

Advanced Separation of Concerns for Component Evolution

Stanley M. Sutton Jr. and Isabelle Rouvellou
IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, New York, USA 10532
{suttonsm, rouvellou}@us.ibm.com

ABSTRACT

Poor separation of concerns impedes the evolution of complex software systems. Advanced separation of concerns supports the flexible yet well-structured evolution of systems according to meaningful units of change. An up-front investment in concern modeling pays dividends in facilitating many kinds of change over the life of a system.

1. INTRODUCTION

Separation of concerns is a long-established principle of software engineering [15]. The goal of separation of concerns is to give independent representations to the independent concerns. Effective separation of concerns achieves advantages throughout the software life cycle, but it is especially important over the extended life of a system with respect to evolution and other sorts of change. Ideally, concerns that have independent representations can be changed independently, thereby simplifying the changes and minimizing the associated cost, effort, and impact. For software processes, separation of concerns may afford flexibility in terms of task assignments and scheduling. Additionally, it can provide a basis for configuration control, versioning, and the management of product families. Separation of concerns is thus a key software engineering technique for the creation of evolvable software components, which in turn can support evolvable systems.

Modern programming languages and design formalisms provide a variety of means to support separation of concerns, including mechanisms for abstraction, encapsulation, modularity, and so on. While these are useful, in practice the benefits they afford are limited. That is because these languages typically allow a system to be decomposed in only a limited number of ways, thereby enabling the separation of only a limited number of concerns. Object-oriented languages, for example, support decomposition (with abstraction, encapsulation, and modularity) according to a particular hierarchy of object classes. However, this does not allow for decomposition (especially concurrent decomposition) along other dimensions such as features or functions, and it allows only

a particular decomposition of object classes to be expressed. This sort of limitation has been characterized as the “tyranny of the dominant decomposition” [19].

This is problematic for system evolution because software is subject to many concerns. At any given time, a system is subject to multiple simultaneous, overlapping concerns. Moreover, over time, the set of concerns to which a system is subject evolves. If a decomposition mechanism allows decomposition in only a small number of dimensions, then some concerns must necessarily be represented by elements that cut across the dominant decomposition. If a crosscutting concern must be changed, that may affect many elements in the dominant decomposition. If the dominant decomposition must be changed, that may affect numerous crosscutting concerns. Either way, when concerns are entangled, changes are generally difficult, effects are not localized, and impacts are difficult to assess.

Advanced separation of concerns attempts to overcome the limitations of a dominant decomposition by supporting the decomposition and composition of software in terms of modules based on concerns. Well-known approaches include subject-oriented programming and design [5, 3], aspect-oriented programming [10], and hyperspaces (multidimensional separation of concerns) [19]. Programs (or other kinds of artifacts) are specified as compositions of concern-specific units. Each concern is represented as a first-class entity, even concerns that might otherwise be crosscutting. A composition tool, such as a compositor or weaver, combines the units. The resulting program (design, etc.) may internally reflect a dominant decomposition, but the details can be hidden from the developer, who can work in terms of concerns as first-class entities.

To support advanced separation of, two general capabilities are needed. One is support for working on software artifacts such as code, designs, and so on, for purposes such as composition, decomposition, and analysis. The other is support for concern modeling and mapping, that is, the representation of concerns and their relationships and the association of concerns to particular work products or system elements. Particular technologies or methods may combine these capabilities to varying degrees.

In this paper we report on our experience in applying advanced separation of concerns techniques to support the flexible composition of a software component. We have been using Cosmos [17], a concern-space modeling schema, to model concerns in terms of which composition is performed. We have been using Hyper/J [6], a composition tool for Java, to effect composition of component code in terms of selected concerns. The component we have focused on is a cache, a common type of component in middleware and other systems software.

The remainder of this paper is organized as follows: Section 2 motivates and gives an overview of Cosmos. Section 3 introduces our example component, the General Purpose Software (GPS) Cache, and illustrates the use of Cosmos to model concerns in the cache. Section 4 describes Hyper/J and discusses our experience with it in generating versions of the GPS cache. Section 5 presents issues and observations based on our overall experience. Finally, Section 6 presents our conclusions.

2. COSMOS: A CONCERN-SPACE MODELING SCHEMA

In this section we discuss concerns in the software life cycle and give an overview of the Cosmos.

2.1 Concerns and Concern Modeling

Most generally, concerns are what we care about in software. A natural English definition of “concern” is a “matter of consideration” [13]. The IEEE defines the “concerns” for a system as “... those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.” [7, p. 4]. Concerns arise throughout the software life cycle, and individual concerns commonly span multiple life-cycle stages and work products.

Concerns are reflected in development work products in various ways. Some work products specifically introduce and characterize concerns of particular types. This is especially evident in requirements analysis, the main purpose of which is to identify concerns incumbent on a system to be developed [11,2]. Other work products may implement, embody, or otherwise affect concerns that they do not directly specify. This is particularly apparent with code, which implements functionality, supports behaviors, and observes properties that are specified separately.

While concerns arise and are addressed throughout the software life cycle and across a range of work products, what is missing from traditional development methods is any global, overarching view of concerns that is

independent of particular stages, formalisms, or tools. In other words, concern modeling is traditionally not itself a separable concern in software development processes; rather, it is entangled in the particular languages, tools, and artifacts used at each stage of development.

Cosmos [17] is a concern-space modeling schema that addresses this limitation. It allows concerns and the relationships among them to be modeled as first-class, independent entities that are separable from particular life cycles, work products, formalisms, and technologies.

2.2 Overview of Cosmos

Cosmos models software concern spaces in terms of *concerns*, *relationships*, and *predicates*. *Cosmos* divides concerns into two categories, *logical* and *physical*. Logical concerns represent concerns viewed conceptually, as “matters for consideration”: issues, problems, “ilities”, and so on. Logical concerns are concerns as we usually think of them. Physical concerns comprise the actual things that constitute our systems, such as specific work products, software units, hardware units, and services. Physical concerns are a way to bring the “real world”, to which our (logical) concerns apply, into the concern-modeling space.

Logical concerns are further typed as *kinds*, *instances*, *properties*, and *topics*. Kinds are categories of concerns (such as functionality, behavior, and state). Instances are particular concerns (usually of some kind, e.g., particular functions, behaviors, states). Properties are characteristics of kinds and instances. Topics are (typically theme-related) groups of concerns of generally different types.

Physical concerns comprise *instances*, *collections*, and *attributes*. Physical instances represent particular system elements (such as source files, design documents, workstations). Collections represent groups of these. Attributes are the specific properties of instances or collections, such as the size of a design document or number of files in a directory.

Cosmos can be illustrated with examples based on the GPS cache design, which we have modeled previously [9,16]: *functionality*, *behavior*, and *state* are kinds of concerns, *create object* and *log operations* are instances of concerns, *performance*, *configurability*, and *robustness* are properties, and *logging* and *algorithms* are topics.

Relationships are divided into four categories: *categorical*, *interpretive*, *physical*, and *mapping*. Categorical relationships reflect fundamental semantics of the concern categories. These relationships include *kind-of*, which relates (sub)kinds to kinds, *instance-of*, which relates instances (both logical and physical) and properties to

kinds, *applies-to*, which relates properties to kinds and instances, *part-of*, which relates instances to instances and properties to properties, *member-of*, which relates physical concerns (instances or collections) to collections, and *relates-to*, which relates any concern to a topic.

Some examples of categorical relationships for concerns expressed in the GPS cache design are as follows: *logging functionality* is a *kind-of functionality*, *toggle logging* is an *instance-of logging functionality*, *performance applies-to logging functionality*, and *logging functionality* and *logging behavior* *relate-to* the topic *logging*. Such relationships are essential to the integrity of Cosmos model in general and apply in any concern-modeling context.

Interpretive relationships relate logical concerns with semantics that are not based on categories. One example is *contributes-to*, which indicates that one concern (e.g., *logging behavior*) contributes in some way to another (e.g., *robustness*). Another interpretive relationship is *motivates*, which indicates that one concern (e.g., *robustness*) motivates another (e.g., *logging*). Such relationships are especially important in understanding the system-dependent semantics of concerns and are typically defined with respect to particular concern-modeling objectives.

Physical relationships represent associations among physical concerns (e.g., composition relationships among Java classes in Hyper/J [6]). Mapping relationships represent non-categorical associations between logical and physical concerns, for example, the implementation of a logical function by a Java class. These are of special importance in modeling concerns for purposes of composing component implementations.

Predicates represent integrity conditions over various relationships and can be classified accordingly. For example, categorical predicates apply to categorical relationships and interpretive predicates apply to interpretive relationships. Examples of the former are that no concern can be both a kind and an instance and that no kind can include both logical and physical instances. Examples of the latter are that, if one concern motivates another, then the second should contribute to the first, whereas if one concern contributes to another, then the concern does not necessarily motivate the first. In the GPS cache, a concern with performance motivates a concern with configurability, so configurability contributes to performance, whereas logging contributes to performance but performance does not motivate logging.

2.3 Discussion

Cosmos represents a kind of system documentation, but it is not intended to replace other documentation or work

products. For example, the details of system requirements should be captured in an appropriate requirements formalism [11,2], where the appropriate concerns can be elaborated in specialized technical detail. Relative to given specific approaches for recording concerns, Cosmos can serve for “early capture” of concerns prior to a full technical treatment, to address concerns of sorts that fall outside the scope of a specialized formalism, to correlate concerns between different models or representations of comparable things (as with intra-system “viewpoints” [14] or inter-system integration), and to span different representations or aspects that arise in different stages or work products of the life cycle (as with “refinements” [1]).

Another view of Cosmos is as a kind of semantic hyper-index of concerns as manifest in various work products across the life cycle. A Cosmos model provides a general overview of system concerns and various significant semantic and structural relationships among them, including relationships between physical and logical concerns (i.e., software artifacts and other systems elements and the considerations that apply to them). In this regard, Cosmos is something like a topic map [8], which is a semantic index into electronic documents. “Concerns” in Cosmos are comparable to “topics” in topic maps, both Cosmos and topic maps reference occurrences in documents, and both enable a variety of semantic relationships to be expressed in an index structure. There are a number of differences between them, however, related to the more specialized purposes of Cosmos. Among these are that Cosmos represents bidirectional relationships between artifacts and concerns (whereas topic maps represent relationships only into documents), and Cosmos incorporates ontological classification fundamentally (which topic maps do not).

3. MODELING COMPONENT CONCERNS

The GPS cache is a general-purpose (software) cache that is intended to be reusable in a variety of applications. It is richly functioned and featured. It supports the usual functionality associated with a cache (such as storing and retrieving objects and tracking expiration times), and it also some less common features such object dependencies, logging, and statistics collection. It is also highly configurable, both statically and dynamically. The GPS cache has been used in web publishing [12] (including IBM web sites for major sports events) and in support of business rule evaluation [4].

We are concerned in this section primarily with modeling concerns in the GPS cache as it was programmed. This modeling serves two purposes. First, it establishes a baseline for the concerns that we hope to address by composition. As such, it provides a prototype around which we can organize conceptions and models of what a

cache is and how it may be subject to evolution [18]. Second, it allows us to see which concerns are addressed in which parts of the cache code. This may provide a basis for decomposing the programmed cache into more specialized units that are suitable for recomposition into alternative versions of the cache.

Because we are interested not only in analyzing concerns in the programmed version of the GPS cache but also in describing a concern space for composing alternative versions of a cache, the concern model that we have created for the programmed cache is somewhat more general than necessary. Even at this stage of some generalizations of concerns in the cache can be anticipated. As we progress with composition we expect to add further concerns.

In principle there are many ways that a model of concerns in the cache might be organized and presented. Cosmos supports the modeling of multidimensional concern spaces, and concerns in the cache can indeed be described multidimensionally. That is, many elements in the cache can be assigned to concerns in multiple dimensions, and any of several dimensions might be distinguished as the root or basis of a view of the concern space.

In practice, particular views of a concern space will be better for particular purposes. In our case, we are looking at code, which is written in Java, which is an object-oriented programming language. Moreover, we are analyzing concerns for purposes of decomposing and then recomposing the code, to create further Java implementations of various caches. In this context, it is natural and effective to organize the presentation of concerns, in at least part of the concern space, around classes and their members. Of course, class inheritance is the tyrannical decomposition of object-oriented languages. However, in this case, the use of classes to organize concerns is not tyrannical, just sometimes preferred. Other organizations of the concerns are possible and may be preferred for other purposes, and composition of caches can be driven by concerns organized in other ways.

3.1 Concerns

Examples of high-level concerns in the implementation of the GPS cache are outlined in Table 1. According to the Cosmos model, concerns are organized as logical or physical, with logical concerns including kinds, instances, properties, and topics, and physical concerns including instances, collections, and properties.

The top-level kinds of concern we distinguish include classes, functionality, behavior, state, properties, and Java code. (Although instances of Java classes are physical

concerns, the kinds under which the Java files fall are considered logical concerns.)

Consistent with the context and purpose of our concern modeling, classes are a recognized kind of concern and are also used to organize subkinds under other kinds (such as functionality, behavior, and state). Under functionality, nine different functional areas are associated with the Cache class, grouping from one to eight methods (unfortunately, space limitations preclude details). Methods in the “printing” group are also included under other groups as they print information relative to those groups (for example, the method “output_stats” is also associated to the statistics-logging group. For the CachedObject class, existing methods fall into three groups, and three additional groups (not initially used) are identified by generalization from the Cache class; these may be populated later.

Behaviors are grouped into those specific to operations, those aspectual to operations, and those not associated with operations. Behaviors in these groups may be related; for example, the behavior of specific operations to turn statistics logging on or off has an effect on the logging of collected statistics, a behavior which is not associated with any particular operation.

Note more generally that a number of themes occur repeatedly under various kinds of concern, for example, “core” (which refers to the state of the cache in terms of the objects it contains), logging, and others. Some of these even occur across multiple classes. Such crosscutting concerns represent other dimensions by which the concern space can be organized. For instance, classes could occur under “core” and under “logging” instead of vice versa. Topics are one way to represent such crosscutting concerns.

Instances are omitted from the table for the sake of brevity. These are the specific things for which kinds provide categories, such as particular functions, behaviors, elements of state, and so on. Some examples are “retrieveObject” (a core function for the class Cache), “delete dependent objects” (an operational behavior associated to the “deleteObject” function in the dependencies group of class Cache), “remove expired objects” (an operational behavior associated to the Cache class, but one for which there is no function), and the count of “deleteObject” method invocations (a statistics counter).

Our list of properties is guided by prior examination of the design goals for the GPS cache [9,16]. Many of these could be inferred from the code (or general knowledge of software engineering), but in general it is possible to infer (or posit) arbitrary properties. Therefore some judgment is required to identify properties that are significant and relevant. The topics are not generally inferable from

Workshop on Engineering Complex Object-Oriented Systems for Evolution--OOPSLA 2001

Logical concerns

- Kinds
 - Classes
 - Cache
 - CachedObject
 - *Other classes ...*
 - Functionality
 - Cache
 - Core (*8 methods*)
 - Object expiration (*1 methods*)
 - Operation enabling (*2 methods*)
 - Dependencies (*5 methods*)
 - Object invalidation (*1 method*)
 - Operation logging (*2 methods*)
 - Statistics logging (*3 methods*)
 - Printing (*4 methods--redundant*)
 - CachedObject
 - Core (*3 methods*)
 - Expiration (*1 method*)
 - Dependencies (*3 methods*)
 - Validation (*0 methods*)
 - Operation logging (*0 methods*)
 - Statistics logging (*0 methods*)
 - *Other classes ...*
 - Behavior
 - Cache
 - Operational
 - Core
 - Object expiration
 - Operation enabling
 - Dependencies
 - Object invalidation
 - Operation logging
 - Statistics logging
 - Printing
 - Aspectual
 - Input checking
 - Operation enabling
 - Operation logging
 - Object expiration
 - Non-operational
 - Statistics logging
 - CachedObject
 - Operational behaviors
 - Core
 - Expiration
 - Dependencies
 - Validation
 - *Other behaviors ...*

- State
 - Cache
 - Objects
 - Dependencies
 - Configurable controls
 - Operation enabling
 - Operation logging
 - Statistics logging
 - Operation log
 - Statistics counters
 - Properties
 - Static properties
 - Dynamic properties
 - Java code
 - Programmed classes
 - Classes
 - Members
 - Decomposed classes
 - *Anticipated for later composition*
 - Instances
 - *Omitted for brevity (examples in text)*
 - Properties
 - Generality
 - Performance
 - Information hiding
 - Concurrency
 - Configurability
 - Correctness
 - *Other properties ...*
 - Topics
 - Dependencies and transitivity
 - Configurable behaviors
 - *Other topics ...*
- Physical concerns**
- Instances
 - com.ibm.ws.abr.gps.Cache.java
 - com.ibm.ws.abr.gps.Cache.class
 - com.ibm.ws.abr.gps.CachedObject.java
 - com.ibm.ws.abr.gps.CachedObject.class
 - *Other classes ...*
 - Collections
 - whimbrel.watson.ibm.com
C:\\$Sutton\Caching\Code\Java\Programmed
 - *Other collections ...*
 - Attributes
 - com.ibm.ws.abr.gps.Cache.java.Size
 - *Other attributes ...*

Table 1. Concern kinds in the implementation of the GPS cache (partial listing).

the code (or any other source). Their purpose is to represent arbitrary matters of interest to developers or users, and they should be identified as needed.

The physical concerns mainly represent the particular Java classes (and class members) that make up the GPS cache program. Of course, many of the logical concerns (e.g., kinds--including functions, behaviors, and state) will map to particular elements within the Java classes. The collections comprise directories in which the Java files are stored.

3.2 Relationships

In previous work on concerns in the design of the GPS cache, we have emphasized categorical and interpretive relationships. With respect to the implementation of the cache, we have focused on categorical relationships, which apply generally, and mapping relationships, which relate logical to physical concerns.

Categorical Relationships Among the categorical relationships, many instances of the kind-of relationship (which relates kinds and subkinds) are found in Table 1. Examples of instance-of relationships are given with the examples of instances in the prior subsection. Some instances of applies-to and relates-to are found in Tables 2 and 3, respectively. (Concerns in these tables are typically based on the example in Table 1 and are designated according to the outline shown there. A wildcard character represents all instances or subkinds of a kind.)

Property	Kinds or Instances
Performance	Behaviors.cache.nonoperational.operation_logging
Performance	Behaviors.cache.nonoperational.object_expiration
Concurrency	Behaviors.cache.nonoperational.object_expiration
Concurrency	Functionality.cache.dependencies.invalidateDependency
Information hiding	Behaviors.cache.operational.retrieve_object_original
Information hiding	Functionality.cache.retrieveObject
Configurability	Behaviors.cache.nonoperational.operation_logging

Table 2. Examples of the property-of relationship.

The member-of relationship is illustrated by the various Java files making up the cache are members-of a particular directory. The part-of relationship among physical concerns is illustrated by the various distinguishable members of a particular class, for example, the "createObject" method and "boolean status" variable are part-of the class com.ibm.ws.abr.gps.Cache. On the logical side, the various elements of the state of a cache, such as

the state of cached objects and the state of control variables, are part-of the state of the cache.

Topic	Related Concern
Dependencies and transitivity	Kinds.Operations.*.Dependencies.*
Dependencies and transitivity	Kinds.Behaviors.Operational.*.Dependencies.*
Dependencies and transitivity	Properties.Generalty
Dependencies and transitivity	Properties.Concurrency

Table 3. Examples of the relates-to relationship.

Mapping Relationships Among possible mapping relationships, for our purposes we are concerned here mainly with a general correspondence relationship which indicates that a particular physical concern, namely a code unit, implements or represents a particular logical concern, namely those representing abstractly the various classes, methods, and variables making up the logical description of the cache. Since these logical concerns have been developed precisely to encompass a natural description of the GPS cache implementation, this correspondence is straightforward and complete. On the other hand, if we were to relate the code of some other cache implementation to logical concerns descriptive of the GPS cache, the correspondence might be problematic.

Note that some other useful associations of physical to logical concerns are already covered under categorical relationships. These include the applies-to relationship, which can associate physical concerns to the properties they impact, and the relates-to relationship, which associates particular logical and physical concerns to topics of interest.

3.3 Predicates

Beyond issues related to the categorical integrity of the model, predicates do not play a large role in the concern space for the GPS cache implementation. We have modeled the concerns to be consistent with basic integrity constraints (categorical predicates). For example, although instances may belong to more than one kind, no kind may be a subkind of more than one kind, and physical concerns and logical concerns do not belong to the same kind.

We can define a number of completeness conditions that might apply to the GPS cache implementation concern model, for example, every function identified should have a corresponding behavior, and every property that applies to a function should apply to the corresponding behavior. Also, if a property applies to a part-of a composite concern then it should apply to the whole of the composite concern. However, the need for completeness mainly depends on the purposes of a model, so such predicates seem to be

typically more conditional than the categorical or interpretive predicates. We are still investigating their role.

3.4 Discussion

Based on our modeling of concerns in the GPS cache implementation, we have achieved two main goals. First, we have identified a substantial set of concerns related to the GPS cache taken as a model for an evolvable cache component. These concerns define a baseline concern space to which we can add further concerns, thereby broadening the concern-space within which we can systematically support cache component evolution. Some generalizations of the GPS cache concern space are already obvious. Second, we can see which parts of the GPS cache code implement or otherwise contribute to various concerns. The GPS cache was programmed in such a way that many concerns are already well separated and that many of the concerns that are not separated (say, combined within a method) are nevertheless readily separable. This gives us reason to believe that useful versions of the cache can be created by concern-based composition and thus that composition can support the evolution of a useful range of cache components. Additionally, it seems that the that the GPS cache code should be decomposable to provide an initial pool of units on which composition and evolution of further cache components can be based.

4. COMPOSING EVOLVABLE COMPONENTS USING HYPER/J

In this section we describe Hyper/J, a tool for concern-based composition, and discuss our experience in using it to compose versions of a cache component.

4.1 Hyper/J

Hyper/J, which has been developed at the IBM T. J. Watson Research Center, is a tool for concern-based composition of Java programs [6].¹ Hyper/J is based on the notion of multidimensional separation of concerns. Individual software units are seen to address or embody multiple concerns simultaneously, and these concerns can be organized in multiple, overlapping dimensions. It is this “hyperspaces” view of concerns that has motivated Cosmos.

Hyper/J works on Java class files and supports both the decomposition and composition of Java programs. We focus here on composition. A composition is specified in three parts. The first defines the concern space, that is, the classes and packages that are available for composition. The second defines the concern mapping, that is, which classes are assigned to which concerns in which

dimensions. The third defines a hypermodule, that is, a composed module comprising units from multiple concerns. The hypermodule specification has two parts. The first is a listing of dimensions and concerns that are involved in the composition. The second is a specification of composition rules indicating how various units assigned to the listed concerns are to be combined. The rules operate on units that include classes, interfaces, operations (abstract methods), actions (concrete methods), and fields.

Composition rules are generally specified in terms of a general strategy plus more specific rules for exceptions from the general rule. Examples of general strategies are *mergeByName*, which merges together all units of a given type having the same name, and *overrideByName*, in which the last specified unit with a given name overrides any earlier units of the same type with that name. Examples of special rules are *equate*, which establishes a “name equivalence” among units with different names, *order*, which constrains the order in which designated units are merged, *noMerge*, which indicates that particular units are not to be merged (when they otherwise would be), and *bracket*, which indicates that a set of methods should be preceded and/or followed by other specified methods.

Although Hyper/J operates on Java class files, the classes used in composition may be fine-grained, addressing individual concerns, or coarse-grained, combining many concerns. Small, concern-specific classes can be combined into larger, more general components, and general components can be refined, tailored, or extended with respect to individual concerns or sets of concerns. These concerns may address features, classes, aspects, artifacts, business rules, or other dimensions of change, customization, management, and use. Thus, the mechanisms Hyper/J offers are capable of supporting many different kinds and purposes of component evolution.

4.2 Composing Evolvable Components

To test our hypothesis that concerns provide a systematic basis for the evolution of software components, we are experimenting with combining concern-space modeling in Cosmos with software composition by Hyper/J, using the GPS cache as our example component. Aspects of this work are described below.

Generalizing the Concern Space The concern model discussed in Section 3 is relatively specific to the existing GPS cache. We believe that this is a suitable starting point for defining a more comprehensive cache concern space. However, there are many ways in which the evolution of a cache component might lead outside of this initial space:

¹ A version of Hyper/J is available from the IBM alphaWorks web site at <http://www.alphaworks.ibm.com/>.

Workshop on Engineering Complex Object-Oriented Systems for Evolution--OOPSLA 2001

- Features that occur in one class can be applied to other classes, for example, operation and statistics logging, which are defined for the Cache class but which can be extended to the CachedObject class
- New operations and associated behaviors can be introduced, for example, invalidateObject (where originally invalidation applied only to separately named dependencies)
- Previously private methods and behaviors can be made public, as with the method removeExpiredObjects
- New behaviors can be associated to existing operations, for example, throwing an exception rather than returning a default value when an unexpected state is encountered
- Alternative implementation strategies can be made available for existing operations and behaviors
- New properties of interest can be identified, for example, the size of the component, which was not previously a concern
- New classes can be introduced into the component architecture, as we have done to abstract the underlying storage structures.

Many other examples are possible. We cannot expect to outline a complete concern space for a component such as a cache. However, we can readily sketch out a broad concern space that includes many interesting and useful variations.

Populating the Concern Space Given a concern space, we need to populate it with units (Java classes) that implement or otherwise address the various concerns and that can be composed into “complete” cache components. We have been doing this by a combination of extraction from the original GPS cache implementation and new programming. For most of the concerns identified in the original implementation, it is possible to extract suitable code that is relatively concern-specific. Where this has not been possible, and to address new concerns, we have programmed new units. These units typically relatively fine-grained, including a single method with associated fields or a functional group with several related methods and fields. Some units represent core functionality, such as adding or deleting an object in the cache, while others represent aspectual features, such as the logging of operations or the checking of input. Different units also embody different behaviors and reflect different properties. We have populated only a subset of the cache concern space, but to date we have defined about 30 units that cover core functionality, some extended functionality, a selection of alternative behaviors and features, and alternative implementation structures.

Mapping Concerns from Cosmos to Hyper/J Cosmos is based on the same intuitions about concern spaces as Hyper/J, but it provides somewhat different modeling notions. Cosmos does not explicitly distinguish

dimensions, which Hyper/J does, and Cosmos allows multilayered concern models, whereas Hyper/J uses two layers (one level of dimensions containing one level of concerns). For dimension, we use selected high-level concerns in Cosmos. To account for the two-layer model in Hyper/J, we flatten the Cosmos concern name space.

Composing Components We have specified and composed (as of the time of writing) about fifteen alternative versions of the cache, typically involving two to four composed classes. The specifications are expressed in terms of concerns and of mappings from concerns to software units. The concerns and mappings were first modeled in Cosmos and then translated into the appropriate representation for Hyper/J. Thus far we have used only the Hyper/J *mergeByName* composition strategy, typically with some additional specific rules, such as *equate*, *rename*, *forward*, and *bracket*, to “fine tune” the composition and to adapt the names in the resulting composition to those desired in the component interface.

Results In the course of composing alternative versions of the cache, we have been able to add, delete, and replace methods, add, remove and replace fields, change the characteristic behaviors of methods, change the methods to which particular behaviors are associated, change aspects associated to methods, introduce and remove features, affect various properties such as performance, size, and robustness, and substitute alternative implementation structures. Additionally, we were readily able to perform an additional composition in which we reused a subset of units from the cache, along with some newly programmed units, to create a different sort of data structure (a buffer for trees of dependent objects). All of these were easily achieved using the Hyper/J composition mechanisms.

Using the Cosmos model of the cache, we can also analyze the feasibility and impact of prospective changes, with respect to modeled concerns and units. For example, suppose we want to create a simpler, smaller version of the cache that does not support inter-object dependency management. We can see that the dependency concern applies to both the Cache and CachedObject classes, that each of those contains a group of methods related to dependencies, and (not shown here) that the “deleteObject” method of the Cache class may also involve dependencies, depending on the particular behavior with which it was implemented. It also allows us to address questions such as the following: If we wish to use a particular implementation class because it supports some desired property (say, performance), what are the other properties (such as, robustness) that it may affect? Conversely, if we wish to address some property (say, configurability or correctness), what are the logical concerns (such as

functions and behaviors) that relate to it and what are the corresponding implementation classes that may affect it?

A given concern model and collection of implementation units support evolution only within a demarcated concern space. However, once these are established, additions of new concerns or implementations is largely incremental. Adding a new concern involves recording it in the model and identifying appropriate relationships to other concerns. Adding a new implementation class involves relating it to the concerns that it implements, embodies, or otherwise affects. A new implementation class must be programmed so as to interoperate correctly with the classes with which it may be composed, as is true for any extension to a system. By developing these components in a controlled environment, we can minimize potential problems such as differing background assumptions and name conflicts. This is facilitated by a Cosmos model that supports concern tracing and by a high level of concern modularity in existing implementation units.

5. Observations and Issues

Our experience in composing variations of the GPS cache demonstrates that concern-based composition can readily and systematically support many kinds of component evolution. There are up-front costs in modeling the concerns associated with a component and in populating a collection of implementation units from which versions of a component may be generated. These costs are repaid at the time of evolution, when the availability of a concern-space model and of well-modularized components can help in understanding the significance of proposed changes, in analyzing the feasibility and impact of those changes, and in propagating the changes. Unanticipated evolutionary directions that arise later can often be accommodated by incremental extensions of the concern model and code base.

Other approaches to component evolution by concern-based composition are possible. Hyper/J does not require a comprehensive concern-space model to effect changes to a component. It is only necessary to model those concerns that are involved in a particular change and relate them to the affected code units, and that may be sufficient in individual cases. However, without more comprehensive concern-space modeling, general analyses of concerns and impacts are not possible, and if code units are not well-modularized with respect to concerns then the potential for code reuse is limited. As a middle ground, a comprehensive concern-space model can be developed initially for an existing, programmed component, and over time concern-specific modules can be introduced in response to particular evolutionary requirements. In this way the component may gradually become compositional.

The approaches we describe still face significant technical issues. We have previously addressed issues in the design and implementation of Cosmos [7], such as determining appropriate types and attributes of concerns and specializing the schema for particular purposes (such as component evolution). In the work described here, we have faced issues in how particular concern spaces should be modeled. One challenge is how to model concern spaces for prospective components, that is, concern spaces within which new components may be developed, as by evolution or reuse. For example, categorization of concerns becomes more difficult because bindings that are fixed in an existing component (say, between behaviors and methods) are potentially variable for new components [18]. Also, this openness of prospective systems must be captured appropriately; under-specified concern models may overly restrict the apparent paths for evolution or reuse, over-specified models may overwhelm with complexity. Additionally, there are significant issues in the effective use of software composition, for example, in addressing the potential for semantic mismatches between units and in accounting for semantic effects induced by composition.

The GPS cache itself represents yet another approach to coping with component evolution, which we call programmed flexibility. By being general-purpose it can accommodate a wide range of application scenarios and thus may forestall evolution in the sense of program change. At the same time, it supports a kind of evolution by (re)configuration. In contrast, the component evolution strategy that we are investigating is one of flexible composition. However, these are not exclusive. Moreover, generality and configurability are themselves concerns, and through compositional means it is possible to control the level of generality or degree of configurability in a component. Of course, both programmed flexibility and flexible composition can be combined with other approaches to component and system evolution.

The experience we describe here relates directly to component evolution and addresses system evolution through the evolution of components. Concern modeling and separation of concerns also apply on the systems level, and we are separately looking at advanced separation of concerns for systems, for example, for system architectures.

6. SUMMARY AND CONCLUSIONS

The evolution of software systems both depends on and entails the evolution of software components. The evolution of software components is often problematic because prevailing development formalisms do not adequately support separation of concerns. Typically only a few concerns are effectively separated, but many concerns crosscut the prevailing decomposition. Thus, changes are

Workshop on Engineering Complex Object-Oriented Systems for Evolution--OOPSLA 2001

often not well contained and their effects are wide ranging and difficult to predict.

Emerging software technologies support advanced separation of concerns. These approaches treat concerns (and the corresponding modules) as first-class entities, and they facilitate systematic software development and evolution in terms of meaningful concerns.

We have been experimenting with the application of advanced separation of concerns for component evolution, using Cosmos, a schema for modeling software concerns and mapping them to software units, and Hyper/J, a tool for composing Java components based on specified concerns. A cache component has been our main test case.

Using these techniques, we have been able to model a wide range of concerns and flexibly compose differing versions of a cache based on selected concerns. These concerns reflect functionality, features, aspects, component architecture, state, and a variety of properties. Although many issue in the development and use of advanced separation of concerns remain, many kinds of component evolution can be addressed by this approach, and the principles may be applicable on the systems level as well.

7. ACKNOWLEDGMENTS

We thank Peri Tarr, Harold Ossher, and their team for help Hyper/J. For insights that have benefited Cosmos we thank them and Stefan Tai, Thomas Mikalsen, Judah Diament, Mohamed Kande, Chris Codella, and Nagui Halim.

8. REFERENCES

- [1] Batory, D. Refinements and Separation of Concerns. Second Workshop on Multi-Dimensional Separation of Concerns, International Conference on Software Engineering, Limerick, Ireland, June, 2000.
- [2] Castro, J., Kolp M., and Mylopoulos, J. Towards Requirements-Driven Information Systems Engineering, 35 pages. To appear in Information Systems, Elsevier, Amsterdam, The Netherlands, 2001.
- [3] Clarke, S., Harrison, W., Ossher, H., and Tarr, P. Towards Improved Alignment of Requirements, Design, and Code. In Conference Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado. ACM SIGPLAN Notices, v. 34, n. 10 pp. 325--339, October, 1999.
- [4] Degenaro, L. Iyengar, A. Lipkind, I. and Rouvellou, I. A Middleware System Which Intelligently Caches Query Results, in Sventek, J. and Coulson G. (eds.), Proceedings IFIP/ACM International Conference on Distributed Systems Platforms, New York, NY, USA April 2000 (Middleware 2000). Springer-Verlag, Berlin, LNCS 1795, pp. 24-44, April, 2000.
- [5] Harrison, W. and Ossher, H. Subject-oriented Programming (a Critique of Pure Objects). In Proceedings of the Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), September, 1993.
- [6] IBM. Hyper/J. <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>
- [7] IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std. 1471-2000. Approved 21 September 2000.
- [8] ISO/IEC. ISO/IEC 13250 Topic Maps. 47 p. Dec. 1999.
- [9] Iyengar, Arun. Design and Performance of a General Purpose Software Cache; in Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99), Phoenix/Scottsdale, Arizona, Feb. 1999.
- [10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W., "Getting Started with AspectJ," *Communications of the ACM*, October 2001.
- [11] van Lamsweerde, A. Requirements Engineering in the Year 00: A Research Perspective. Invited paper, Proceedings 22nd International Conference on Software Engineering, ACM Press, 2000, pp. 5--19.
- [12] Levy, E., Iyengar, A., Song, J., and Dias, D. Design and Performance of a Web Server Accelerator, in Proceedings of IEEE INFOCOM '99, New York, New York, 1999.
- [13] Merriam-Webster Collegiate Dictionary on line, <http://www.m-2.com/>
- [14] Nuseibeh, B., Kramer, J, and Finkelstein, A. Expressing the Relationships Between Multiple Views in Requirements Specification. Proceedings of the 15th International Conference on Software Engineering (ICSE 15), Baltimore, Maryland; IEEE, 1993, pp. 187-196.
- [15] Parnas, D. L. On the Criteria to be used in Decomposing Systems into Modules. *CACM*, v. 15, n. 12, December, 1972.
- [16] Sutton Jr., S. M. and Rouvellou, I. Concerns in the Design of a Software Cache. Workshop on Advanced Separation of Concerns in Object-Oriented Systems. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, Minnesota, Nov. 2000.
- [17] Sutton Jr., S. M. and Rouvellou, I. Issues in the Design and Implementation of a Concern-Space Modeling Schema. Workshop on Advanced Separation of Concerns in Software Engineering. 23rd International Conference on Software Engineering (ICSE), Toronto, Canada. May 2001.
- [18] Sutton Jr., S. M. and Rouvellou, I. Applicability of Categorization Theory to Multidimensional Separation of Concerns. In submission to Workshop on Advanced Separation of Concerns, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa, Florida, October 2001.
- [19] Tarr, P., Ossher, H., Harrison, W. and Sutton Jr., S. M. N Degrees of Separation: Multidimensional Separation of Concerns. In Proceedings of the 21st International Conference on Software Engineering. ACM, New York, 1999, pp. 107--119.