Using Objects for Systems Analysis

By drawing on distinctive models of representation we are able to analyze the role of objects in domain modeling. This method leads to a clearer understanding of various aspects of the OO approach.

Jeffrey Parsons and Yair Wand

The object-oriented approach originated as a programming and software design discipline offering advantages such as reusability, extendibility and portability. Recently, object thinking has been applied extensively to systems analysis [1, 3, 4, 8, 10]. It has even been claimed that “the real payoff (of the object-oriented approach) comes from addressing front-end conceptual issues, rather than back-end implementation issues” [10]. However, in this evolution, programming concepts have crept into some analysis methods. This is demonstrated by statements such as: “Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation” [10] or “An object is any thing, real or abstract, about which we store data and those methods that manipulate the data” [8].

We believe that improper adaptation of programming concepts has impeded the successful application of object-oriented systems analysis. Systems analysis involves modeling a domain. It is therefore fundamentally different from software design, which is implementation-oriented. According to Kilov and Ross [7], “the library of generic programming concepts is at too low a level for analysis.”

Of course, object-oriented analysis provides input for designing software. Accordingly, software design objectives such as modularity and reuse could be seen as relevant during analysis. We argue, however, that this perspective may interfere with understanding the domain by drawing attention to implementation considerations. This can be detrimental to system success, since “focusing on implementation issues too early . . . often leads to an inferior product” [10].
We argue the key to applying object concepts in systems analysis successfully is viewing objects as representation, rather than implementation, constructs. While many object concepts may be common to representation and implementation [6], applying them with an implementation focus during analysis may have undesirable consequences.

What, then, is a suitable basis for using objects in systems analysis? We propose the answer can be found in models of representation, specifically ontology and cognition. Such models may have important implications and practical consequences to object-oriented systems analysis.

**Representation Models of Objects**

We start with a basic assumption: An information system represents knowledge about things in some domain.

Based on this, we apply two views of representation to interpret object concepts. One is that an information system represents things in the application domain, such as employees, products, and customers. Therefore, a theory of the nature of things, an ontology, can guide us in using objects in systems analysis. The second view is that an information system represents human knowledge about a domain. Hence, a theory of the structure and organization of knowledge about things, that is, a theory of concepts, can also serve as our guide. Ontology and concept theory are not the only possible foundations. For example, another foundation can be sought in linguistics. However, they are appropriate in light of the basic assumption we make here.

**The Ontological Approach**

Ontology has a tradition dating back to ancient Greek philosophers, however, no universally accepted ontology has emerged. Hence, the first step in the analysis is choosing an ontological model. We base our analysis of object concepts on Mario Bunge’s ontological work [2]. This work has already been used to formalize object concepts [12]. Here we use it to interpret these concepts for systems analysis, beginning first by introducing its main constructs and assumptions. Hereafter, “ontology” refers to Bunge’s ontological model.

**Things and properties:** The basic assumption is that the world is made of things that possess properties, but a thing is not just a bunch of properties. A property can be intrinsic to a thing, or mutual to several things. For example, height of a person is intrinsic, while being employed is mutual to a person and an employer.

No two things have identical properties, thus every thing is unique. Properties can be restricted by laws relating to one or several properties, for example, a limit on a salary, or on a salary at a certain rank. A special form of a law is precedence of properties, wherein the existence of one property implies the existence of another, preceding, property. For example, the ability to walk is preceded by the ability to move.

**State of a thing:** Things possess properties independently of what we know. In contrast, attributes are characteristics assigned to things by humans. Every property can, in principle, be represented by attributes. An attribute may, or may not, represent a property. The attributes used to describe a thing are called its “functional schema.” Each of these attributes is a state variable and their values at a certain time comprise the state of the thing. The possible states are constrained by the laws of the thing.

**Dynamics of a thing:** Ontology assumes that every thing must change. A principle called “nominal invariance” states that a thing can change properties and still be the same thing. Since the world is made of things, every change involves a change of things. Every change of a thing is manifested as a change of its state. Hence, dynamics is described in terms of state changes, termed events. Events are governed by transformation laws that define the allowed changes of state.

**Interaction:** Based on dynamics, interaction is defined as the ability of one thing to affect the state evolution of another. An interaction is a mutual property of the interacting things.

**Composition of things:** Things can be combined to form a composite thing. A property of a composite that is not possessed by any of its components is called an “emergent property.” Ontology requires that every composite thing has emergent (holistic) properties.

**Classification of things:** A kind is the set of things possessing a given set of properties. A natural kind is defined by a set of properties and the laws connecting them.

In the ontology-based view of objects, objects are representations of things [12]. The attributes of the
thing are represented as attributes of the object and
the allowed state transformations as the operations of
the object. Interactions are represented as communi-
cations between objects.

The Classification Theory Approach
Classification theory has its roots in psychological
research on how humans structure knowledge about
things by constructing concepts that describe cate-
gories of similar things. There are several theories of
classification [11], differing in the view of what a
class is. We use a view of classification which has
previously been applied to object concepts [9] to
interpret object characteristics for systems analysis.
The main elements of this view are instances, prop-
erties, and concepts.

Instance: An instance denotes knowledge of the
(believed) existence of a thing in the world.

Property: Properties constitute knowledge about
things beyond the fact they exist. An instance pos-
sesses properties of three types [9]: Structural—
which belong to the thing itself; relational—which
relate the thing to other things; and behavioral—
which determine possible changes to the values of
structural and relational properties.

Concept: A concept is defined by the common
properties possessed by a set of instances. A concept
is intensional; in other words, defined by properties,
not by the instances that possess them. Specialized
concepts can be derived by adding properties to the
definitions of existing concepts.

A concept is more than an arbitrary collection of
properties. Concepts are meaningful only if they are
useful to humans for efficient storage and retrieval of
knowledge about things. Since usefulness depends
on context and purpose, concepts are neither prede-
termined nor fixed, but may vary among individuals
or over time. Usefulness of concepts is determined
by the principles of cognitive economy and inference
[11]. Cognitive economy means reducing the mental
work required to keep track of knowledge about
instances. Inference refers to the ability to infer char-
acteristics of instances from concepts.

Composite instance: Some instances (wholes) are
composed of simpler instances (parts). A composite
instance possesses emergent properties.

The view of objects based on classification theory is
as direct representations of cognitive instances, having
structural relational, and behavioral properties [9].

The two representation approaches we chose differ
in both their foundation and formalization. Neverthe-
less, both provide a view of objects in modeling with
no reference to implementation issues. We now inter-
pret object-oriented concepts using these approaches.

Analysis of Object Concepts
The most common description of “object” is as an
identifiable thing that remembers its own state [1,
6], and that can respond to requests for operations
with respect to this state [5]. Objects have two types
of properties: state (instance) variables, and opera-
tions (methods). Objects interact by requesting ser-
sices or information from each other. In response to
a request, an object may invoke an operation that
might change its state.

For our analysis, we have compiled a list of com-
monly cited concepts from the object literature (such
as [1, 3, 4, 5, 8, 10]) and categorized them into four
groups: objects as distinct units, classification, asso-
ciation and interaction, and object behavior.

Objects as Distinct Units
An object has a unique unchangeable identity [1, 10],
“which provides a means to denote or refer to the
object independent of its state or behavior” [5]. This
view emphasizes identity as a reference to the object.
The representation models provide a different inter-
pretation: Identity stands for existence and unique-
ness. Ontology states that a thing is more than just
its properties and that no two things have exactly the
same properties; classification theory distinguishes
the existence of an instance from its properties.

Encapsulation, or data abstraction, means that data,
along with all operations on that data, should be pack-
aged in a self-contained unit. Encapsulation is also a
behavior abstraction in that internal, implementation
details, are hidden inside the object [5, 10], and has
been referred to as “information hiding” [1].

Both representation models hold that structure
and behavior aspects of things (or instances) are
inseparable. In ontology, there are no “thingless
events” or “changeless things” [2]. In classification
theory, an instance possesses structural, relational,
and behavioral properties where the last is defined in
terms of the previous two.

Persistence means the existence of an object “trans-
scends time. . . . and/or space” [1], and has been
described as a “temporal” property of objects [6].
This concept is usually related to the independence
of data or of processes running in an operating sys-
tem once they have been created.

In ontology, persistence can be related to the prin-
ciple of nominal invariance, which means that a
thing can acquire and lose properties and still be
considered the same thing.

An object-oriented environment is homogeneous if
everything, including variables and messages, is an
object. In Smalltalk, for example, both instance vari-
ables and classes are objects. Homogeneity has been
justified on the grounds of providing "a more uniform implementation and greater functionality for solving complex problems" [10]. In practice, complete homogeneity is impossible since, if instance variables (or methods) are objects, they will have instance variables and methods, and so on.

This implementation perspective has influenced some object-oriented analysis methods. For example, Rumbaugh et al. and Booch view classes as objects [1, 10]. Similarly, Embley et al. suggest that relationships be viewed as objects [4].

In contrast, neither the ontological nor the classification view support complete homogeneity. Things (or instances), properties, and classes are viewed as distinct constructs. Therefore, neither properties nor classes are considered objects. In particular, classes are abstractions, reflecting similarity of things, but are not things.

In a homogeneous object-oriented programming environment, every program component is an object. However, from a representation perspective, not every component represents a thing in the application domain. For example, a class is implemented as an object but represents an abstraction rather than a thing. Hence, software objects can be categorized into two types: one represents things in the domain, the other consists of implementation artifacts. This has been recognized by Booch who claims: "Real-world objects are not the only kinds of objects that are of interest to us in software design" [1].

Classification of Objects

Classification/Instantiation is widely assumed in object-oriented environments. A class specifies the common structure (attributes, instance variables) and behavior (operations, methods) of a set of objects, allowing just one implementation of the behavior of these objects.

It follows there are two views of a class. The extensional view is "a set of objects that share a common structure and a common behaviour" [1]. The intensional view of a class is a template for objects, and is sometimes termed "type" [5, 6].

In ontology a natural kind is defined by a set of properties and the laws constraining them. This is a combined intensional-extensional view. Since things in a natural kind are similar in some respects, a common representation can be used for all instances of the same kind. A class of objects can be viewed as a representation of a natural kind. In the classification theory model, an object class is defined by intension as a representation of a concept, that is, as a set of properties (structure and behavior) shared by some instances.

In most object-oriented environments, objects are grouped into classes that may be arranged taxonomically. A subclass inherits the properties (instance variables and methods) of its parent class(es). This supports reuse of code. For example, Martin and Odell claim that "class inheritance makes the data structures and operations of a class physically available for reuse by its subclasses" and tie it to "code sharing rather than code redefinition" [8]. Inheritance may be single (one immediate superclass) or multiple (several immediate parent classes) [5]. Not all object-oriented environments support multiple inheritance.

In both representation models, classes are specializations of others if they have additional properties. Both models support multiple inheritance. Since classes are defined by intension (in terms of properties), different classes must include different sets of properties. In addition, to support cognitive economy, a new class should provide information beyond what can be inferred from existing classes. Consequently, a subclass should have properties possessed by none of its parents.

Association and Interaction of Objects

Composite objects can be described as combinations or aggregations of simpler objects [6, 8, 10]. Compo-
Composition reduces complexity “by treating many things as one” [8]. In object-oriented databases, composition is used to assure integrity, and storage and retrieval efficiency. In the context of analysis, composition is claimed to reflect the way humans think about a problem domain [3].

The assumption that things can associate into other, more complex things, is fundamental in both ontology and classification theory. An ontological premise states that a composite must possess emergent properties. Object-oriented environments typically do not require that composites possess emergent properties. However, the ODP-2 Reference Model supports a form of emergent behavior, as the behavior of a composite object is the corresponding composition of the behaviour of the component objects” [6].

Some form of communication or interaction between objects appears in all object environments and models. Communication entails exchange of requests between objects, where a request specifies what is to be done, but not how. An object is “sovereign” to determine whether to take an action in response to a request and how to perform it. This supports the ideas of encapsulation and objects as independent communicating entities. A common communication mechanism is message passing [5], which can be viewed as a special case of generalized communication.
object interaction. The latter views objects as “participating in” behavior, and does not distinguish between a sender and a recipient [5, 7, 8].

In the ontological model, communication is established via interaction. Interaction implies the existence of a mutual property (a property common to two or more things). Since laws are also properties of things, they can be mutual as well [12]. In the classification theory model, communication can be modeled by behavioral properties which restrict state changes of two or more instances. Both models are consistent with the generalized object interaction model. Neither explicitly specifies message passing as a communication mechanism.

Several models of objects incorporate relationships [4, 5], sometimes termed “instance connections” [3], or links and associations [10]. In Rumbaugh et al., relationships are not formally defined, but can be inferred from examples (for example, works-for [10]). Kilov and Ross define generic types of relationship as invariants, which are conditions that are true before and after operations [7].

There is no explicit relationship construct in either of the representation models. Instead, both recognize properties that are joint to several things. In ontology, such properties are termed mutual or relational and reflect interaction. Similarly, in classification theory, a property of an instance can be relational, that is, link an instance to other instances.

**Issues Related to Object Behavior**

**Polymorphism** implies the same request may invoke different implementations of an operation [5]. Different receivers may respond differently to the same request. Polymorphism is viewed as a useful feature in software construction as it allows specialization of a generic operation from a superclass to its subclasses. However, this idea is somewhat paradoxical. Martin and Odell claim: “The arbitrary overriding and cancelling of inherited features is a questionable practice. Logically it is incorrect because, by definition, all features of a type apply to its subtypes” [8].

The ontological view suggests a way out of this contradiction. Ontology recognizes the notion of precedence of properties [2]. Property $P_1$ precedes property $P_2$ if and only if every thing that possesses $P_2$ possesses $P_1$. An inherited property can be viewed as preceding its specialized forms and polymorphism as a special case of property precedence. For example, fish swim and land mammals walk. Swimming and walking are both preceded by the property “being able to move.” That would mean that fish and land mammals are both animals and animals can move. Using the message metaphor, fish and land mammals respond differently to the message “move.” The notion of precedence applies to static properties (attributes) as well.

**Dynamic (late) binding** means the action taken in response to a message is determined only at run time, when the specific variables are instantiated (bound) [1]. Dynamic binding enables polymorphism in environments where variable types can change at run time. Dynamic binding is a pure software concept which has no meaning in the representation models.

**Concurrency** means that several objects can be active at the same time. In implementation, an active object is a running process (an actor).

The representation models do not address concurrency explicitly, as neither relates to the notion of an active process. However, the idea of concurrency does not contradict the models since things (existing or perceived) can be active at the same time.

We have suggested that the use of object-oriented concepts in systems analysis should be driven by representation, rather than implementation, objectives. Therefore, we interpreted object concepts in ontological and cognitive terms. This perspective can influence the use of objects in systems analysis. The table featured here compares the implementation and representation views and suggests implications for analysis.

**Some Implications for Systems Analysis**

**Encapsulation.** From a representation perspective, encapsulation means that structure and behavior are inseparable. This suggests that when using objects to model a domain, static and dynamic properties should be considered together. This view is not always followed in object-oriented analysis. For example, Embley et al. [4] distinguish between an object relationship model and an object behavior model, and Rumbaugh et. al. [10] between object modeling and dynamic modeling.

**Homogeneity.** The representation models view things, properties, and classes as different constructs. This has implications for object-oriented analysis. First, classes should not be treated as objects. Hence, the notion of class properties is meaningless. What can be meaningful are properties of a composite object consisting of the instances of the class. For example, the employees of a certain company belong each to the class “employee.” However, the set of instances of this class may be viewed as the aggregate object “employee body of the company.” Such an aggregate object has properties, but is not the same as the employee class which is just a concept. Second, in some object-oriented environments, a new object is created by sending a message to its class object. In
the representation models, this metaphor has no meaning in describing real-world situations. For example, in an order processing system, a new order is created by a sales person or by a customer, but definitely not by an order class.

**Specialization/Inheritance.** In the representation models, classes are defined by a set of properties (both static and dynamic). Hence, multiple inheritance is strongly supported. In addition, the principles of cognitive economy and inference imply that a subclass should have properties in addition to all those of its superclasses. This means, among other things, that subclasses cannot be defined by value of a property ("all employees who live in New York") or by intersection ("employees who are customers") without adding any new properties. These additional properties should be identified in the analysis. By contrast, some object-oriented analysis approaches do not require subclasses to have properties in addition to their parents (examples can be found in [3, 10]).

**Composition.** In the representation models, composites must have emergent properties. Kilov and Ross describe composition with emergent properties as an "elementary association" [7]. However, most of the object-oriented analysis literature does not recognize the importance of emergent properties. We propose that an analyst examine candidate composite objects for their emergent properties. If none can be found, the composite may not be a proper object. For example, in a payroll system, an employee and his/her desk might not reasonably be considered a composite object.

**Communication.** The representation models do not specify message passing as a communication mechanism. Consequently, we suggest caution in modeling a domain in terms of messages passed between objects. For example, consider an order processing system. It would be wrong to describe placing an order as "a customer sends a message to the inventory item." In reality, the order is placed with a sales person. This is what the model should show.

**Conclusion**

The object-oriented approach emerged as an implementation paradigm, motivated by the objective of building better software more efficiently. However, the use of objects in systems analysis should be driven by the objective of effectively representing a domain. The essence of our proposal is that, for analysis, representation-based foundations are more suitable than implementation-based approaches. We used ontological and cognitive theories of representation to interpret object-oriented concepts and derive guidelines for using them in systems analysis.

Alternative representational paradigms may be adopted. These include other ontological and cognitive models (such as prototype-based classification), as well as models form other fields such as linguistics. These alternatives may generate different guidelines for object-oriented analysis. The ultimate test is to evaluate these guidelines in practice. To that end, we are developing an analysis methodology based on the representation models we've presented.

**References**


This research was supported by grants from the Natural Sciences and Engineering Council of Canada and the Social Sciences and Humanities Research Council of Canada to Jeffrey Parsons and to Yair Wand.

**Jeffrey Parsons** (jeffreyp@plato.ucs.mun.ca) is an associate professor of Information Systems in the Faculty of Business Administration, Memorial University of Newfoundland, St John’s, Canada.

**Yair Wand** (wand@ubc.ca) is a professor of information systems in the Faculty of Commerce and Business Administration, The University of British Columbia, Vancouver, Canada.