

Concern Modeling for Aspect-Oriented Software Development¹

Stanley M. Sutton Jr. and Isabelle Rouvellou

Separation of concerns is a fundamental principle of software engineering. Of course, concerns are modeled in a variety of guises in contemporary software development, but the modeling approaches used typically depend on the development method, development stage, artifact formalism, and other project-specific factors. Concerns in various representations are also the focus of aspect-oriented software development (AOSD) techniques. However, concerns as such are still not modeled independently, and concern-modeling is still not a distinguished activity in software development.

In this chapter we argue that concerns must be first-class entities and concern modeling must be a first-class activity in AOSD. That is, concern modeling should be an explicit and integral part of AOSD methods, and concerns should be modeled in their own appropriate formalisms, separately from their representations in requirements, design, code, and other software artifacts. We discuss the meaning of *concern* and consider the characteristics

¹ The final version of this article appears as Chapter 21 (pp. 479—505) in [Aspect-Oriented Software Development](#), edited by Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, Addison-Wesley, 2004, 775 p.

2 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

of concerns in the life cycle. We show that, while existing modeling approaches address concerns in specific contexts for specific purposes, a general-purpose concern-modeling capability is still needed. We describe requirements for a concern-modeling language and discuss the role of concern modeling in the software process. Finally, we give an overview of a general-purpose concern-space modeling schema, Cosmos, which we illustrate with an example based on the transformation of an individual component into a product family.

21.1. INTRODUCTION

Separation of concerns (SOC) is a long-established principle in software engineering [30]. It has received widespread attention in modern programming languages, with constructs such as modules, packages, classes, and interfaces, supporting properties such as abstraction, encapsulation, and information hiding. SOC has also received attention in software architecture and design, with techniques such as composition filters [1] and design patterns [12]. While advances in all of these areas have had significant benefits, problems due to inadequate separation of concerns remain [14]. This has led to recent work on “advanced separation of concerns” (ASOC), including subject-oriented programming and design [8, 13], aspect-oriented programming [11, 21], and multi-dimensional separation of concerns [37]. These bring several innovative ideas to programming in particular and to software development in general, which are now beginning to mature and coalesce under the heading of aspect-oriented software development (AOSD).

Surprisingly, concerns themselves have so far remained something of second-class citizens in ASOD. Current ASOD tools provide only limited support for explicit concern modeling. Representations of concerns tend to be tied to particular tools or artifacts, and concern modeling usually occurs only in the context of a particular type of development activity, such as coding or design [8, 22, 37]. A global perspective on concerns, one that spans the life cycle and is independent of particular development tools or artifacts, has been lacking.

Concerns do not play a second-class role in software development. They arise at every stage of the life cycle, spanning activities, artifacts, methods, and tools. If AOSD is to be fully realized, concerns must be treated as first-class entities throughout the life cycle. Concern modeling must be a recognized and essential part of AOSD methods, and concerns must have independent representations on the same level as requirements, architecture, design, and so on.

Sections 21.2 and 21.3 below define concerns and give a characterization of concerns in the software life cycle. In Section 21.4 we argue that general-purpose, independent concern modeling is needed. Section 21.5 presents some requirements for a concern-modeling formalism and discusses some process issues related to concern modeling. Section 21.6 then gives an overview of the Cosmos concern-modeling schema, followed by an example of concern modeling in Section 21.7. Section 21.8 describes related work in the form of four early-life-cycle approaches to software modeling that address concerns in particular contexts. Section 21.9 provides some additional discussion of issues, and Section 21.10 presents a summary.

21.2. WHAT IS A CONCERN?

Although most software developers have a good intuitive sense of what a concern is, good definitions of *concern* are hard to come by. *Aspects* are one category of concern: an aspect is a (program) property that cannot be cleanly encapsulated in a “generalized procedure” (such as an object, method, procedure, or API) [21]. A later but comparable definition is that an aspect is a program property that forces crosscutting in the implementation [11]. These definitions identify a critical property of some concerns that makes separation of concerns problematic in conventional programming languages. However, it is too focused on program structure (and code) to qualify as a general definition of concern. Tarr and others [37] define a concern as a predicate over software units. This definition is not limited to code and appropriately spans the life cycle, but it is still based on software units.

4 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

To promote concerns to first-class status in software development, they must be defined independently of any specific type of software artifact and even of software artifacts in general. One dictionary definition of *concern* is “a matter for consideration” [27]. More specifically to software, an IEEE standard 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” [17, p. 4]. We take *concern* generally to be any matter of interest in a software system. This may not seem like a particularly technical definition, but it is open, simple, intuitive, encompasses other definitions, and is suitable for many purposes. We give some examples in Section 21.7.

Based on this definition, it follows that concerns are fundamentally *conceptual*. They are *not*, in general, artifacts, although artifacts may represent concerns, and artifacts may be of concern (say, as work products in a development task). Concerns are *not*, in general, requirements, although requirements represent concerns and are of concern at many points in a development process. Similarly, a concern model is *not*, in general, a domain model, although a domain model may contribute concerns (or define a domain over which concerns are expressed).

21.3. A VIEW OF CONCERNS

Given the above definition of concerns as matters of interest in a system, what can we say about them as a prospective domain for modeling? What do we know about them *a priori* based on our overall knowledge of software development and early experience with AOSD?

First, concerns arise at all stages of the software life cycle. This is implicit in the different purposes, activities, formalisms, and artifacts associated with the different stages of development. A variety of concerns, in various forms, are documented for requirements specification [40, 42], architecture and design [8], coding [13, 22, 37], testing [32], and maintenance and evolution [14].

Individual concerns also span multiple phases of the life cycle, relate to multiple instances and types of artifacts, and crosscut phases and artifacts in different ways [4, 32, 37]. For example, a concern for performance may be associated with a set of use cases in requirements, a choice of architectural styles and design patterns that may facilitate performance, algorithms for efficiently implementing specific logical components, and so on.

Finally, concerns are dynamic and relative. The concerns relevant to a particular software product or individual unit change over time, and they also depend on the perspective or purpose of the developer, user, or other stakeholder who considers the software [17, 23, 28, 32].

Based on previous studies, we believe that concern spaces are multidimensional; that is, many concerns of multiple types may apply to a particular software unit at any one time [28, 32, 37]. For example, functionality, behavior, performance, reliability, and understandability may all apply to a particular class or method. Additionally, we believe that a concern space is highly structured; that is, that concerns can be organized by multiple relationships of multiple types, that these relationships may be independent or dependent, and that they commonly have a hierarchical or lattice-like organization [32, 33].

21.4. WHY DO WE NEED CONCERN MODELING?

Most, if not all, of software development can be viewed as modeling related to concerns. We don't call the various activities involved "concern-modeling," though, because their primary focus is on other things: requirements, architecture, design, implementation, and so on. The modeling approaches currently used in software development enable modeling many kinds of concerns in many forms. However, none of them models concerns in general or in the abstract. For example, none of them allows us to say simply that performance is a concern, although they enable us to say that achieving high performance is a goal, or to select an architectural style based on performance properties. In this section we justify modeling concerns explicitly as first class entities.

21.4.1. Specification and Analysis of Concerns across the Development Lifecycle and Artifacts

Concerns, collectively and individually, occur across the development life cycle and development artifacts. Therefore, we need to be able to specify and analyze concerns across the development lifecycle and artifacts. Concerns exist *ab initio* with respect to the lifetime of a system. Although concerns arise throughout system evolution, the system itself arises from some concerns that preceded it, and these concerns exist before and apart from any representation they may have in the system or associated work products. Concerns also span multiple representations in particular artifacts and formalisms, and the relevance of particular concerns crosses development methods, processes, and stages. We can see examples of this with specific concerns such as performance.

21.4.2. Enhancement of Traditional Development Tasks

Concern models have many potential applications that complement or supplement traditional development tasks. One area of application is based on the analysis and understanding of concerns themselves. Individual concerns may be defined, have associated attributes (e.g., related to ownership, priority, or process stage), be related to other concerns (e.g., to show refinement, motivation, dependence), and individually or collectively constrained (e.g., to be used exclusively or in combination). Formal representations of concerns would enable analysis for their completeness, correctness, and consistency. In turn, the analysis of concerns can help in assessing the completeness, correctness, and consistency of software artifacts in which concerns are represented.

Another area of application for concern modeling arises from the relationship of concerns to particular work products. Concerns in a first-class model can be associated to particular requirements statements, design elements, code units, and so on. These may be artifacts from which the concerns are derived, in which they are implemented, through which they are affected, and so on. An independent concern model, with meaningful links into an ar-

tifact set, can be viewed as a kind of semantic, hyper-index to the artifacts (comparable to a topic map [18]). Such a model has many possible uses, including:

- Organizing and associating product elements according to the concerns to which they are related (the refinements of [4] would be an extended case).
- Tracing of concerns across artifacts and indirectly across associated formalisms, development-stages, processes, and organizational units.
- Supporting impact analysis by enabling the determination of concerns affected when artifacts are changed and of artifacts affected when concerns are changed.
- Facilitating the propagation of changes to artifacts linked by shared concerns.
- Supporting rationale capture and analysis, by linking artifacts to the concerns that they implement or influence and by semantically linking those concerns to other concerns that may provide motivation for them or receive contributions from them.
- Contributing to reuse by providing a semantic context in which artifacts (or elements of them) can be considered or presented for reuse.
- Enabling additional analyses for completeness, consistency, correctness of an artifact base with respect to a given concern space.

These applications do not depend on the use of other AOSD technologies or methods. Concern modeling can be used, for example, to facilitate traceability between development stages [26], to analyze a system in preparation for evolution, or to evaluate a candidate component for reuse, all within the context of traditional development methods.

21.4.3. Support of AOSD

Finally, if we are to develop a discipline of aspect-oriented software development based on advanced separation of concerns, it seems appropriate to consider concerns, in general and in the abstract, as first-class entities in their own right. Additionally, concern modeling lends support to specific AOSD approaches. For example, in aspect weaving such as with AspectJ [22], code units that represent a particular aspect such as logging are integrated or “woven” into a base program. However, the base program certainly represents a combination of concerns, and the aspect code, although it typically emphasizes a particular concern, will inevitably involve multiple concerns. For example, logging affects performance and recoverability, and it may serve purposes such as auditing, and analysis. Effective aspect weaving thus depends on the compatibility of concerns in the base model and the aspect.

For code composition with Hyper/J [16], Java classes are organized into a multidimensional concern space. Units are related to specific concerns in specific dimensions, and composition is specified in terms of the dimensions and concerns to be integrated. Modeling of concerns is thus a part of the Hyper/J method. An example of the use of Cosmos with Hyper/J, which motivates the integration of concern modeling with other aspect-oriented tools, is given in Section 21.7.

In general, if the goal of AOSD is to enable us to specify the composition and decomposition of systems in terms of concerns, then the specification, analysis, and interrelation of concerns should be a fundamental part of any fully developed AOSD life cycle. The modeling of concerns, including capturing how they relate to each other and how they are either individually or collectively constrained, is key to allowing a more systematic and therefore manageable use of AOSD techniques. It is a fundamental element to enable AOSD approaches to “scale” to address composition (and decomposition) of “enterprise” systems (e.g., systems where the relevant level of abstraction is not code units or systems that are very complex because of richness of aspects).

21.5. CONCERN MODELING AS A FIRST-CLASS UNDERTAKING

To make concern modeling a first-class undertaking in software development (AOSD, OOSD or otherwise), a schema for the modeling of concerns must be adopted and the activity of concern modeling must be integrated into development processes. In this section, we consider requirements on a concern-modeling schema and discuss issues in integrating concern modeling into the software life cycle.²

21.5.1. Requirements for a Concern-Modeling Schema

Since a concern model is an information model, any language for modeling information or representing knowledge may have some applicability to concern modeling. The initial models we constructed when we began our investigations [33] were informal. Though informal models may be useful for ad hoc or experimental purposes, formal models are more powerful and offer a firmer foundation for an aspect-oriented software engineering discipline. However, different formal languages offer different advantages for different purposes. Thus, while some benefits may indeed be gained by applying existing formalisms to concern modeling, we believe that consideration should be given specifically to the requirements of modeling concerns.

We have identified desirable properties that are useful for a concern-modeling schema, including generality, independence, appropriateness, completeness, and ease of use.

- Generality is important for serving many purposes and allowing users to capture a variety of concerns.

² Tooling and methods are also necessary, but these depend in part on which AOSD technologies adopted and are beyond the scope of this paper.

10 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

- Independence is important for minimizing the effect of other choices of development methods, tools, and approaches on concern modeling.
- Appropriateness is important for facilitating expressing the kinds of concerns that users have and the kinds of information that they want to express about them.
- Completeness is important for allowing users to express all important aspects of a concern model.
- Ease of use is important for facilitating adoption and use.

These desired general characteristics can be addressed through a number of recommendations more specific to concern modeling.

For completeness, it is important to be able to model not only concerns but also relationships among concerns and consistency conditions on concerns. For generality, the schema should allow users to define arbitrary concerns, relationships, and conditions. For completeness and generality, it should be possible to view these relationships and constraints as concerns themselves. Since not all kinds of concerns and relationships can be anticipated, the formalism should include elements for allowing users to define their own concern kinds. This also supports appropriateness, since it would allow specific schemas to be tailored to specific projects. To facilitate ease of use, some specific but commonly useful types of concern and relationship should be included.

To support completeness, generality, and appropriateness, it should be possible to organize concerns in various ways, i.e., by classifications (including multiple classifications), aggregations, and other groupings.

For independence, a general-purpose concern-modeling schema should be independent from other modeling or implementation languages, artifact types, life-cycle stages, and development methods. For independence and generality both it should also be possible to capture concerns (or related data) that are not necessarily captured in other development formalisms.

In asserting that a concern-modeling schema should be independent, we do not mean to suggest that it cannot be based on or used closely with other

languages. Rather, it should be possible for the concern-modeling schema to be meaningful and usable on its own. (For example, a concern-modeling schema may be relational, but it should not depend on the use of relational modeling in other parts of the development process.) Also, while we are advocating a general-purpose concern-modeling schema, it is possible to define more specialized approaches to be integrated with (and dependent on) particular development languages, artifact types, or methods. Within a specialized context, this may increase the appropriateness and ease of use of the formalism.

Finally, many of the potential applications of concern modeling depend on the ability to relate concerns to associated artifacts, that is, artifacts that may represent, define, implement, affect, or otherwise be of significance for the concerns. To some extent this depends on the nature of the artifacts, for example, on the ability to reference and access the artifacts and their constituent parts. Where feasible, concerns should be related directly to artifacts, although it may be problematic to establish the reverse relationships, i.e., from artifacts to concerns (as the artifact representations may not accommodate these references). As an alternative, a model of the artifact space may be constructed, analogous to the domain models used in domain specific software architectures (DSSA) [39] (with the artifacts, in effect, as the domain). In this case, concerns are related to the artifact model, and relationships from the artifact model to the concern model can be easily represented. A similar approach is used in UML (v. 1.4 [29]), in which the physical elements of systems are represented by “nodes” and “artifacts,” and design elements, such as “classes” and “components,” are associated to them.

21.5.2. Process Considerations

In general there are many ways to use concern modeling within a software development process, just as there are many ways to specify requirements or to model architecture. The role of concern modeling depends on the development method and the process objective. The following scenarios suggest the range of possibilities:

12 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

- In a process centered on using commercial, off-the-shelf software (COTS), a concern model of the system under development (SUD) may be constructed for evaluating the compatibility and potential contribution of candidate COTS products. These products can be characterized in terms of the concerns they address and evaluated against the concern framework of the SUD. The suitability of particular products can be determined based on the consistency or inconsistency of concerns, the range of concerns addressed, the need for tailoring to assure compatibility, and the concerns remaining to be addressed.
- In new development, a limited concern model can be elaborated at the start of development to represent the initial concerns that motivate or constrain the project. As development progresses through requirements, architectural design, detailed design, and so on, the concern model can be elaborated. Concerns can be related to the various work products in which they are defined, implemented, or otherwise addressed, and relationships between concerns can be drawn to capture semantic, operational, and other dependencies. The concern model can be analyzed to assess the consistency and coherence of the concerns associated to the project. As the life cycle iterates, changes made at various stages can be validated against the concern model, and the concern model can be used to help propagate updates through both the concern space and the artifact space.
- In the retroactive extension of a product line based on an existing product, concern modeling can be used to characterize the potential feature space and to characterize the product variants within that space. An initial concern model can be developed of the existing product. The product can be decomposed into units corresponding to specific concerns. Concerns representing additional features can be identified and related to the existing model, and additional work products can be developed to support implementation of the additional concerns. Variants within the product family can then be

specified by selecting concerns of interest from within the product-family concern space. Compositional technologies may then be used to compose a product variant using implementations associated to the selected concerns. This sort of scenario is discussed in more detail in Section 21.7.

As these scenarios may suggest, concern modeling may be done before initial development, during development, or as specific needs arise afterwards. It may be done independently of, or in close association with, other development activities. It may be done in one step or incrementally. It may be for particular work products or sets of concerns or comprehensively for a product or complete concern space. Finally, it may be done for general purposes or to address specific problems.

21.6. COSMOS: A CONCERN-SPACE MODELING SCHEMA

Cosmos is a general-purpose concern-space modeling schema that addresses the requirements outlined in Section 21.5.1. *Cosmos* models software concern spaces in terms of *concerns*, *relationships*, and *predicates*. We give here a concise overview and example of *Cosmos*. A more detailed discussion can be found in [36].

Cosmos divides concerns into two main categories, logical and physical. Logical concerns represent conceptual concerns, as “matters of interest”: issues, problems, “ilities,” and so on. Physical concerns deal with 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. They play a role analogous to that of nodes and artifacts in UML [29].

Table 20-1 Cosmos Concern Model Elements: Outline

- | | |
|--|--|
| <ul style="list-style-type: none"> •Concerns <ul style="list-style-type: none"> ○ Logical <ul style="list-style-type: none"> ▪ Classifications ▪ Classes ▪ Instances ▪ Properties ▪ Types ○ Physical <ul style="list-style-type: none"> ▪ Collections ▪ Instances ▪ Attributes ○ Predicates ○ Relationships ○ Groups •Predicates <ul style="list-style-type: none"> ○ // subtypes not elaborated | <ul style="list-style-type: none"> •Relationships <ul style="list-style-type: none"> ○ Categorical <ul style="list-style-type: none"> ○ Classification ○ Generalization ○ Instantiation ○ Characterization ○ Topicality ○ Attribution ○ Membership •Interpretive <ul style="list-style-type: none"> ○ <i>Contribution</i> ○ <i>Motivation</i> ○ <i>Admission</i> ○ <i>Logical implementation</i> ○ <i>Logical composition</i> ○ <i>Logical requisition</i> •Physical <ul style="list-style-type: none"> ○ Physical association ○ <i>Physical requisition</i> •Mapping <ul style="list-style-type: none"> ○ Mapping association ○ <i>Physical implementation</i> |
|--|--|

Note: Items in normal font are part of the core schema; items in italic font are representative schema extensions used in particular concern models

For completeness, and consistent with a multidimensional perspective on concerns, predicates and relationships are also classified as concerns. In other words, a predicate over concerns or a relationship between concerns may itself represent a matter of interest. In many situations, though, it is natural to think of predicates and concerns as top-level elements of the concern space, which is how we discuss them here. There is also a fifth kind of concern, concern groups. These represent the idea that groups of concerns may also be of concern. Concern groups are parameterized by the type of concern they can contain. The element type can be any type in the hierarchy

of model elements, so specific concern groups can be made inclusive or exclusive as necessary.

Logical concerns are further categorized as classifications, classes, instances, properties, and topics. Classifications are for modeling systems of classes and allow for multiple classification of concerns [35]. Classes are for categorization of concerns, for example, by functionality, behavior, or state.³ Instances represent particular concerns, usually of some class, e.g., particular functions, behaviors, or states. Properties are characteristics, such as performance, configurability, robustness, that may apply to classes and instances. Topics are groups of concerns of generally different types, typically related to a theme of user interest, for example, the classes, instances, and properties that are related to the theme of “logging.”

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.

Relationships are divided into four categories: categorical, interpretive, physical, and mapping. *Categorical relationships* reflect fundamental semantics of the concern categories. For example, generalization, which relates (sub)classes to (super)classes, instantiation, which relates instances (both logical and physical) classes, and characterization, which relates properties to kinds and instances.

Interpretive relationships relate logical concerns according to user-assigned semantics. One example is contribution, which indicates that one concern (e.g., logging behavior) contributes in some way to another (e.g., robustness). Another interpretive relationship is motivation, which indicates that one concern (e.g., robustness) motivates another (e.g., logging). Such

³ Classes of concern in Cosmos should not be confused with classes in design notations or programming languages. Cosmos classes serve the purpose of classification of concerns in a taxonomic or ontological sense, which we believe is essential to concern modeling.

relationships are especially important in understanding the system-specific semantics of concerns.

Physical relationships associate physical concerns, such as composition relationships that associate Java classes in Hyper/J [16] or connectivity relationships among nodes in a network.

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 important (along with interpretive relationships) for purposes such as dependency analysis, impact assessment, and change propagation. They are also important for assessing component reuse and composition potential.

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. An example of the latter is that, if one concern motivates another, then the second should contribute to the first (whereas the converse is not required).

21.7. A CONCERN-MODEL EXAMPLE

This section provides an example of concern-modeling based on the GPS cache, a general-purpose (software) cache that is intended to be reusable in a variety of applications [19]. The GPS cache is richly functioned and featured. It supports the usual functionality associated with a cache and it has 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 [24] (including IBM web sites for major sports events) and in support of business rule evaluation [9].

We have been interested in the GPS cache as the potential prototype for a “component family,” a collection of specialized caches with various, more or less overlapping combinations of features, functions, behaviors, and properties. Our approach to developing this family of caches was based on as-

pect-oriented ideas and tools. In particular, we wanted to allow a cache to be specified in terms of a particular set of desired concerns, and then to enable a cache that addressed those concerns to be composed from reusable, concern-specific fragments of Java code. To achieve this, we used Cosmos for concern modeling and Hyper/J [16] for composition of Java units. This work is described at more length in [34].

Here we consider our initial modeling of concerns for the GPS cache, primarily as it was programmed, but also including some obvious generalizations. This modeling served two purposes. First, it established a baseline for the concerns that a cache may address and provided a core around which we could organize conceptions and models of what a cache is and how it may be subject to variation [34]. Second, this modeling allowed us to see which concerns were addressed in which parts of the cache code. This provided a basis for decomposing the code into more specialized units that were suitable for recomposition into alternative variants.

Examples of high-level concerns in the implementation of the GPS cache are outlined in the Tables “Selected Concerns from the GPS Cache” Parts 1 and 2. (For discussion of relationships and predicates see [36].)

According to the Cosmos model, concerns are organized as logical or physical, with logical concerns here including concern classes, instances, properties, and topics, and physical concerns including instances, collections, and properties. The top-level concern classes we distinguished included implementation object classes, functionality, behavior, state, properties, and Java units.

Implementation-object classes are concerns in and of themselves and are also used to organize subclasses under other concern classes (such as functionality, behavior, and state). Under functionality, nine different functional areas are associated with the `Cache` class, grouping from one to eight methods (omitted for brevity). Methods in the “printing” group are also included under other groups as they print information relative to those groups.

Table 20-2 Selected Concerns from the GPS Cache (Part 1)

Logical Concerns: Classes

- Object classes
 - Cache
 - CachedObject
 - *Other object classes ...*
- Functionality
 - Cache
 - Core
 - Object expiration
 - Operation-enabling
 - Dependencies
 - Object invalidation
 - Operation logging
 - Statistics logging
 - Printing
 - CachedObject
 - Core
 - Expiration
 - Dependencies
 - *Other functionalities ...*
 - *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
 - *Other operation behaviors*
 - *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 the `cachedObject` class, existing methods fall into three groups, and additional groups (not initially used) can be identified by generalization from the `Cache` class; these can be populated later.

Table 20-2 Selected Concerns from the GPS Cache (Part 2)

Logical Concerns: Other Categories

Instances

- Omitted for brevity

Topics

- Dependencies and transitivity
- Configurable behaviors
- Other topics ...

Properties

- Generality
- Performance
- Information hiding
- Concurrency
- Configurability
- Correctness
- Other properties ...

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

- whimbrl.watson.ibm.com
C:\\$Sutton\Caching\Code\Java\Prog
rammed
- Other collections ...

Attributes

- com.ibm.ws.abr.gps.-
Cache.java.Size
- Other attributes ...

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.

A number of themes occur repeatedly under various kinds of concern, for example, “core,” “logging,” and others. Such crosscutting concerns rep-

resent other dimensions by which the concern space can be organized. For instance, object classes could occur under “core” and under “logging” instead of vice versa. Topics are one way to represent such crosscutting 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 classifications, and any of several classifications might be distinguished as the basis of a view of the concern space.

We continued our work on the cache product family by decomposing the cache implementation into small units focused on specific concerns. We then elaborated the concern model to include concerns of interest that were not addressed in the original cache, programming implementations for these. Using Hyper/J [16], and transforming the Cosmos concern-space model into the Hyper/J representations for a hyperspace specification and concern mapping, we composed about fifteen alternative versions of the cache. In these we added, deleted, and replaced methods; added, deleted, and replaced fields; changed the behaviors of methods; changed the methods to which particular behaviors were associated; changed aspects associated to methods; introduced and removed features; affected properties such as performance, size, and robustness; and substituted implementation structures. Using the same units, many additional cache variants could be composed to address additional combinations of concerns.

21.8. RELATED WORK

In this section we consider approaches to modeling at the early stages of the software life cycle. Early-stage activities, such as requirements specification and architectural design, are of particular interest because these are activities whose primary purpose is to introduce concerns into a development project. We consider two sets of modeling approaches: “traditional”, non-aspect oriented modeling approaches, and new aspect-oriented modeling approaches.

21.8.1. Traditional (Non-Aspect Oriented) Modeling

We describe here some representative modeling approaches in which concerns are identified in the form of requirements or design elements but in which concerns, as such, are not first-class entities. The lack of consideration for concerns as first-class entities is one of the principal motivations for Cosmos.

21.8.1.1. Requirements Engineering: I* and Tropos

I* is a framework for modeling organizations in terms of actors, goals, and dependencies. Tropos is a methodology that applies this to early requirements and provides a basis for extending early requirements to late requirements, architectural design, and detailed design [7, 42].

Tropos emphasizes the need to identify organizational concerns, separate them from implementation concerns, and give them first-class treatment. Toward this end, Tropos posits five main classes of concern: actors, resources, goals, soft goals, and tasks. A particular requirements model will contain multiple instances of each of these classes. Properties are not represented directly in the Tropos schema but may be captured in hard or soft goals (e.g., “increase friendliness of customer service”). Tropos (and I*) incorporate several types of concern relationship, including decomposition, means-ends, and dependency relationships.

21.8.1.2. Requirements Engineering: KAOS

KAOS [5], like I*, offers a goal-dependency model of requirements and takes a view of concerns that is appropriate to requirements and separated from implementation. KAOS also provides some downstream continuity, from requirements to architectural refinement.

In some contrast to Tropos, KAOS adopts a more explicitly multidimensional perspective on requirements. Conceptually, requirements in the abstract and elements in a model can be associated with different “aspects” (in their terms) such as “why,” “who,” “when” and “what.” Requirements also have a dual linguistic dimension, including both an “outer” semantic net

for general types and semantic relationships and an “inner” assertion language for detailed temporal and logical semantics. Goals can be further classified according to their domain, for example, robustness, safety, efficiency, and privacy. Goals are also subject to disjunctive and conjunctive refinement. The KAOS system also supports multiple views of the requirements, including refinement, operationalization, entity-relationship, and agent.

As in Tropos, properties *per se* are not a first class construct in KAOS, but are represented indirectly by other constructs such as goals and constraints. In addition to refinement, KAOS relationships also include operationalization and responsibility.

21.8.1.3. Architectural Design: ABAS

ABAS are “attribute-based architectural styles” [23]. These styles address specific quality attributes and can be analyzed in terms of these attributes.

ABAS modeling is explicitly multidimensional. It incorporates a number of classifications, including classification of architectural attribute information (in terms of external stimuli, architectural decisions, and responses), classification of architectural elements (in terms of components, connectors, and properties), classification of ABAS specification elements (such as problem description and stimulus/response attribute measures), and classification by property. Additionally, parameters in each of the categories of information are subject to multiple simultaneous classifications. For example, stimuli are classified with respect to mode, source, and regularity, and responses are classified with respect to latency, throughput, and precedence.

ABAS, in contrast to the requirements methods described above, treat properties explicitly and prominently. All architectural styles are classified with respect to the properties of performance, modifiability, and availability. These crosscut the information categories, so that the kinds of information used to describe stimuli, architectural decisions, and responses for performance are different from those used for modifiability. For example, performance responses can be described in terms of latency and throughput, modifiability responses in terms of extent of impact and effort of change.

The sorts of relationships that are emphasized in the requirements modeling approaches (dependency, responsibility) are not highlighted in ABAS. Of course, ABAS describe kinds of physical relationships (connections) among components that observe various architectural styles. Additionally, ABAS include potentially detailed and quantitative analytical models that describe how specific property measures relate to architectural elements and changes to those elements.

21.8.1.4. Architectural Design: ABAS

Domain-specific software architectures (DSSA) make (repeated) use of a domain model, reference requirements, and reference architectures common to a family of applications [39]. The domain model usually includes a lexicon, ontology, and taxonomy of terms and entities belonging to the domain. The domain may be characterized in terms of objects, relationships, products, behaviors, and so on. The architecture, depending on the style and representation, typically comprises elements such as components, connectors, constraints, dependencies, responsibilities and capabilities. The reference requirements are typically expressed in terms of the domain model and linked to elements in the reference architecture.

All the elements of a DSSA—domain model, requirements, and architecture—express concerns of some sort. The domain model defines a domain of concern (or a domain of concerns) but in a way that is abstract from the motivations and objectives of any particular project (i.e., from the things that make a “matter of interest” interesting). Requirements express concerns in the form of specific needs and goals relating to applications in the domain. Architecture begins to introduce concerns about how those needs and goals are to be addressed.

21.8.1.5. Observations on Non-Aspect Oriented Modeling Approaches

The approaches described above are advanced techniques for modeling software requirements or architecture. These all represent concerns, or enable

users to represent concerns, in a particular form for a particular purpose. For example, ABAS identify concerns, in the form of architectural features and properties, which are associated with particular architectural styles but independent of a domain. DSSA, in contrast, present domain-specific concerns in the form of a domain model, requirements, and architecture. Tropos/I* and KAOS support users in specifying concerns in the form of requirements. Thus, the outcome of these approaches are elements specific to requirements and design, even though the concerns they represent typically cut across activities and artifacts from the whole life cycle. Cosmos is intended to support a concern modeling approach that can transcend particular activities and artifacts and span the whole life cycle.

Each of the modeling approaches discussed above also reflects some explicit or implicit assertion about how concerns of various types should be represented. However, these approaches typically draw on many of the same modeling techniques, such as enumeration, single and multiple classification, views, templates, relationships and associations, abstraction, properties and attributes, and conditions and constraints. These modeling techniques are thus generic with respect to modeling approaches and domains, and Cosmos likewise makes use of many of them.

21.8.2. Aspect-Oriented Modeling

There has been a flurry of recent work in the areas of aspect-oriented requirements engineering and architectural analysis, as well as in the area of more general concern modeling. In contrast to the modeling approaches reviewed above, a major goal of these recent approaches, as for Cosmos, is to identify concerns or aspects as such. We discuss some examples here.

21.8.2.1. Aspect-Oriented Requirements Engineering and Architectural Analysis

Aspect-oriented requirements engineering (AORE), like all requirements-engineering, aims to capture requirements. However, it explicitly recognizes that at least some requirements represent aspects or concerns that will cross-

cut both requirements and downstream life-cycle artifacts. Thus, AORE approaches include techniques for explicitly modeling aspects or concerns, at least in the context of requirements specification. For example, Brito and Moiera [6] define a process for separation of concerns in requirements that includes steps for identifying concerns, specifying concerns, identifying crosscutting concerns, and composing concerns. Baniassad and Clark [2] propose the Theme/Doc approach to identify crosscutting behaviors that represent aspects in requirements documentation. Tekinerdogan [38] argues that explicit mechanisms are needed to identify, specify, and evaluate aspects in architectural design. He proposes ASAAM, an Aspectual Software Architecture Analysis Method, that provides a set of heuristic rules to allow architectural aspects to be identified based on usage scenarios. Bass, Klein, and Northrop [3] are developing a method to derive a software architecture from required quality attributes and propose that these attributes often represent architectural aspects that can be carried through detailed design and implementation.

21.8.2.2. Concern Modeling

Concern modeling in a still more general sense is now addressed by several approaches. Wagelaar [41] proposes a concept-based approach called CoCompose for the modeling of early aspects. In CoCompose the concepts involved in a software system are first modeled independently of any implementation; the conceptual models can then be processed to automatically generate an implementation. Lohman and Ebert [25] propose a generalization of the Hyperspaces [16, 37] approach to concern modeling that replaces orthogonal dimensions of concerns with nonorthogonal clusters of concerns and allows a unit to be assigned to more than one concern in a dimension. Lohman and Ebert distinguish primary and secondary dimensions of concern in which the primary dimensions are based on artifacts and the secondary dimensions represent user interests that are not derived from corresponding artifacts, although these concerns may still be related to artifacts.

Finally, IBM, in part as a successor to Hyper/J [16], initiated development of the Concern Manipulation Environment (CME), a platform for the

development and application of cross-life cycle aspect-oriented technologies [31, 15]. The CME includes a concern-management component, ConMan, that supports general-purpose, multidimensional concern modeling, including concerns, relationships, predicates, and various ways to group and associate these. Concerns may be related to artifacts or independent of them, and concerns may be used to organize artifact composition and extraction, to support querying and analysis, and for other purposes with or without artifacts. The CME is now an Eclipse Open-Source project [10].

21.8.2.3. Observations on Aspect Oriented Modeling Approaches

The aspect-oriented approaches to requirements and architecture analysis take a step toward generality of concern modeling: they enable particular requirements or architectural elements to be specified, and they recognize that these elements represent more general aspects or concerns that crosscut multiple development stages and artifacts. However, concern modeling in these approaches is still focused mainly on particular activities and on artifact representations appropriate to those activities. Approaches such as those in [41] and [25] take concern modeling a further step toward generality. They provide very general notions for the modeling of concerns, although ties to artifacts are still prominent in these models. ConMan in the CME provides the most general and artifact-neutral approach to concern modeling. ConMan, like Cosmos, is a generalization of the Hyperspaces approach, although the basic concepts in Cosmos are at a somewhat higher semantic level than those in ConMan. For instance, ConMan includes a wide variety of constructs for grouping concerns, whereas Cosmos has just a few simple constructs for grouping concerns but also includes constructs for semantically categorizing concerns. In the future we hope to implement Cosmos concepts on top of ConMan.

21.9. ADDITIONAL DISCUSSION

In this section we address two fundamental questions: Can we achieve a useful degree of formalization for such a general notion as “concern”? And what is the nature of the contribution that concern modeling may make to AOSD?

21.9.1. Formalization of the Notion of “Concern”

We have purposely adopted a notion of concern that is general and intuitive. It is therefore reasonable to wonder in what ways and to what extent this notion can be formalized.

We should note first that our interest has been primarily with representing concern *spaces* rather than individual concerns. That is, we have focused on categories of concern and on organizational and semantic relationships among concerns rather than on how best to represent the details of particular concerns. The latter does deserve attention, but we expect that different domains of concerns will have different, specialized languages for description. For example, distinct formalisms may be used to describe concerns relating to functionality (perhaps algebraic specifications), performance (perhaps numerical models), and consistency (perhaps Object Constraint Language (OCL) [29]). Moreover, it should be kept in mind that the description of a concern in a concern modeling schema such as Cosmos is not intended to replace the representation of that concern in other software artifacts. In Cosmos a concern represents a subject of interest, the details of which should be found mainly in the associated artifacts such as requirements, architecture, designs, and code, through which system-specific details are defined and implemented.

With regard to the representation of concern spaces, it is possible to introduce a level of formality that is comparable to that found in other sorts of information modeling used in software development, including domain modeling [20, 39], semantic dependency modeling [7, 42], or design modeling [29]. These kinds of modeling typically entail some representation of entities, kinds of entities, properties of entities, and various sorts of semantic or

structural relationships among entities. With Cosmos we have demonstrated that this sort of modeling is possible with concerns.

The level of formality (or formalizability) in the Cosmos schema is sufficient to have allowed us to construct object-oriented models of the schema in UML [29]. In this model, for example, `LogicalInstance` is defined as an extension of `LogicalConcern`, which is defined as an extension of `Concern`, which is defined as an extension of `ConcernModelElement`; operations on instances of the schema (that is, on specific concern models) can take place at any of these levels of abstraction. Given a Cosmos model, it is possible to algorithmically define many sorts of analyses. Questions about a Cosmos model that can be answered by analysis include, for example: Are there cycles in the concern-class structure? Are there classes without instances or instances without classes? Does every concern have a motivation or make a contribution? Additionally, it is possible to define consistency conditions for a concern model. For example, no logical class should contain both logical instances and physical instances; and, for any two concerns X and Y , if X motivates Y then Y should contribute to X . For a UML model of the Cosmo schema we have defined such constraints using OCL [29].

Thus, it appears possible to formalize modeling concern spaces. As with other sorts of information modeling in software development (and elsewhere), the purpose such formalization is to take notions that are intuitive and informal and render them precisely. Formalization allows using information not just in one specific tool, but in a variety of ways.

21.9.2. Nature of the Contribution to AOSD

AOSD provides new tools and methods for developing software based on concerns. It uses advanced techniques for separation of concerns, with an emphasis on separating and composing crosscutting concerns. In general, AOSD allows for the separate, modular representation of information associated with specific concerns. Examples of such information include requirements, designs, and code. In particular, AOSD seeks the modular representation of concerns that might be scattered across conventional representations, such as object-oriented models or code, in which a particular decomposition

predominates. AOSD also automates the weaving or composition of concern-specific units according to explicit specifications. In this way, it controls and systematizes the distribution of concern-specific elements across differently organized application architectures. AOSD thus depends on an understanding of concerns and their interrelationships.

Concern modeling allows us to directly represent and reason about concerns and their interrelationships, thereby creating an externalized abstraction for organizing, analyzing, managing, and composing artifacts according to concerns. This affords several advantages with respect to aspect-orientation. It reveals crosscutting relationships among concerns (as when one concern is defined with respect to another). It provides a reference model of concerns that may be organized and distributed in different ways in different artifacts (for example, as when requirements are organized by functional decomposition but designs are organized by object decomposition). Finally, for software rich in aspectual properties, it helps with managing complexity, such as when multiple concerns are tangled within a single artifact, or when individual concerns scattered across multiple artifacts. All these contributions are important for helping AOSD to scale from individual components and applications to product families and enterprise-level systems.

21.10. SUMMARY

The notion of concerns has long received attention in software development. However, until now, concerns have remained something of second-class citizens. Although existing software development activities address concerns in specific ways for specific purposes, there has been no effort to model concerns as such, in general and in the abstract.

The advent of aspect-oriented software development and new techniques for advanced separation of concerns have made plain the need to treat concerns as first-class entities and concern modeling as a first-class activity. There are many reasons to do this. They include the need to address concerns that occur across life-cycle stages and artifacts, the ability to enhance tradi-

tional development tasks with concern-based analyses and technologies, and the need to support aspect-oriented software development.

If concern modeling is to become a first-class undertaking in software development, then a formalism for the modeling of concerns must be adopted and the activity of concern modeling must be integrated into development processes. At a high level, the requirements for a concern-modeling formalism are generality, independence, appropriateness, completeness, and utility. The integration of concern modeling into development processes may be done in many ways, but in general concern modeling may be done before, during, or after initial development, and it may be adopted comprehensively or incrementally.

Cosmos is a schema designed specifically for general-purpose concern-space modeling. Cosmos models concerns spaces in terms of concerns in several categories, relationships of several types, and predicates. A notable example of the use of Cosmos (along with Hyper/J for Java composition), is in the reengineering of a monolithic, general-purpose software component. Decomposing the component according to concerns, we created a varied product line of specialized components tailored according to selected concerns of interest.

Concerns can be considered the essence of software. All software must address and embody concerns, and software development is fundamentally about identifying, realizing, and reconciling concerns. Aspect-oriented software development with concern modeling provides new tools and methods for developing software based on concerns. It allows for the separate, modular representation of information (such as requirements, designs, or code) that is associated with particular concerns, including concerns that might be scattered and tangled in conventional representations (such as object-oriented models or code). It also automates the weaving or composition of concern-specific units according to concern-based specifications, thereby controlling and systematizing the distribution of these units across differently organized application architectures. AOSD thus depends on understanding and leveraging concerns and their interrelationships and constraints. Concern modeling externalizes the abstractions of concerns and their interrelationships and constraints, and it allows them to be specified, analyzed, and understood as in-

dependent, first-class entities. Concern modeling is thus fundamental to aspect-oriented software development, and it holds the potential to enhance other approaches to software development as well.

ACKNOWLEDGMENTS

For discussion on MDSOC and Hyper/J, we thank Peri Tarr and Harold Ossher. For sharing his experience with Cosmos, we thank Juri Memmert. For help with the GPS cache, we thank Arun Iyengar and Lou Degenaro. For further discussions on MDSOC we thank Stefan Tai and Thomas Mikalsen.

REFERENCES

1. AKŞIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. 1993. Abstracting object-interactions using composition-filters. In *Object-Based Distributed Processing*, R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds. LNCS, vol. 791. Springer-Verlag, Berlin, 152–184.
2. BANIASSAD, E. AND CLARKE, S. 2004. Finding aspects in requirements with Theme/Doc. In *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (AOSD)* (Lancaster, UK). <http://trese.cs.utwente.nl/workshops/early-aspects-2004/Papers/Baniassad-Clarke.pdf>
3. BASS, L., KLEIN, M., AND NORTHROP, L. 2004. Identifying aspects using architectural reasoning. In *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (AOSD)* (Lancaster, UK). <http://trese.cs.utwente.nl/workshops/early-aspects-2004/Papers/BassEtAl.pdf>

32 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

4. BATORY, D. 2000. Refinements and separation of concerns. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE)* (Limerick, Ireland).
<http://www.research.ibm.com/hyperspace/workshops/icse2000/Papers/batory.pdf>
5. BERTRAND, P., DARIMONT, R., DELOR, E., MASSONET, P., AND VAN LAMSWEERDE, A. 1998. GRAIL/KAOS: An environment for goal driven requirements engineering. *Research Demonstration and Handout, 20th Int'l Conf. Software Eng (Kyoto)*.
6. BRITO, I, AND MOREIRA, A. 2004. Integrating the NFR framework in a RE model. In *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (AOSD)* (Lancaster, UK).<http://trese.cs.utwente.nl/workshops/early-aspects-2004/Papers/BritoMoreira.pdf>
7. CASTRO, J., KOLP, M., AND MYLOPOULOS, J. 2002. Towards requirements-driven information systems engineering: The Tropos project. *Information Systems* 27, 6, 365–389.
8. CLARKE, S., HARRISON, W., OSSHER, H., AND TARR, P. 1999. Subject-oriented design: Towards improved alignment of requirements, design and code. In *14th Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (Denver). ACM, 325–339.
9. DEGENARO, L., IYENGAR, A., LIPKIND, I., AND ROUVELLOU, I. 2000. A middleware system which intelligently caches query results. In *Middleware 2000: IFIP/ACM Int'l Conf. Distributed systems platforms* (New York). LNCS, vol. 1795. Springer-Verlag, Berlin, 24–44.
10. ECLIPSE.ORG. 2004. Concern Manipulation Environment Project, <http://www.eclipse.org/cme>
11. ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-oriented programming. *Comm. ACM* 44, 10 (Oct.), 29–32.

12. GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.
13. HARRISON, W. AND OSSHER, H. 1993. Subject-oriented programming—a critique of pure objects. In *8th Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (Washington, D. C.). ACM, 411–428.
14. HARRISON, W., OSSHER, H., AND TARR, P. 2000. Software engineering tools and environments: A roadmap. In *Conf. Future of Software Engineering (Limerick)*. ACM, 261–277.
15. IBM CORPORATION. Concern manipulation environment (CME): a flexible, extensible, interoperable environment for AOSD. <http://www.research.ibm.com/cme/>
16. IBM CORPORATION. Hyperspaces. <http://www.research.ibm.com/hyperspace/>.
17. IEEE. 2000. IEEE recommended practice for architectural description of software-intensive systems. IEEE Std. 1471-2000.
18. ISO/IEC. 1999. ISO/IEC 13250 topic maps.
19. IYENGAR, A. 1999. Design and performance of a general purpose software cache. In *18th Int'l Performance, Computing, and Communications Conf. (IPCCC)*, (Phoenix). IEEE, 329–336.
20. JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. 1999. *The Unified Software Development Process*. Addison-Wesley, Reading, Massachusetts.
21. KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. Getting started with AspectJ. *Comm. ACM* 44, 10 (Oct.), 59–65.

34 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

22. KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming, 11th European Conference*, M. Akşit and S. Matsuoka, Eds. LNCS, vol. 1241. Springer-Verlag, Berlin, 220–242.
23. KLEIN, M. AND KAZMAN, R. 1999. Attribute-based architectural styles. Tech. Rep. CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University. Oct.
24. LEVY, E., IYENGAR, A., SONG, J., AND DIAS., D. 1999. Design and performance of a web server accelerator. In *INFOCOM '99 (New York)*. IEEE, 135–143.
25. LOHMANN, D., AND J. EBERT., J. 2003. A generalization of the hyperspace approach using meta-models. In *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (AOSD) (Boston, USA)*. <http://www.cs.bilkent.edu.tr/AOSD-EarlyAspects/Papers/LohEbe.pdf>
26. MEMMERT, J. 2001. Personal communication.
27. Merriam-Webster. Merriam-Webster collegiate dictionary on line. <http://www.merriam-webster.com/>.
28. NUSEIBEH, B., KRAMER, J., AND FINKELSTEIN, A. 1993. Expressing the relationships between multiple views in requirements specification. In *15th Int'l Conf. Software Engineering (ICSE) (Baltimore, Maryland)*. IEEE, 187–196.
29. OBJECT MANAGEMENT GROUP. 2001. OMG Unified Modeling Language specification, version 1.4.

30. PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15, 12 (Dec.), 1053–1058.

31. SABBAAH, D. 2004. Aspects--from promise to reality. In 3rd International Conference on Aspect-Oriented Software Development (AOSD) (Lancaster, UK). ACM, 1-2.

32. SUTTON JR., S. M. 1999. Multiple dimensions of concern in software testing. In *Workshop on Multi-Dimensional Separation of Concerns (OOPSLA)*, (Denver).

<http://www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws13-sutton.pdf>

33. SUTTON JR., S. M. AND ROUVELLOU, I. 2000. Concerns in the design of a software cache. In *Workshop on Advanced Separation of Concerns (OOPSLA)*, (Minneapolis).

<http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/sutton.pdf>

34. SUTTON JR., S. M. AND ROUVELLOU, I. 2001. Advanced separation of concerns for component evolution. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) (OOPSLA)*, (Tampa, Florida). <http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml>

35. SUTTON JR., S. M. AND ROUVELLOU, I. 2001. Applicability of categorization theory to multidimensional separation of concerns. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA)*, (Tampa, Florida).

<http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/05-sutton.pdf>

36. SUTTON JR., S. AND ROUVELLOU, I. 2002. Modeling of software concerns in Cosmos. In *1st Int'l Conf. Aspect-Oriented Software Development (AOSD)*, (Enschede, The Netherlands), G. Kiczales, Ed. ACM, 127–133.

36 Chapter 21 Concern Modeling for Aspect-Oriented Software Development

37. TARR, P., OSSHER, H., HARRISON, W., AND SUTTON JR., S. M. 1999. *N* degrees of separation: Multi-dimensional separation of concerns. In *21st Int'l Conf. Software Engineering (ICSE)*, (Los Angeles). IEEE, 107 – 119.

38. TEKINERDOGAN, B. 2003. ASAAM: aspectual software architecture analysis method. In *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (AOSD)* (Boston, USA). <http://www.cs.bilkent.edu.tr/AOSD-EarlyAspects/Papers/Tekinerdogan.pdf>

39. TRACZ, W. 1994. DSSA frequently asked questions. *Software Engineering Notes* 19, 2, 52–56.

40. VAN LAMSWEERDE, A. 2000. Requirements engineering in the year 00: A research perspective. In *22nd Int'l Conf. Software Engineering (ICSE)*, (Limerick, Ireland). IEEE, 5–19.

41. WAGELAAR, D. 2003. A concept-based approach for early aspect modelling. In *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design Workshop (AOSD)* (Boston, USA). <http://www.cs.bilkent.edu.tr/AOSD-EarlyAspects/Papers/Wagelaar.pdf>

42. YU, E. S. AND MYLOPOULOS, J. 1994. Understanding “why” in software process modeling, analysis, and design. In *16th Int'l Conf. Software Engineering (ICSE)* (Sorrento, Italy). IEEE, 159–168.

43.