

B. Henderson-Sellers D. Firesmith I. M. Graham

The Benefits of Common Object Modeling Notation

We introduced the third generation, full lifecycle methodology OPEN (Object-oriented Process, Environment and Notation) in our last column[1] where we focused on the most important element of OPEN: its process.[2] Another important component of OPEN and other OO methods is their underlying modeling language, which includes both 1) a metamodel defining the syntax and semantics of the underlying OO concepts and 2) a (typically graphical) notation for expressing these concepts when modeling. The two industry standard modeling languages currently being promulgated are the OPEN Modeling Language (OML) and Rational's Unified Modeling Language (UML). OML consists of a metamodel (derived from the COMMA metamodel[3]) and a notation known as COMN: Common Object Modeling Notation (shown in Fig. 1). In this column, we will outline the characteristics of COMN, which is the preferred notation for most users of the OPEN method. While some may choose the UML notation instead, there are a number of aspects of the OPEN approach in which some things that can be expressed readily in COMN cannot be expressed or will not be properly expressible within the semantics of UML (e.g., responsibilities, rulesets, exceptions, cluster encapsulation). Furthermore, as we described in a previous article,[1] some UML practices (bidirectional associations) violate OPEN principles; for example, that of supporting true object-orientation with a responsibility-driven flavor. Three typical examples of OPEN's emphasis on pure object-orientation are:

1. OML emphasizes objects, types, classes, and their responsibilities rather than the early identification of properties, as do some notations strongly based on entity relationship attribute (ERA) relational database modeling techniques (e.g., OMT, UML, Shlaer/Mellor, Coad, Fusion).

2. OML emphasizes unidirectional relationships over bidirectional relationships because unidirectional relationships are the default in the object paradigm (i.e., objects use internal properties to reference other objects). When needed, bidirectional relationships are derived from two unidirectional relationships that are semistrong inverses of one another and require all of the additional operations to ensure referential integrity.[4]

3. COMN draws aggregation arcs from the aggregate to the part (because that is how aggregate objects are defined and reference their parts in the object-oriented world) rather than from the part to the aggregate (which is how relational database tables are joined via foreign keys). The full documentation is

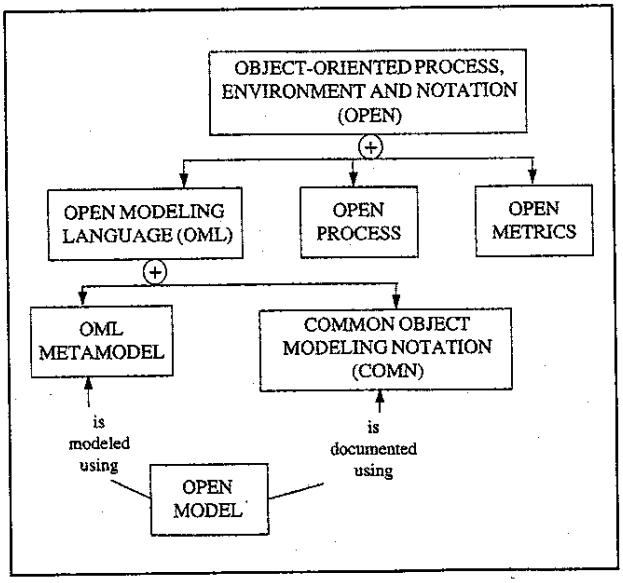


Figure 1. Partial metamodel for OPEN (after Firesmith *et al.*).

Brian Henderson-Sellers is Professor of Computer Science (Object Technology), School of Computer Science & Software Engineering, Swinburne University of Technology, Hawthorn, Victoria, Australia. He may be contacted at brian@csse.swin.edu.au.

Donald G. Firesmith has recently joined Knowledge Systems Corporation as a senior member of the technical staff. He may be contacted at dfiresmith@kscary.com or 73664.3513@compuserve.com.

Ian Graham is Chairman of Graham Associates. He may be reached at 101710.3061@compuserve.com

available in Firesmith *et al.*;[5] so here, it will be sufficient to describe the core elements, or "COMN Light" to give you the flavor of the notation and the thinking behind it.

USABILITY

Notation is the way of communicating between software developers, domain experts, users, customers, managers, and quality assurance personnel. It should therefore be designed with usability and human-computer interaction (HCI) issues in mind.[6] A poor notation can still be learned but is likely to take more time and be a poorer communication vehicle. Most well-known OO notations, for example, are sometimes counter-intuitive or contain arbitrary elements which have to be learned by rote—not a good HCI practice.

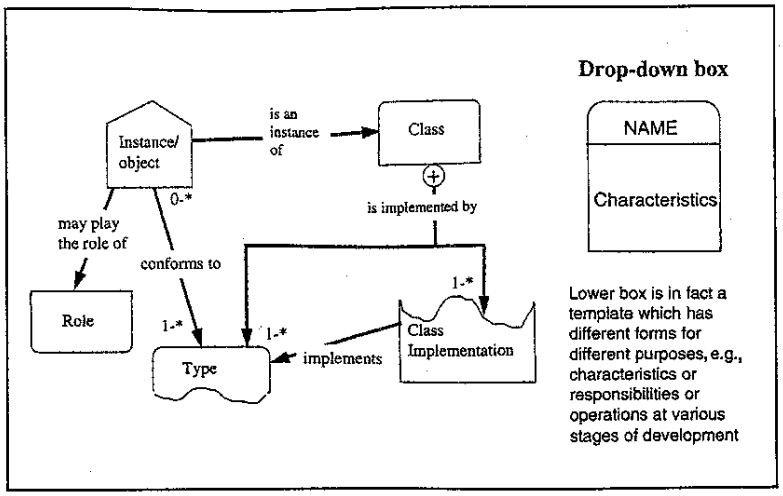


Figure 2. Notation for Object, Class, Type, Role, and Implementation. The Class icon is "torn apart" into the Type (external/interface) and the Class Implementation (internals). All icons can have a drop down box in which information pertinent to the lifecycle stage is displayed.

Intuitiveness

COMN provides support for both the novice and the sophisticate. For many of us, once we have learned a notation we find no barriers—we can use the notation easily and fluently. It is like learning a natural language. Some natural languages are harder to learn than others. It is generally appreciated that Chinese and English (for non-native speakers) can present almost insurmountable problems. For a francophone, on the other hand, learning Italian is relatively easy. Even becoming fluent with the basic alphabet (choose from Roman, Japanese, Cyrillic, Arabic, Hebrew, and many others) can be a challenge for adults with no previous exposure. So, an interface, here the alphabet, that is unfamiliar or does not include intuitive symbols makes the syntax and semantics hard to learn. So it is with an OO modeling notation. The OPEN preferred notation, COMN, has been designed with intuition, usability, and learnability in mind. Granted, we can't find internationally recognizable symbols amenable to all types of novices in every country; however, if we assume we are trying to pictographically describe the main, commonly understood elements of object technology such as encapsulation, interfaces, blackbox and whitebox inheritance, and a discrimination between objects, classes, and types, then designing a broadly acceptable notation becomes possible. Because OML is intended to communicate object-oriented models to humans including non-software professionals, it must be unambiguous, consistent, and comply with our best understanding of iconic design principles. Because object-oriented modeling will continue to be new to most modelers for the next few years, it is critical that the notation be intuitive and imply what it means. In other words, it should not give incorrect cognitive/visual signals. While practical, COMN avoids representing concepts by arbitrary icons and symbols that must be remembered by rote. For example, COMN annotates interface (whitebox) inheritance and implementation (blackbox) inheritance arcs with a white box and black box respectively, and COMN does not use arbitrary symbols (e.g., $ to represent class versus instance-level characteristics). To the extent practical, COMN implies what is intended. For example, COMN draws the aggregation arc from the aggregate to its parts and uses a plus sign to represent that "the whole is (more than) the sum of its parts." Similarly, COMN uses arrowheads on all arcs to represent the direction of dependency and visibility.

Semiotic underpinnings

Semiotics is the study of signs and symbols. Those semiotic ideas are fully integrated into COMN, as are more recent studies in interface design and notational design. COMN has no hereditary biases from an earlier, data-modeling history. COMN has been designed from the bottom up, with intuition and usability in mind, by a small team of methodologists who have worked on these issues during the last decade. It has a responsibility, not data, focus.

State of the art

Based on over a decade of experience, we have learned a great deal about how to make a notation easy to understand, learn, and use. More than 90% of those who will be doing OO modeling in the future have yet to learn any OO modeling technique, and they will not be primarily software professionals who are used to arcane graphical jargons. Therefore, it is critical that methodologists be willing to abandon their obsolete, nonintuitive notations which largely had to be learned by rote and replace them with the currently best available notation based on established HCI principles. Due to the current emphasis on method convergence, now may represent the industry's last, best chance for significantly improving the notation before it is forever "chiseled in granite." Thus, the members of the OPEN Consortium have largely taken a revolutionary rather than evolutionary approach to notation in which no previous, traditional notation has dominated COMN.

Ease of drawing

COMN is easy to draw by hand and easy for upper CASE tool vendors to implement. Indeed, regardless of upper CASE tool availability, most initial modeling will continue to be done on whiteboards. To the extent practical, COMN has avoided mandating conventions (e.g., italics, boldface, color) that are hard to do by hand, while allowing individual CASE tool vendors to create competitive advantage by using such conventions (e.g., drawing exception objects and classes in red). COMN also uses the same notation when drawn by hand and by CASE tool.

Simplicity

COMN Light concentrates on documenting the key modeling constructs. It relegates language-specific concepts to extensions of the core notation. Too many details can clutter up a diagram, making it hard to understand, and should best be relegated to either drop-down boxes or pop-up screens that can present a relatively unlimited amount of information. The primary purpose of a diagram is not to provide enough information to generate code but rather to communicate with human beings. At the same time, a CASE tool should store all information in such a way as to permit the forward engineering of code from models and the reverse engineering of models from code. For example, COMN light does not attempt to display the visibility (e.g., public, protected, private) of C++ characteristics (e.g., data members, member functions).

Scalability

COMN is usable on smaller, informal projects and larger, more complex projects. COMN currently comes in two forms: COMN Light (a subset for beginners and small, simple applications—as discussed here) and the complete Standard COMN for experienced users.[5] Large projects are supported by OML's inclusion of powerful clustering constructs and COMN's inclusion of diagrams (e.g., Configuration Diagrams, Layer Diagrams, Deployment Diagrams) that allow the developer to attack large applications. The OPEN Consortium intends to eventually augment Standard COMN with extensions for particular situations and for use in specific subdomains to meet the needs of advanced users on complex projects.

The goals of ease of learning and simplicity far outweigh any advantages of using a less effective, yet more traditional notation.

Consistency with previous notation

COMN does not include new icons just to be new, but rather reuses traditional icons wherever practical and where they introduce no possibility of ambiguity or misinterpretation. COMN only introduces new notations when traditional notations violate other goals of the OML, or if a sufficiently better notation exists. Since most of the industry has yet to transition to object-orientation, we feel that the goals of ease of learning and simplicity far outweigh any advantages of using a less effective, yet more traditional notation, perhaps one that may reflect someone's vested interests.

COMN LIGHT

Here we describe only the basic elements needed and, indeed, those which will be found in over 80% of all applications. In a nutshell, we need to have symbols for:

- instance versus class versus type versus role versus implementation. All icons have optional drop down boxes for information relevant to the particular phase of the lifecycle. Drop-down boxes may contain information on characteristics, responsibilities, requirements, or stereotypes, for instance. These are all types of traits.
- basic relationships of association, aggregation, containment, and inheritance (specialization, specification and implementation inheritance—these must all be clearly differentiated)
- a state-transition model (dynamics of individual objects and classes)
- an interaction model (dynamics of interactions)
- a use case model (or an extension thereof)

Different models (semantic, interaction, state, or scenario) provide different views of a single overall model and are thus tightly interconnected. Figure 2 depicts the basic icons for class and object. Both class and object are similar; however, a problem domain object is "more real" than a class, so the icon is represented by a sharper icon, whereas the class icon is smooth. We note that in coding, the reverse is probably true—a runtime object is more ethereal than a compile time class. However, we believe that since OT is really about providing solutions to business problems by the use of modeling, the business view should take precedence. The class icon itself is also unmistakable and cannot be confused with rectangles as used in structured hierarchy charts, for example. In MIS systems, we usually use class icons since we generally have one concept (e.g., bank account) with very many instantiations. Thus we only use object icons for specific message-passing sequences (on collaboration diagrams). On the other hand, in real-time systems it is usually object icons that are most prevalent since there is frequently only a single occurrence of a class and the control features dominate. Another sign used is that of a dotted line to indicate a more ethereal notion. Abstract/deferred classes are more ethereal than classes, so they get a dotted outline. (An alternative is to label it as an abstract stereotype.) The

icon for role in COMN is that for a class but inverted and is reminiscent of the Greek tragedy role player's mask. The other interesting item here is how to design a notation that will "last" throughout the lifecycle. The answer we have come up with is "drop-down boxes." These are attached below the icons (for all icons) and can contain information relevant to the particular phase of the lifecycle. They can document traits of various kinds: descriptions, responsibilities, stereotypes, characteristics (e.g., operations, properties, etc). Although drop-down boxes are primarily designed as a way of dynamically providing flexibility when using an upper CASE tool, they can and have been drawn statically on whiteboards and paper. Figure 2 also shows how these concepts are related by the core metamodel. Here we introduce for the first time into an OO notation the well-understood notion that a class is composed of an interface plus an implementation. Graphically we "tear apart" the class icon to get the two complementary icons: the type and the class implementation. An object is then an instance of a class that also conforms to a type and may play a role. The diagram is an example of a semantic net (a.k.a. class diagram) that uses COMN to describe its own metamodel. It can either be read as an example of COMN notation or as a description of a metamodel (actually it is both!).

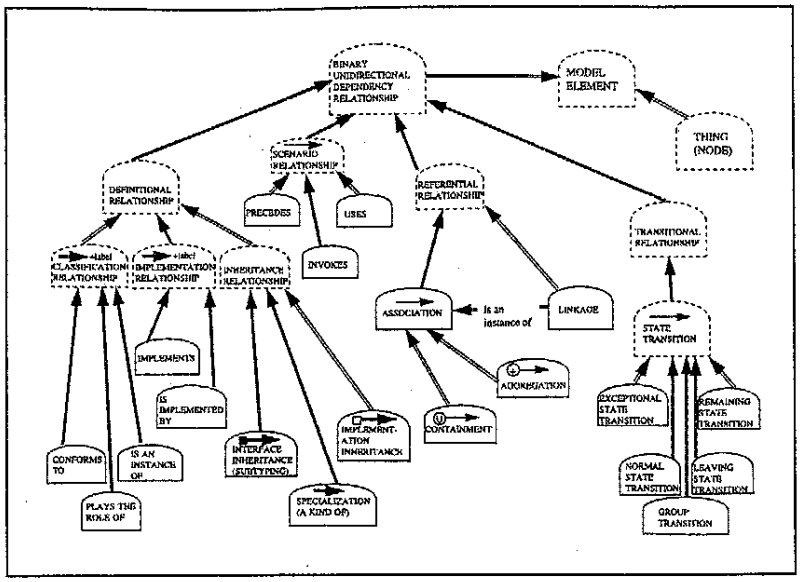


Figure 3. Various forms of relationship in the OML. Here we see unidirectional associations, aggregations, and the containing relationship.

In Figure 3 we see the major relationships illustrated: specialization, unidirectional associations (mappings), aggregations, and containment. Again the icons chosen are self-explanatory. Specializations are very close bindings between sub- and superclass/type. They have a broad arrow—a double arrow is chosen to represent strong coupling as well as being easier to draw by hand than a thick one and easier to see when different magnifications are used in a drawing tool. It is also less common than an association/linkage which therefore is allocated the easiest arrow to draw (single thickness). A label can be used to indicate the discriminator used for the subclassing. Specialization, the default, is an is-a-kind-of relationship. Other types of "inheritance" can be shown but the basic, encouraged is-a-kind-of gets the easiest-to-draw line. All relationships are unidirectional—as indicated by the arrowhead. Consequently, an association is always, by default, unidirectional. If no decision has yet been made, the arrowhead may be left off until a later decision adds it. This is more reasonable than permitting an unarrowed line to mean bidirectional. Leaving off arrowheads may be carelessness rather than a design decision! Mappings as used here do not break encapsulation; whereas bidirectional associations do;[4] if needed, these are represented by double-headed arrows as a shorthand for a pair of unidirectional associations which are semistrong inverses of one another. We believe in supporting a true object-oriented paradigm as default. Unidirectional associations minimize coupling and thus enhance reuse as well as simplifying forward and reverse engineering.

Aggregation, although historically not as well-defined as we would all like it to be, represents a fairly permanent binding: an "is-composed-of" or "part-whole" relationship. At the same time, we can see this symbol as a plus sign which represents the fact that the aggregate (here the car engine) is (usually more than) the sum of its parts—an aggregate has at least one emergent property.

Often confused with aggregation is the looser "collection" concept. A good example here is what you store in the trunk of your

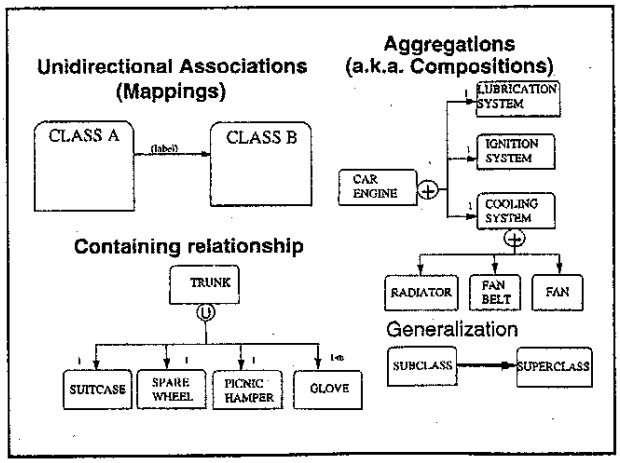


Figure 4. The relationship metamodel hierarchy for OML. The arrow style is dictated by the position in this hierarchy. All definitional relationships are indicated by a double arrow, all referential relationships by a single arrow. Three styles of inheritance are also represented: is-a-kind-of (generalization—the default), subtyping (black box), and implementation inheritance (white box).

car. The items are connected with trunk, but that connection may be highly temporary. We replace the plus symbol with a cup symbol to give the visual clue suggested by this icon of a cup and its contents. An aggregation results in a structure having relationships between parts, whereas when using containment there is no such interpart relationship.

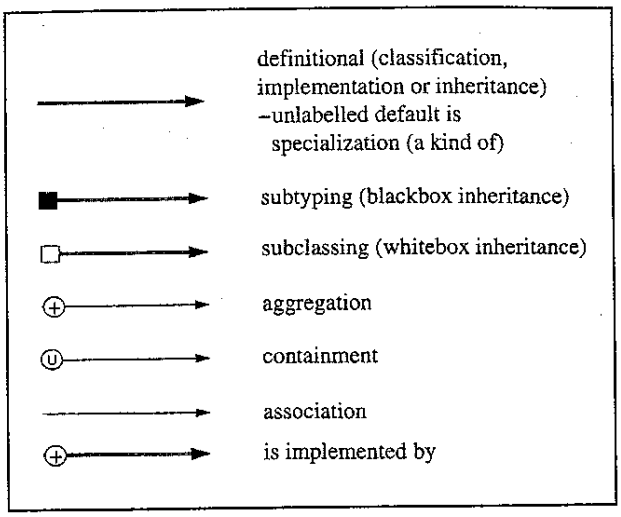


Figure 5. The major relationship arrows in COMN.

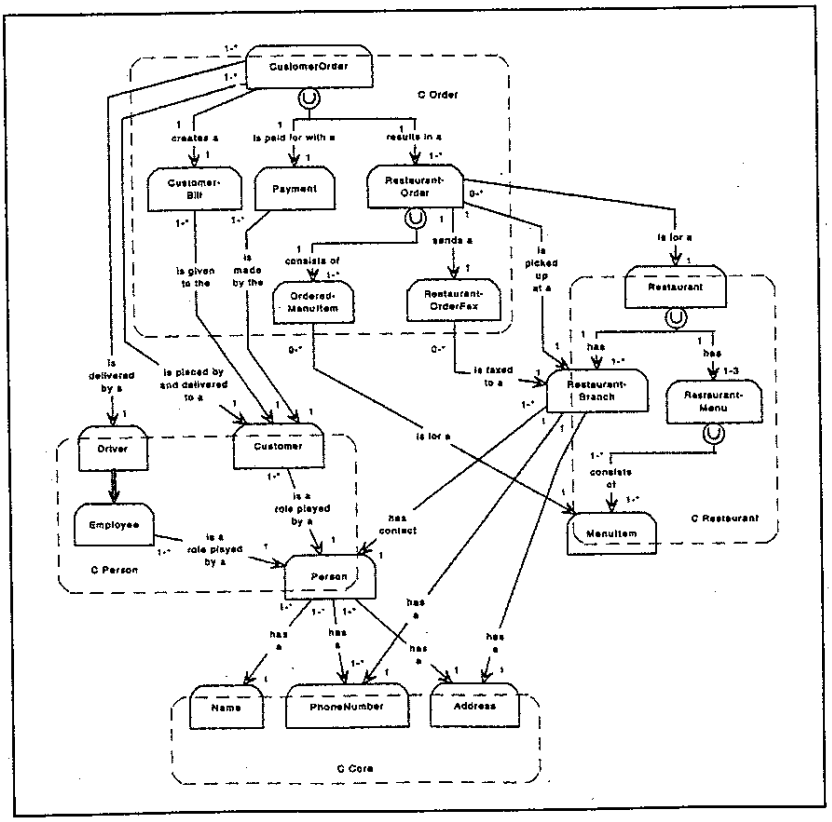


Figure 6. Cluster diagram showing classes for a good delivery order entry application (after Firesmith *et al*.[5]).

BASIC RELATIONSHIPS IN COMN

One pleasing aspect of the relationship model is its carefully structured metamodel (shown in Fig. 4). All relationships, as noted previously, are binary, unidirectional dependencies or mappings. These can be of two major types (four when we include dynamic models). The two static relationship types are referential (in which one thing "knows about" another) and definitional (in which one thing is defined by relationship to another).

Referential relationships

All referential relationships use a single width arrow. Associations and linkages (associations for classes, linkages for instances) have an unadorned solid single arrow whereas for aggregation and containment it is adorned at the "whole" end (see Figs. 4 and 5).

Definitional relationships

We use a double arrow for the tighter definitional relationship. Our default definitional node, the easiest to draw, is the is-a-kind-of which thus gets an unadorned double arrow. An is-a-kind-of relationship is good for both knowledge representation (in say user requirements/analysis) and in support of polymorphism, through dynamic substitutability. Since we discourage simple subtyping (specification inheritance) and implementation inheritance, they are represented as adornments (a blackbox and whitebox respectively at the subclass end of the arrow, to represent blackbox and whitebox inheritance)—(see Fig. 5). All other definitional relationships (also with double arrows) carry a textual label. These can be grouped into classification relationships (conforms-to, is-an-instance-of, plays-the-role-of) and implementation relationships (implements, is-implemented-by).

Transitional and scenario relationships

There are, in fact, two further types of relationships: transitional and scenario. These are more advanced features, not part of COMN Light and thus not discussed here.[5] Transitional relationships are not used in the static model (semantic net or class models) but only in the dynamic models and scenario relationships in use case/task script models. Beginners can use any state transition model they choose before investigating the OML state model. Similarly, although we prefer a task script/use case model for large systems, any version of this to help gain understanding of user requirements will be satisfactory for learning the overall OPEN approach. Interconnections between diagrams are similarly reserved for the full notation discussion.[5]

DIAGRAM TYPES

Since COMN is used to model business domains and applications with a great deal of inherent complexity, no single view or diagram type is adequate to capture all important aspects of the resulting model. COMN therefore offers set diagram types that provide relatively orthogonal views of the single underlying model. Some diagrams document static architecture, whereas others document dynamic behavior. Some diagrams view the model at a very high, blackbox level of abstraction, whereas others open up the blackbox to display encapsulated details. The OML metamodel[5] provides a way to capture the single, underlying model and check for consistency. Because they all describe different aspects of a single system, it is critical that changes made to one diagram are synchronized across and reflected in appropriate changes being made to all the other diagrams, probably by using an automated CASE tool. These views are relatively orthogonal in that each diagram provides a view of the underlying model that can be understood and studied separately. However, these diagrams are related to each other in interesting and useful ways because they provide views of a single underlying model, and this allows CASE tool vendors to provide coherent cross-referencing and consistency checking. There are four major kinds of diagram types in OML: semantic nets, scenario class diagrams, interaction diagrams, and state transition diagrams.

Semantic nets

The semantic net, of which there are six subtypes in OPEN (see Table 1), is the most important and most widely-used diagram type in the OPEN method. OPEN uses semantic nets instead of extended entity relationship diagrams because:

- Semantic nets from the artificial intelligence community are designed to document static architecture in terms of modeling elements and the semantically important relationships among them.
- Semantic nets naturally capture classification (is-a), specialization (a-kind-of),* and aggregation (has-part) relationships.

Table 1. Six subtypes of OPEN Semantic Net.

Diagram Type	Purpose
1. Context Diagrams: System Context Diagrams Software Context Diagrams	Depicts scope and environment
2. Layer Diagrams	Depicts the overall architecture in terms of layers.
3. Configuration Diagrams	Depicts the overall architecture in terms of clusters.
4. Cluster Diagrams	Describes the elements comprising each cluster.
5. Inheritance Diagrams	Depicts all or part of an inheritance graph.
6. Deployment Diagrams	Depicts the allocation of software to hardware in a distributed system.

- These diagrams primarily document objects and classes, rather than relational database tables. Entity relationship diagrams from data models are therefore misleading.
- These diagrams should not be called class diagrams (as they are by UML) because they document internal and external objects, types, roles, and clusters as well as classes (see Fig. 6).

Scenario class diagrams

Scenario class diagrams may be used for 1) task scripts, 2) use cases, or 3) mechanisms. All three are supported in COMN to document a set of collaborating scenario classes and the invocation and precedes relationships between them.

Interaction diagrams

Interaction diagrams show interactions (message passing and exception raising), may be either collaboration or sequence diagrams, and are fairly standard in OO notations. Collaboration diagrams have a graph structure similar to semantic nets (see Fig. 7) and are typically used to provide summary information, whereas sequence diagrams use the standard fence notation and are used to show sequencing. Major advantages of COMN over UML interaction diagrams are the ability to handle exceptions and the availability of logic boxes to handle branching, looping, and interleaving due to

* The AI community also does not confuse is-a and a-kind-of relationships, which is, unfortunately, common in the object community.

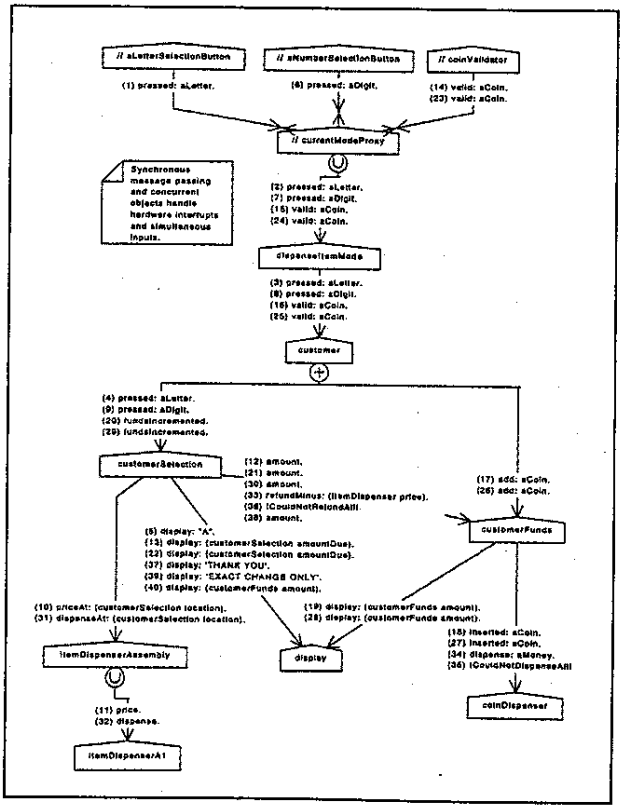


Figure 7. Scenario collaboration diagram for a vending machine application (after Firesmith *et al.*).

concurrency, thereby greatly decreasing the number of diagrams that need to be developed and maintained.

State transition diagrams

While state transition diagrams may have many flavors, the underlying concepts are usually credited to Harel.[8] The details of the COMN STD are in Firesmith *et al.*[5]—there are no startling differences compared to other STD approaches, just differences of detail. COMN STDs were developed from Harel statecharts incorporating later work from Embley *et al.*[9] and Selic *et al.*[11] States and transitions are represented with events triggering changes between states. Guards decide whether any particular event will trigger an event change or whether the state remains unchanged.

SUMMARY

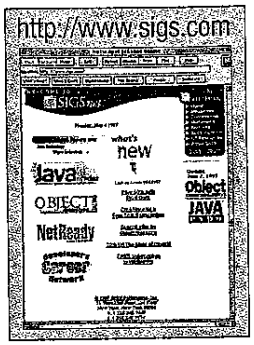
We have outlined the main elements of the COMN "Light Notation," essentially describable on the "back of an envelope." We have aimed for minimality in semantics and icon set while acknowledging that the notation is likely to evolve, especially with niche extensions such as hard real-time and distribution. This will particularly result as more "satellite" methods are merged into the mainstream of OPEN. ■

Acknowledgments

This is Contribution number 97/25 of the Centre for Object Technology Applications and Research.

References

1. Henderson-Sellers, B., I. M. Graham, and D. Firesmith. "Methods Unification: The OPEN Methodology," *Journal of Object-Oriented Programming*, 10(2): 41–43, 55, 1997.
2. Graham, I., Henderson-Sellers, B., and H. Younessi. *The OPEN Process Specification*, Addison-Wesley, Wokingham, UK, 1997.
3. Henderson-Sellers, B. and D. Firesmith. "COMMA: Proposed Core Model," *Journal of Object-Oriented Programing*, 9(8): 48–53, 1997.
4. Graham, I. M., J. Bischof, and B. Henderson-Sellers. "Associations Considered a Bad Thing," *Journal of Object-Oriented Programing*, 9(9): 41–48, 1997.
5. Firesmith, D., B. Henderson-Sellers, and I. Graham. *OPEN Modeling Language (OML) Reference Manual*, SIGS Books, New York, 1997.
6. Constantine, L. L., and B. Henderson-Sellers. "Notation Matters: Part 1—Framing the Issues; Part 2—Applying the Principles," *Report on Object Analysis and Design*, 2(3): 25–29 and 2(4): 20–23, 1995.
7. Kilov, H., and J. Ross. *Information Modeling. An Object-Oriented Approach*, Prentice Hall, Englewood Cliffs, NJ, 1994.
8. Harel, D. "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, 231–274, 1987.
9. Embley, D. W., B. D. Kurtz, and S. N. Woodfield. *Object-Oriented Systems Analysis. A Model-Driven Approach*, Yourdon Press, Englewood Cliffs, NJ, 1992.
10. Firesmith, D. G. *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach*, Wiley, New York, 1993.
11. Selic, B., G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*, Wiley, New York, 1995.



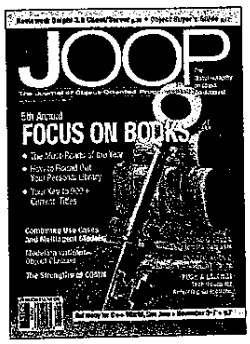
JOOP

The Journal of Object-Oriented Programming

The Global Authority on Object Development

September 1997 Vol. 10, No. 5

FOCUS ON BOOKS



Cover photo: Romilly Lockyer/Image Bank