COMMA: Proposed core model

B. Henderson-Sellers
School of Computer Sciences and Software Engineering,
Swinburne University of Technology,
Hawthorn, Victoria, Australia
email: brian@csse.swin.edu.au

D. Firesmith
KSC,
4001 Weston Parkway,
Cary,
NC, 27513, USA
email: Dfiresmith@ksccary.com

Printed: September 28, 1998

[Column for Journal of Object-Oriented Programming (Sept 1996)]

Address for Correspondence
Professor B. Henderson-Sellers
School of Computer Science and Software Engineering
Swinburne University of Technology
John Street
PO Box 218
Hawthorn
Victoria 3122
AUSTRALIA
fax: +61 3 9819 0823
email: brian@csse.swin.edu.au
In previous columns\textsuperscript{1,2}, we described the overall project structure for COMMA (the Common Object Methodology Metamodel Architecture) and some sample results from our metamodelling 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 need 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 which defines when a concept does or does not apply to an individual object\textsuperscript{3}. The extension on the other hand, defines the collection of individuals currently satisfying the intension.

Whilst object-oriented models in methodologies typically deal with object types, in the sense described above, the word used is often 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\textsuperscript{a} nomenclature of:

\textsuperscript{a} There is arguably a fourth: rôle\textsuperscript{4}. Whilst this may eventually be recognized as important to the core, it is missing from most current methodologies.
• an object type which is at the conceptual level displaying the externally (public) viewable characteristics of a specific kind of objects. It is this interface which defines the public view of the class (q.v.)

• a class which is the object type plus its implementation. The class therefore defines the intension — the rules which determine whether individuals do or do not belong. It is a means by which to define and instantiate members of the extension (see above for definition). 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 which can be instantiated), the alternative abstract class (Figure 1) being a concept which has no instances. A class consists of an interface (which specifies the object type) and an implementation (an aggregation shown in Figure 1 by a directed arrow with a circle with 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 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 (Figure 2).

Whilst these are three distinct notions, there are many occasions when a collective word is needed for all three (a generalization or supertype in OO parlance). Examples in the literature are O/Cs\(^6\) or just objects\(^7\). In COMMA, we explicitly introduce the metalevel concept of CIT (= the class or instance or type) for this generalized term (see Figure 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 metamodelling concept equivalent to the use of the word CIT above.

OBJECT TYPEs should have VISIBLE RESPONSIBILITYs (Figure 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 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 focusses 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 (Figure 1). The class interface has a number of visible or public responsibilities and visible characteristics (Figure 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 (Figure 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 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\textsuperscript{11} that all attributes should be hidden and only accessible via a responsibility implemented by one or more methods. However, in some methodologies and OOPLs, attributes appear to be publicly visible e.g. OMT. Whilst this may be regarded as semiotically\textsuperscript{b} dangerous\textsuperscript{12,13} since it is likely to give the wrong signals, and indeed is regarded by some as non-OO 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\textsuperscript{8} — as suggested in Figure 4. Attributes are hidden properties which may reference an internal

\textsuperscript{b} Semiotics is the study of signs and symbols and the way they provide representation and communication of ideas to the reader.
OBJECT or be simple data types (as advocated in OMT, for example). Other properties are represented as LINKs to other, external OBJECTs. PARTs, which relate to the notion of aggregation, are also HIDDEN PROPERTYs which may reference internal or external OBJECTs.

Whilst 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 OO 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 which gives direct access to the HIDDEN CHARACTERISTICs.
Compatibility of proposed core with existing metamodels

In this section, 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_

OMT 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 RDD 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 proposed core metamodel except for OBJECT TYPE; whilst 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; although 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 ref. 16). However, there are no clearly described implementation details in Martin and Odell\textsuperscript{3} — these are considered in a very recent publication\textsuperscript{17}.

BON
BON’s simple metamodel focuses on features (a synonym for responsibility) and formality (through ASSERTIONs which 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 responsibilities 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 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 towards the COMMA metamodel described here.

*OBA*
OBA’s focus is on instances, rôles 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 since, 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

Whilst not yet an officially published method, the Unified Method of Booch and
Rumbaugh\textsuperscript{19} has produced publicly available documentation (ver 0.8) as of October 1995\textsuperscript{C}. The metamodel is fully documented in both graphical form and in the form of a dictionary
set of definitions.

CLASSDECL is a concept which defines a name and class structure (attributes and
operations). It is not necessarily a type. This sounds roughly equivalent to CLASS in Fig-
ure 6. However, whilst 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 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.

\textsuperscript{C} The next version, the Unified Modeling Language (UML) (Ver 0.9) is scheduled for release some time in 1996.
Whilst responsibilities 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”; whilst 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.

Whilst 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 document.

Summary

A static core metamodel has been proposed for object modelling based on an analysis of 14 existing OOAD methods, supplemented by discussions with the authors. Whilst 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 which is close to almost all methods but which 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 (published 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. Thank you.

Acknowledgements

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


Figure Legends

Figure 1 CIT as a generalization of OBJECT TYPE, CLASS and OBJECT. Only CLASS has a CLASS IMPLEMENTATION. [These are drawn using the emerging UML/OPEN notational standard (but with the basic COMMA icon) in which a double arrow indicates inheritance, a directed line (possibly labelled) 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.]

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.

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

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 which summarizes and merges the main features of earlier, more detailed diagrams.