationships in which they are involved. These interactions are extremely important for evaluating impact of change on other concerns when working with a particular concern. Concern intersections also often turn out to be useful concerns themselves (e.g., style checking of binary operations, an intersection of feature and class concerns). Hyperspaces provide for the identification and reification of such concerns. Further, the structure of a hyperspace aids the identification of other types of "hidden" concerns. For example, the definition of dimensions precludes situations where two concerns in the same dimension overlap. This property helps identify many kinds of potential encapsulation errors, ranging from failure to identify concerns, to poor separation of concerns, to coupling of concerns.

Evolvability, limited impact of change, and traceability: Hyperspaces greatly facilitate many aspects of evolution. First, they enable *additive*, rather than *invasive*, extension, customization, and extraction of hyperslices, often even when the changes were not anticipated, as the example shows. This is a significant part of the goal of achieving limited impact of change. Second, the addition of units, concerns, and dimensions to hyperspaces is clearly straightforward, with little impact on existing concerns. Moreover, the process of adding units to a hyperspace (and of defining new concerns), forces developers to determine how the new units (concerns) affect existing concerns (units), even if by only indicating that the new units (concerns) do not affect any existing concerns (units).

Hyperspaces also help to limit the impact of removing concerns, which is typically the most invasive and high-cost evolutionary activity. A common problem in removal is that such changes tend to cascade throughout large

parts of a software system. By including the declarative completeness requirement as part of the definition of hyperslices, we have ensured that removal of units, and hence, concerns, can, at worst, affect nothing more than the set of hypermodules in which those units and concerns played explicit roles in relationships. This is because the model eliminates direct dependencies among hyperslices by using declarative completeness. Thus, while removal of a unit from a hyperslice, or a hyperslice itself, from a hyperspace may end up eliminating units with which units in other hypermodules had been *related*, the breaking of these can be followed, *non-invasively*, by the establishment of new relationships to other units in other hyperslices that fulfill the intended semantics of a given concern relationship. The impact of removal can, therefore, be limited to the particular hyperslice it affects and to the relationships within hypermodules. The "dangling relationships" that result from removing units can also be used to identify concerns that might not have been separated appropriately, and to identify other concerns that should also be removed (e.g., when removing a display requirement, we would certainly want to remove any design and code concerns that satisfy the requirement). Thus, the model promotes both traceability and limited impact of change.

6. RELATED WORK

In a prior paper [tar99], we discussed many modern approaches that introduced novel modularization mechanisms related to multi-dimensional separation of concerns: subject-oriented programming [har93,oss96], aspect-oriented programming [kic97], contracts [hol92], role models [and92, van96], adaptive programming [mez98], Viewpoints [nus94] and Catalysis [dso98]. All of these, except Viewpoints, pertain to object-oriented systems, in which the dominant decomposition is by class. Each introduces a mechanism, analogous to hyperslices, to segregate design or code that addresses other, non-class concerns. All these approaches provide some of the benefits of hyperslices, in terms of identification and encapsulation of concerns that are not in the dominant decomposition dimension. Many of these approaches also provide some of the benefits of hypermodules—some degree of flexibility in composition of concerns along some useful dimensions [tar99]. As such, they all make valuable contributions by satisfying some of the goals of multi-dimensional separation of concerns, but none of them satisfies all.

One key distinguishing characteristic of hyperspaces relative to all other mechanisms known to us is the support for on-demand remodularization: the ability to extract hyperslices to encapsulate concerns that were not separated in the original software artifact. This lowers the entry barrier, greatly facili-

tates evolution, and opens the door to non-invasive refactoring and reengineering. Other important characteristics of hyperspaces that help to differentiate them from other approaches include:

– Hyperspaces do not restrict the nature or number of dimensions of concern permitted. They allow new concerns and dimensions to be added at any time, and these can apply to existing units, using on-demand remodularization, or to new units.
– Hyperslices are grouping constructs that collect together all software that pertains to a particular concern. Contracts and role models are similar, but not aspects, composition filters or propagation patterns, which are finer-grained, each dealing with part of a concern (e.g., methods that share a weave specification or the filtration needs of a particular object).
– Integration and other kinds of relationships are separate from artifacts. This reduces coupling, which has many benefits. Hyperslices are reusable; different integration relationships can be used in different contexts to specify details of how they are to be reused in those contexts. Separate relationship specifications also permit the hyperspace approach to be applied without changing artifact languages. Composition filters are similar, in that attachment is specified separately from filter code, whereas aspects contain weave specifications and code bundled together.
– Hyperslices are declaratively complete, separately understandable, and reusable. It is possible to understand a hyperslice in isolation, and conceptually possible to specify its semantics. It is also conceptually possible to understand a hypermodule by combining one's understanding of the component hyperslices and the composition relationships; it is not necessary to create the composed artifact and examine it. Details of semantic specification of hyperslices and composition remain an important area of future research.
– Hyperspaces support primitive units at the granularity of declarations (e.g., functions or members). This is a limitation, but one that simplifies understanding of hyperslices and hypermodules, as well as implementation of composition [oss98, tar99].
– Concerns can span lifecycle phases. Many of the details of making this a practical reality remain to be worked out.

Hyperspaces also relate to, and incorporate results from, other areas of related work. Loose binding is an accepted means of helping to limit the impact of some forms of change. Work in the area of software architecture (e.g., [sha96, all97]) has identified the need to separate software *components* (like hyperslices) from *connectors* (like relationships). Similarly, earlier work on Precise Interface Control (PIC) [wol89] identified benefits of representing a particular kind of inter-component interaction: *provides* and *re-*

quests. The declarative completeness requirement and use of separately specified composition relationships are in the spirit of these, and similar, approaches. Barrett et. al. [bar96] describe a spectrum of mechanisms to achieve connections among components, ranging from tightly to loosely bound, and from early to late binding. We have attempted to choose a point on this spectrum that balances the need to limit the impact of change (by not permitting software components to know about each other) with the need for analyzability (most readily accomplished in the presence of tighter binding).

7. CONCLUSIONS AND FUTURE WORK

A number of important problems in software engineering have resisted general solution, including problems related to the "ilities:" comprehensibility, traceability, and evolvability. We believe that these problems share a common cause: failure to identify and encapsulate, *simultaneously*, all concerns of importance in a software system, and the inability to use different dimensions of concern for different purposes throughout a system's lifetime. This paper presented multi-dimensional separation of concerns as an ambitious set of goals that need to be achieved to address these problems fully. It also presented our approach to achieving them, called *hyperspaces*, and its realization in tool support for Java™, called Hyper/J™.

Hyperspaces permit the identification, encapsulation, and integration of any kinds of concerns in standard software, either during initial system development or in retrospect, as the need arises during the course of evolution. They allow the set of concerns of interest to grow and change. They permit explicit representation of relationships among units, concerns and dimensions, loose coupling among concerns, and on-demand remodularization. They even allow for the representation of concerns that span the software lifecycle—for example, the "display" concern has requirements, design, code, and test module units associated with it. Because developers have the choice of when and how to apply hyperspaces, hyperspaces do not interfere with existing software processes—though developers may choose to modify their processes to take advantage of the extra flexibility hyperspaces give them—and the entry barrier for developers to use them is quite low.

This work is clearly at an early stage, largely unproven in practice as 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, and hence an important area for research. Many questions remain, such as: What sorts of concerns are important in real software development projects? What are their structure and the nature of their interactions, for concerns both within and across software ar-

tifacts? How can one understand, and perhaps specify, encapsulated concerns and their integration? What mechanisms are needed to achieve the goals of multi-dimensional separation of concerns? How do they scale? What tool support is needed? How can the software process be improved to take advantage of these ideas and tools? As these and many other open questions are answered, and tools are built, it will become possible to apply multi-dimensional separation of concerns to real development, and thus to explore its benefits and its limits.

REFERENCES

[all97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July, 1997.

[and92] 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.

[bar96] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.

[cla99] S. Clarke, W. Harrison, H. Ossher and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications,* pages 325–339, November, 1999. ACM.

[dso98] D. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[gos96] James Gosling, Bill Joy and Guy L. Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[har93] 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, September 1993. ACM.

[hol92] 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. LNCS 615.

[hyp99] Hyperspace web site, http://www.research.ibm.com/hyperspace.

[jac90] M. Jackson. Some complexities in computer-based systems and their implications for system development. In *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 344–351, 1990.

[kel99] R. K. Keller, R. Schauer, S. Robitaille and P. Pagé. Pattern-Based Reverse-Engineering of Design Components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 226–235, May, 1999.

[kic97] G. Kiczales. Aspect-oriented programming. *In ECOOP '97: European Conference on Object-Oriented Programming*, 1997. Invited presentation.

[kim99] Doug Kimelman, Multidimensional tree-structured spaces for separation of concerns in software development environments. Position paper, OOPSLA ’99 Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99.

[mez98] 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*, October, 1998.

[nus94] 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):760–773, October, 1994.

[oss88] H. Ossher. A case study in structure specification: A Grid description of scribe. *IEEE Transactions on Software Engineering*, 15(11), 1397–1416, November, 1989.

[oss96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[oss98] H. Ossher and P. Tarr, Operation-level composition: A case in (join) point. In *ECOOP ’98 Workshop Reader*, pages 406–409, July 1998. Springer Verlag. LNCS 1543.

[par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[par76] D. L. Parnas, On the Design and Development of Program Families. In *IEEE Transactions on Software Engineering*, 2(1), March 1976.

[rum98] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[sha96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[tar93] P. L. Tarr and L. A. Clarke. PLEIADES: An object management system for software engineering environments. In *Proceedings of the ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 56—70, December, 1993.

[tar99] 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, May 1999.

[tur98] 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*, 162–164, April, 1998.

[van96] 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, October 1996. ACM.

[wol89] A. L. Wolf, L. A. Clarke, and J. C. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.