Applicability of Categorization Theory to Multidimensional Separation of Concerns

Stanley M. Sutton Jr. and Isabelle Rouvellou
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{suttonsm, rouvellou}@us.ibm.com

ABSTRACT

Prototype theory is a cognitive theory of categorization that describes many aspects of MDSOC better than the classical theory of hierarchical categories. Prototype theory implies that composition is a more natural means of specifying components than is inheritance. Prototypes are useful in organizing information and workflows in component composition systems.

1. INTRODUCTION

Multidimensional separation of concerns (MDSOC) [6] inherently entails problems of categorization. Software units are categorized into one or more concerns, concerns are grouped into dimensions or higher-level concerns. Categorization provides MDSOC with a means for organizing, describing, accessing, and operating on software artifacts, in particular with parts (or subparts) of artifacts related to particular concerns.

In the context of MDSOC, the use of categories is highly technical, but categorization is fundamentally a natural human cognitive process. There has been considerable work on categorization outside of computer science in the areas of cognitive science, anthropology, linguistics, and philosophy.[1] Work in these fields over the past decades has lead to a view of categorization known as "prototype theory" [3], in which categories are defined by prototypical members. Prototype theory stands in contrast to the more traditional ("classical") view of categorization in terms of taxonomic hierarchies of well-defined categories.

In this paper we investigate the applicability of such categorization theories, particularly prototype theory, to MDSOC. Section 2 summarizes the theory and section 3 describes the development context in which we consider it. Sections 4 and 5 describe applications of the theory and remaining categorization issues, respectively. Section 6 indicates how we are using prototypes to simplify the design of a software composition system, while Section 8 presents a summary and our conclusions.

2. PROTOTYPE THEORY OF CATEGORIZATION

The prototype theory of categorization is based on the work of many scholars and researchers. To present it briefly, we quote in its entirety a summary by Lakoff (bracketed comments added) [3]:

- Some categories, like *tall man* or *red*, are graded; that is, they have inherent degrees of membership, fuzzy boundaries, and central members whose degree of membership (on a scale of zero to one) is one.
- Other categories, like *bird*, have clear boundaries; but within those boundaries there are graded prototype effects--some category members are better examples of the category than others. [For example, wrt birds: {robin, sparrow} > {owl, duck} > {penguin, ostrich}.]
- Categories are not organized just in terms of simple taxonomic hierarchies. Instead categories "in the middle" of a hierarchy are the most *basic*, relative to a variety of psychological criteria: gestalt perception, the ability to form a mental image, motor interactions, and ease of learning, remembering, and use. Most knowledge is organized at this level. [Categories above and below the basic level may be added but these are more intellectual than cognitive constructs.]
- The basic level depends upon perceived part-whole structure and corresponding knowledge about how the parts function relative to the whole.
- Categories are organized into systems with contrasting elements.
- Human categories are not objectively "in the world", external to human beings. At least some categories are *embodied*. [That is, they have a physiological basis or manifestation.] Color categories, for example, are determined jointly by the external physical world, human biology, the human mind, plus cultural considerations. Basic-level structure depends on human

---

[1] We deliberately omit mathematical category theory, although it, too, may have applications to MDSOC.

perception, imaging capacity, motor capabilities, etc.

- The properties relevant to the description of categories are *interactional properties*, properties characterizable only in terms of the interaction of human beings as part of their environment. Prototypical members of categories are sometimes describable in terms of *clusters* of such interactional properties. These clusters act as gestalts: the cluster as a whole is psychologically simpler than its parts.
- Prototype effects, that is, asymmetries among category members such as goodness-of-example judgments, are superficial phenomena which may have many sources.

A further point is that the members of a category need not share any common characteristic. They may be grouped by "family resemblance", for example, such that each member shares some characteristics with some other members.

In contrast, the classical view of categorization is hierarchically oriented, with membership in a category based on clearly defined necessary and sufficient conditions. As a consequence, there are certain properties that all members of a category have in common, and no member of a category is a better representative of the category than any other.

We believe that it is worthwhile to investigate the prototype theory of categorization for purposes of MDSOC for two main reasons:
- It reflects issues or conditions that we have encountered in trying to organize software concerns
- It acknowledges and accommodates the human factors affecting categorization, which we necessarily bring to MDSOC

We also recognize that there are reasons that the prototype theory might not apply to MDSOC:
- Software is "unnatural" or, more particularly, artificial and abstract
- Hierarchy is important in software, for managing complexity and for other purposes
- Multiple dimensions of classification apply simultaneously (a condition not addressed in the prototype or classical theories)

In the remainder of this paper we consider how, and how well, prototype theory may address MDSOC, then identify limitations or complications of the theory in the context of MDSOC, and finally describe how we are applying the notion of prototypes to simplify the design of a component composition system.

3. CONTEXT OF CONCERN MODELING

We are investigating the use of MDSOC-based compositional techniques to create customized software components, especially systems components. We are using a cache as our main example. In previous work [4] we have described the concerns associated with the design of a general-purpose software cache, the GPS cache [2]. We are now modeling concerns in the code of the GPS cache, specifically its Java implementation. One of our objectives is to be able to start from a pool of relatively small Java units and compose a cache that is more or less equivalent to the GPS cache in terms of the concerns addressed. Another is to be able to generate specialized subsets and variants of the cache, where the specializations are based on selected concerns. We are using Hyper/J [1] as our composition tool.

In terms of the Cosmos concern-space modeling schema [5], our analysis of concerns in the design of the cache dealt entirely with logical concerns (representing concepts, properties, issues), whereas our work with the implementation deals with concerns that are both logical and physical (representing software units). The association of logical concerns to physical concerns, for example, the association of particular properties, functions, behaviors, and states to particular Java classes, methods, and variables, is a major part of concern-space modeling for composing component implementations.

In designing an application to support the specification and generation of customized caches, we face a number of fundamental questions that can be viewed as categorical. For example:
- For what categories of component (caches or otherwise) do we want to support composition?
- How do we categorize a cache as a component?
- What categories of concern are important in characterizing caches, either to distinguish them from other types of components, or to describe useful customizations within the category?
- How do different categories of concern relate? How are concerns in different categories correlated?

As a result of considering these and related questions, we can make some preliminary observations about the applicability of prototype theory to component composition based on MDSOC.

4. Applicability

Virtually all of the general characteristics of the prototype theory of categorization have some application to, or corresponding aspect in, software composition based on MDSOC.

Graded categories: The types of software components such as caches may qualify as graded categories, that is, ones for which there are fuzzy boundaries and degrees of membership. Compared to an ideal notion of a cache, a prospective cache component may be more than a cache (having additional functions and features, even to the extent that "cacheness" becomes a minor part of its character), less than a cache (lacking in some features, for example, expiration times, that are commonly considered integral to a cache), or just different (for example, using different replacement algorithms or access disciplines). Thus it seems that any attempt to define "cache" precisely will be at least somewhat arbitrary.

Additionally, some of the categories of concern that apply to caches and other software components are also graded, especially properties. Examples include performance (fast versus slow), size (small versus large), usability (easy versus hard), maintainability (easy versus hard), and scalability (scalable versus not scalable). As we have noted previously [4], this makes it difficult to categorize software in some dimensions and to capture and analyze relationships among some concerns and associated units.

Definite prototypic categories: Not all of the categories of concern in the cache are graded with fuzzy boundaries. For example, in the functionality dimension, we have been able to divide operations on the cache and related classes into definite categories without ambiguity. For example, we have distinguished operations that affect the state of the cache in terms of the objects it contains (e.g., add, delete, retrieve, and update), operations related to inter-object dependencies, operations related to enabling and disabling the cache, and others. Within these groups, however, it seems that not all operations are created equal. For example, with respect to the core operations, it seems that adding, retrieving, and deleting objects are most central, that updating an object is somewhat less central, and that an operation to purge the cache is less central still (although not necessarily less useful for that). With respect to the cache-enabling operations, the operation to enable or disable the cache is most central, whereas the operation to test whether the cache is enabled or disabled is less central. A similar situation exists with respect to behaviors.

Basic levels of categorization: Our interest in components with the granularity of a cache may imply that this level of organization of software reflects a "basic level" of categorization. A basic level of categorization is one that is most natural and generally useful for organizing concepts, knowledge, and activities. When we think of a cache or comparable component, we think of a module that embodies a useful set of functionality, that exhibits properties and behaviors as a whole, that can be plugged into and used directly in a range of applications, and that is a natural unit of work with respect to software engineering practices such as design, coding, testing, configuration, versioning, release, reuse, and so on.

In traditional object-oriented approaches to component definition, a useful component such as a cache might be defined by inheritance through several levels of more general classes; however, these more general classes are likely to be too general for direct use by applications and may even be abstract. Components below this level may be too limited or specific to be useful in many applications.

The significance of basic levels of categorization in cognition and language may lend support to concern-based compositional approaches to software development. Since these approaches can operate directly at the level of useful components, they may be more natural, and thus perhaps more workable, than approaches based on the classical notion of hierarchical categorization and inheritance.

Part-whole structure: A software component, such as a cache, is a whole that combines a number of parts, such as operations and elements of state. The component as a whole is understood in terms of these parts and the way in which they interact to produce an overall effect. For example, an attempt to retrieve an object that has not been added to a cache should fail, whereas an attempt to retrieve an object that has been added to a cache should succeed (ignoring object expiration and other effects that may cause object to be removed implicitly). Moreover, types of components with similar parts may be distinguished by the way in which those parts interact in their contribution to the whole. For example, both caches and buffers may have operations to add and retrieve objects, but their behaviors over an extended period or sequence of operations may vary due to differences in factors such as replacement algorithms, access protocols, object expiration, and so on.

The importance of part-whole structures in categorization may lend further support for composition as a natural approach to software development. Composition is about assembling parts into wholes and orchestrating their interactions. Component specialization by selective composition is a direct way to obtain a whole comprising just the parts of interest. In contrast, inheritance-based approaches are not about assembling parts but rather about successive refinements or extensions of components. The result may not appear as an integrated whole because its

parts (and their interactions) may not be found or understood locally; rather they may need to be located and comprehended over a sequence of stages.

Other features: Several other features of prototype theory also apply to MDSOC, although space limitations preclude detailed discussion here. The importance of contrast between categories has implications for the selection of concerns within dimensions. Non-objectivity is reflected in subjective and context-dependent concerns. Interactions with components in a software-engineering sense may serve to categorize roles for artifacts in development methods. And use or convention may give rise to a variety of superficial (non-inherent) prototype effects.

5. Complications

Although many aspects of the MDSOC approach to software composition can be described in terms of prototype theory, there remain problems of categorization to be resolved.

Approximate categories: It is natural to try to characterize a system for composing components in terms of the kinds of components that can be composed using it. We find it convenient to say that we are designing a system for composing caches. However, that characterization must be understood as an approximation, for several reasons:
- We have no general test for "cacheness" that we can apply to prospective products (in other words, "cache" is a graded category with fuzzy boundaries)
- The composition technology (Hyper/J) is not cache-specific, and there are limits on the extent to which processes for the use of that technology can restrict the categories of products produced with it.
- The code units from which we compose products, even those extracted from a cache, can be used in other kinds of components (or in components that are over- or under-qualified to be classified as a cache)
- At least part of most user's notions of a cache incorporate aspects related to use (e.g., in locally storing remotely obtained data) that are beyond the control of a composition system

Thus, while "cache" may serve as a useful notional prototype for the kind of component for which we want to support composition, it does not suggest the range of components for which we might actually be able to support composition or the range of applications for which those components might be suitable. Consequently, we face the question of how we can characterize the products of our system without being simplistic or unduly restrictive.

Interaction at multiple levels: Prototype theory states that we tend to categorize entities at a certain level of generality that corresponds to the level at which we most naturally interact with or think about them. Software composition involves working with software components at multiple levels of multiple types, including levels of categorization, granularity, complexity, and completeness. Moreover, levels of one sort may not correlate with levels of another.

One example is that developers must often reason about effects on the whole based on features of the parts (or, conversely, understand parts in terms of their potential effect in the whole). To complicate this, the whole and parts may be at different levels of granularity (as a class may be composed from individual members), the parts may be selected based on different types of concerns representing different dimensions of categorization (some methods selected by function, some for particular behaviors, and some to support specific properties), and parts may be specified at different levels of generality (we may select all behaviors that support correctness, specific behaviors that support robustness, and all behaviors that support generality except certain specific behaviors).

Thus, counter to our natural ability to reason and work best at a particular level on a particular scale, developers must reason and work at and across different levels on different scales. A challenge for MDSOC is helping developers to traverse and span such categorical gaps. One of the goals of Cosmos [5], for example, is to capture different levels of generality within different dimensions of concern and to allow these to be meaningfully interrelated.

Multiple dimensions of categorization: Both the prototype and classical theories of categorization address classification within a single dimension or scheme (although an individual classification may depend on a combination of multiple factors). Multidimensional separation of concerns, of course, addresses multiple dimensions of classification individually. The prototype and classical theories provide no direct guidance for accommodating multiple dimensions in a single model or analysis. When a component is described in terms of multiple dimensions it is not clear how these should be organized and presented. For example, for the cache, we have independent dimensions of concern (top-level concerns in the Cosmos sense) relating to classes, functionality, behavior, state, and properties such as performance, correctness, safety, and configurability. None of these is *a priori* a root dimension for all (or any) purposes, and conceivably any of them may be of primary interest to a developer in a particular context.

A further complication that arises especially in the case of software composition is that the association of categories in different dimensions is not fixed. In other words, the bindings between categories are potentially variable. For example, in describing an existing cache, we can categorize

elements definitely with respect to class, operations, state, behavior, and specific properties, and the correlations between these are established. In contrast, when composing a new cache, we may have the option to vary the classes involved, the methods and variables belonging to those classes, the behaviors associated to the methods and classes, the properties resulting for the classes and methods, and so on. The correlations between dimensions vary, and it is not generally possible to systematically describe a member of one category of one dimension in terms of categories in other dimensions.

Although theories of categorization do not contribute much directly to the management of multiple dimensions of concern, the idea of multiple dimensions of concern may contribute to the prototype theory of categorization. In particular, the applicability of multiple dimensions of concern may be a source of asymmetric or graded effects in categories based on prototypes. For example, our sense of the goodness-of-fit of a component in a category such as "cache" may be affected by dimensions of concern other than those in terms of which a category is defined. For example, a general-purpose cache, with a rich variety of features, may clearly qualify as a cache, but it may be seen as a more peripheral sort of cache than one with a more typical range of features. (Does the prototypical cache include logging or inter-object dependency management? Probably not, but a cache that offers these is still a cache.)

Decoupling of interaction: The prototype theory of categorization places a strong emphasis on interactional properties as the basis for classification. Those are properties related to human interactions with a categorized entity. With software components, we can also consider programmatic interactions, that is, interactions within a program. However, such interactions may not always be a useful basis for categorization. For example, many data structures offer substantially comparable interfaces that allow for data to be written, read, updated, deleted, etc. The interfaces for a cache, a buffer, and a queue might be virtually identical in this respect (with that for a stack not very different, if at all). Our interactions with these data structures are all basically the same. This general sort of interaction may distinguish data structures from other kinds of components, but it fails to make distinctions between types of data structures that are different in other significant ways (for example, access disciplines and effects of operations on structure state).

Related to interactions are the uses to which components (or other elements) may be put in a program. For example, any data structure or variable may be used by an application as a cache, depending on what needs to be cached or what variables or structures are convenient for that purpose. A single integer value that is difficult to obtain or compute

might be cached in a local variable. However, when we consider a system to support the composition of customized caches, we do not have in mind a system that will allow the composition of arbitrary variables or data structures. We intend to support the composition of components that are relatively restricted in many dimensions.

Conversely, the purposes for which a component may be used by a program are not in general restricted. A component that is intended to serve as a cache may provide a useful data structure in other roles. For example, the GPS cache may be used as a store for local data by an application that can benefit from features such as logging, collection of usage statistics, or dependency management.

Overall, interactions or uses seem to provide a not very useful basis for categorization of software components for many purposes. The multidimensional nature of software largely accounts for this, as categorizations in terms of interaction or use often do not correlate well with categorizations in other dimensions that may be of more importance. This also relates to our need to work with (and to think about) software at different levels of classification.

6. Role in the Design of a System to Support Multidimensional Composition

In considering the design of a system to support the specification and composition of customized software components, we initially faced a number of questions related to the scope of the system. For example, what was the range of component for which we wanted to support composition, and what would it mean to say we support the composition of a particular kind of component? Answers to such questions have significant implications for the prospective system in the areas of concern-space modeling, related information modeling, and workflow.

To simplify the design space, we decided to focus initially on caches. However, that still left many design decisions open, as the notion of a cache is not well defined, what support we would provide for composition of caches would not necessarily be restricted to caches, and there are many ways of approaching the customization of a cache (or other component) by selective composition.

To further constrain the design space, we have adopted an approach based on strong, flexible prototyping. More specifically, we have identified a particular set of classes that we take as prototypical of a software cache. We call this the "application model" for the cache, as it represents an idealized model of the caches produced by the system, as seen by an application. For the application model classes, we have introduced a set (initially fixed) of methods, states

elements, behaviors, and associated properties that can be variously composed into the model classes.

The approach is strong in that classes that do not conform to the model cannot be produced, and their methods, behaviors, and so on are limited to combinations of those provided. The approach is flexible in that many combinations of the provided elements are possible (including some that may push the limits of what we may ordinarily think of as a cache).

The prototypical classes of the application model provide a framework for organizing information related to concerns in other dimensions (e.g., operations, behaviors, properties, etc.). Classes may be considered as a prototype dimension in organizing our concern space. They are not a dominant decomposition, but they serve as a featured orientation for organizing the presentation of information and structuring prototypical workflows. Additionally, they will not be the only dimension of organization within the system, and paths based on other featured orientations will be provided.

Once we understand better how to categorize and present concerns for a particular type of component, we expect to open up the system to various types of extension. These include the addition of new implementation classes for use in composing the original application model, the addition of new elements to the original application model, and the addition of new classes to those for which composition is supported. The latter we expect will involve the introduction of new prototypes for new categories of application-level components. In keeping with the principle that a system of categories should represent contrasting elements, we need to understand more about what truly distinguishes one type of component from another for the purposes of developers and to be sure that the appropriate concerns and relationships are reflected clearly in our concern-space models.

7. Summary and Conclusions

Multidimensional separation of concerns and its applications fundamentally involve problems of categorization. The prototype theory of categorization contains many elements that are descriptive of phenomena related to MDSOC. These elements include graded categories, prototypical effects, basic levels, part-whole structure, and others. The prototype theory seems to fit MDSOC better than the classical theory of hierarchical categories. Moreover, elements of prototype theory imply that composition should be a more natural technique than inheritance.

Although prototype theory characterizes MDSOC well in several respects, there are issues related to categorization that are not yet resolved. These include the prevalence of approximate categories (where approximation has several sources), interaction at multiple levels, applicability of multiple dimensions, and the decoupling of interaction from other meaningful bases for categorization.

Many aspects in the design of a system to support the composition of software components, such as caches, are unconstrained. The adoption of various notions of prototype are helpful in restricting the design space. We have used a prototypical application model of a cache to organize the system information model and workflow. With this we expect to gain experience with the use of prototypes and ultimately to learn how to systematically expand the scope, flexibility, and utility of component composition systems.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] IBM. Hyper/J. http://www.research.ibm.com/ hyperspace/HyperJ/HyperJ.htm

[2] 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.

[3] Lakoff, George. Women, Fire, and Dangerous Things--What Categories Reveal about the Mind. The University of Chicago Press, Chicago, 614 pp., 1987.

[4] 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.

[5] 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.

[6] 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.