ROAD

COMMA: Proposed core model

Brian Henderson-Sellers

Donald G. Firesmith

ROAD: Report on Object Analysis & Design

In previous columns,[1,2] we described the overall project structure for COMMA (the Common Object Methodology Metamodel Architecture) and some sample results from our metamodeling effort of fourteen methodologies. Based on these 14 models, in this article, we make some tentative proposals for a common, core metamodel.

AN EMBRYONIC CORE METAMODEL

The static model for objects, classes, types In any object model, there needs to be at least objects and classes/types. Here we are discriminating between the individual instance (model of a single thing in the real world or application implemented as computer memory containing values of the thing's properties) and the concept that captures a single kind of such individual instances. At the conceptual level, we can describe concepts by their *intension*—the complete definition of the object type that defines when a concept does or does not apply to an individual object.[3] The *extension*, on the other hand, defines the collection of individuals currently satisfying the intension.

While object-oriented models in methodologies typically deal with object types, the word most often used is "class." Use of the word class to mean concept or object type is at variance with its use in programming languages to mean the implementation details of a particular object type.

This then leads to the need for a three-part* nomenclature of:

- An *object type* that is at the conceptual level displaying the externally (public) viewable characteristics of a specific kind of object. It is this interface that defines the public view of the class (q.v.)
- A *class* that is the object type *plus* its implementation. The class therefore defines the intension—the rules that determine whether individuals do or do not belong. It is a way to define and instantiate members of the extension. This is a "collective" notion only in the sense that a class can be regarded as a "template" or "factory"[5] by which individuals (objects) can be instantiated. This then leads to the concept of a *concrete class* (a class that can be instantiated), the alternative *abstract class* (see Fig. 1) being a concept that has no instances. A class consists of an interface (that specifies the object type) and an implementation (an aggregation, shown in Figure 1 by a directed arrow with a circle containing a plus sign at the aggregate end of the relationship).
- An *object* (a model of a thing in the real world or application) implemented as an instance of a class. Classes may be viewed

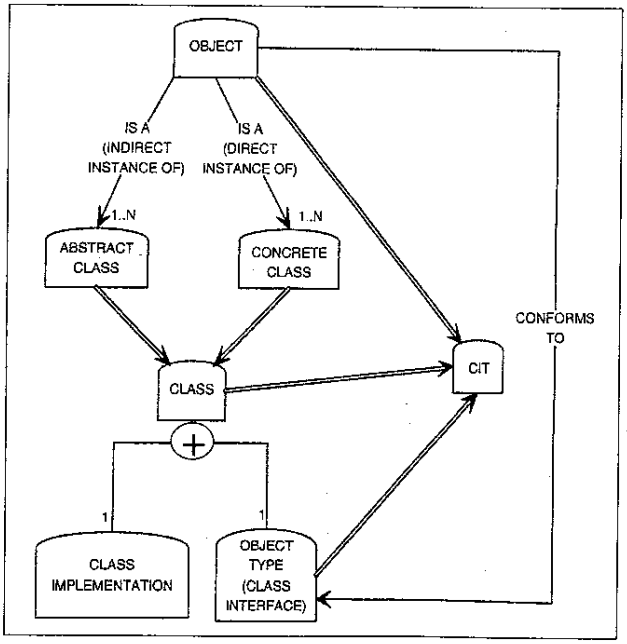


Figure 1. CIT as a generalization of OBJECT TYPE, CLASS and OBJECT. Only CLASS has a CLASS IMPLEMENTATION. (These are drawn using the emerging OPEN notational standard in which a double arrow indicates inheritance, a directed line [possibly labeled] an association, and where aggregation is shown by an arrow with a circle with a plus sign [a "Philips screw-head"] at the aggregate end of the line.)

* There is arguably a fourth part: rôle.[4] While this may eventually be recognized as important to the core, it is missing from most current methodologies.

B. Henderson-Sellers is professor of computer science in object technology at Swinburne University of Technology, Victoria, Australia. Don Firesmith is a Senior Member of Knowledge Systems Corporation. He can be reached at dfiresmith@ksccary.com.

as objects (instances of metaclasses) and scenarios as instances of use cases. Thus "instance" is a broader term and we can talk of CLASS_INSTANCE (a.k.a. OBJECT) as the most useful (for the current discussion) subtype of INSTANCE (see Fig. 2).

While these are three distinct notions, there are many occasions when a collective word is needed for all three (a generalization or supertype in O-O parlance). Examples in the literature are O/C[6] or just objects.[7] In COMMA, we explicitly introduce the metalevel concept of CIT (the class, instance, or type) for this generalized term (see Fig. 1).

Inspection of the metamodels for specific methodologies does show frequent occurrence of the notion of CLASS versus OBJECT where CLASS frequently confounds both CLASS and TYPE. Some of these deal with OBJECTs and CLASSes separately but many of these also invent an abstract metamodeling concept equivalent to the use of a CIT.

OBJECT TYPEs should have VISIBLE RESPONSIBILITYs (see Fig. 3). A visible responsibility is any high-level purpose, obligation, or required capability of a CIT, typically provided by a cohesive set of one or more characteristics (i.e., visible operations, properties, rules—see below). A responsibility for doing something is provided by one or more visible operations; a responsibility for knowing is provided by one or more visible properties and associated visible operations. Although not present in many methodologies, it is increasingly clear from recent articles and public discussions at international conferences that we also need responsibilities to describe rules for the CITs. In this respect, COMMA attempts to be proactive in creating its core metamodel. Thus the core contains responsibilities for enforcing, provided by properties and associated exception(s). Not shown here as part of the proposed core, but seriously encouraged, is that RESPONSIBILITYs should be enforced by associated CONTRACTs.[8-10]

Encapsulation. The external/internal dichotomy is critical to object technology. Few current methodologies stress which part of their object model relates to the visible and which to the hidden characteristics. This is probably because most of the OOA/D methodological development focuses on ADTs (actually OBJECT TYPEs) and seldom delves deeply into the implementation details within a class. Nevertheless, any object metamodel should support this external/internal split. In the COMMA core metamodel we need to be clear.

The external view is the object type, also known as the class interface. The class is then this class interface plus the class implementation (see Fig. 1). The class interface has a number of visible or public responsibilities and visible characteristics (see Fig. 3). These are themselves linked to the class implementation details, and thus the internal view of the class is that of a set of hidden responsibilities and hidden characteristics (see Fig. 4). Responsibilities and characteristics thus provide the conduit between the external view and the internal view (i.e., they have both hidden and visible parts).

Viewed from inside the class, we thus see HIDDEN RESPONSIBILITYs (for doing, enforcing, and knowing) implemented by

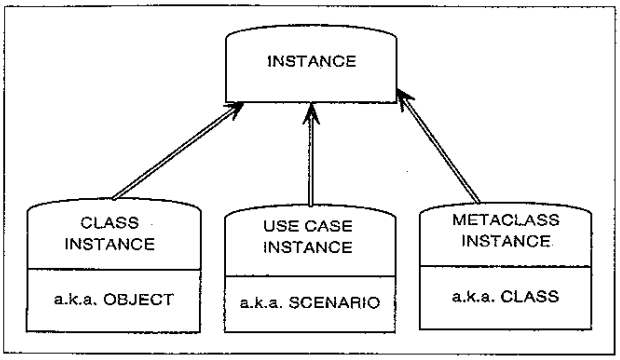


Figure 2. Supertype INSTANCE with some of its subtypes.

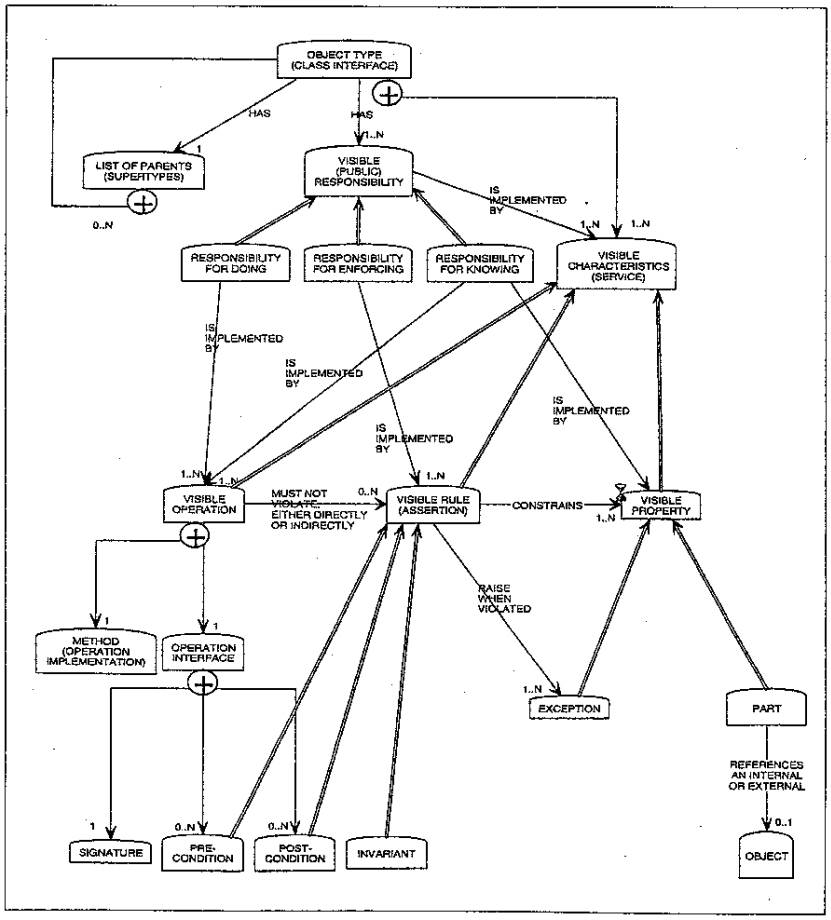


Figure 3. An OBJECT TYPE has VISIBLE RESPONSIBILITYs. These can be subdivided into responsibilities for knowing, doing, and enforcing, which are then implemented by VISIBLE OPERATIONs, RULEs, and PROPERTYs.

HIDDEN CHARACTERISTICS. These characteristics comprise HIDDEN OPERATIONS (OPERATION INTERFACE plus the associated METHOD), HIDDEN RULEs, and HIDDEN PROPERTYs (EXCEPTIONs, LINKs, PARTs, and ATTRIBUTEs). This is in agreement with the recommendation that all attributes should be hidden and only accessible via a responsibility implemented by one or more methods.[11] However, in some methodologies and OOPLs, attributes appear to be publicly visible, e.g., OMT. While this may be regarded as semiotically† dangerous[12,13] because it is likely to give the wrong signals, and indeed is regarded by some as non-O-O by breaking encapsulation, there is an argument for supporting apparently publicly visible attributes in the sense that they can be read simply as shorthand for associated accessor functions.

It may also be useful to divide operations into procedures and functions[8] (as suggested in Figure 4). Attributes are hidden properties that may reference an internal OBJECT or be simple data types (e.g., as advocated in OMT). Other properties are represented as LINKs to other, external OBJECTs. PARTs, that relate to the notion of aggregation, are also HIDDEN PROPERTYs that may reference internal or external OBJECTs. While Figure 3 depicts the external view (TYPEs) and Figure 4 the internal view (CLASS IMPLEMENTATIONs), the connection between these two is depicted in Figure 5 in terms of CHARACTERISTICS, which may be Visible or Hidden (a Boolean attribute of the metalevel CHARACTERISTIC). This figure links the two critical viewpoints needed for a full description of objects, types, and classes.

Figure 6 merges together Figures 1–5. This COMMA core metamodel is much more than either the union or the intersection of existing methodologies, preferring to keep at the leading edge of O-O thinking, both published and unpublished.

Figure 6 is the proposed COMMA core metamodel (static).

It is worth noting that types have logical properties whereas instances and classes have both logical and physical properties. This is well represented in the metamodel. OBJECT TYPE contains RESPONSIBILITYs which are then mapped through to CHARACTERISTICs. On the other hand, an OBJECT is a direct instance of one or more CONCRETE CLASS(es) (and hence of a CLASS), which has as a direct component the CLASS IMPLEMENTATION that gives direct access to the HIDDEN CHARACTERISTICs.

Compatibility of proposed core with existing metamodels Here, we undertake a very brief comparison of the proposed static COMMA metamodel, shown in Figure 6, and the individual methodology metamodels, some of which were illustrated in more detail in our previous column.[2]

Booch. Booch has low-level attributes and operations but nothing equivalent to the CIT responsibilities or rules. Neither does it differentiate between object types and classes—not unusual. Being primarily a design method, it surprisingly does not spell out implementation details; although within the text there are sufficient rules and ideas to implement a design in C++. The dichotomy between internal and external views is not evident in Booch's methodology[14] but is recognized explicitly in a later publication.[15]

OMT. The Object Modeling Technique has no generic CIT concept. OBJECTs and CLASSes have attributes and operations (there is no word like *responsibility, characteristic,* or *service* to conjoin these) and there is no mention of rulesets. With these constraints and some terminology differences, the OMT metamodel (corresponding to the proposed core ONLY) is identical with that of Booch.

RDD. Although Responsibility Driven Design doesn't have a CIT concept, it does have many of the other elements. Rulesets are not included as such but the concept of CONTRACTs is paramount. Messages are more of a focus than in other methodologies and we have included them in this metamodel.

MOSES. MOSES captures much of the pro-

† Semiotics is the study of signs and symbols and the way they provide representation and communication of ideas to the reader.

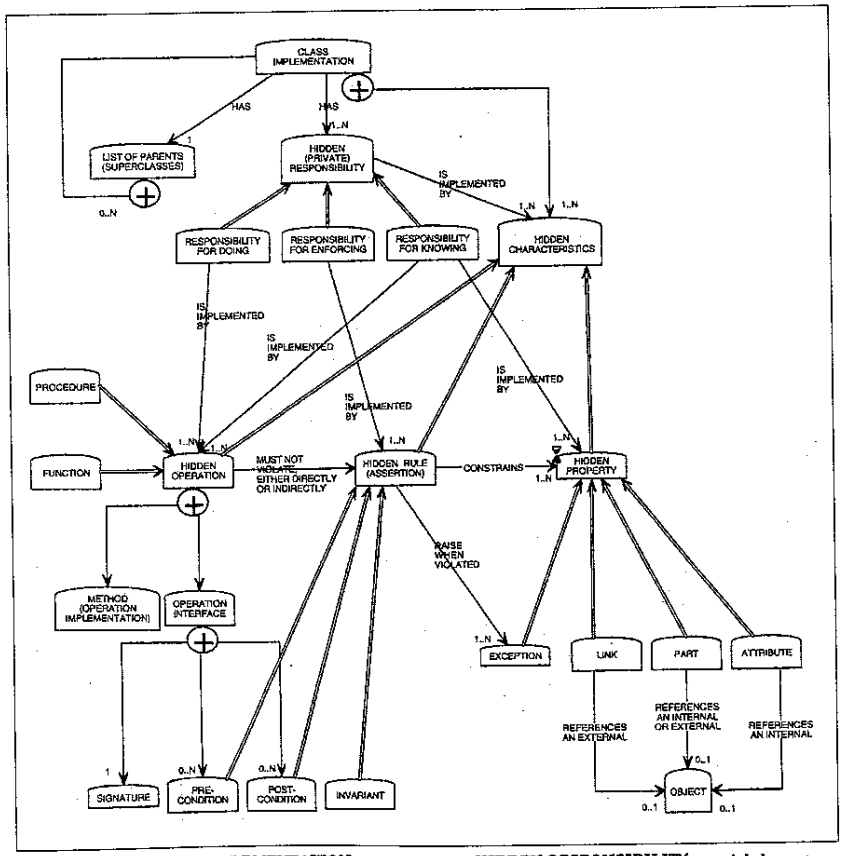


Figure 4. CLASS IMPLEMENTATION consists of HIDDEN RESPONSIBILITYs, which are implemented by HIDDEN CHARACTERISTICs (OPERATIONs, RULEs, ATTRIBUTEs, PARTs, LINKs, and EXCEPTIONs).

posed core metamodel except for OBJECT TYPE; RESPONSIBILITY FOR ENFORCING, although present, needs considerable improvement, not being of significant enough focus to be included in the derived MOSES metamodel. In addition, the concept of CONTRACT is important, as it is in RDD, BON, and Firesmith. No real discussion of implementation details are given although some code constructs such as friends, client/server, and embedding are included; however, MOSES does reserve the term "attribute" for internal, hidden information.

SOMA. It is interesting to note that SOMA, one of the more recently published methodologies, has all the elements shown in Figure 6 augmented by the CONTRACT concept, with the exception of some of the detailed implementation details and the notion of OBJECT TYPE.

Martin/Odell. This methodology has different terminology and a somewhat different focus. For instance, EVENTS are more important (not discussed here). OBJECT TYPE is included, as are CLASS and OBJECT (note that OBJECT TYPE is in accord with OMG recommendations), although there is no concept at the generalized CIT level. OBJECT TYPES (and their implementations as classes) have properties and operations. Properties seem to be viewed primarily as associations (in OMT and MOSES, there is an interchangeability between these two concepts—see also OBJECTEXPO EUROPE CONFERENCE PROCEEDINGS[16]). However, there are no clearly described implementation details in Martin and Odell[3]—these are considered in a very recent publication.[17]

BON. BON's simple metamodel focuses on features (a synonym for responsibility) and formality (through ASSERTIONs that represent the RULEs needed for contracting). BON does not even stress a distinction between CLASS and OBJECT and its focus on responsibilities is less explicit than in Figure 6. No generic OBJECT TYPES or CITs are included, nor any real implementation details (at the metalevel).

Fusion. Fusion is one of the few methodologies to explicitly separate analysis and design. For instance, methods are only considered to be part of the interface during design and not analysis. Messages are also a feature of this approach and some implementation concepts are included. However it does contain many of the elements of Figure 6, with the exception of RESPONSIBILITYs FOR ENFORCING (or RULEs) and CIT.

OOSE. OOSE is in many ways similar to OMT and Booch but in this case, the concepts of attributes and operations seem to be more closely connected with the OBJECT concept than with the CLASS concept (although this may turn out to be more terminological than methodological). OOSE has no obvious CIT concept and no discussion at all of responsibilities, the main focus of OOSE being the more functionally-oriented use case (not included in the current stage of the COMMA project). Some mention is made, however, of implementation details in OOSE.

Coad. The Coad approach shows a very different approach to the relationships between OBJECTs and CLASSes. The term CLASS-&-OBJECT is defined as "a CLASS and the OBJECTs in that CLASS." While this suggests a possible aggregation relationship, the author has advised us that the "class with objects" symbol should be regarded as being equivalent to CONCRETE CLASS, whereas the concept CLASS is equivalent to ABSTRACT CLASS in Figure 6. OBJECTs still belong to concrete classes although they are not able to be portrayed graphically independently in the associated Coad notation. CONCRETE CLASSes (CLASS-&-OBJECT) then have ATTRIBUTEs and CHARACTERISTICs. As an analysis method, extended into design, it is not surprising to find no implementation details.

Shlaer/Mellor. Shlaer/Mellor has CLASSes (called objects), ABSTRACT CLASSes, ATTRIBUTEs, and OPERATIONs (called published operations) only.

Firesmith. In Firesmith,[18] there is a clear separation between internal and external viewpoints (perhaps the clearest of all the methods considered). The implementation (called BODY) contains much of the proposed metamodel core including the division of OPERATIONs into PROCEDUREs and FUNCTIONs (called MODIFIERs and PRESERVERs). Firesmith essentially has all the elements shown in Figure 6, although RESPONSIBILITY FOR ENFORCING is most clearly seen in the O/C bodies in the form of

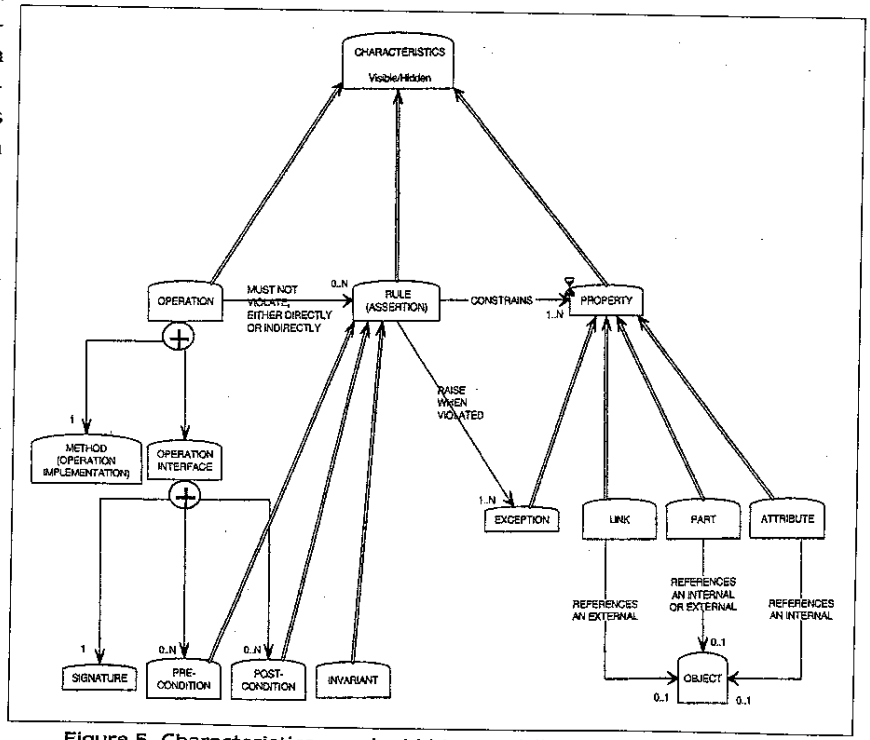


Figure 5. Characteristics may be hidden or visible. This attribute (at the metalevel) is inherited by OPERATIONs, RULEs, and PROPERTYs.

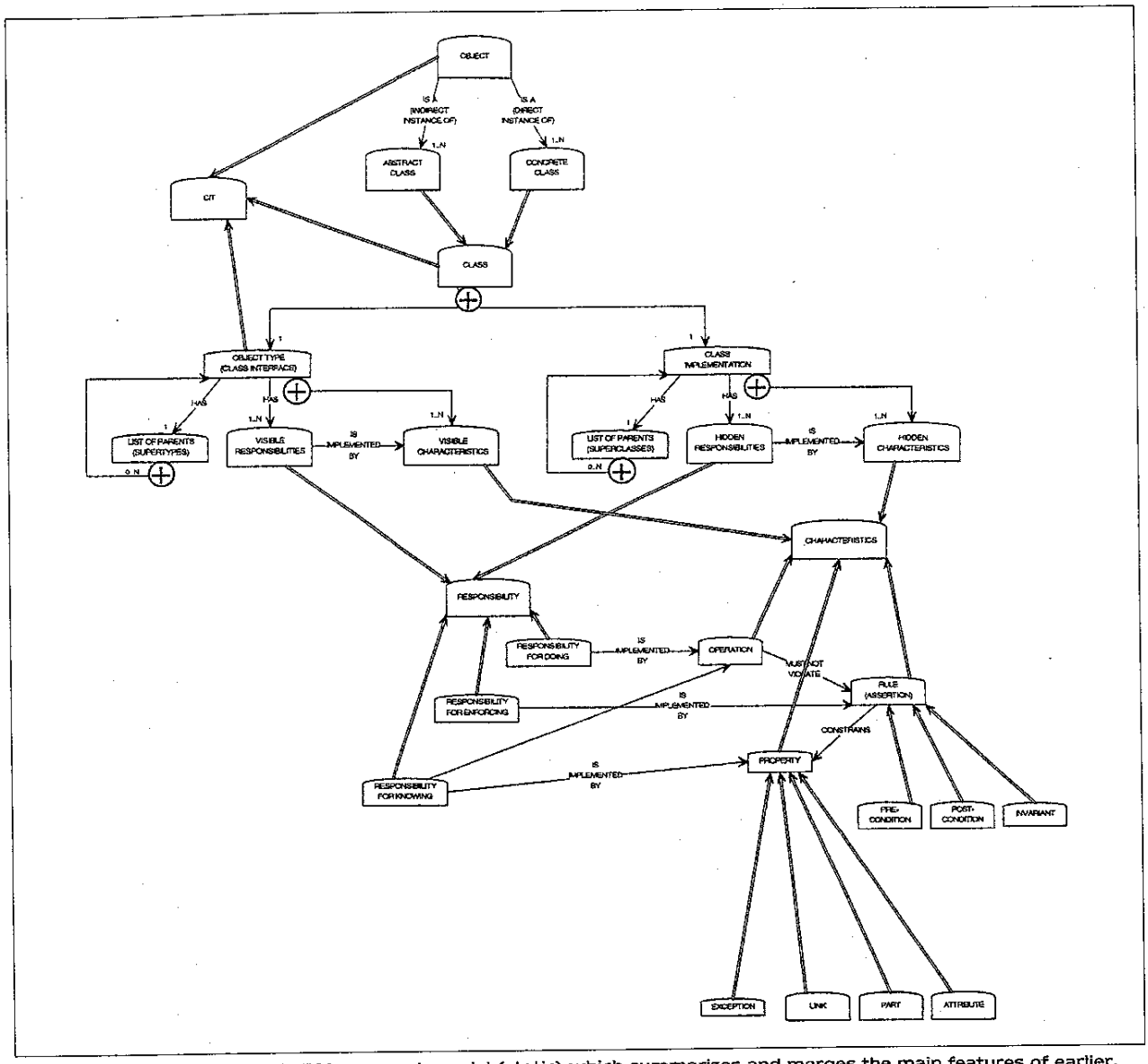


Figure 6. The proposed COMMA core metamodel (static) which summarizes and merges the main features of earlier, more detailed diagrams.

assertions and class invariants. Also, the LIST OF PARENTS is seen as part of the class specification rather than the class body. It should be noted that these comments pertain to the published (1993) version of the methodology and that the author is moving rapidly toward the COMMA metamodel described here.

OBA. OBA's focus is on instances, roles, and responsibilities. It does not obviously discuss the CIT/ABSTRACT CLASS/CONCRETE CLASS model on the left hand side of Figure 6. There are no further implementation details because, naturally, OBA is an analysis method, considering only non-implementation concerns. This should therefore not be regarded as any deficiency in this approach.

The Unified Method. While not yet an officially published method, the Unified Method of Booch and Rumbaugh[19] has produced publicly available documentation (version 0.8) as of October 1995.‡ The metamodel is fully documented in both graphical form and in the form of a dictionary set of definitions.

CLASSDECL is a concept that defines a name and class structure (attributes and operations). It is not necessarily a TYPE. This sounds roughly equivalent to CLASS in Figure 6. However, while not being a TYPE, it does inherit from TYPEDECL, which declares a TYPE name in some scope. This seems on the face of it contradictory. Another metalevel concept CLASS is the superclass/supertype of CLASSDECL and is "a definitional entity that has instances with identity." It is noted that "If abstract, then it cannot have direct instances." Thus the definition is that of an entity with instances

‡ The next version, the Unified Modeling Language (UML) (version 0.9) was released in July 1996.

but that may not have instances—again paradoxical. Certainly we can say that the concept CLASS encompasses both CONCRETE CLASS and ABSTRACT CLASS in Figure 6.

While RESPONSIBILITYs feature strongly in the metamodel, they are attached to LOGICAL ELEMENT, which is the ancestor class of the internal concepts of ATTRIBUTE and OPERATION. This suggests that responsibilities are only viewed as "hidden" and only then at a "method level"; while mostly the use of RESPONSIBILITY is either as part of the external view of the class or may be subdivided between PUBLIC and PRIVATE, i.e., at the CLASS or TYPE level.

While there are overall similarities, the early nature of this metamodel suggests that there are likely to be near-future changes. Consequently, we will not pursue the comparison to any further depth in this article.

CONCLUSION A static core metamodel has been proposed for object modeling based on an analysis of 14 existing OOAD methods, supplemented by discussions with the authors. While none of the methods fully support the totality of the proposed core, the core aims to both coalesce, facilitate, and encourage agreement on a standard—a standard that is close to almost all methods but clarifies some poorly explained portions of all methodologies as well as attempting to reflect current (1996) understanding of object technology to enhance the information published in the 14 methodology text books (over the last six years).

The participants in the COMMA project are keen to receive comments on the results of the project, specifically on these tentative ideas towards a core metamodel. Please email comments to brian@csse.swin.edu.au. ■

Acknowledgments This is Contribution number 96/3 of the Centre for Object Technology Applications and Research.

References

1. Henderson-Sellers, B. and A. Bulthuis. The COMMA project: First steps, REPORT ON OBJECT ANALYSIS AND DESIGN, 2(7):49–52, 1996.
2. Henderson-Sellers, B. and A. Bulthuis. COMMA: Sample metamodels, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, 9(7):44–48, 1996.
3. Martin, J. and J.J. Odell. OBJECT-ORIENTED METHODS: A FOUNDATION, PTR Prentice Hall, Upper Saddle River, NJ, 1995.
4. Reenskaug, T., P. Wold, and O.A. Lehne. Working with Objects, THE OORAM SOFTWARE ENGINEERING MANUAL, Manning, Greenwich, CT, 1996.
5. Cox, B.J. OBJECT ORIENTED PROGRAMMING: AN EVOLUTIONARY APPROACH, Addison-Wesley, Reading, MA, 1986.
6. Henderson-Sellers, B. and J.M. Edwards. BOOK TWO OF OBJECT-ORIENTED KNOWLEDGE: THE WORKING OBJECT, Prentice Hall, Sydney, 1994.
7. Graham, I.M. MIGRATING TO OBJECT TECHNOLOGY, Addison-Wesley, Wokingham, UK, 1995.
8. Meyer, B. OBJECT-ORIENTED SOFTWARE CONSTRUCTION, Prentice Hall, Hemel Hempstead, 1988.
9. Wirfs-Brock, R.J., B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Upper Saddle River, NJ, 1990.
10. Waldén, K. and J.-M. Nerson. SEAMLESS OBJECT-ORIENTED ARCHITECTURE, Prentice Hall, Upper Saddle River, NJ, 1995.
11. Wirfs-Brock, A. and B. Wilkerson. Variables limit reusability, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, 2(1):34–40, 1989.
12. Constantine, L.L. and B. Henderson-Sellers. Notation matters: Part 1—Framing the issues, REPORT ON OBJECT ANALYSIS AND DESIGN, 2(3):25–29, 1995.
13. Constantine, L.L. and B. Henderson-Sellers. Notation matters: Part 2—Applying the principles, REPORT ON OBJECT ANALYSIS AND DESIGN, 2(4):20–23, 1995.
14. Booch, G. OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS (2ND EDITION), Benjamin/Cummings, Redwood City, CA, 1994.
15. Booch, G. OBJECT SOLUTIONS: MANAGING THE OBJECT-ORIENTED PROJECT, Addison-Wesley, Menlo Park, CA, 1996.
16. Premerlani, W. Object model transformations, OBJECTEXPO EUROPE CONFERENCE PROCEEDINGS, SIGS Conferences, New York, 1994.
17. Martin, J. and J.J. Odell. OBJECT-ORIENTED METHODS: PRAGMATIC CONSIDERATIONS, Prentice Hall, Upper Saddle River, NJ, 1996.
18. Firesmith, D.G. OBJECT-ORIENTED REQUIREMENTS ANALYSIS AND LOGICAL DESIGN: A SOFTWARE ENGINEERING APPROACH, John Wiley, New York, 1993.
19. Booch, G. and J. Rumbaugh. UNIFIED METHOD DOCUMENTATION VERSION 0.8, Rational Software Corporation, Santa Clara, CA 1995.

An empirical study

continued from page 47

evidence also showed that it is very difficult to produce a comprehensive problem statement. Function list or use case methods are more suitable in an industrial programming team.

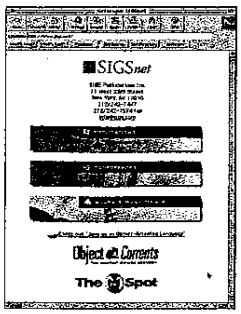
One factor that may influence the outcome of a software project may be measured by the union of the programmer domains within a project team. A project has a higher chance for success if the union of all the programmer domains cover most of the enclosing application domain.

This is a preliminary study into the programming domains and their impact on the OOA&D process. More research is needed to further understand the characteristics of the programming domains. A more challenging task is to develop a framework to quantitatively measure the programming domains. ■

References

1. Walz, D.B., J.J. Elam, and B. Curtis. Inside a software design team: Knowledge acquisition, sharing, and integration, COMMUNICATIONS OF THE ACM 36(10):62–77, 1993.
2. Holtzblatt, K. and H. Beyer. Making customer-centered design work for teams, COMMUNICATIONS OF THE ACM 36(10):93–103, 1993.
3. Zultner, R.E. TQM for technical team, COMMUNICATIONS OF THE ACM 36(10):79–91, 1993.
4. Booch, G. OBJECT-ORIENTED ANALYSIS AND DESIGN, 2ND EDITION, Benjamin-Cummings, New York, 1994.
5. White, I. USING THE BOOCH METHOD, Benjamin/Cummings, Redwood City, CA, 1994.
6. Jacobson, I., et al. OBJECT-ORIENTED SOFTWARE ENGINEERING, Addison-Wesley, Reading, MA, 1992.
7. Martin, J.C. DESIGNING OBJECT ORIENTED C++ APPLICATION USING THE BOOCH METHOD, Prentice-Hall, Upper Saddle River, NJ, 1995.
8. Conger, S. THE NEW SOFTWARE ENGINEERING, Wadsworth Publishing, Belmont, CA, 1994.
9. Pressman, R.S. SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, McGraw Hill, New York, 1992.

http://www.sigs.com



JOURNAL OF OBJECT-ORIENTED Programming

January 1997
Vol. 9, No. 8