ELSEVIER

Information and Software Technology 41 (1999) 139–156

INFORMATION AND SOFTWARE TECHNOLOGY

Comparing OPEN and UML: the two third-generation OO development approaches

B. Henderson-Sellers[a,1,*], D.G. Firesmith[b]

[a]*School of Information Technology, Swinburne University of Technology, Australia*
[b]*Storage Technology, Louisville, CO, USA*

Received 20 June 1998; received in revised form 27 November 1998; accepted 1 December 1998

Abstract

Recent efforts have been made to coalesce object-oriented methods and object-oriented modelling languages and, at the same time, to put them on a more rigorous footing by the use of metamodelling techniques. Two so-called third-generation approaches, OPEN (a full methodology) and UML (a modelling language) are described and compared here. These two approaches are compared by focusing on two main areas: (1) process and lifecycle support and, predominantly, (2) metamodel and notation. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Object technology; Methodology; Modelling language; Metamodels; Notation

1. A plethora of OO methods

It was clear for several years that there are too many object-oriented (OO) methods (a.k.a. methodologies). Whilst there are many similarities, there are a few basic differences, particularly in philosophy. Perhaps the major difference is that between methodologies based on extensions of data modelling (examples are OMT [1], Coad and Yourdon [2]) and those based on behavioural or responsibility modelling (examples are RDD [3], BON [4], MOSES [5] and OPEN [6,7]). Baudoin and Hollowell [8, p. 305] epitomize this difference in noting that in data modelling the focus is on creating a *public* data representation to be manipulated through a query language. Such an approach is the antithesis of information hiding. While public data have been eschewed in OO programming [9], current modelling languages[2] like UML [10,11] may lead to confusion in the way they focus on attributes at all lifecycle stages. The distinction between logical and physical attributes can easily be lost [12] which means that the principle of information hiding may be too easily forgotten by modellers. In contrast, responsibility-driven methods hide all attributes, focusing on the integrated or holistic behaviour and deferring discussion of attributes to much later in the lifecycle than OO analysis and design (the focus of today's OO modelling languages). These two categories also align fairly well with another classification which is often applied: that of evolutionary versus revolutionary approaches, perhaps typified in languages terms by C++ and Smalltalk respectively. More recently, the notion of a use-case driven methodology has arisen, as advocated by, for instance, Jacobson [13] and Booch [14]. Such a methodology focuses on the description of functionality as the driving factor in software development – although many question whether use cases are OO at all [15] or suggest that they ''simply become an excuse for functional decomposition'' [16,17].

Whilst competition can be creative, it can also stultify the technology transfer of these exciting new research ideas into the realm of industrial software development. Consequently, many of the world's leading methodologists have been actively pursuing a goal of method unification which will consolidate, formalize and lessen the choice by creating a small number of OO methods – so-called ''third-generation''. This convergence and unification has resulted in two methodological approaches: OPEN[3] and UML[4]. These are examined critically in this article by two members of the OPEN Consortium.

* Corresponding author. Present address: Basser Department of Computer Science, University of Sydney, Madsen Building, F09, NSW, Sydney 2006, Australia. Tel.: + 61 3 9214 8524; fax: + 61 3 9819 0823; e-mail: brian@cs.usyd.edu.au

[1] Currently on leave and at Sydney University, Sydney, NSW, Australia.

[2] A modelling language consists of a metamodel plus notation.

[3] OPEN is an acronym for OO Process, Environment and Notation.

[4] UML is an acronym for Unified Modeling Language.

PII: S0950-5849(98)00127-X

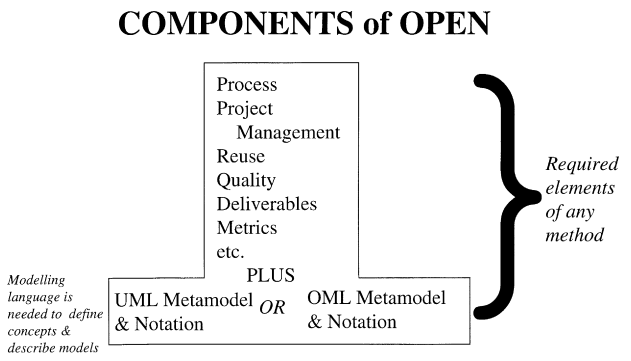

Fig. 1. A method contains many elements. Only two of these are found in UML: notation and metamodel.

2. Methods unification

The idea of creating one or more third generation, OO methodologies arose around 1993/1994. The idea was to proactively merge existing second generation methods by active involvement of the methodology authors. It became clear that the future success of OO methods needed methods convergence, collaborations between methodologists and methods standardization. UML is a modelling language (metamodel + notation), which was initially created by the merger of ideas from three approaches (OMT [1], Booch [18], OOSE [19]. In contrast, OPEN is a full, process-focussed OO methodology which was created from SOMA [20], MOSES [5] and Firesmith [21] with contributions from a total of 33 researchers and methodologists – this group of 33 forming the OPEN Consortium. OPEN also requires a modelling language. Its preferred modelling language is OPEN Modelling Language (OML [22]) although UML can also be used (Fig. 1). Both UML and OML have been given a formal underpinning by the use of metamodelling techniques [22–24]. Progress in this area has been accelerated by the thrust of the Object Management Group (OMG) towards standardizing an OO metamodel in mid-1997 and by the results of the Common Object Methodology Metamodel Architecture (COMMA) project [25] which is based in part on the work of Eckert and Golder [26].

Table 1
Comparison of UML and OPEN under the categories itemized by [27]

Category	UML	OPEN
Lifecycle process	N	Y
Full set of concepts	Y	Y
Rules and guidelines	Partly	Partly
Deliverables	Partly	Y
Notation	Y	Y
Techniques	N	Y
Metrics	N	Y
Organizational rôles	N	Y
Project management	N	Y
Reuse	N	Y

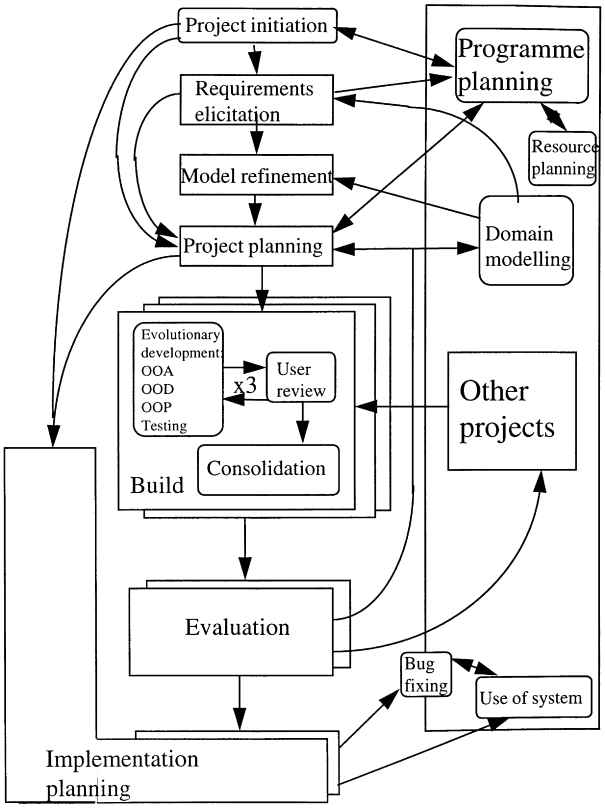

Fig. 2. An example instantiation of OPEN's contract-driven process lifecycle model. Rounded boxes are more open-ended than those with rectangular corners which can be more accurately delineated in time, perhaps using a timeboxing technique. Arrows indicate possible workflow directions (adapted from [6]). Reprinted by permission of Addison-Wesley Longman Ltd.

3. Third-generation OO software development approaches

3.1. What is a methodology?

It has been proposed that an appropriate lifecycle methodology for OO developments must contain *all* of the following components [27]:

- a full lifecycle process for both business and technological issues;
- a full set of concepts and models which are internally self-consistent;
- a collection of rules and guidelines;
- a full description of all deliverables;
- a workable notation; ideally supported by third party CASE/ drawing tools and designed for optimal usability [28];
- a set of tried and tested techniques;
- a set of appropriate metrics, standards and test strategies;
- identification of organizational rôles e.g., business analyst, programmer;
- guidelines for project management and quality assurance;
- advice on library management and reuse.

Table 2
Example OPEN tasks grouped under seven loose headings (three entries per section have been selected from the full suite of OPEN tasks described in [6])

1. Tasks which focus on modelling or system construction[a] include:
   Analyze user requirements
   Construct the object model
   Undertake usability design
2. Database focused tasks include:
   Design and implement physical database
   Identify user database requirements
   Map logical database scheme
3. Tasks which focus on user interactions and business issues include:
   Identity user requirements
   Model and re-engineer business process(es)
   Develop business object model
4. Tasks which focus on large scale architectural issues include:
   Undertake architectural design
   Optimize the design
   Construct frameworks
5. Tasks which focus on project management issues include:
   Develop software development context plans and strategies
   Develop and implement resource allocation plans
   Undertake feasibility study
6. Tasks which focus on reuse issues include:
   Optimize reuse (with reuse)
   Create new reusable components
   Manage library of reusable components
7. Tasks focusing on quality issues include:
   Evaluate quality
   Test
   Undertake in-process review
8. Tasks which focus on distribution issues include:
   Identity user requirements for DCS
   Establish distributed systems strategy
   Develop security plan

[a] These tasks deal specifically with the 'technology' of object technology and utilize those tips and techniques which are often all there to an OO ''methodology''. In OPEN, these Tasks are only a small portion of the overall approach. For further details, consult standard texts or the forthcoming OPEN manuals and user guides.

Table 1 summarizes the degree to which OPEN and UML meet these criteria. It is thus self-evident from Table 1 and Fig. 1 that UML cannot be described as being a method(ology) whereas OPEN can. However, this needs to be stressed as many training companies advertise ''UML methodology'' courses and even in the literature this misdirection abounds e.g., [12, pp. xv, 1,29,30] wherein UML is said to be the successor of many previous OOAD methods.

A direct comparison of OPEN and UML is thus inappropriate; rather we can compare (in Sections 4 and 5 as explained later) the modelling language element of OPEN known as OML [22] which has exactly the same *scope* as the UML. Finally in this article, we undertake an analysis of the availability of support (Section 6), particularly in terms of web-based documentation.

3.2. What is OPEN?

OPEN consists of a full lifecycle process-centred OO methodological framework with emphases on, inter alia, reuse, quality, organizational issues including people and project management [31] – all the elements listed earlier as being necessary. The architecture of OPEN, at the meta-level, is a set of lifecycle Activities[5] represented as objects (Fig. 2). These objects have contacts with each other so that the flow of control can pass between these objects in any order so long as the contracts are met. This gives the necessary flexibility and tailorability to the overall architecture.

On the left hand side of Fig. 2, which shows one specific instantiation of the OPEN framework, are Activities which are associated with a single project; on the right hand side, in the shaded box, are those Activities which transcend a single project and are associated more with strategic planning and implementation concerns e.g., resources across several projects; reuse strategies; delivery and maintenance aspects. OPEN includes both project and organizational software strategies. Details of these Activities is outside the scope of this article but are described elsewhere [6].

Each Activity has a number of associated Tasks which describe what is to be done. They are the smallest unit of work [32]. OPEN Tasks can be loosely grouped (Table 2). Some occur typically earlier in the lifecycle; others group around a particular domain such as distribution or database management. Detail of these Tasks is outside the scope of this article but are described in [6].

How the goals specified in these Tasks are achieved is described by a set of Techniques. Techniques are ways of doing things. They include the ways that have been tried and tested over the last decade, but also may include new techniques that are more experimental. Often choices are available[6].

A technique is designed to help in the performance of either a single or only a small number of tasks and can thus be loosely grouped by the tasks that they implement. They are thus somewhat akin to the tools of the tradesperson – a carpenter's toolbox contains many tools, some of which have superficial resemblances to one another but may have operational affinity to tools of different outward appearance. Not all tools are usable for all jobs (tasks). A full description of the OPEN toolbox of Techniques (currently well over 150 techniques have been catalogued) is a book in itself [7].

As indicated earlier, OPEN has a preferred modelling language, OML, although UML can also suffice. This arose because OML was developed in parallel with UML and indeed contributed towards one of the six submissions to the OMG in January 1997. Thus the scope of OML is that of a metamodel and a notation – as is the scope of UML (see later). The OML metamodel is built upon the COMMA

[5] Activities are the OO equivalent of phases in a waterfall model. They are high level statements of things which need to be done.

[6] For example, in order to find objects, the choice may be between, say, using noun analysis, identifying concepts and their responsibilities, using CRC cards, using textual (noun) analysis, using use cases or task scripts etc. In reality, many tasks are best accomplished by a mixture of techniques rather than just one.

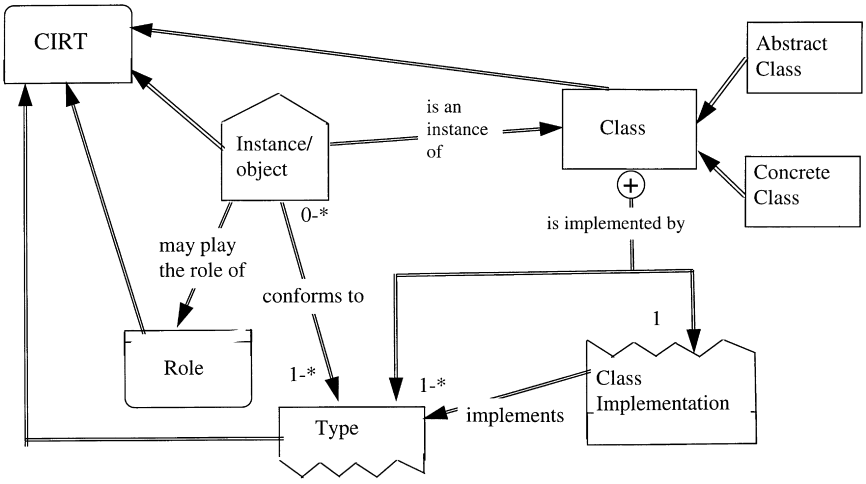

Fig. 3. OML metamodel for the inter-relationships between CIRT, class, instance/object, rôle, type and class implementation drawn using the OML/COMN notation for illustrative purposes only. The class icon is ''torn apart'' into the type (external/interface) and the class implementation (internals). An unadorned double arrow is specialization/generalization inheritance (is-a-kind-of) (modified after [34]).

metamodel [33] and has a strong focus on responsibilities and unidirectional associations together with strongly defined semantics for aggregation (see Section 4). Following the endorsement of the UML by the OMG in November 1997, OML will continue to promulgate quality modelling by offering ideas to clarify and extend this OMG standard [34].

3.3. What is UML?

UML, the Unified Modeling Language, consists of a metamodel and a notation *only*, fully described in a series of documents from Rational. Version 0.8 was the first released version (October 1995) followed by Versions 0.9 (released July 1996) and 0.91 (released October 1996) which were addenda to this document. Version 1.0, was then released in January 1997 as part of Rational's submission to the OMG. Whilst being adopted by the OMG in November 1997, it soon became clear that revisions, and corrections of both errors and ambiguities was necessary. At the time of writing of this article, the revisionary task force (RTF) is still working having released Version 1.1 and (in July 1998) Version 1.2 which is little more than an editorial clean-up of Version 1.1. Further versions (up to 1.4) are planned for the period up to April 1999. The most stable/standard version of UML to date is effectively Version 1.1 – which is the version considered here. The UML metamodel is documented in [10] and its accompanying notation in [11]. It is most compatible with a use-case driven process. UML associations are by default bidirectional and responsibilities are a tagged value, a tagged value permitting ''arbitrary information to be attached to any model element''[10, p. 55].

4. Metamodels

The metamodel in OML is derived strongly from the COMMA metamodelling work [33] together with later influences (in Version 1.1) from the evolving OMG standard [35]. OML's metamodel is characterized by and emphasizes responsibilities; undirectional associations; hidden versus visible criterion (for characteristics and responsibilities); and the inclusion of rôles (based on the work in OOram [36]). In addition, classes, instances, rôles and types (Fig. 3) are collectively referred to as CIRT (an acronym for class or instance or rôle or type). Responsibilities are either for knowing, doing or enforcing and these are implemented by characteristics (i.e., properties, operations and rules/assertions respectively). Properties are variously (sub) classified; including LINK, PART, ATTRIBUTE, ENTRY, MEMBER and EXCEPTION.

UML, in contrast, is characterized by and emphasizes use cases, relationships as reifiable classes, its obvious data modelling heritage, its use of bidirectional associations and rôles from OMT and its increasing reliance on stereotypes. Indeed, stereotypes played a major rôle in UML having been first introduced in Version 0.9 [37]. Whilst the original idea stems from Rebecca Wirfs-Brock [38], UML extends the notion to create a flexible, yet often ill-defined extension to the basic UML.

Without repeating the whole of the UML documents, we highlight several of the features of the OML metamodel [22] here for comparison with the UML metamodel [24]. We discuss Version 1.1 with some references back to Versions 0.8 and 1.0 in order to gain insight into the documented refinement process undergone by this group of methodologists as well as the current version.

4.1. Static model

4.1.1. OML

Fig. 4 shows the major elements of OML's static

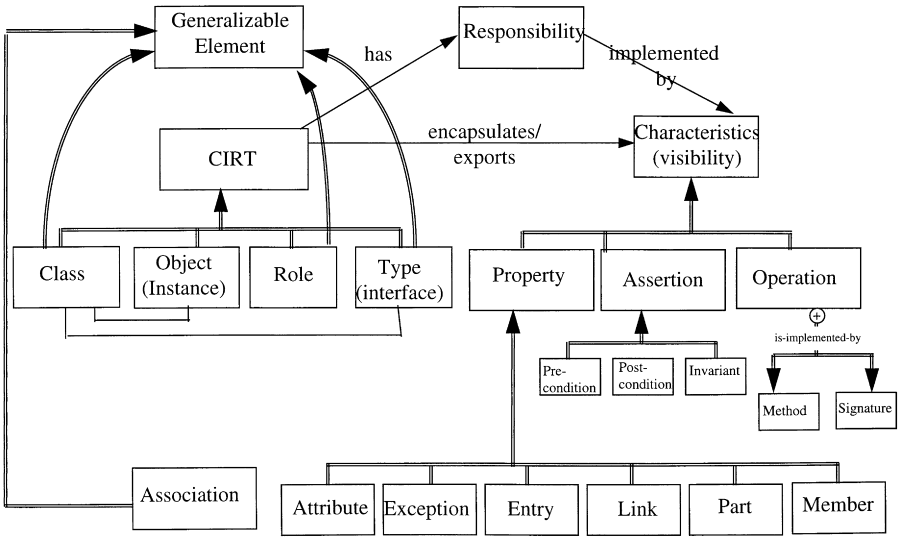

Fig. 4. Main structural elements of the OML metamodel. Additional notation is a referential relationship (association) which is a directed arrow from client to server.

architectural model. Currently, there are four main metalevel concepts:

- An *object* (a model of a thing in the real world or application) which is implemented as an *instance* of a class. Objects may also play rôles (q.v.).
- An *object type* which is at the conceptual level, displaying the externally (public) viewable aspects of the objects. It is this interface which defines the public view(s) of the class (q.v.), which may, in turn, exhibit multiple interfaces (multiple types). The type defines the intension – the rules which determine whether objects do or do not belong.
- A *class* which is the object's type(s) *plus* its implementation (Fig. 3). It is a means by which to define and instantiate members of the extension (the current members of a set). This is a ''collective'' notion only in the sense that a class can be regarded as a ''template'' or ''factory'' by which individuals (objects) can be instantiated; it is not a collection in the sense of a set of individuals which *define* the collection. This then leads to the concept of a *concrete class* (a class which can be instantiated), the alternative *abstract class* being a concept which has no instances but is used for creating other classes via inheritance (Fig. 3). Classes may, in certain contexts, be usefully viewed as objects (i.e., instances of metaclasses).
- A *rôle* is a partial object (type and implementation). A rôle is said to be played by instances of one or more classes where these classes need not be closely related by inheritance. An instance plays a particular rôle or several rôles at any given time, but *over* time, these rôles may change. Rôle modelling supports dynamic classification.

In the metamodel fragment of Fig. 3, these ideas are reflected and formalized. CLASS is-implemented-by CLASS IMPLEMENTATION and TYPE (shown as a directed double arrow with a circle with plus sign at the ''aggregate'' end of the relationship). There are two subtypes of CLASS: ABSTRACT CLASS (a class which can have no direct instances, only indirect) and CONCRETE CLASS (a class which can be instantiated). An individual OBJECT can be a *direct* instance of a concrete class or an *indirect* instance of an abstract class. The supertype (generalization) of OBJECT, CLASS, TYPE and RÔLE is the metalevel concept of CIRT. Finally an OBJECT conforms to TYPE.

The metamodel is clear about the distinctions between the definitions of object, class, type and rôle. Unlike UML, OML applies these concepts broadly [22, p. 263] (e.g., an object is an instance of a class, a package is an instance of a package class, a link is an instance of an association). It also makes clear that the total class consists of one or more visible interface(s) (TYPE) together with a hidden implementation. It is this dichotomy (external view versus internal view) which is crucial to understanding and using an OO modelling approach (see further discussion in Section 4.3). In addition, the support for multiple types is consistent with and offers good support for Java interfaces.

CIRTs in OML have responsibilities which are implemented by characteristics. These may be hidden or visible. Visible characteristics comprise logical properties, operations and assertions in the interface and are then matched into their complements of the hidden characteristics. Operations are implemented by methods and logical properties by methods or attributes. Other connexions result in the need for exceptions, entries, links, parts and members as subtypes of the metaclass PROPERTY. In addition, there are three kinds of ASSERTION (Fig. 4).

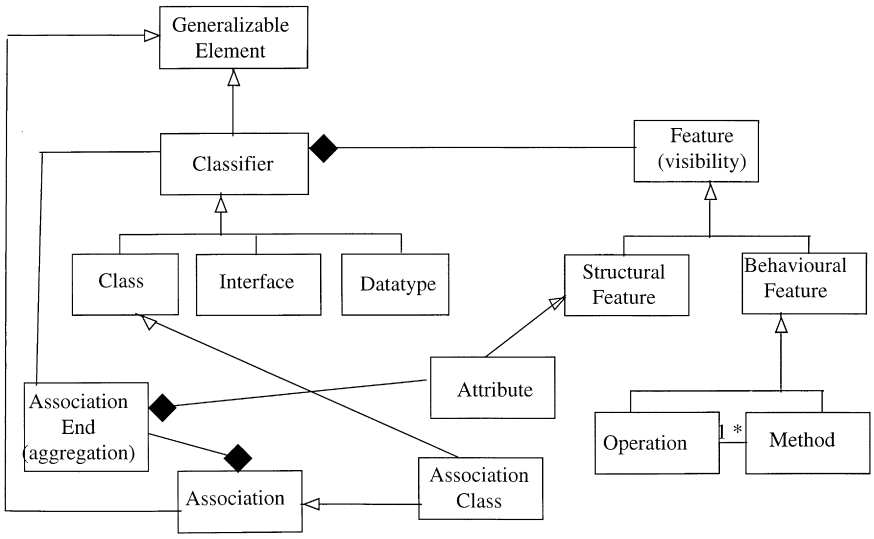

Fig. 5. Main structural elements of the UML metamodel. The notation is UML in which the black diamond indicates a composite aggregation, the white arrowhead and is-a-kind-of inheritance (after [34]).

4.1.2. UML

Fig. 5 shows the equivalent static architectural model for UML in which there are three major metalevel concepts (collectively called Classifier which is an element which declares a collection of features with a name unique within the namespace enclosing the said classifier [10, p. 22]:

- A *class* describes a set of objects sharing a collection of features thus describing both its state and behaviour. A class may also realize zero or more interfaces.
- An *interface* is ''a declaration of a collection of operations that may be used for defining a service offered by an instance'' [10, p. 24]. In other words, it represents operations but no attributes, associations or methods.
- A *datatype* is a type whose values have no identity but are simply pure values. All operations of a datatype are pure functions (they can access but not change values).

In UML, an *object* is an instance of a class, but is not defined in the core model package. An object in UML has only attributes and no operations.

Rôles are only shown in the form of ClassifierRoles on collaboration diagrams. A ClassifierRole is described as ''a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration'' [10, p. 82].

While Class and Classifier are full metalevel concepts (metaclasses), there are other relevant stereotypes: Type and Implementation Class (Fig. 6) where the Implementation Class is said to realize the Type. One area of current concern with this model (which is likely to be changed in

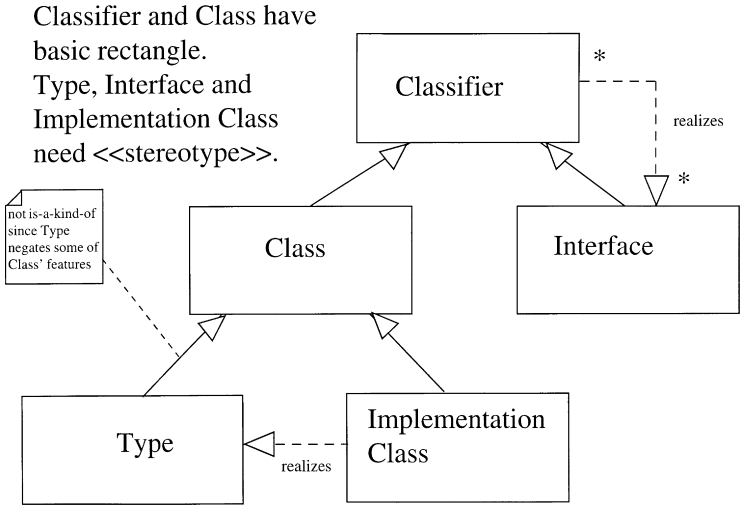

Fig. 6. Relationships between UML metaclasses of Classifier, Class and Interface and the two stereotypes of Class: Type and Implementation Class.

future versions of UML) is that the nature of a stereotype is that of specialization inheritance. Thus, in Fig. 6, we should be able to say that ''a type is a special kind of class''. This is, however, not true as a class can have methods but a type cannot. Thus the type metaclass (shown as such in Fig. 6) subtracts from its ''parent'' — recognized many years ago by Brachman [39] as a bad modelling strategy.

Classifiers are composed of (black diamond notation) features which have an associated visibility (Fig. 5). In UML, features can only be structural or behavioural — there is no equivalent of the assertion subtype of Fig. 4. The behavioural model is identical with operations being implemented by methods. However, on the structural branch, the only subtype is the Attribute. No specific metatypes exist in UML for parts, links etc. as in Fig. 4.

An additional metamodel fragment exists in UML but not in OML. An Association Class (Fig. 5) is declared to inherit multiply from Association and Class. In addition, there is a metaclass Association End in UML not found in OML. This is used to carry cardinality information and, importantly, metaattributes which are used to specify composite and shared aggregation (black and white diamonds in the UML notation).

4.2. Responsibilities

4.2.1. OML

A visible responsibility is any high-level purpose, obligation or required capability of a CIRT, typically provided by a cohesive set of one or more characteristics (i.e., visible operations, properties, rules – see later). 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.

In the COMMA and OML metamodels (see [33] for full details), OBJECT TYPEs have VISIBLE RESPONSIBILITYs. These may be for DOING, ENFORCING and KNOWING. These are implemented by VISIBLE CHARACTERISTICs – OPERATIONs, ASSERTIONs and (logical) PROPERTYs respectively. VISIBLE PROPERTYs are restricted to PARTs and EXCEPTIONs. OPERATIONS are composed of the operation's INTERFACE plus its IMPLEMENTATION.

In parallel, but viewed from *inside* the class, there are HIDDEN RESPONSIBILITYs (for doing, enforcing and knowing) implemented by HIDDEN CHARACTERISTICS. These characteristics comprise HIDDEN OPERATIONS (OPERATION INTERFACE plus the associated METHODs), HIDDEN RULEs, and HIDDEN (logical) PROPERTYs (EXCEPTIONs, LINKs, PARTs, MEMBERs, ATTRIBUTEs and ENTRYs). 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. The main differences between visible and hidden characteristics, other than names, is the inclusion of LINKs, ENTRIES, MEMBERs and ATTRIBUTEs which, in pure OO fashion, *must* be hidden within the CLASS IMPLEMENTATION. (OPEN does not support visibility of ATTRIBUTEs in contrast to OMT). It may also be useful to divide operations into procedures and functions (or commands and queries). Attributes are hidden properties which reference an internal OBJECT. Other properties are represented as LINKs to other, external OBJECTs. PARTs, which relate to the notion of aggregation, and MEMBERs, which relate to the notion of membership, are also HIDDEN PROPERTYs which may reference internal or external OBJECTs. In a similar vein are ENTRYs which refer to the elements stored in a container.

4.2.2. UML

A Responsibility in UML was originally defined as ''an obligation of an entity'' which ''may include information to maintain or behavior to perform'' – there was no recognition of a responsibility to enforce. It may be ''fulfilled by structure (attributes and associations), procedure (operations and transitions), or delegated to other entities''. In the diagrams of the metamodel specification, the Responsibility class only appeared in Fig. 2 of Version 0.8 describing Logical Elements and in this was connected (by association) to the Logical Element class (see the comment mentioned earlier). However, in Version 1.0 a Type was modelled as an aggregate (composition) of Responsibility, now defined by: ''A responsibility is a contract by or an obligation of the type to which it is attached''. Unfortunately, contract and responsibility are *not* synonyms (e.g., [40]); neither is there any stated relationship to the services offered by the class (operations, attributes) as there should be. In Version 1.1, the term ''responsibility'' does not appear in the index or in the main semantics document describing the metamodel and is merely defined in Appendix A2 as in Version 1.0 (as a Tagged Value – now on Classifier) and in the Glossary as ''A contract or obligation of a type or class''. In other words, metalevel support for the concept of responsibility is weak or non-existent. Indeed, modelling responsibilities using UML requires the manual introduction of two Dependency relationships. ''Specifically, you can have a dependency from the responsibility to the class AND you can, if your process demands it, have a dependency from the responsibility to individual elements such as attributes and operations (see p. 43 of the UML 1.1 semantics)'' [41].

4.3. Encapsulation and cross-boundary connectivity

4.3.1. OML

The object type is the external view. Object type is also known as class interface, although strictly we would talk about interfaces of many other modelling elements other than just class. The class itself is then composed of this class interface plus the class implementation (Fig. 3). Multiple views in terms of a class supporting multiple types also ensure that a high degree of encapsulation occurs. The class interface has a number of visible or public responsibilities

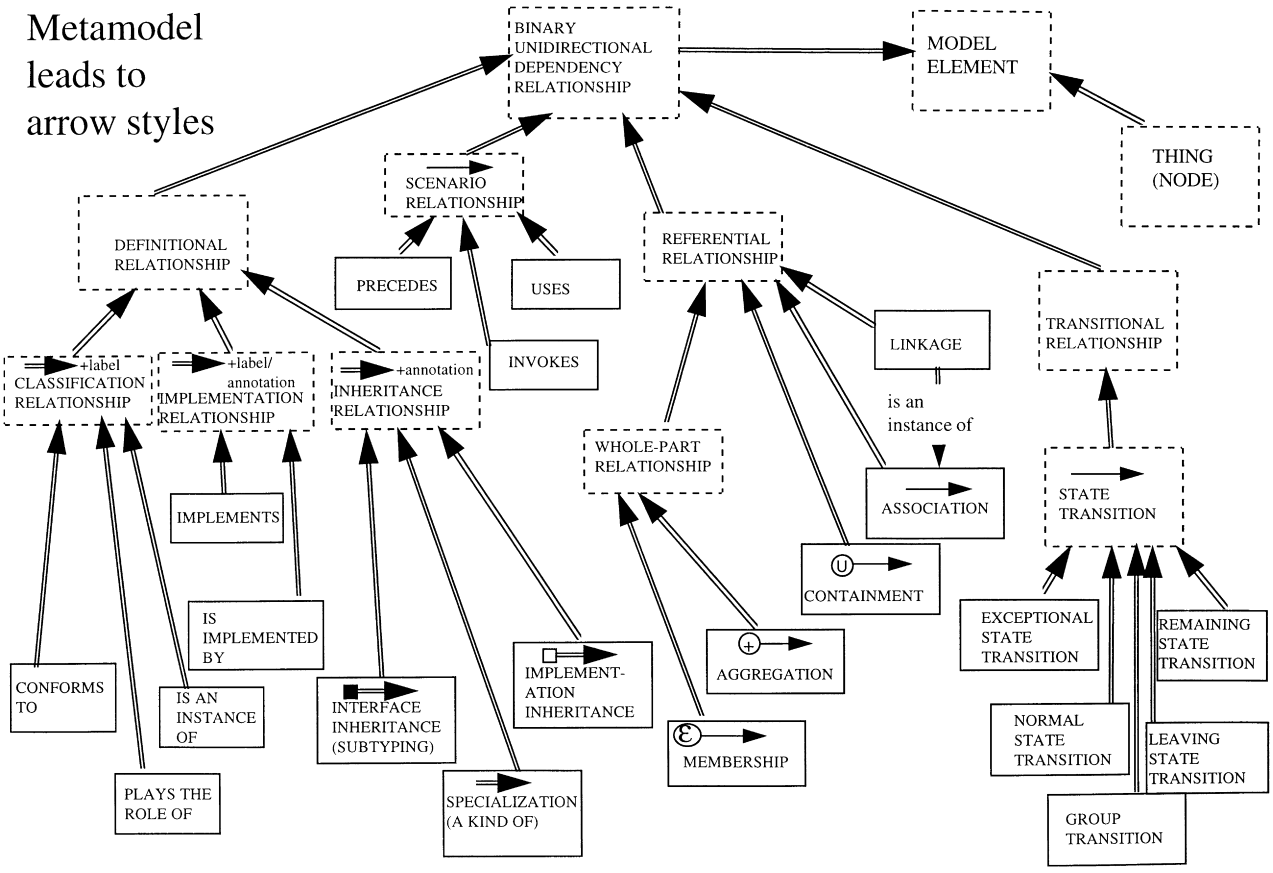

Fig. 7. The relationship metamodel hierarchy for OML. The arrow style is dictated by the position in this hierarchy. All definitional relationships are indicated by a double arrow, all referential relationships by a single arrow. Three styles of inheritance are also represented: is-a-kind-of (generalization – the default), subtyping (■) and implementation inheritance (□) (revised from [60]).

and visible characteristics. 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. In other words, the CLASS IMPLEMENTATION is fully encapsulated. OML not only encapsulates at the CIRT level, but also at the package level – a mechanism to apply encapsulation at the multiclass level i.e., at a higher level of abstraction (see e.g., Figs. 4.5 and 4.6 of [22] in which visibility is show in by having class icons straddle the boundary of the cluster icon).

The external/internal dichotomy is critical to object technology. Responsibilities and characteristics provide the conduit between the external view and the internal view (i.e., they have both hidden and visible parts). It has long been known [42] that analysts and designers view objects in terms of their interfaces (the external view) while programmers care predominantly about the internal implementation. This can sometimes lead to tensions and confusion, particularly if the notation/method does not depict these two viewpoints and discriminate cleanly between them. However, 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. This is also in agreement with the recommendation that all attributes should be hidden [9] and only accessible via the methods (i.e., implementations) of public or private operations. However, in some methodologies and OOPLs, attributes appear to be publicly visible e.g., OMT. Whilst this may be regarded as semiotically[7] dangerous as 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, with practice, be read simply as shorthand for accessor functions.

4.3.2. UML

There was some notion of the external/internal dichotomy as seen in Version 1.0's specialization/realization and essence/manifestation descriptions [24]; for instance, Type/Class and Operation/Method. However, Attributes seem to be both specification and realization (although

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

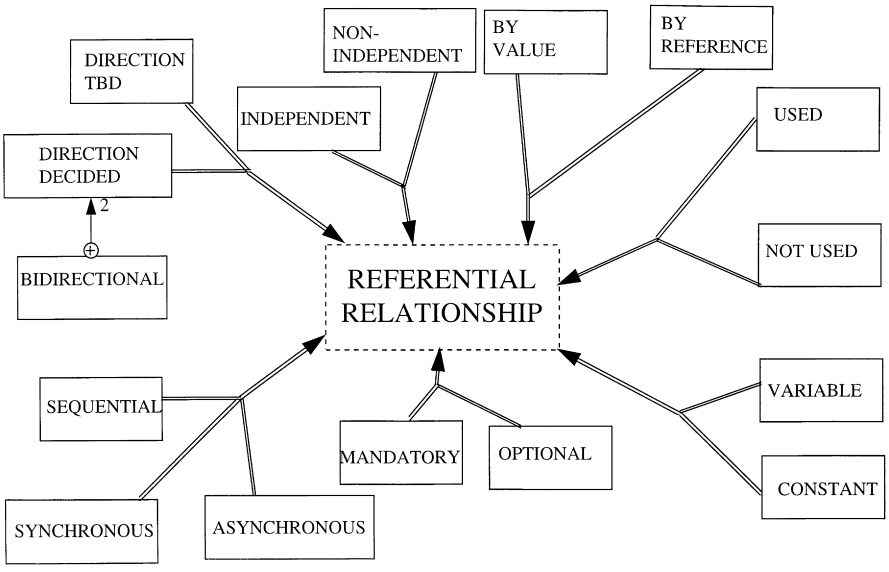

Fig. 8. Metamodel for REFERENTIAL RELATIONSHIP showing seven orthogonal partitions (after [34]).

some visibility information can be provided by use of C++-like annotations of +, #, − for public, protected and private visibility respectively) and the aggregation relationship is unclear on this point of external visibility. In Version 1.1, such external/internal considerations are less apparent. Encapsulation is only provided at the abstraction level of the class. Whilst UML supports packages (defined as ''a grouping of model elements'') at a higher level of abstraction, these are not used to represent any higher level encapsulations. Finally, UML advocates bi-directional associations which have been shown [43] to violate the principle of encapsulation.

4.4. Relationships

4.4.1. OML

The metamodel for relationships is shown in Fig. 7. The model discriminates between relationships which are (i) definitional, (ii) referential and (for the dynamic models only) (iii) transitional and (iv) scenario. A definitional relationship is any binary unidirectional dependency relationship from one modelling element to another whereby the first modelling element depends on the definition provided by the second. Definitional relationships include inheritance, classification and implementation. Inheritance is the definitional relationship from a child to a parent, whereby the child is a new definition (e.g. class, type rôle) created from the existing definition provided by the parent. Classification is the ''is-a'' relationship from an instance to its definition. An implementation relationship is any definitional relationship whereby one modelling element provides the implementation of another. There are two implementation relationships: ''implements'' and ''is implemented by''. The classification relationship has three subtypes: ''conforms to'', and ''is an instance of'' and ''plays the rôle of''. There are also three subtypes of Inheritance Relationship: specialization inheritance (is a kind of), interface (or specification) inheritance (type conformance) and implementation inheritance.

In contrast, referential relationships such as associations and aggregations merely connect together existing nodes (i.e., they provide a reference between nodes). Referential relationships include association, aggregation, membership and containment where aggregation and membership can be viewed as subtypes of a whole-part relationship and are described in the metamodel as concepts in their own right i.e., they are represented as metaclasses. These are all then subtypes of REFERENTIAL RELATIONSHIP, as indicated in Fig. 7. Referential relationships are unidirectional. When ''bidirectional'' relationships are required, a pair of semi-strong inverses is used. Further elaboration of the possible combinations of stereotypes and metatypes for referential relationships (Fig. 8) are discussed in [44]. Their provision in OML permits the sophisticated description of a wide range of modelling concerns; in UML, specific combinations are often compounded (e.g., by-value and linked lifetimes (black diamond) versus by-reference and non-exclusive ownership (white diamond) [45]).

Transitional relationships represent various forms of state transitions (exceptional, normal, leaving and remaining – see discussion of Fig. 11 later); and, finally, there are the three relationships used in scenarios, use cases and task scripts: precedes, invokes and uses.

4.4.2. UML

Relationships were originally defined (Version 0.8, p. 8) as either Generalization or Association. Generalization implies ''is-a-kind-of'' – although some of the uses of generalization in the metamodel itself seem not to be true

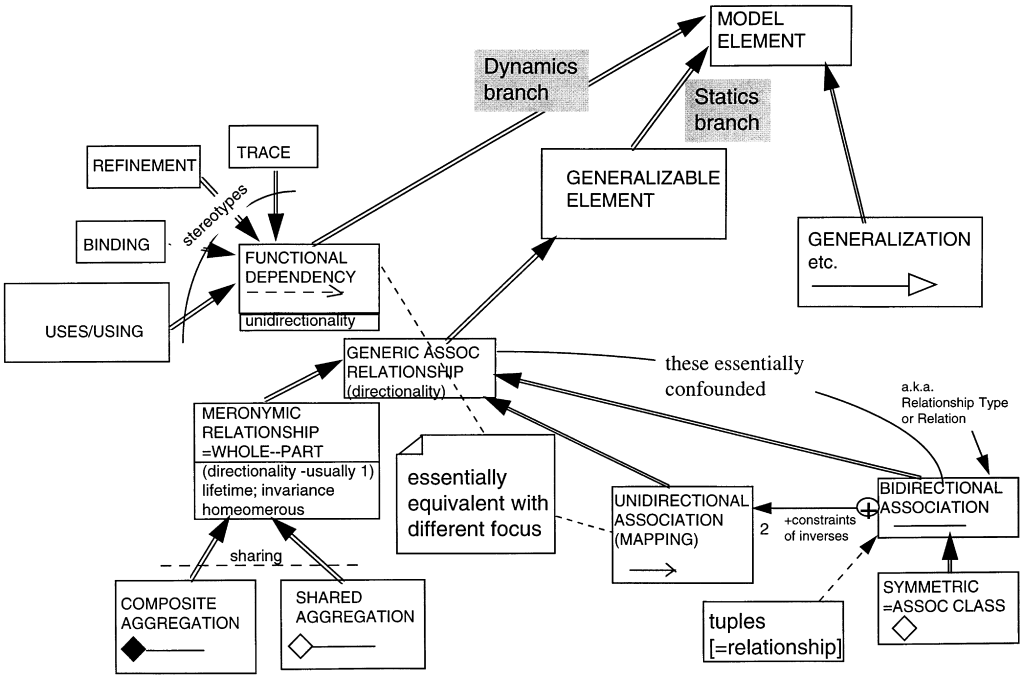

Fig. 9. UML Version 1.1 instantiation of proposed unifying metamodel fragment drawn using COMN (after [61]) ©SIGS

''is-a-kind-of'' relationships (for example, *Generalizable-Element* is a specialization of *Namespace*). UML also supports a discriminator (as does OML) and an And label. In UML, Association subsumes usage, links, association, aggregation etc. of Booch, of OMT and of OOSE. OMT rôles are retained in UML – but consciously rejected in OML (as noted earlier). Rôle in UML means ''simply an end of an association where it attaches to a class'' (Version 0.8, p. 9). There appears to be no explicit metamodel hierarchy for relationships in UML viz. nothing for direct comparison with Fig. 7.

Version 1.0 introduced a third relationship: Dependency. Dependency is ''a unidirectional using relationship from a source (or sources) to a target (or targets)'' which appears to encompass all relationships not specifically identified as association or generalization. In the Version 1.1, Dependency and Association have no common root. However, discussion on the OTUG newsgroup has been extensive, suggesting links such as ''an Association is a type of Dependency'' and ''a Dependency is a subtype of Association''; the former finally prevailing – despite there being no statement at all to this effect in the official UML metamodel documentation [10]. Further analysis in [46] supported the distinctiveness of the Dependency and Association branches of the metamodel (Fig. 9) but suggests a tenuous link defined as a dynamic versus static focus.

Thus Version 1.1 of UML supports these three major relationships: Generalization, Dependency and Association plus links in static diagrams and, on STDs, transitions (Fig. 10). Note that, although not stated explicitly in the metamodel, Generalization and Dependency are unidirectional and Association bidirectional. Aggregation is *not* a concept in the metamodel; although two types of aggregation are used in the notation [11]. These are (i) aggregation and (ii) a ''strong form of aggregation known as *composition*'' [47, p. 40; 10, pp. 148–150] – at odds with Odell [48] who clearly states that aggregation and composition are synonyms. Indeed, UML Version 1.1's aggregation has been a major discussion point in a newsgroups and in the literature (e.g., [46,49]). Shared aggregation (white diamond in the UML notation) is loosely defined with composite aggregation (black diamond) being defined in terms of (a) a whole-part relationship, (b) a part instance only belongs to one composite instance at any one time and (c) propagation semantics, in which ''some of the dynamic semantics of the whole is propagated to its parts'', are supported [10, p. 38]. This leads to lifetime dependency so that when the whole is deleted, so are the parts – this is only a small part of the suite of aggregation models described in, for example, [50,51].

4.5. State model

4.5.1. OML

Fig. 11 shows OML's metamodel for state transitions – a type of dynamic model. OML's state model is based on the work of [51–53] plus previously unpublished work by Firesmith (see [54]). The approach is based on OO concepts (i.e., state is defined in terms of an equivalence class of property values that have the same qualitative behaviour, triggers are operations and exceptions).

In OML, each OBJECT has one or more STATE MACHINE(s), defined by the CLASS, each of which is an aggregate of a number of STATEs and TRANSITIONs. A

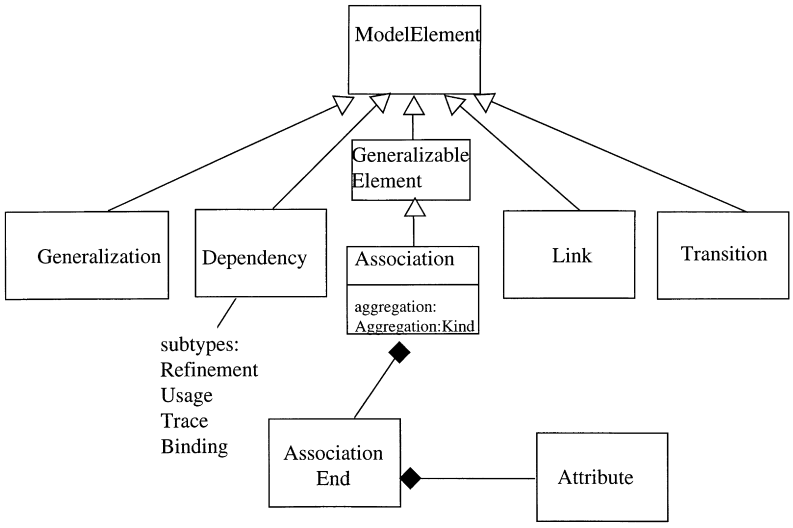

Fig. 10. Metalevel hierarchy for relationship types in UML Version 1.1 drawn using UML.

TRANSITION is a change of STATE effected at the submetatype level (as both STATE and TRANSITION are abstract). The occurrence of the transition is caused by a TRIGGER (a.k.a. EVENT). As a trigger must change the value of state properties, triggers are messages (or associated operations) and exceptions (and associated exception handlers). A transition may be controlled by a GUARD, which is generally of an enumeration type whose value determines the result of the transition.

The metaclasses STATE and TRANSITION have, in OML, many metasubclasses. Two important subclasses are STOP STATE and START STATE. Important transition subtypes are NORMAL/EXCEPTIONAL and LEAVING/REMAINING. Note that unlike OMT we do not have an ACTIVITY class associated with STATE. In OMT, this is merely something that occurs continuously throughout the state. It is therefore a manifestation of the STATE itself and does not need its own metaclass i.e., it is not an important concept, merely another view/interpretation of what being in a given state actually means.

Other elements of the OML dynamic metamodel are elaborated graphically in Fig. 11 and not discussed here in any further detail (see e.g., [22]).

4.5.2. UML

The UML State Model is derived from Harel's statecharts. It uses traditional state modelling based on traditional concepts (e.g., events). The metamodel is shown in Fig. 12 in which we see the STATE MACHINE again consisting of STATEs and TRANSITIONS. Events are triggers and should be interpreted as either messages or exceptions. An Event occurs at a point in time and causes a state change. In addition, a state is modelled as an aggregation of transitions which seems hard to defend.

In UML, there is a notation (H inside a circle) to represent the fact that an object can remember its previous state. Thus, when it transitions back to the superstate it enters the correct substate. This is called a history state and is one of the kinds of available Pseudostates, themselves a submetatype of State Vertex [10, pp. 98,101]. In addition to history, UML also permits the sending of events between communicating state machines.

4.6. Interaction modelling

Interaction diagrams show interactions (message passing and exception raising) and may be either collaboration or sequence diagrams (often called fence diagrams) and are fairly standard in OO notations. Collaboration diagrams have a graph structure similar to semantic nets and are typically used in OML to provide summary information, whereas sequence diagrams use the standard fence notation and are used to show sequencing.

4.6.1. OML

OML supports both collaboration diagrams and sequence diagrams. Major differences between COMN and UML interaction diagrams are the ability in COMN to handle exceptions and the availability of logic boxes (Fig. 13) in OML Version 1.1 [35] to handle branching, looping and interleaving owing to concurrency, thereby greatly decreasing the number of diagrams that need to be developed and maintained. OML also supports class-level collaboration diagrams documenting interactions between class internals (operations and attributes).

Sequence diagrams in OML come in whitebox and blackbox varieties: the former documents interactions between objects and classes involved in a single mechanism or use case path, the latter describes the interaction between externals and the application viewed as a blackbox.

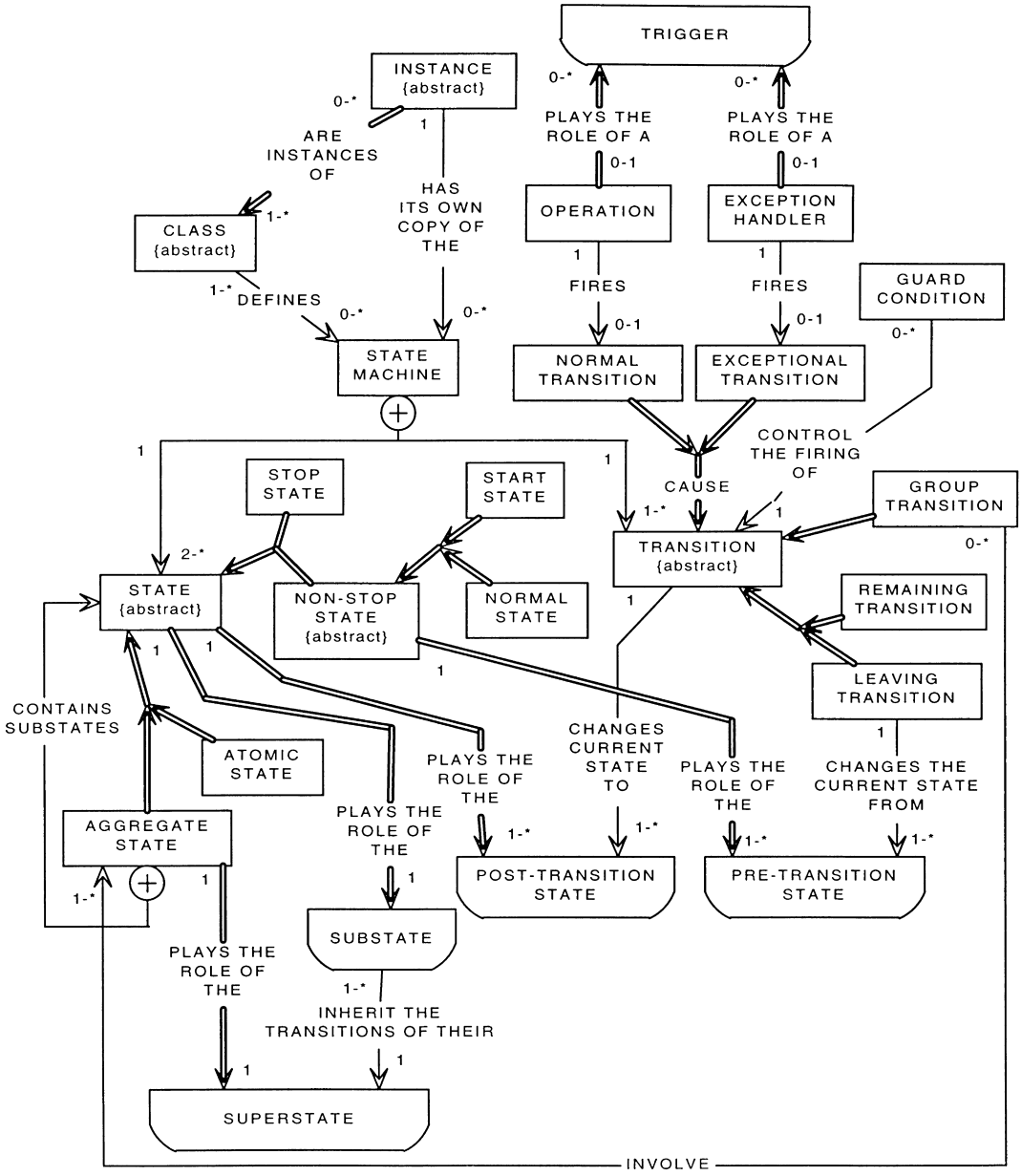

Fig. 11. OPEN's dynamic metamodel (main elements) (after [22]). ©CUP.

4.6.2. UML

UML also has both collaboration diagrams and sequence diagrams (again both subtypes of interaction diagram). Unlike OML, UML's collaboration diagrams have the same scope as the UML sequence diagrams (a use case path) and several CASE tools can automatically generate one from the other. UML concentrates on messages and is very weak on exception throwing and handling.

4.7. Scenarios

Another kind of dynamic model is used to represent scenarios/use cases/task scripts in both OML and UML.

4.7.1. OML

A scenario class model (and its associated diagram) consists of the application, its externals, scenario classes and the relationships between them.

In OML, a scenario class diagram may depict either use cases, task scripts or mechanisms (which are themselves all stereotypes of scenario class). There are three relationships: PRECEDES, INVOKES and USES which are all conceptualized in the metamodel [22, p. 154].

4.7.2. UML

The use case model (and its associated diagram) consists of the use cases and the objects as actors in the model. There is also a link through to events.

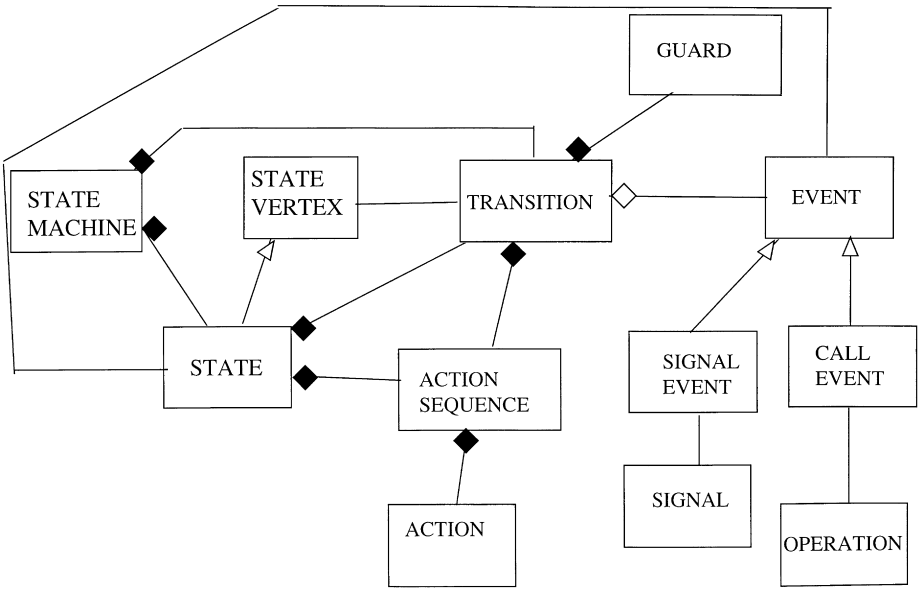

Fig. 12. UML's state (meta)model (main elements) (Version 1.1). New notation is the UML white diamond which indicates a shared aggregation.

The use case metamodel in UML Versions 0.8 and 0.9 was fairly simple and straightforward. A USE CASE consisted of several SNAPSHOTs/SCENARIOs[8] and links with OBJECTs via a ''Participation'' association – this in contrast to the more normal definition [55] that ''a scenario is an instance of a use case''.

In UML Version 1.0, there was no metamodel at all for use cases. The concept (metatype) use case appears only twice in the Semantics Document: once to define it as a subtype of metatype Type (p. 24) and once to relate it to the Diagram metatype (p. 93).

In UML Version 1.1, the use case is objectified, being composed of attributes and operations, with its own interface and connections (unspecified in the UML documentation) to Association End (Fig. 14) – a metamodel almost identical to that for Class [10, p. 35]. Two relationships of «uses» and «extends» are shown as stereotyped Generalization relationships – a modelling approach criticized earlier [56].

5. Notation

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 interact (HCI) issues in mind. Some efforts have been made along those lines (e.g. [28]). Many well-known OO notations, however, use arbitrary symbols requiring rote memorization, are sometimes counter-intuitive, are obsolete, and are often the result of historical accident – not a good HCI practice. Formal usability studies on notation, particularly in the context of OML and UML notations, are proceeding in parallel with this study.

5.1. OPEN's recommended notation: COMN

Semiotics is the study of signs and symbols. Those semiotic ideas are fully integrated into the OML notation [22] known as COMN (Common Object Modelling Notation), as are more recent studies in interface design and notational design. COMN has no hereditary biases from an earlier, data-modelling history. COMN has been designed from the bottom up, with intuition and usability in mind, by a small team of methodologists who have over the last decade

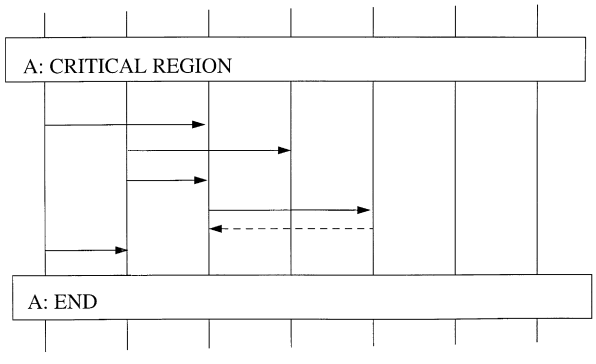

Fig. 13. Example of logic box-used here to delineate a critical region for a concurrent system. Vertical lines are timelines (time increases vertically downwards) for individual objects. The rectangular logic box spans those object timelines to which it applies. Solid arrows indicate messages; dotted arrows exceptions (modified from [35]).

[8] Snapshots and scenarios are both defined as ''A value of the system state, manifested as a network of objects''. Whereas the snapshot is static, the scenario is dynamic and represents a summation of all the snapshots over time.

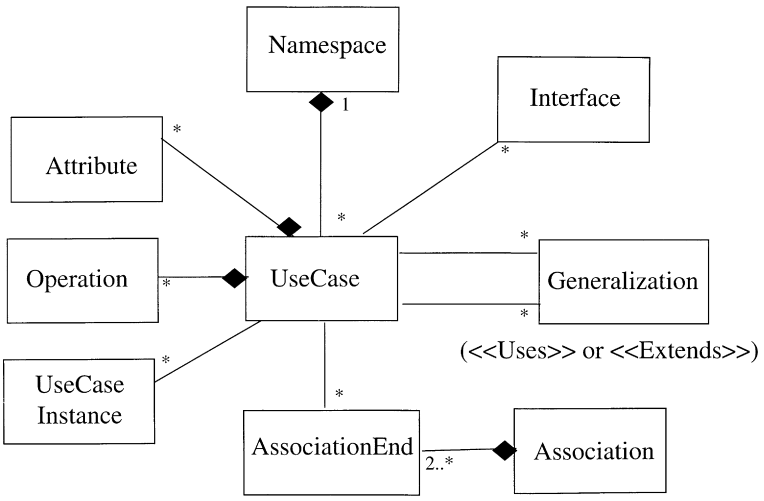

Fig. 14. Use case metamodel from UML Version 1.1 [10]. Stereotypes are shown in guillemets (« ») ©OMG, 1997.

worked on these issues. It has a responsibility, not data, focus.

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, e.g. Chinese and English (for non-native speakers) can present almost insurmountable problems. So it is with an OOAD notation. COMN is designed to be intuitive and easy to learn. It emphasizes a high level of abstraction, logical modelling and a pure OO approach.

Here we analyse only the basic elements needed and, indeed, those which will be found to make up over 90% of all applications. In a nutshell, we need to have symbols for:

- Instance versus class versus type versus rôle versus implementation (Fig. 3). All icons have optional drop down boxes (Fig. 3) for information relevant to the particular phase of the lifecycle. Drop-down boxes (Fig. 15) may contain information on characteristics, responsibilities, requirements or stereotypes, for instance. These different kinds of information are known as traits.
- Basic relationships of association, aggregation, containment and inheritance (generalization/specialization, specification/interface and implementation inheritance) (Fig. 16) – these are clearly differentiated (Fig. 7) – together with instantiation and implementation relationships.
- A state-transition model (dynamics of individual objects and classes).
- An interaction model (dynamics of interactions).
- A scenario class model (or an extension thereof).

Different models (semantic, interaction, state, scenario class) provide different views of aspects of a single overall model and are thus tightly interconnected.

This section primarily uses COMN Light – the essential subset of the complete COMN notation (see Appendix B of [6]). It aims for minimality in semantics while acknowledging that the notation is likely to evolve, especially with niche extensions such as hard real-time an distribution. This will particularly result as more ‘‘satellite’’ methods are merged into the mainstream of OPEN.

It should also be noted that, in the light of the OMG OAD Facility, endorsed by the OMG in November 1997, OPEN also permits, but, for obvious reasons, does not advocate, the use of UML’s notation (see later).

5.2. UML’s notation

The basic class/type (they are not differentiated notationally except by a stereotype) icon is a rectangle with three parts: name, attributes and operations. There is no easy national support for assertions, exceptions or rôles as in OOram. Whilst the icon for an object in Version 0.8 was a

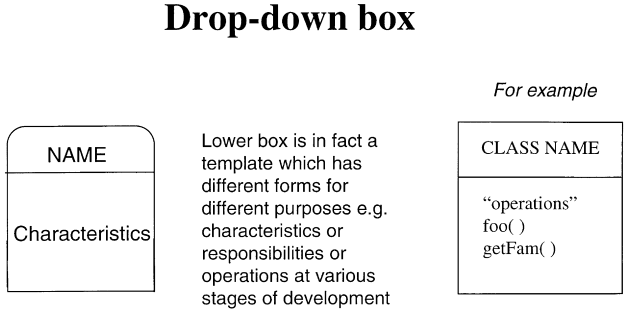

Fig. 15. Notation for OML’s drop down box in which information pertinent to the lifecycle stage is displayed.

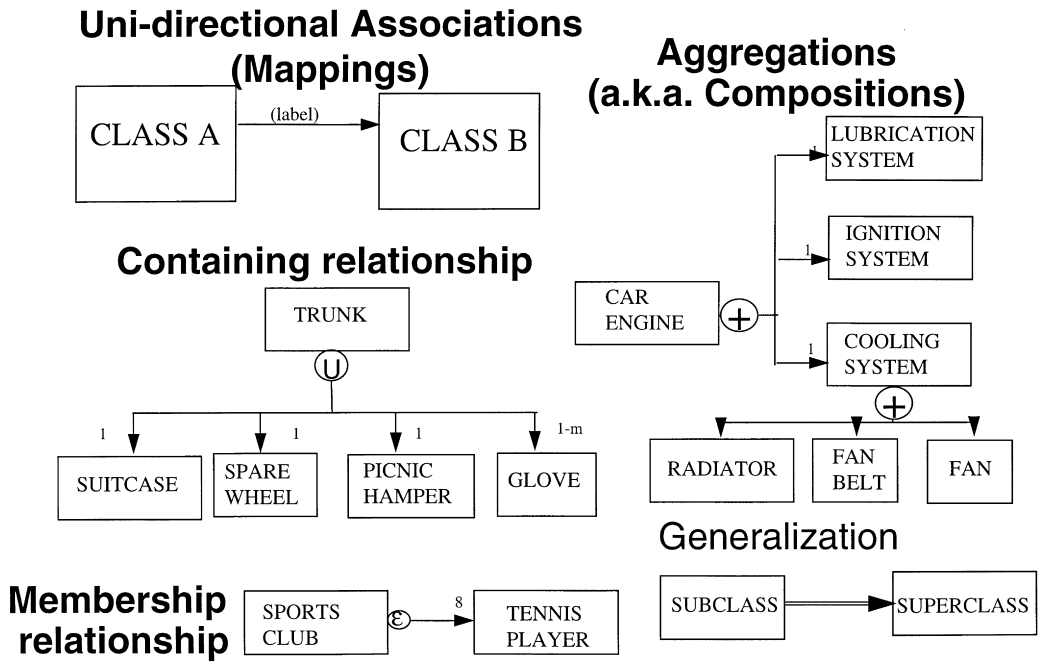

Fig. 16. Various forms of relationship in the OML. Here we see uni-directional associations, aggregations, containing, membership and generalization relationship (modified from [60]).

hexagon, in Version 0.91 this was changed to a rectangle also. The differentiator between class and object is that for objects the name is underlined (Fig. 17). Interfaces are shown as ''lollipops'' – inheritance (types) between interfaces is not supported directly as in COMN.

The UML notation for association is a line (an arrowhead is optional and discouraged) and for generalization a line with a white (changed from black in version 0.9) headed arrow. Two-way associations (the default) are not arrowed; TBD (to-be-decided) associations cannot therefore be supported as they are in COMN. Ternary associations are supported with a large diamond as in OMT.

The relationship for aggregation is the OMT (non-arrowed) line with a diamond at the aggregate end, drawn from the part of the aggregate (Fig. 18) – giving the impression of an arrowhead pointing at the aggregate (reverse visibility). This certainly makes sense if designing a relational part table with a foreign key pointing back to the whole table; but this is the reverse logic to the client-server/contracting/responsibility model of object technology.

Annotations of +, #, − are used for public, protected and private attributes and operations (as noted earlier). These annotations are arbitrary and must therefore be learned by rote. Associations can have attributes themselves and classes can have qualifiers shown by boxes abutting the class/object icon. ''Parallel inheritance'' (multiple partitioning) uses a discriminator but this has to be written alongside *all* arrows rather than on a yoke as in OML.

In addition to these elements of the static architectural models, UML supports a dynamic model, by a typical STD-extension, and a use case model (as noted in Section 4). Large scale structures use the package as a combination of the older UML-supported category (now renamed package) and subsystem with an icon of a ''tab card''.

6. Support available

The OPEN website is at http://www.csse.swin.edu.au/cotar/OPEN/OPEN.html with mirrors in USA and Eurasia. Files for process, metamodel and notation are available – readable in html format and variously downloadable in post-script or Word (RTF) formats.

Books describing OPEN and OML have been published

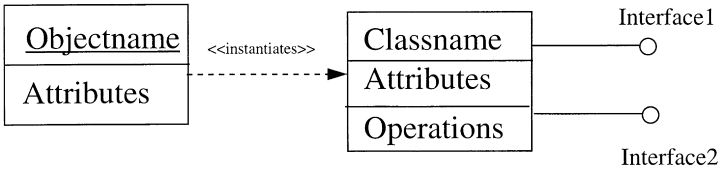

Fig. 17. UML Notation for Object, Class and Interface (derived from [11]).

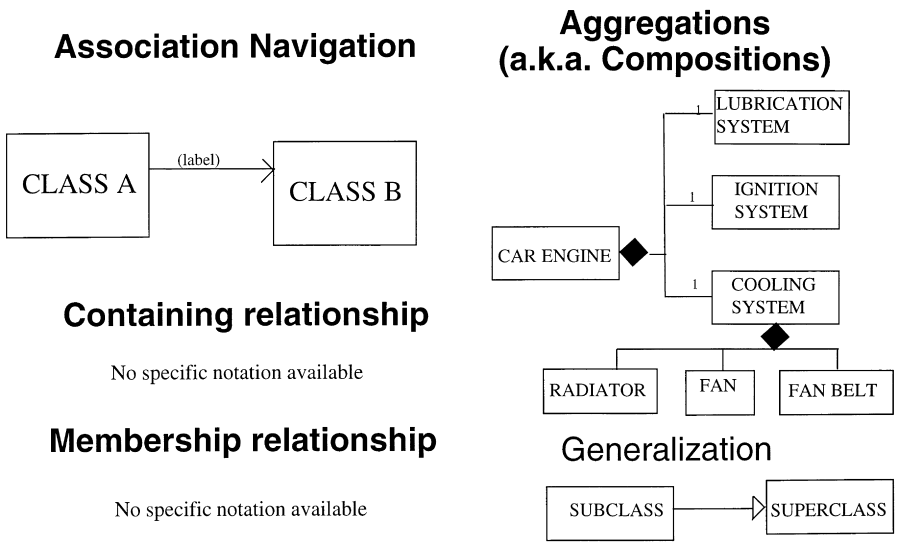

Fig. 18. Various forms of relationship in the UML. Here we see bi-directional associations, aggregations and the inheritance relationship.

[6,7,22,57] and texts utilizing the process and notation in real industrial settings as well as those with a more pedagogic bias are in press.

The UML website is at http://www.rational.com. Files for the metamodel and notation are available – typically in Acrobat format.

There are many UML books, written by third parties, published and three by the original developers scheduled for publication 1998/1999.

UML is currently more visible in the commercial marketplace and, whilst CASE tool support from various tool companies is rapidly increasing for both UML and OPEN/OML, UML support is typically being finalized ahead of OML support. The US-based UML has stronger marketing support whereas OPEN appears to have wider support in the number of researches and methodologists worldwide who are directly involved.

Education and training are well supplied for both UML and OPEN. There are companies around the world skilled in both approaches with numerous companies (development and consulting) and universities around the world offering training in both OPEN and UML.

7. Summary and conclusions

The rapid uptake of OO approaches to software development have seen the creation of a large number of OO methodologies. Metamodelling has been used to establish formal underpinnings to such methodologies, accelerated by the impetus provided by the OMG in their Request for Proposals (RFP) which had a deadline of 17 January 1997. Two of these proposals include full or partial descriptions of emerging, the so-called third generation approaches: UML and OPEN. Both have an underpinning metamodel and an associated notation; but only OPEN offers lifecycle process support (Fig. 1). These two approaches were described and contrasted here, primarily in terms of the modelling languages of UML and OML. Only OPEN provides process and lifecycle support, whereas there is a choice to be made for the metamodel and notation. UML has more of a traditional data modelling and use-case driven flavour, whereas OML (the metamodel and notation underpinning OPEN) has more of a responsibility-driven flavour and strong compatibility with OO databases, i.e. it supports a purer OO modelling approach. UML tends to be evolutionary and C++-oriented whereas OML is more revolutionary and language-independent supporting equally Java, Smalltalk, Eiffel, C++ etc. (Table 3).

It should be stressed that for supporting commercial software development, a notation is inadequate alone and a full method(ology) with process support, business concerns, quality and testing etc. is vital [58]. UML contains no process at all and thus offers no realistic support for full lifecycle development. It therefore must be complemented by a process and care is needed to ensure that they are completely compatible. In other words, in choosing a process, it is probably wiser to choose its accompanying notation and metamodel as well [59] as a more ‘‘holistic’’ package. [It should be noted that we are focusing solely on public domain choices – if a proprietary method is required, the choice is different and we understand that a proprietary (not public domain) process is currently (1998) offered by Rational.]

The comparison has also mentioned some of the areas where the published UML metamodel and notation still appear to be deficient (e.g. lack of adequate support for responsibilities, weakly defined aggregation model, notation not based on semiotics, no discrimination between various types of inheritance, implicit use of implementation inheritance in metamodel) and it is to be hoped that studies such as these will improve the quality of the OO

Table 3
High level characteristics of OPEN and UML

OPEN	UML
Revolutionary	Evolutionary
Responsibility-driven	Data-driven, use-case driven
Intuitive	Arbitrary (in parts)
Pure OO	Hybrid OO
Smalltalk, Eiffel and Java-oriented	C++-oriented
Strong OO database compatibility integrative, full process	Strong relational database compatibility

metamodels and will contribute to the OMG's ongoing efforts to create a more standard environment for OO development.

Acknowledgements

We wish to thank Rob Allen for his very constructive comments on an earlier draft of this manuscript. This is Contribution no 97/17 of the Centre for Object Technology Applications Research (COTAR).

References

[1] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object Oriented Modelling and Design, Prentice- Hall, Englewood Cliffs, NJ, 1991 p. 500.
[2] P. Coad, E. Yourdon, Object-Oriented Analysis, 2, Prentice-Hall, Englewood Cliffs, NJ, 1991 p. 233.
[3] R. Wirfs-Brock, B. Wilkerson, L. Wiener, Designing object-Oriented Software, Prentice- Hall, Englewood Cliffs, NJ, 1990 p. 368.
[4] K. Waldén, J.-M. Nerson, Seamless Object-Oriented Architecture, Prentice-Hall, Englewood Cliffs, NJ, 1995 p. 301.
[5] B. Henderson-Sellers, J.M. Edwards, BOOKTWO of Object-Oriented Knowledge: The Working Object, Prentice-Hall, Sydney, 1994 p. 594.
[6] I. Graham, B. Henderson-Sellers, H. Younessi, The OPEN Process Specification, Addison-Wesley, UK, 1997 p. 314.
[7] B. Henderson-Sellers, A.J.H. Simons, H. Younessi, The OPEN Toolbox of Techniques, Addison-Wesley, UK, 1998.
[8] C. Baudoin, G. Hollowell, Realizing the Object-Oriented Lifecycle, Prentice-Hall, Englewood Cliffs, NJ, 1996 p. 508.
[9] A. Wirfs-Brock, B. Wilkerson, Variables limit reusability, J. Obj.-Oriented Programming 2 (1) (1989) 34–40.
[10] OMG, 1997, UML Semantics, Version 1.1, 15 September 1997, OMG document ad/97-08-04.
[11] OMG, 1997, UML Notation, Version 1.1, 15 September 1997, OMG document ad/97-08-05.
[12] M. Fowler, K. Scott, UML Distilled. Applying the Standard Object Modeling Language, Addison-Wesely, Reading, MA, 1997 p. 179.
[13] I. Jacobson, Objectory is the unified process, Object Magazine 8 (2) (1998) 67–69.
[14] S. Zamir, Taking UML from innovation to usage (Interview with Grady Booch), Component Strategies 1 (2) (1998) 15–20.
[15] D.G. Firesmith, Use cases: the pros and cons, Report on Object Analysis and Design 2 (2) (1995) 2–6.
[16] T. Korson, The misuse of use cases, Object Magazine 8 (3) (1998) 18–20.
[17] B. Meyer, UML: the positive spin, Amer. Programmer 10 (3) (1997) 37–41.
[18] G. Booch (Ed.), Object-Oriented Analysis and Design with Applications 2, Benjamin/Cummings, Menlo Park, CA, 1994, pp. 589.
[19] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, New York, USA, 1992 p. 524.
[20] I.M. Graham, Migrating to Object Technology, Addison-Wesley, Wokingham, UK, 1995.
[21] D.G. Firesmith, Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach, Wiley, New York, 1993 p. 575.
[22] D. Firesmith, B. Henderson-Sellers, I. Graham, OPEN Modeling Language (OML) Reference Manual, SIGS Books, New York, USA, 1997 Cambridge University Press, Cambridge, New York, 1998, p. 271.
[23] D. Monarchi, G. Booch, B. Henderson-Sellers, I. Jacobson, S. Mellor, J. Rumbaugh, R. Wirfs-Brock, Methodology standards: help or hindrance? Proc. Ninth Annual OOPSLA Conference ACM SIGPLAN 29 (10) (1994) 223–228.
[24] Rational, UML Semantics, Version 1.0, 13 January 1997, (Unpublished), available from http://www.rational.com., 1997.
[25] B. Henderson-Sellers, A. Bulthuis, Object-Oriented Metamethods, Springer, New York, 1998 p. 158.
[26] G. Eckert, P. Golder, Improving object-oriented analysis, Inf. Software Technol. 36 (2) (1994) 67–86.
[27] B. Henderson-Sellers, Who needs an OO methodology anyway?, J. Obj.-Oriented Programming 8 (6) (1995) 6–8.
[28] L.L. Constantine, B. Henderson-Sellers, Notation matters: Part 1 – framing the issues; Part 2 – applying the principles, Report on Object Analysis and Design, 2 (3) (1995) 25–29 and 2 (4) 20–23.
[29] J. Suzuki, Y. Yamamoto, Making UML models exchangeable over the Internet with XML: UXF approach, Proc. «UML»; 1998. Beyond the Notation, 3/4 July, 1998, Mulhouse, France, pp. 65–74.
[30] I. Khriss, M. Elkoutbi, R.K. Keller, Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams, Proc. «UML» 1998, Beyond the Notation, 3/4 July, Mulhouse, France, pp. 115–126.
[31] OPEN Consortium, Proposing an open standard, Object Expert 2 (1) (1996) 14–15.
[32] A. Goldberg, K.S. Rubin, Succeeding with Objects. Decision Frameworks for Project Management, Addison-Wesley, Reading, MA, 1995 p. 542.
[33] B. Henderson-Sellers, D. Firesmith, COMMA: proposed core model, J. Obj.-Oriented Prog. (ROAD) 9 (8) (1997) 48–53.
[34] B. Henderson-Sellers, OML: proposals to enhance UML, Proc. «UML» 1998, Beyond the Notation, 3/4 July, Mulhouse, France, pp. 319–329.
[35] B. Henderson-Sellers, D.G. Firesmith, Upgrading OML to Version 1.1: Part 2 – Additional concepts and notations, JOOP/ROAD 11 (5) (1998) 61–67.
[36] T. Reenskaug, P. Wold, O.A. Lehne, Working with Objects. The OOram Software Engineering Manual, Greenwood Press, Westport, CT, 1996 p. 366.
[37] I. Jacobson, G. Booch, J. Rumbaugh, Unified standard moves closer, Object Expert 1 (6) (1996) 64–65.
[38] R.J. Wirfs-Brock, Adding to your conceptual toolkit what's important about responsibility-driven design, Report on Object Analysis and Design 1 (2) (1994) 39–41.
[39] R.J. Brachman, I lied about the trees or defaults and definitions in knowledge representation, The AI Magazine 6 (3) (1985) 80–93.
[40] B. Meyer, Applying design by contract, IEEE Computer 25 (10) (1992) 40–51.
[41] E.G. Booch, Post on 24 June to otug@rational.com newsgroup entitled RE: (OTUG) OML, 1998.

[42] Z. Urlocker, Breaking technical barriers in the 1990's, J.Obj.-Oriented Programming 2(5) (1990) (1990) 78–80.
[43] I.M. Graham, J. Bischof, B. Henderson-Sellers, Associations considered a bad thing, J. Obj.-Oriented Programming 9 (9) (1997) 41–48.
[44] D.G. Firesmith, B. Henderson-Sellers, Upgrading OML to Version 1.1, Part 1. Referential relationships, JOOP/ROAD 11 (3) (1998) 48–57.
[45] T. Quatrani, Visual Modelling with Rational Rose and UML, Addison-Wesley, Reading, MA, 1998 p. 222.
[46] B. Henderson-Sellers, OPEN relationships – composition and containment, JOOP 10 (7) (1997) 51–55.
[47] Rational, UML Notation Guide, Version 1.0, 13 January 1997, (Unpublished), available from http://www.rational.com., 1997.
[48] J.J. Odell, Six different kinds of composition, J. Obj.-Oriented Prog. 5 (8) (1994) 10–15.
[49] F. Civello, Roles for composite objects in object-oriented analysis and design, Procs. OOPSLA, (1993) 376–393.
[50] M. Saksena, M.M. Larrondo-Petrie, R.B. France, M.P. Evett, Extending aggregation constructs in UML, Procs. «UML» 1998. Beyond the Notation, 3-4 July, Mulhouse, France, pp. 273–280.
[51] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Computer Program. 8 (1987) 231–274.
[52] D.W. Embley, B.D. Kurtz, S.N. Woodfield, Object-Oriented Systems Analysis. A Model-Driven Approach, Yourdon Press, Englewood Cliffs, NJ, 1992 p. 302.
[53] B. Selic, G. Gullekson, P.T. Ward, Real-Time Object-Oriented Modelling, Wiley, New York, 1995 p. 525.
[54] B. Henderson-Sellers, D.G. Firesmith, I.M. Graham, OML metamodel: relationships and state modelling, Journal of Object-Oriented Programming (ROAD section) 10 (1) (1997) 47–51.
[55] I. Jacobson, Public communication on OOPSLA 94 panel: methodology standards: help or hindrance? 1994.
[56] J. Rumbaugh, Getting started using use cases to capture requirements, J. Obj.-Oriented Programming 7 (5) (1994) 8–12.
[57] D.G. Firesmith, G. Hendley, S. Krutsch, M. Stowe, Object-Oriented Developing Using OPEN: A Complete Java Applications, Addison-Wesley, Reading, MA, 1998, p. 404 + CD.
[58] B. Henderson-Sellers, Notations are not enough, in: W. Gens (Ed.), Proc. STJA 98: Objektorientierte Sprachen, Konzepte und Systeme, 1998, pp. 84–89.
[59] B. Henderson-Sellers, R.J. Kreindler, S. Mickel, Methodology choices-adapt or adopt?, Report on Object Analysis and Design 1 (4) (1994) 26–29.
[60] B. Henderson-Sellers, D.G. Firesmith, I. Graham, The benefits of common object modelling notation, JOOP 10 (5) (1997) 28–34.
[61] B. Henderson-Sellers, OPEN relationships – associations, mappings, dependencies, and uses, JOOP 10 (9) (1998) 49–57.

Brian Henderson-Sellers *is Professor of Computer Science (Object Technology) at Swinburne University of Technology, Hawthorn, Victoria, Australia. He is currently Visiting Professor at the University of Sydney, NSW, Australia. He can be contacted at brian@cs.usyd.edu.au.*

Donald G. Firesmith *works at StorageTek as a senior member of the technical staff. He may be contacted at firesdg@sweng.stortek.com.*