RC 21452(96717)16APR99 Computer Science/Mathematics

Research Report

Multi-Dimensional Separation of Concerns in Hyperspace

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

Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

Multi-Dimensional Separation of Concerns in Hyperspace

Harold Ossher and Peri Tarr
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

Despite the well-known benefits of *separation of concerns*, and despite the presence of mechanisms to achieve separation of concerns in all modern software formalisms, software artifacts continue to exhibit properties associated with poor separation of concerns. Comprehensibility degrades over time; impact of change is high; reuse and traceability are limited.

We have hypothesized that these limitations are largely caused by the "tyranny of the dominant decomposition:" existing languages and formalisms generally provide only one, "dominant" dimension along which to separate concerns–e.g., by object or by function. Achieving many software engineering goals depends on the ability to separate *all* concerns of importance. We therefore introduced the notion of *multi-dimensional separation of concerns*: simultaneous separation according to multiple, potentially *overlapping* concerns.

This paper explores the structure of the space of concerns, to which we refer as *hyperspace*, partially formalizing our earlier model. We discuss how the model facilitates the identification and encapsulation of those portions of a system pertaining to a given concern, whether or not that concern is "dominant," and how it helps identify, introduce, change and remove concerns during evolution. We also show how this approach promotes two crucial aspects of evolvability: traceability and limited impact of change.

Keywords: Multi-dimensional separation of concerns, hypermodules, hyperslices, evolution, traceability, limited impact of change, software decomposition and composition, concern spaces, hyperspace

1 Introduction

The notion of *separation of concerns* [12] is at the core of software engineering. Done well, it permits the isolation and encapsulation of all concerns of importance in a software system. Proper separation of concerns results in a number of properties of recognized importance, including traceability, reduced impact of change, plug-and-play and mix-and-match, and improved comprehensibility, evolvability, and reusability.

Despite the well-known benefits of separation of concerns, and despite the presence of mechanisms for achieving it in all modern software formalisms, software artifacts across the software lifecycle continue to exhibit negative properties commonly associated with poor separation of concerns. Software comprehensibility degrades over time (if, indeed, it is present at all). Many common maintenance and evolution activities result in high impact of change. Artifacts are of limited reusability, or are reusable only with difficulty. Traceability, both within and across the various software artifacts, is limited.

In a previous paper [14], we hypothesized that these limitations are caused, in large part, by the "tyranny of the dominant decomposition:" existing artifact formalisms generally provide only one, "dominant" dimension along which concerns can be separated (e.g., by object or by function). The ability to achieve the goals of software engineering depends fundamentally on our ability to separate *all* concerns of importance. These concerns often overlap and interact with one another; e.g., a method `print()` is part of concerns in both the object and function dimensions. Concerns often vary over time, as requirements change. Representing and satisfying different concerns may even suggest different artifact decompositions and architectures. To begin to address these problems, we proposed *multi-dimensional separation of concerns*: simultaneous separation according to multiple, potentially *overlapping* and *interacting* concerns.

Our previous work presupposed the identification of "dimensions" of concern, and of "concerns" and their interrelationships, including points of overlap and interaction—concepts of which most software engineers have an intuitive grasp but no precise definition. This paper makes several novel contributions. It partially formalizes these notions, and our earlier model, by introducing and modeling multi-dimensional *concern spaces*, which we call *hyperspaces*. It discusses how hyperspaces

facilitate identification and encapsulation of concerns, whether or not they are "dominant," and whether or not they were identified during original development. Finally, it demonstrates how this approach promotes the "ilities," which, for the purposes of this paper, we restrict to mean comprehensibility, traceability, evolvability and limited impact of change.

Section 2 introduces an example to motivate the work and to illustrate the key concepts presented in this paper. Section 3 presents our model of hyperspaces. In Section 4, we apply this model to the example introduced in Section 2, illustrating how hyperspaces can be used to identify and encapsulate concerns and to guide the structuring of systems. We also demonstrate how the model promotes the "ilities." Section 5 discusses related work. Section 6 presents some conclusions and future work.

2 Motivation and Concepts

In this section, we introduce an example (adapted from [14]) to motivate this work and to introduce key concepts and issues.

Motivating Example: A Matter of Concern in an SEE

The example involves constructing and evolving 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 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 and displays it, and a display capability, which depicts an expression textually.

Based on these requirements, we design the system using UML, as partially depicted in Figure 1. It 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()` and `display()`, 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.

In using the SEE, clients discover that unnecessary reevaluations occur, causing performance problems. They request an enhancement to the SEE to permit

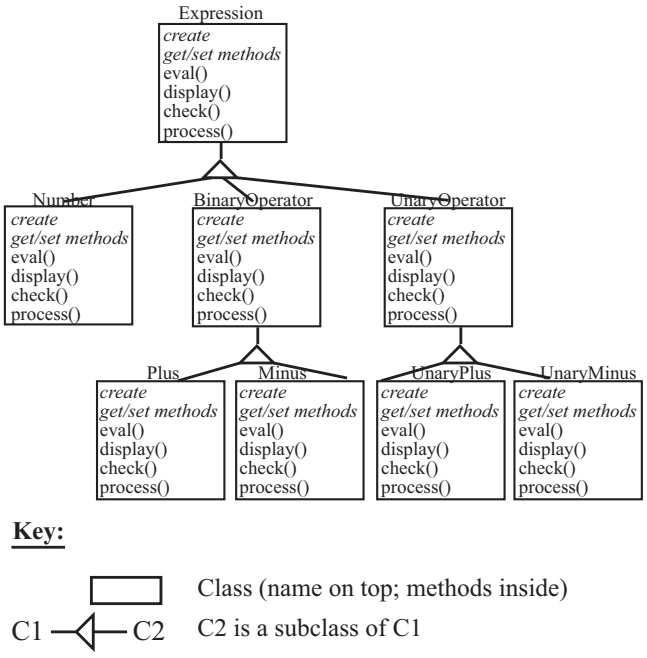

Figure 1: Initial (Partial) Design Artifact for SEE.

caching of evaluation results.

Unfortunately, this seemingly straightforward enhancement has a significant impact on the design and code. A simple implementation of caching requires adding a "cached result" instance variable and its accessor methods to the `Expression` base class, modifying all existing setter methods, in all classes, to invalidate the cached result, and changing the implementation of the `eval()` method in all classes that implement it to return a cached result if one is available. Clearly, this represents a non-trivial, invasive change to the design and code [14]. Further, code to support caching becomes tangled with other code, impeding comprehensibility and future evolution.

A World of Concerns

This example illustrates a few fundamental and pervasive problems in modern software engineering, pertaining to *separation of concerns*. This concept is essential to reducing the complexity of software systems to a level where people can cope with it, as it permits people to focus on a small set of details and ignore others. Clearly, however, the separation of concerns in this example, while "clean," was insufficient to facilitate traceability, limited impact of change and evolution:

Numerous different dimensions of concern may affect a software system, and the concerns and dimensions may evolve over time, as the system evolves. Informally, a dimension of concern is a means of decomposing a software system into a set of concerns.

A myriad of different dimensions of concern exist, each possessing different properties that make them more or less able to facilitate particular kinds of evolution. This example illustrates several common ones. *Object* is a key dimension, and each of the design and code classes is a concern in this dimension. *Feature* is another dimension; it encompasses the kernel (core AST), display, and evaluation concerns. Traditionally, *artifact* is also an important dimension; in this example, it includes the requirements, design, and code artifacts. The significance of this dimension is reflected in the great importance given to the traceability property. *Unit of change* is a dimension of less importance initially, but it becomes increasingly important, as a system evolves, to be able to encapsulate units of change as individual concerns on which developers focus.

Note that the dimensions of concern facilitate very different kinds of changes. For example, objects, as units of data abstraction, help localize the effects of adding or making changes to data, but they result in the scattering of functions across multiple classes (e.g., each class in the AST design and implementation has its own `eval()` method). Conversely, features help localize the effects of adding or making changes to units of functionality, but, since different features may manipulate the same data, feature-based separation of concerns results in scattering of data across multiple features. The fact that different kinds of concerns facilitate only a subset of the types of changes that occur during the course of software evolution leads inevitably to the conclusion that different kinds of evolution require the use of different dimensions of concern. The reason that the addition of caching was difficult is that it represents a concern in a different dimension—feature—from the dominant one used to decompose the design and code artifacts—object.

Multiple concerns may affect the same software system simultaneously, but only a small number can, traditionally, be separated. This is, in essence, the "tyranny of the dominant decomposition"—despite the clear need for different dimensions of concern to promote and limit the impact of different kinds of evolution, modern artifact formalisms generally provide only a single dimension of decomposition.

Whereas artifact development formalisms generally permit decomposition along only one dimension, mechanisms outside of these formalisms can help to achieve some secondary decompositions along other dimensions. For example, the separation into different kinds of artifacts is enforced by the use of different formalisms to represent different artifacts; and units of change can be encapsulated, to some extent, using version control and configuration management systems. Unfortunately, these mechanisms are insufficient to achieve the "ilities," even for those dimensions of decomposition that they support. This is because any given piece of a software system may affect concerns in multiple dimensions (e.g., the "display" requirement affects design and code, plus all of the classes in the object dimension); conversely, any given concern may affect multiple pieces of a software system. Thus, concerns are interrelated. Existing mechanisms cannot represent or manage these interrelationships.

Separation of concerns involves both identification and segregation. Traceability, and to some extent, comprehensibility, depend on *identification*—determining which pieces of software pertain to a given concern. Identification is necessary for determining how concerns impact software systems. While it can *demarcate* the scope of a change to a given concern, however, it cannot itself *limit* the impact of such a change; we must still determine how the change affects other concerns and propagate its effects to those other concerns, which may cascade. Thus, evolution also requires the ability to *segregate* and encapsulate concerns, which localizes the effects of a change to a given concern. For example, if we intend to modify the evaluation feature to include caching, and to have the effects of this change not cascade across other concerns, evaluation must be encapsulated as a feature.

"Clean" separation of concerns along a single dimension cannot, in general, promote the "ilities." Comprehensibility, traceability, limited impact of change, and evolvability all depend on separation and encapsulation of *all* concerns of importance, and they depend on the ability to identify, understand, and manage interactions among concerns.

3 A Model of Hyperspace

To begin to address these problems, we now describe a model of software that permits the explicit representation of all concerns of importance in a system and the identification and management of interactions among those concerns. This model is intended to facilitate both segregation and identification of concerns, and includes several features aimed at achieving limited impact of change. This section elaborates the model, while Section 4 describes specifically how it supports the "ilities."

Units Organized within Hyperspace

Software consists of *artifacts*, which comprise descriptive material in suitable formalisms. A *unit* is a syntactic construct in some such formalism. A unit might be, for example, a declaration, a statement, a state chart, a class, an interface, a requirement specification, or any other coherent entity that can be described in a given formalism. We distinguish *primitive* units, which are treated as atomic, from *compound units*, or *modules*, which group units together. Thus, for example,

a method, instance variable, or performance requirement might be treated as a primitive unit, while a class, package, or collaboration diagram might be treated as a module.[1] In the expression SEE example, we treat methods (e.g., `Number.display()`, `Plus.eval()`) as primitive units, and classes (e.g., `Number`, `Plus`) as modules.

A *concern space* encompasses all the software in some domain of discourse, such as a set of software systems, product families, or component libraries. For example, a concern space for the SEE contains all of the software artifacts described in Section 2, for both the initial system and the extension.

The job of a concern space is to organize the units in the domain of discourse so as to separate (identify and segregate) all important concerns, and to indicate how software components and systems can be built and integrated from the units that address these concerns.

Every software artifact (or set of artifacts), in whatever formalism it is written, can be mapped to some concern space that organizes it so that some set of concerns it addresses are separated. The structure of each such concern space depends on the formalism(s) used and the particular decomposition approach. For example, object-oriented formalisms help to identify *object* concerns. Thus, a concern space for an object-oriented system separates out object concerns; the particular objects it separates depends on the set of classes chosen for a given system.

As we have noted, we believe that achieving the "ilities" requires the representation of *multiple* kinds of concerns. Our model therefore introduces the notion of a *hyperspace*, which, broadly speaking, is a concern space that facilitates multi-dimensional separation of concerns using our approach. The remainder of this section describes our model of hyperspaces.

Formally, a *hyperspace* is a tuple (U, M, H), where

- U is the set of units in the hyperspace.
- M is a *concern matrix*, which simultaneously organizes the units in U according to all concerns of importance. M supports *identification* of concerns in the hyperspace.
- H is a set of *hypermodules* [14], which specify how to build components from the units in U. H promotes *segregation* of concerns.

Identification: The Concern Matrix

As noted in Section 2, many concerns are of interest in a software system, and they often overlap, since a single unit can address many concerns. For example, the unit `Number.display()` in the expression SEE is involved in the Number object concern and the Display feature concern. Since units simultaneously address multiple concerns, usually of different types (e.g., object vs. feature), we represent concern matrices as *multi-dimensional* matrices of units. The dimensions represent types of concerns, and the points on the dimensions are the specific concerns of those types.

We model a *concern* as a predicate, c, over units in U. It indicates whether a unit addresses that concern. The *unit set* induced by concern c is:

$$\mathcal{U}(c) = \{u \in U \mid c(u)\}$$

Concerns are said to *overlap* if their unit sets are not disjoint.

A *dimension of concern* is a set of concerns whose unit sets partition U. This partitioning property is important, and is standard in multi-dimensional spaces: a point in the space projects onto exactly one coordinate in each dimension. It implies that the concerns within a dimension cannot overlap, and must cover all the units in U.

Each dimension, d, has a special concern, which we call the *none concern* for d, N_d. All units that do not address any other concern in d address N_d. For example, function units do not address concerns in a dimension based on data decomposition; thus, they address N_d for that dimension. The *none* concern is important: it is the set of all units that are unaffected by evolutionary changes that occur within its dimension. Thus, every dimension has one, even if it happens to be empty.

We define a *concern matrix* over a set of units, U, as a tuple (C, D), where C is a set of concerns, and D is a set of concern dimensions, such that:

- Every concern in C is in exactly one dimension in D.
- Every dimension in D partitions U.

Each unit in U will fall at exactly one point (i.e., address exactly one concern) in each dimension, and its coordinates in the matrix identify these concerns.[2]

The structure of a concern matrix makes it possible to choose any concern or dimension as the primary focus, and be able to see directly which units in the hyperspace affect that concern, or each concern in that dimension.

Segregation: Hyperslices and Hypermodules

Concern matrices support the identification of different concerns, in different dimensions, simultaneously. They do not segregate concerns, however. Concerns can be

[1] Exactly what syntactic constructs are regarded as primitive units or modules depends upon how this model is instantiated for use with particular formalisms, a process described in [14]. The details are not, however, relevant to elaboration of the model.

[2] Note that it is possible for multiple units to fall at the same coordinates.

segregated only if the units' artifact formalism(s) provide explicit constructs (typically in the form of modules) to do so. This is the primary cause of the "tyranny of the dominant decomposition."

To address the need for segregation and encapsulation of concerns, we use *hyperslices* and *hypermodules* [14]. A hyperslice is a set of units. Like modules, hyperslices are units of encapsulation, intended to encapsulate concerns, but they are not bound by artifact formalisms; hyperslices can include any units.

Hyperslices are the building blocks of software. Each unit in a hyperspace belongs to at least one hyperslice. When new units are added, they must be added in hyperslices. Hyperslices can be grouped into hypermodules, which specify how they are related, and how they may be integrated to form a single hyperslice. Systems are hypermodules that satisfy a particular "completeness" constraint (described below).

In the rest of this section we describe the properties of hyperslices, model them, and discuss how they are composed into hypermodules.

Declarative Completeness
Units are typically related in a variety of ways; for example, one procedure 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, defeating the goal of segregation. To decouple them, we require hyperslices to be *declaratively complete*: they must themselves declare (though not necessarily implement) everything to which they refer (e.g., every function they invoke and variable they use).

For example, suppose a Display hyperslice contains a unit, u_1, for its `display()` method, which uses a `getOperand()` accessor function, u_2, defined in a Kernel hyperslice. To make Display declaratively complete, it must contain another unit, $u_{2_{decl}}$, which *declares* `getOperand()` (without necessarily implementing it). u_1 must use $u_{2_{decl}}$ instead of u_2. This eliminates the coupling between Display and Kernel, in favor of the assertion that $u_{2_{decl}}$ must eventually be "bound" to a unit in *some* hyperslice that provides the required implementation.

This approach makes use of a common mechanism for achieving looser, later binding: the separation of "*declarations*" from "*implementations*."[3] We assume a predicate, *decl*, on units that identifies declarations. The declarative completeness requirement typically results in many declarations of the same entity, in different hyperslices. These declarations are sometimes different, where the different hyperslices have different needs, but often look just the same. We do not wish these similar declarations in different hyperslices to create "false overlap." We therefore define the *implementation set* of a hyperslice as

$$\mathcal{I}(hs) = \{u \in hs \mid \neg decl(u)\}$$

and use it when determining overlap. The implementation set of a concern is defined similarly.

This property of declarative completeness means that each hyperslice is self-contained, from the perspective of declarations. Declarative completeness of hyperslices is crucial to achieving limited impact of change, as we will discuss in Section 4.

Correspondence
Total isolation of hyperslices is useless: e.g., to create a working SEE, some hyperslice must provide a unit that can be bound to $u_{2_{decl}}$ to provide an implementation. We refer to this kind of association among units as *correspondence*. Correspondence occurs within the context of a particular software component or system—the same declarative unit may be associated, for example, with different implementation units in different systems. This context is a hypermodule (described later in this section). Once corresponding units have been identified, they can be *composed*, or integrated, within the hypermodule to form a software component or system. Thus, the composition of corresponding units u_2 and $u_{2_{decl}}$ results in u_2 being called by u_1 at runtime.

Correspondence represents a fairly loose form of binding, which is a critical property for evolvability. Hyperslices do not depend on each other directly. Instead, systems are subject to a *completeness constraint* in which each declaration unit in a system must correspond to compatible implementation(s) in some other hyperslice(s). Replacing an implementation is non-invasive on hyperslices; it merely requires the redefinition of correspondence relationships.

Different types of correspondences can occur, beyond the association between declarations and implementations. 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 the cached result before evaluating.

Clearly, the issue of whether corresponding units are "compatible" (e.g., whether an implementation unit sat-

[3] Neither "declaration" nor "implementation" should be taken as referring solely to code units. Taken loosely, they apply equally well to any entity for which it is possible to separate some sort of declaration (e.g., a requirement name, test plan name, or function signature) from a full definition (e.g., the statement of a requirement, the contents of a test plan, or a function body). Declarations merely specify requirements; implementations provide details that satisfy requirements (though they might also have requirements of their own).

isfies a declaration unit's requirements, or whether a design unit satisfies a requirement) involves both syntactic and semantic issues. How to characterize and check such for 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.

Correspondence provides great flexibility, 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 compositions.

Hyperslices

Formally, a *hyperslice* in a hyperspace (U, M, H) is a declaratively complete concern ($hs \in C$). Thus, hs is closed under references of all kinds: if any unit $u_1 \in hs$ refers in any way to any other unit, $u_2 \in U$, then $u_2 \in hs$.

Hyperslices differ from other concerns only in that other concerns need not be declaratively complete. We require all hyperslices to occur in distinguished dimensions, called *hyperslice dimensions*.

Because they occur in separate dimensions, hyperslices, by definition, overlap other concerns. This is deliberate: their very purpose is to segregate units belonging to specific concerns. We say that a hyperslice *affects* a concern if and only if their implementation sets intersect. We say that a hyperslice *perfectly encapsulates* a concern if and only if their implementation sets are equal.

Hypermodules

Hyperslices provide a means of segregating concerns. Once segregated, however, it must be possible to integrate them [7]. A *hypermodule* is used to integrate a set of hyperslices into a new, unified hyperslice that addresses some or all of the concerns encapsulated in the original hyperslices. This integration may be used, for example, to create a system, a software artifact, or any meaningful software component that addresses some set of concerns of interest.

To integrate a set of hyperslices in a hypermodule, it is necessary to identify the set of correspondence relationships that exist among the hyperslices, and determine how to *compose* them together to effect integration. Thus, we define a hypermodule as a tuple, (HS, CR), where HS is a set of hyperslices and CR is a set of *composition relationships*. A composition relationship is itself a tuple, (I, r, f, o), where

- I is a tuple of *input units* drawn from the hyperslices in HS.
- r is a *correspondence relationship*, which characterizes the way in which the units in I are interrelated.
- f is a *composition function*, $f : (I \times r) \nrightarrow U$, which indicates how to compose the units in I as appropriate for r. If correspondence relationships are being used solely for traceability, f need not be specified.
- o is an *output unit* produced using f.

As noted earlier, many different types of correspondence relationships may be of interest. Some noteworthy ones are:

- Binding of declarations to implementations.
- Correspondence of multiple implementations, with the intent of integrating their respective capabilities.
- Elaboration. For example, a design unit might elaborate a requirement unit. Elaboration relationships are appropriate for modeling refinement.

Correspondence relationships are deliberately left abstract in this model, as their details depend on many factors, including the formalism(s) in which units are written, which of the formalisms' constructs are treated as units and modules [14], the extent of environment support for correspondence, etc. Our intent was to provide a framework within which multiple semantics for correspondence could be specified.

We previously defined a hypermodule as a set of hyperslices together with a *composition rule*, rather than *relationships* [14]. Composition rules are high-level, *intensional* specifications of composition relationships. They are as succinct as possible, relying on defaults and uniformity to specify composition primarily at the hyperslice level rather than at the individual unit level. A *compositor* tool generates from such a rule the set of unit-level composition relationships used in this model. We use the *extensional* unit-level approach here to facilitate discussion of impact of change, but we expect environmental support for hyperspaces to include composition rules and compositors.

Hypermodules can be used to model many kinds of software artifacts, components, and fragments thereof. 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. As we shall see in the next section, many system structures and "packagings" of concerns are also possible, and very useful.

4 Applying the Model

In this section we discuss the use of the model just presented, in the context of the SEE example from Section 2, to promote comprehensibility, traceability, evolution, and limited impact of change.

The Original SEE in Hyperspace

We identified three dimensions of concern in the original SEE:

1. **Artifact,** comprising concerns Requirements, Design, and Code.
2. **Feature,** comprising concerns Kernel, Display, and Check.
3. **Object,** comprising concerns Expression, Number, BinaryOperator, UnaryOperator, Plus, UnaryPlus, Minus and UnaryMinus.

Structurally, all of the object concerns are fairly similar and equally well illustrate important features of the model, so, for expository simplicity, we will model only one concern in the object dimension—Plus—while modeling all concerns in the other dimensions.

The SEE may undergo evolution based on any of these dimensions. For example, if we chose to add the ability to multiply expressions, we might want to focus on the **Object** dimension and add a new concern, Multiply; if we wanted to add a tool to the environment to check the correctness of expressions, we would focus on the **Feature** dimension and add a Check concern; and if we needed to add a test plan, we would focus on the **Artifact** dimension and add a TestPlan concern. Thus, we define two hyperslice dimensions: one to represent **Feature** hyperslices and one to represent **Artifact** hyperslices.[4]

Two hyperslice dimensions are shown in the figure, **ArtifactHD** and **FeatureHD**, reflecting decompositions of the system by artifact and by feature. They contain hyperslices that perfectly encapsulate the concerns in the **Artifact** and **Feature** dimensions. **ArtifactHD** comprises hyperslices RequirementsHS, DesignHS, and CodeHS, and **FeatureHD** contains hyperslices KernelHS, DisplayHS, and EvaluateHS. These hyperslices include any declaration units introduced to make them each declaratively complete (see Section 3): hyperslice concerns are needed, in addition to the concerns they encapsulate, precisely to provide an appropriate location for these declarations in the matrix. Thus, for example, the EvaluateHS hyperslice, which encapsulates Evaluate, contains all the units that Evaluate does, along with declaration units for `display()` (since the evaluation feature depends on being able to depict results or errors, but depiction is not itself part of the evaluation concern) and AST accessor methods (which are part of the Kernel concern).

To create any needed expression SEE artifact, we simply write a hypermodule that encapsulates the hyperslices (concerns) the artifact must address, plus a set of composition relationships that describe the correspondence relationships among units in the hyperslices and how to compose the artifact from the hyperslices. Thus, for example, we can represent the entire SEE, with all features and all its artifacts, by defining either of the hypermodules shown in Table 1. This table shows high-level composition rules, rather than unit-level composition relationships. The *merge* rule establishes correspondences by name, and merges corresponding units [11]. Both these hypermodules create the same system, comprising all of the concerns identified in this hyperspace, with all declaration units merged with appropriate implementation units.

Perhaps a more interesting feature of hyperspaces is that they can also facilitate the creation and integration of concerns based the *intersection* of hyperslices. Consider, for example, the situation where we would like to create a code artifact that includes just the Kernel and Display features. The code units that satisfy these concerns appear in the sets $\mathcal{U}(\mathsf{Kernel}) \cap \mathcal{U}(\mathsf{Code})$ and $\mathcal{U}(\mathsf{Display}) \cap \mathcal{U}(\mathsf{Code})$. Thus, these sets represent a concern that has no corresponding hyperslice. We can encapsulate them by creating a new hyperslice dimension, **CodeFeatureHD**, which will have two hyperslice concerns: KernelCodeHS and DisplayCodeHS. We can use hypermodules to create these two new hyperslices, and then define a hypermodule to integrate them, to produce a new hyperslice, KernelDisplayCodeHS, as shown in Table 2. This table shows the use of the composition rule *declaratively complete intersection*, which simply takes the intersection of two hyperslices and adds any necessary declaration units to make the resulting hyperslice declaratively complete.[5]

As Concerns Turn: Extending the Environment

Clients of the SEE requested caching of the results of evaluating expressions, to improve performance. Examining the concern matrix, we determine that caching cuts across multiple concerns in every dimension (all classes in the object dimension, the evaluation and kernel concerns in the feature dimension, and all artifacts in the artifact dimension). This means that caching belongs in a new dimension. Adding a dimension to a concern matrix is generally straightforward; the new di-

[4] Recall that we are focusing on a single object concern, so we do not need a hyperslice dimension to represent object hyperslices.

[5] This composition rule presumes the availability of intra-hyperslice def/use analysis, a capability we presume elsewhere as well. Since it is only intra-slice, this analysis is not difficult to perform.

Hypermodule	Hyperslices	Composition Rules
EnvironmentAll$_1$	KernelHS DisplayHS EvalHS	*merge*
EnvironmentAll$_2$	RequirementsHS DesignHS CodeHS	*merge*

Table 1: Alternative Hypermodule Definitions for the SEE.

Hypermodule	Hyperslices	Composition Rules
KernelCodeHS	KernelHS CodeHS	*declaratively complete intersection*
DisplayCodeHS	DisplayHS CodeHS	*declaratively complete intersection*
KernelDisplayCodeHS	KernelCodeHS DisplayCodeHS	*merge*

Table 2: Creating New Hypermodules from Existing Ones.

mension is unlikely to affect existing units, so adding the dimension generally requires nothing more than updating the coordinates of each existing unit to indicate that the unit is in the *none* concern in the new dimension, in this case **Caching**.

Adding new units that address caching to the concern matrix requires defining the units within the context of one or more hyperslices, as noted in Section 3. We choose to do it in a single one, CachingHS, encapsulating all details of caching; portions that intersect with specific other concerns, such as objects or artifacts, can be extracted by intersection. CachingHS must, of course, be declaratively complete: where there is a need to refer to units in the existing hyperslices, local declarations must be introduced. We put CachingHS in a new hyperslice dimension, **CachingHD**. We must also determine the correct location of each new unit in the other dimensions. For example, the code unit that declares the variable to hold the cached result belongs in the Expression, Kernel, Code and Caching concerns.

After adding caching to the concern matrix, we can describe any number of interesting software artifacts based on it, by defining the appropriate hypermodules and composition rules. We can create any or all of the SEE artifacts, and they can provide caching or omit it; they can include or exclude any of the features (except for the kernel). Table 3 depicts some of the possible artifacts that can be created using the SEE hyperspace.

Removal of Concerns

Removal is typically the most invasive of evolutionary activities. In the context of hyperspaces, there are several scenarios, some remarkably simple. We consider here just the most difficult: removing a concern, along with all its units, from the hyperspace.

The most difficult aspect of removing a concern from a hyperspace (U, M, H) is removing its units. After that, the concern itself can merely be removed from M. A unit, u, can simply be removed from U, which will simultaneously remove it from all concerns it affects, determined by it coordinates in M. The difficulty is that at least one of these will be a hyperslice, and declarative completeness must be maintained and composition relationships must remain valid.

To maintain declarative completeness, if $u \in hs$ is an implementation unit that is used by other units in hs, u must be transformed into a declaration unit. For example, if u represents a `display()` method with an implementation, it must be transformed to remove the implementation. A declaration that is used within its hyperslice cannot be removed without removing or changing the using units.

To maintain the validity of compositions, it is necessary to examine the correspondence relationships of all hypermodules that include each affected hyperslice, once all unit removal has been completed. For any composition relationship, (I, r, f, o), such that $u \in I$, it is necessary to remove u from I, compensate for effects on the relationship, and then reapply f (possibly revised). In some cases, effects might be nonexistent or purely local—for example, if all declarations in I have also been removed. In other cases, the unit might really be needed—for example, it might be the only implemen-

Hypermodule	Hyperslices	Composition Rules, Comments
CachingEnv$_1$	EnvironmentAll$_1$ CachingHS	*merge* Environment with caching
CachingEnv$_2$	EnvironmentAll$_2$ CachingHS	*merge* Environment with caching
CachingReqsHS	CachingHS Requirements	*declaratively complete intersection* Requirements for the caching capability
AllReqsHS	RequirementsHS CachingReqsHS	*merge* Requirements for environment with caching
Caching$_E$HS	CachingHS Eval	*declaratively complete intersection* Eval part of caching capability
CachingEvalHS	EvalHS Caching$_E$HS	*merge* Eval feature with caching

Table 3: Some software artifacts that can be created from the SEE hyperspace.

tation in I that satisfies corresponding declarations. In such cases, the developer must determine whether other units in existing hyperslices can fulfill the need, possibly with some "transformational glue" that can be specified in a revised composition function, f. If not, the developer must either reinsert u (at a different location in the matrix), write a new implementation unit, or, in the worst case, rewrite existing units—those related to the other units in I by def/use relationships within their hyperslices. Only the last case involves invasive change; further research is needed to determine how often it occurs in practice.

Analysis: Achieving the "Ilities"

The representation of the SEE using hyperspaces demonstrates some important properties of our model with respect to achieving the "ilities." First, the hyperspace, by its structure, enables software engineers to focus in on any concern of importance that is represented in the hyperspace—they need only examine the hyperplane containing the concern and orthogonal to its dimension. This facilitates comprehensibility.

Second, given the ability to separate, simultaneously, all concerns of importance, it is possible 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 composition relationships in which their encapsulating hyperslices 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, as the example demonstrated, and the model of hyperspace provides for the identification and reification of such concerns. Further, the structure of a hyperspace aids the identification of other types of "hidden" concerns. In particular, the definition of dimensions precludes situations where two concerns in the same dimension overlap. This property helps identify many kinds of errors, ranging from weak identification and separation of concerns to coupling of concerns.

Third, the addition of units, concerns, and dimensions in the model is clearly straightforward, with little impact on existing concerns. Moreover, the process of adding units forces software engineers to determine how the new units affect existing concerns, though they can choose to indicate that the units affect no existing concerns by using *none* concerns. Extension is *additive*, rather than invasive—a significant part of the goal of limited impact of change.

Fourth, hyperspaces 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. In 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 composition relationships. This is because the model eliminates direct dependences among hyperslice concerns 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 *connected*, the breaking of these connections (embodied by correspondence relationships) can be followed, *non-invasively*, by the establishment of new connections to other units that fulfill the intended semantics of a given composition relationship. The impact of removal can, therefore, be limited to the particular hyperslice it affects and to the composition relationships of 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.

5 Related Work

In a prior paper [14], we discussed various modern approaches that have introduced novel modularization mechanisms that relate to multi-dimensional separation of concerns: subject-oriented programming [5, 11], aspect-oriented programming [8], contracts [6], role models [2, 15], adaptive programming [9], Viewpoints [10] and Catalysis [4]. All of these, except Viewpoints, deal with object-oriented systems, in which the dominant decomposition is by Object. Each introduces a mechanism, analogous to hyperslices, to segregate design or code addressing other, non-object 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—namely, some degree of flexibility in composition of concerns along some useful dimensions [14]. We believe that all of these approaches fit well into hyperspaces, which, in turn, offer additional benefits to them. These benefits include help with selecting and modeling additional dimensions of separation, organization and comprehension of concerns, identifying and managing overlap among concerns, and enhanced traceability.

Our model has relationships to, and has used results from, other areas of related work. The utility of loose binding to help limit the impact of some forms of change is accepted. Work in the area of software architecture (e.g., [13, 1]) has identified the need to separate software *components* from *connectors*. Similarly, earlier work on Precise Interface Control (PIC) [16] identified benefits of representing a particular kind of inter-component interactions: provides and requires. The declarative completeness requirement and use of separately specified composition relationships are in the spirit of these, and similar, approaches. Barrett et. al [3] describe a spectrum of mechanisms available 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 impact of change (by not permitting software components to know about one another other) with the need for analyzability (most readily accomplished in the presence of tighter binding).

6 Conclusions and Future Work

A number of important problems in software engineering have resisted general solution, notably ones 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 introduced and modeled *hyperspaces*, which provide a foundation for addressing these problems. This model does not replace, but rather supplements, existing artifact notations. It facilitates the separation, encapsulation, and ultimately, integration of multiple concerns, along multiple dimensions, simultaneously, providing uniform treatment of "dominant" and other concerns. This permits software engineers truly to achieve the comprehensibility promise of "clean separation of concerns:" the ability to ignore any part of a system that is not of importance for a particular purpose. The structure of hyperspaces facilitates traceability, identification of how a change affects other concerns, and localization and limited impact of change. The ability to introduce, modify, and remove concerns and dimensions of concern further enhances evolvability.

Much work remains, both to address limitations of and issues with the model and to apply hyperspaces to the software engineering lifecycle. First, environment support for hyperspaces will be critical to their use, and is an open research issue. Second, issues of scalability, including possible problems with proliferation of hyperslices, concerns and dimensions must be addressed. *Restructuring* of hyperspaces, to change or consolidate dimensions and concerns, is also important, and should facilitate software reengineering. We believe that sophisticated environment support is the key to dealing with these issues. Third, it will be important to identify and specify more precisely the types of correspondence and composition relationships required to achieve the "ilities" in practice.

Finally, it is particularly important to instantiate the model of hyperspaces for a variety of formalisms, to explore issues that arise when using different methodologies. These instantiations must be used for real development, to evaluate them and to create new development methods that explore their strengths; to explore issues in cross-concern and cross-formalism correspondence and composition; to explore the potential for mix-and-match and plug-and-play; and to explore the impact of this approach on areas like development methodologies, software process, analysis, testing, reverse engineering, reengineering, and software architecture.

This work is clearly at an early stage, largely unproven

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. The model of hyperspaces we propose goes beyond existing efforts by providing a basis for, finally, realizing the full potential of "separation of concerns."

REFERENCES

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

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

[14] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. International Conference on Software Engineering (ICSE'99)*, 1999. (To appear.).

[15] 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.

[16] 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.