ROAD983.TEX

Upgrading OML to Version 1.1

Part 2. Additional concepts and notation

B. Henderson-Sellers and D.G. Firesmith

for ROAD/JOOP September 1998: Volume 11 no 5

In this second instalment describing the new, updated version of the OPEN Modelling Language (OML), we describe clarifications and extensions to the book published describing OML Version 1.0 (ref. 1). In particular, we focus on the crucial support for responsibilities, the underpinning concepts (and notation) for classes, instances, rôles and types, extensions regarding patterns, packages, scenario types, stereotypes and interaction diagrams. We also extend the modelling language to cater for the important areas of concurrency and distribution.

RESPONSIBILITIES

Definitions

- **responsibility**: any purpose, obligation or required capability of an object, class, type or rôle (CIRT). A responsibility can be a responsibility for doing, a responsibility for knowing or a responsibility for enforcing. It is a modelling element which captures knowledge belonging to a CIRT which is at a higher level of abstraction than a single operation or property and one which has a meaning within the problem domain being modelled.
- **CIRT = class, instance, rôle or type**: a metasuperclass to represent the generic concept when the actual metaclass (CLASS, INSTANCE, RÔLE, TYPE) is not yet determined or is unimportant (e.g. the information being captured is equally relevant to class, instance, rôle or type).

OML Metamodel

Figure 1 shows a partial metamodel for responsibilities that captures the following information:

- CIRTs have responsibilities
- Classes, instances, rôles and types thus also have a cohesive set of responsibilities inherited from CIRT (although how they are implemented is likely to be different)
- Responsibilities may be (1) for doing, (2) for knowing, (3) for enforcing
- A responsibility corresponds to and is implemented by one or more characteristics
- Characteristics may be visible or hidden
- A CIRT exports visible characteristics

Characteristics are then described by the metamodel shown in figure 3.4 of ref. 1. However, it should be noted that while there are now six kinds of hidden property (PART, LINK, ENTRY, ATTRIBUTE, EXCEPTION and MEMBER), only three of these (EXCEPTION, PART and MEMBER) are permitted to be visible characteristics.

Responsibilities are implemented by characteristics — responsibilities for doing are implemented by operations; responsibilities for knowing by operations or properties; and responsibilities for enforcing by rules/assertions (for further detailed diagrams see ref. 2).

While the COMMA core model (ref. 2) supports both public and private responsibilities, public (or visible) responsibilities are the prime focus in OML Version 1.1 (those of a private nature being rarely considered on real projects). Each (public) responsibility is then implemented by one or more public characteristics — the implementation of these public characteristics may be by delegation to hidden characteristics or even to responsibilities of other classes.

COMN Notation

COMN V1.1 notation for responsibilities is unchanged. They are a kind of trait and listed in the appropriate drop-down box.

Rationale

Responsibilities are crucial in good OO design. While RESPONSIBILITY was given as a trait kind in OML Ver 1.0 (ref. 1), the metamodel was not expanded sufficiently as had been shown possible in ref. 2. The details expounded there (in the context of the COMMA project) are now resurrected to be included in the OML Ver 1.1 metamodel.

In addition, users of OML have drawn to our attention the idea that sometimes it is highly beneficial to regard a use case as a system-level responsibility — much in keeping with the responsibility-driven focus of the OPEN methodology.

METALEVEL CLASS, TYPE and INSTANCE

Definitions

No change.

OML metamodel

Figure 3.1 of ref. 1 states that a CLASS implements a TYPE, an INSTANCE conforms to a TYPE and an INSTANCE is an instance of a CLASS (Figure 2). TYPE, CLASS, RÔLE and INSTANCE can also be generalized (i.e. be part of a generalization network at the model level). We can thus introduce a metaclass into OML Ver 1.1 of GENERALIZABLE_ELEMENT from the OMG metamodel to ensure better compliance of the OML metamodel with the emerging OMG standard.

New stereotypes of CLASS are introduced in OML Version 1.1. These are:

- **rôle class**: a class representing rôles played by other classes
- **test class**: a class which exists primarily to test other classes (e.g. test cases, test sets, test drivers)

COMN Notation

Since the majority of OO methods, including UML, now use a rectangle to denote a class, COMN Ver 1.1 will adopt this "standard" to represent a class also (ref. 3). In accordance with Ver 1.0, we maintain the idea of a class consisting of one (or more) type(s) plus an implementation so that upon horizontally tearing apart the class (rectangular) icon, we still have a readily understandable symbol for type and class implementation (Figure 3).

The symbols for INSTANCE, CLASS_IMPLEMENTATION and RÔLE remain as before — in UML these are shown by stereotypes on classifiers but an option in the documentation (ref. 4) permits as an alternative the introduction of new icons, as we do here — thus ensuring that OML V1.1 meets the OMG standard.

One element of the Version 1.0 metamodel which is valuable is the relationship between CLASSes, TYPEs, CLASS_IMPLEMENTATION and RÔLEs (figure 3.1 in ref. 1). An additional concept, much used in MOSES (an OPEN progenitor) and in other modelling papers, is the CIRT which subsumes CLASS, INSTANCE, RÔLE and TYPE into a single metalevel concept. In Ver 1.0 we had no explicit symbol for this and in Ver 1.1 we reintroduce the tablet icon (Figure 4) which was used in Ver 1.0 for class (and now replaced by a rectangle — see above).

Rationale

Changes to the notation are made to bring COMN more in line with the OMG standard. At the same time, the semiotic focus of COMN leads us to take the option of icons rather than stereotypes — within the OMG specification. An object (instance) is shown by a peaked rectangle, or “house”, icon. As an additional mnemonic, our users have advised us to think of the class icon as a blueprint (which is represented on a (rectangular) sheet of paper) with its instantiation, the object, represented by a house-shaped icon (a house being the instantiation of a blueprint in real life).

Usage Guidelines and Discussion

The notions of class, instance and type (and their interconnectedness shown in Figure 2) are in fact highly generic. They can be applied when dealing with objects (the one most commonly thought of) but they can also be applied when dealing with scenarios (scenario classes, scenario types and scenario [instances]), packages (package [classes], package types and package instances) and referential relationships (associations or referential relationship classes, referential relationship types and links or referential relationship instances) — see Figure 5 and later discussion. In all cases, the type relates to the interface, the class to the interface plus the implementation (both of these at the intension level) and an individual or instance (i.e. a member of the extension of the class). In contrast, in UML, this metastructure is essentially applied only to the object domain.

RÔLES

Definitions

- A rôle is a partial object which has a cohesive set of responsibilities and therefore encapsulates a cohesive set of characteristics. Unlike types, rôles may have implementation as well as interface.

OML Metamodel

No changes.

COMN notation

No change. The inverted tablet is retained as in Figure 4.

Discussion

Rôles occur when an instance of a class takes on additional responsibilities, perhaps on a temporary basis and can be regarded as adding an extra interface. In OML, rôles are used in two ways: as a partial object (as defined above) with its own notation (Figure 4) and as a stereotype {rôle} on a class (see Figure 9 below). The former supports instances of classes not linked by inheritance fulfilling a single rôle — it is relatively static; the latter is needed for dynamic flexibility (see Usage Guidelines below). Since it is frequently the case that a class plays a single rôle and a rôle is played by a single class, many methodologists and developers have typically confused these two concepts.

Often a number of instances from the same class will play different rôles simultaneously, perhaps even in a single collaboration — for example, an instance of class PERSON plays the rôle of EMPLOYEE interacting with a second instance of PERSON playing the rôle of MANAGER. It is also possible that a single object may play two rôles simultaneously; for example MyBank object may play the rôle of PayeeBank in one collaboration and the rôle of PayerBank in a second, concurrent collaboration. Objects of classes unrelated by inheritance may play the same rôle if they conform to the type (interface) of the class. This is useful in pattern identification and description.

Usage guidelines

Rôle modelling in OML largely derives from the work of Reenskaug *et al.* (ref. 5) in the OOram methodology. Rôles are put together to form a rôle model which is, in many cases, an embryonic pattern. In Figure 6, two partial objects are represented by the rôles of Observer and Subject (the observer pattern). In this instance, the object MyGUI plays the rôle of Observer and MyModel the rôle of subject. The MyGUI and MyModel objects are interacting in accordance with the defined rôle model. However, other pairs of objects may interact in this same way and follow the same rôle model. Thus the rôle interaction (or pattern or rôle model) represented in the lower part of this figure is the basis of a reusable pattern whereas classes and instances describe specific implementations of that pattern. Thus rôle icons appear in analysis and design diagrams alongside classes and instances — in OPEN we treat rôle as a peer at the metalevel as seen in the notion of CIRT (= class or instance or rôle or type) (see above).

Whilst rôles (in OOram) are specifically focussed at the object level, since it is specific objects that play these rôles (as seen in Figure 6), a more general statement, e.g. PERSON may play the rôle of CHAUFFEUR, is often needed in analysis. OML supports the "plays the rôle of" relationship which we propose here can be used at the class level as well as the object level much as we use referential relationships between classes to indicate the existence of links at the object level. Since we use the same notation in OML (for referential relationships) at both class (referential relationship) and object (link) level, we do not propose any new notation for a "class rôle" (which could be added to the metamodel of Figure 5). Thus the statement in Figure 7 that PERSON plays the rôle of CHAUFFEUR is a high level surrogate for the statement that instances of the PERSON class have the ability/opportunity to act as a chauffeur object. Some instances of PERSON class will take this opportunity, others will not.

Since the rôle is really a partial object which is a cohesive set of responsibilities, one specific design/implementation is as a type (e.g. a Java interface). In Figure 8, the class PERSON has not only a CHAUFFEUR interface (Rôle 1) but also a SPOUSE interface (Rôle 2). The disadvantage of this is that at the code level, the PERSON class needs the interface code to represent *all* of its foreseen rôles. In other words, the structure of Figure 8 is static —no rôles can be added without recoding the class.

A second approach which is much more dynamic is to use referential relationships (Figure 9). Here we use a rôle stereotype such that when a CHAUFFEUR instance needs to access those parts of it not defined locally, it delegates to a PERSON object, which in this case is acting in the rôle of model. Thus if a CHAUFFEUR now becomes a robot, the referential relationship from the PERSON instance to the ROBOT instance can be effected easily.

What is *not* an appropriate model for rôles is the use of the generalization relationship (Figure 10). This is an error endemic in the literature. Since an object can only be a member of any one class or subclass during its lifetime (in todays OOPLs), there is no easy/clean way that a CHAUFFEUR object, once created, can relinquish this rôle and change, say, to a SPOUSE object. The relationship does not describe any fundamental difference between PERSONs and CHAUFFEURs since the notion of chauffeur is ephemeral.

PATTERNS

Definitions

- **pattern**: a set of collaborating rôles which experience has shown to solve a common problem within a given context.

OML Metamodel

Pattern is represented in the metamodel as a stereotype of package [instance] that contains rôles instead of classes/instances.

COMN notation

A dashed rectangle with rounded corners represents a pattern — since a pattern is a kind of package, we use the package notation. If need be, a note may be attached to add clarity (since rounded rectangles with dashed lines also represent packages — this is done purposefully).

PACKAGES

Definitions

- **package**: a cohesive set of collaborators. Packages are a mechanism for controlling complexity by offering a unit of modularity larger than a class.

Packages exist as package instances, package classes and package types. Their definitions remain the same as in OML V1.0 (ref. 1) but in which the word cluster is replaced by the word package. Thus, the definition above is refined, at the different levels, as

- **package = package class**: contains [object] classes, types and packages
- **package type** contains visible [object] classes, types and packages
- **package instance** contains objects, rôles and package instances

OML Metamodel

Figure 5 shows how the meta-elements for various package elements are part of a broader, coherent scheme in which the class/instance/type triad is reflected in the metamodel for packages and classes themselves (see Figure 2), but also for scenarios and referential relationships (see below).

Rationale

Package is the term used to replace "cluster" in Version 1.0 in accordance with emerging industry standard nomenclature (e.g. Java and UML). Whilst UML Ver 1.1 uses the term subsystem as well as package and see the subsystem as a stereotype of package, OML prefers a more straightforward terminology and reverts to the ideas of UML Ver 1.0 of "logical" package and "physical" package which are stereotypes of PACKAGE.

SCENARIO TYPES

Definitions

- **scenario**: any specific, contiguous set of interactions that is not implemented as a single operation within an instance or class, an instance of a scenario class.
- **scenario type**: any declaration of the externally visible aspects of a scenario class, typically including name, parameters (if any), preconditions and postconditions.
- **path**: any contiguous set of referential relationships that are traversed by a scenario

OML Metamodel

In the OML V1.0 metamodel (figure 3.24 of ref. 1), a SCENARIO is an instance of a SCENARIO_CLASS. In Ver 1.1, we include for consistency and coherence the notion of SCENARIO_TYPE (Figure 5). As in V1.10, scenario classes may be either task scripts, use cases or mechanisms.

Notation

In Figure 5 we saw that a SCENARIO_TYPE is implemented by a SCENARIO_CLASS, an instance of which is a SCENARIO. Here a ScenarioType can be thought of as exhibiting responsibilities including assertions and the implementation part of the ScenarioClass being documented using a whitebox sequence diagram showing the inter-object interactions. Consequently, the OML V1.1 notation for all scenarios parallels that for classes: the ScenarioClass is represented by the basic icon (ellipse – as before) which can now be "torn apart" to give a upper ellipse with jagged bottom to represent a ScenarioType supplemented by its implementation which is its visual complement (Figure 11). We can then quite reasonably talk about and document the fact that a particular ScenarioClass implements (double arrow between different metatypes) a ScenarioType (Figure 12). [The complementary concepts of ScenarioType and ScenarioClass are not supported in UML — they are confounded in the UseCase metaclass.] Since SCENARIO_CLASS is generalizable (Figure 5), we can also have inheritance relationships between SCENARIO_CLASSes and could thus foresee the use of (now supported) abstract scenario classes (with a dotted outline or the stereotype label of {abstract}).

Whilst interactions (invokes and precedes) between scenario classes remain the same, the use of a dotted arrow for the "uses" relationship to show the user/system interactions seems unnecessary, especially since uses, invokes and precedes are all types of referential relationship which use solid lines. Since it involves an external icon, it is unambiguous and thus it will be clear if we use the simpler solid arrow in Ver 1.1.

Usage Guidelines

Figure 13 gives additional information to clarify the use in OML of these in the context of scenario classes, scenario instances and scenario type. Different kinds of scenarios have different scope. In particular, a task script has a very strong business focus as compared to use cases which are software focussed. Indeed, a task script makes no assumption about the existence of a (software) application — for a use case it is essential that an application exists since the purpose of a use case is to document a functionally cohesive set of interactions between a blackbox application and one or more externals (e.g. a human user). A mechanism is a scenario class of collaborating internals (i.e. a mid-sized pattern of collaborating rôles (ref. 1, p57) in that it can be used to document fragments and thus can also be regarded as relevant to collaborations. Finally, as noted earlier, it seems reasonable to consider a use case as a system-level responsibility.

Finally, the definition of path in ref. 1 required clarification (see above). A path is used to divide a potentially huge number of scenarios into a much smaller number of equivalent “classes” for testing.

ASSOCIATION METACLASS

OML Metamodel change

REFERENTIAL RELATIONSHIP CLASSes (i.e. ASSOCIATIONs) and REFERENTIAL RELATIONSHIP TYPES (i.e. ASSOCIATION TYPEs) are generalizable in OML Version 1.1 because they are submetatypes of CLASS and TYPE respectively (Figure 5), which are themselves submetatypes of CIRT which is a submetatype of GENERALIZABLE_ELEMENT. As can be seen from Figure 5, LINKs (referential relationship instances) are also generalizable.

STEREOTYPES and TRAITS

Revised Usage Guidelines

In OML version 1.0 (ref. 1, p63), the list of acceptable trait kinds included stereotypes. This is now recognized as incorrect. In OML V1.1, we use the definition of stereotype from UML which permits the user to identify partitions of new, user-defined meta-subtypes. Such a user-created metasubtype need not appear in the OML metamodel, although OML does define a few standard partitions (see ref. 3). However, in the user's locally-defined metamodel, they could be shown as subtypes of the parent metaclass. In that sense, they are similar to, for instance, the OML-defined subtypes of, for instance, scenario class (see earlier section). An OML stereotype can be applied to any (and indeed all) trait kinds. A stereotype is *not* a trait kind in itself, it is applied to a trait kind in order to create a partition. An example is shown in Figure 14.

INTERACTION DIAGRAMS

OML Metamodel

There are (in both OML and UML) two types of interaction diagram: (i) collaboration diagrams which show object and package-level CIRTs and their connections laid out much along the lines of a semantic net and (ii) sequence diagrams (a.k.a. fence diagrams), which show interactions between timelines. Using an orthogonal classification, we could identify a diagram which shows potential interactions (collaboration diagram with no message sequencing shown) and those which show actual message sequences (two diagrams: collaboration diagrams annotated with message numbers and sequence diagrams). Whilst there has been much discussion regarding how best to metamodel the above situation, OML Ver

1.1 endorses and clarifies the suggestions in both UML Ver 1.1 and OML Ver 1.0 in terms of the metamodel of Figure 15.

COMN Notation (Sequence diagrams)

On sequence diagrams (whitebox and blackbox), OML Ver 1.0 introduced the notation of logic boxes which are rectangular boxes drawn over the timelines and used to document (i) branching, (ii) loops, (iii) concurrency and (iv) temporal requirements. In Ver 1.1 we augment their use by one further application: to delineate critical regions (Figure 16) which are regions in which the sequence of messages has to run from start to finish without being interrupted. This is one way to guarantee thread safety.

CONCURRENCY

COMN Notation

The basic concurrency notation in OML Ver 1.0 is a pair of parallel, slanted lines. The annotation was for objects or classes (not packages, scenario classes or associations). Additions to this notation are now added in Version 1.1 — summarized in Figure 17. The parallel lines are retained for indicating the parallel processing of a separate thread. If the class is, in addition, threadsafe, then a "safety" symbol (a shield) is superimposed to give the icon for concurrent, threadsafe. If the shield icon is not present, then it is not threadsafe. In addition, the concurrency annotation is now also available at the operation level (Figure 17).

DISTRIBUTED SYSTEMS

Definition

- **distribution unit**: a stereotyped package instance, the objects of which are distributed (i.e. deployed or moved) as a group.

COMN Notation

For distributed systems, deployment diagrams become useful. However, as described in both UML and OML Ver 1.0, these only address the static allocation of software (distribution units, classes, packages and objects) to processors (hardware allocation). For distributed designs and implementations, an intermediate step is required, called virtual node in ref. 6. This is a logical grouping of objects and/or classes and may or may not correlate to the deployment on to specific hardware. We thus, in OML Ver 1.1, introduce the idea of a "distribution unit" which is essentially a logical package of objects, classes and possibly other packages which has some logical coherence and is the unit for consideration for actual deployment. We thus use the package icon (dashed, rounded rectangle). Thus, in fact, we have broadened the packaging ideas of Version 1.0 into the idea that Distribution Units can interact and the particular form of distribution unit is identified by the label which can have values of package, class or object (Figure 18). These distribution units are then allocated to hardware using the existing deployment diagrams now labelled as distribution unit (Figure 19). In Version 1.1, we use a migration diagram to handle these issues of dynamic distribution.

Consequently, we can think of a deployment diagram in a distributed application as being essentially analogous to a semantic net, being architectural in nature, with a migration diagram analogous to a collaboration diagram, showing not messages but the movements of software distribution units rather than messages/exceptions along referential relationships.

CONCLUSIONS

We have described in this and last month's column the extensions to OML Version 1.0 as described in ref. 1 in creating the current Version 1.1, now in accord with, but also enhancing, the emerging OMG standard. We have focussed here on the critical need to model responsibilities, as well as clarifying the metamodel and notation for classes, rôles and other elements. A new and broader notation for concurrency and distribution has also been added in this OPEN Modelling Language update to Version 1.1.

Acknowledgements

This is contribution number 98/3 of the Centre for Object Technology Applications and Research.

References

1. Firesmith, D.G., Henderson-Sellers, B. and Graham, I., 1997, *OPEN Modeling Language (OML) Reference Manual*, SIGS Books, New York, USA, 271pp

2. Henderson-Sellers, B. and Firesmith, D.G., 1997, COMMA: proposed core model, *J. Obj.-Oriented Prog.,* **9(8)**, 48–53

3. Firesmith, D.G. and Henderson-Sellers, B., 1998, Upgrading OML to Version 1.1: Part 1. Referential relationships, *JOOP/ROAD,* **11(3)**, June 1998

4. OMG, 1997, UML Notation. Version 1.1, 15 September 1997, OMG document ad/97–08–05

5. Reenskaug, T., Wold, P. and Lehne, O.A., 1996, *Working with Objects. The OOram Software Engineering Manual*, Manning, Greenwich, CT, USA, 366pp

6. Rasmussen, G., Henderson-Sellers, B. and Low, G.C., 1996, An object-oriented analysis and design notation for distributed systems, *J. Obj.-Oriented Prog.,* **9(6)**, 14–27

7. Henderson-Sellers, B., Simons, A.J.H. and Younessi, H., 1998, *OPEN's Toolbox of Techniques*, Addison-Wesley, London, UK

Figure legends

Figure 1 OML Version 1.1 metamodel fragment for RESPONSIBILITY

Figure 2 Metamodel fragment showing that both CLASS and TYPE are GENERALIZABLE ELEMENTs, that CLASS implements TYPE, that INSTANCE conforms to TYPE and that INSTANCE is an instance of CLASS

Figure 3 OML Ver 1.1 icons for Type and Class Implementation which together visually coalesce to form the Class icon (rectangle)

Figure 4 Metamodel and notation for Object, Class, Type, Rôle and Implementation. The Class icon is "torn apart" into the Type (external/interface) and the Class Implementation (internals). All icons can have a drop down box in which information pertinent to the lifecycle stage is displayed (after ref. 7)

Figure 5 Metamodel fragment (OML V1.1) showing that the notion of class-level can be applied not only to the object model but also to referential relationships, scenarios and packages. Similarly, the instance level is appropriate to the same four domains (objects, links, package instances and scenarios). Type, on the other hand, does not apply to rôles

Figure 6 OOram rôle model (after ref. 7)

Figure 7 Class level representation of the statement that objects of class PERSON may play the rôle of chauffeur

Figure 8 A Person may play the rôle of both Chauffeur and Spouse by supporting two interfaces (after ref. 7)