UML data models from an ORM perspective: Part 6

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This paper first appeared in the December 1998 issue of the Journal of Conceptual Modeling*, published by InConcept.*

This paper is the sixth in *a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 discussed historical background, design criteria for modeling languages, object reference and single-valued attributes. Part 2 covered multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints, as well as instantiation of associations. Part 4 contrasted ORM nesting with UML association classes, ORM co-referencing with UML qualified associations, and ORM exclusion constraints with UML or-constraints. Part 5 discussed ORM subset and equality constraints, and how to specify them in UML. Part 6 examines subtyping in ORM and in UML.*

Subtyping: semantics and motivation

Both UML and ORM support *subtyping*, using substitutability ("is-a") semantics, where each instance of a subtype is also an instance of its supertype(s). For example, declaring Woman to be a subtype of Person entails that each woman is a person, and hence Woman inherits all the properties of Person. Given two object types, *A* and *B*, we say that *A* is a *subtype* of *B* if, for each state of the database, the population of *A* is included in the population of *B*. For data modeling, the only subtypes of interest are *proper* subtypes. We say that *A* is a proper subtype of *B* if and only if (i) *A* is a subtype of *B*, and (ii) there is a possible state where the population of *B* includes an instance that is not in *A*. We could have a database state in which all people are women, and another in which some people are men, but we never have a state in which a woman is not also a person. From now on, we use "subtype" as short for "proper subtype".

In both ORM and UML, *specialization* is the process of introducing subtypes, and *generalization* is the inverse procedure of introducing a supertype. Both ORM and UML allow single *inheritance* as well as multiple inheritance (where a subtype has more than one direct supertype). For example, AsianWoman may be a subtype of both AsianPerson

and Woman. In UML, “subclass” and “superclass” are synonyms of “subtype” and “supertype” respectively, and generalization may also be applied to things other than classes (e.g. interfaces, use case actors and packages). We confine our attention here to subtyping between object types (classes).

In ORM, a subtype inherits all the roles of its supertypes. In UML, a subclass inherits all the attributes, associations and operations / methods of its supertype(s). An operation implements a service and has a signature (name and formal parameters) and visibility, but may be realized in different ways. A method is an implementation of an operation, and hence includes both a signature and a body detailing an executable algorithm to perform the operation. In an inheritance graph, there may be many methods for the same operation (*polymorphism*), and scoping rules are used to determine which method is actually used for a given class. If a subclass has a method with the same signature as a method of one of its supertypes, this is used instead for that subclass (*overriding*). For example, if Rectangle and Triangle are subclasses of Shape, all three classes may have different methods for display(). Since the focus of this series of articles is on data modeling, not behavior modeling, we restrict our attention to inheritance of static properties (attributes and associations), typically ignoring operations or methods.

Subtypes are used in data modeling to do at least one of the following:

- assert typing constraints
- encourage reuse of model components
- show a classification scheme (taxonomy)

In this context, typing constraints ensure that subtype-specific roles are played only by the relevant subtype. As we will see, ORM has a stronger approach to typing constraints than UML. Both approaches use subtyping for reuse. Since a subtype inherits the properties of its supertype(s), only its specific roles need to be declared when it is introduced. Apart from reducing code duplication, the more generic supertypes are likely to find reuse in other applications. At the coding level, inheritance of operations / methods augments the reuse gained by inheritance of roles / attributes / associations. Using subtypes to show taxonomy is of little use, since taxonomy is often more efficiently captured by predicates. For example, the fact type Person is of Sex {male, female} implicitly provides the taxonomy for the subtypes MalePerson and FemalePerson.

Display of subtypes

Data modeling approaches typically depict subtyping graphically using either Euler diagrams or Directed Acyclic Graphs. *Euler diagrams* depict relationships between subtypes spatially (unlike Venn diagrams, with which they are sometimes confused). For example, Figure 1 depicts the following information: *B*, *C* and *D* are subtypes of *A*, and *E* is a subtype of both *C* and *D*. Moreover, *B overlaps* with *C* (i.e. they may have a common instance) and *C* overlaps with *D*, but *B* and *D* are *mutually exclusive* (cannot have a

common instance). For example: *A* = Person; *B* = Asian; *C* = Consultant; *D* = American; *E* = TexanConsultant.

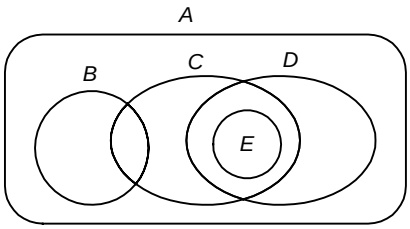

Figure 1: An Euler diagram

Euler diagrams provide intuitive displays for simple cases, and are used in some ER modeling tools (e.g. Designer/2000). However Euler diagrams are too cumbersome for complex subtype patterns often found in real applications, where an object type might have a large number of subtypes, possibly overlapping. Moreover, individual subtypes may have many specific details recorded for them, and there is simply no room to attach these details if the subtype nodes are crowded inside their supertype nodes.

For such reasons, Euler diagrams are eschewed for non-trivial subtyping. Instead *directed acyclic graphs* (DAGs) are often used, as is the case for both ORM and UML. A directed graph is simply a graph of nodes with directed connections, and "acyclic" means there are no cycles (a consequence of proper subtyping). The subtype pattern in Figure 1 is represented in DAG form in Figure 2, with (a) ORM and (b) UML versions shown. Here an arrow from one node to another shows that the first is a subtype of the second. ORM uses a solid shaft and arrowhead, while UML uses a thin arrow shaft with an open arrowhead. As an alternative notation, UML also allows separate shafts to merge into one, with one arrowhead acting for all (see Figure 3 later). Since subtypehood is transitive, indirect connections are omitted (e.g. since *E* is a subtype of *C*, and *C* is a subtype of *A*, it follows that *E* is a subtype of *A*, so there is no need to display this implied connection).

Figure 2: Previous Euler subtype diagram depicted as directed acyclic graphs in (a) ORM and (b) UML

As shown, ORM and UML both show subtypes outside, connected by arrows to their supertype(s). Although less intuitive than Euler diagrams, this is preferable since it allows us to express subtype patterns of any complexity, with enough space at each subtype node to add specific details. By default, ORM subtypes may overlap, and subtypes need not collectively exhaust their supertype. However ORM allows graphic constraints to be added to indicate that subtypes are mutually exclusive (a circled "X" connected to the relevant subtypes via dotted lines, as in Figure 2, collectively exhaustive (a circled dot) or both (a circled, crossed dot). Exhaustion constraints are also called "totality constraints". Although exclusion and totality constraints for subtypes are part of ORM, VisioModeler does not yet support them. As we will see presently, explicit depiction of such constraints is not a necessity, since other constraints in conjunction with formal subtype definitions typically imply the relevant exclusion and totality constraints.

In UML the only default for generalization appears to be "incomplete", i.e. not all subtypes have been specified. The opposite of "incomplete" is "complete"—this probably means the same as totality (collective exhaustion) in ORM, i.e. the supertype equals the union of its subtypes. However the wording in the UML standard [7] does not make this clear. In UML, constraint keywords may be added in braces besides dotted lines connecting the relevant subtypes, as shown in Figure 2. The following four keywords are predefined in UML for this purpose: "overlapping" (the subtypes overlap), "disjoint" (the subtypes are mutually exclusive), "complete" (all subtypes have been declared), and "incomplete" (some more subtypes may be introduced later). Other keywords may be added by users. When more than one arrowhead is involved, UML requires the keyword to be written beside a single dotted line that connects the relevant subtypes. I have assumed that this line may include elbows (as shown in Figure 2 for the disjoint constraint); without elbows or a similar device, some cases can't be specified.

As Figure 2 shows, ORM's depiction of inter-subtype constraints is less cluttered than UML's. ORM's approach is that exclusion and totality constraints are enforced on populations, not types. For example, an overlapping "constraint" does not mean that the populations must overlap, just that they may overlap. Hence from an ORM viewpoint, this is not really a constraint at all, so there is no need to depict it.

For any subtype graph, the topmost supertype is called the *root*, and the bottom subtypes (those with no descendants) are called *leaves*. In UML this can be made explicit by adding "{root}" or "{leaf}" below the class name. If we know the whole subtype graph is shown, there is little point in doing this; but if we were to display only part of a subtype graph, this notation makes it clear whether or not the local top and bottom nodes are also like that in the global schema. For example, from Figure 3, we know that globally Customer has no supertype, and MalePerson and FemalePerson have no subtypes. However, since Organization has not been marked as a leaf node, it may have other subtypes not shown here. Currently ORM does not include such a root/leaf notation (apart from adding a text box with this information), but it would be simple to add it. UML also allows an ellipsis "…" in place of a subclass to indicate that at least one subclass of the parent exists in the global schema, but its display has been suppressed on the diagram.

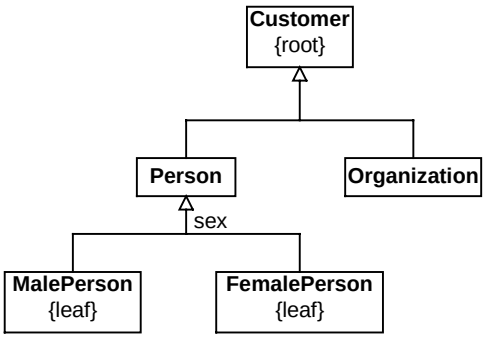

Figure 3: An incomplete UML subtype graph: Organization may have other subtypes not shown here

UML also distinguishes between *abstract* and *concrete* classes. An abstract class cannot have any direct instances, and is shown by writing its name in italics or by adding "{abstract}" below the class name. Abstract classes are realized only through their descendants. Concrete classes may be directly instantiated. This distinction seems to have little relevance at the conceptual level, and is not depicted explicitly in ORM. For code design however, the distinction is important (e.g. abstract classes provide one way of declaring interfaces, and in C++ abstract operations correspond to pure virtual operations, while leaf operations map to nonvirtual operations). For further discussion, see [3, pp. 85-8] and [1, pp. 125-6].

Subtype definitions

Like other ER notations, UML provides only weak support for defining subtypes. A *discriminator* label may be placed near a subtype arrow to indicate the basis for the classification. For example, Figure 3 includes a "sex" discriminator to specialize Person into MalePerson and FemalePerson. The UML standard [7] says that the discriminator names a "a partition of the subtypes of the superclass". In formal work, the term "partition" usually implies the division is both exclusive and exhaustive. In UML, the use of a discriminator does not imply that the subtypes are exhaustive or complete, but it does seem that they must be exclusive (e.g. [3], p. 78). If that is the case, there does not appear to be any way in UML of declaring a classification scheme for a set of overlapping subtypes. The same discriminator name may be repeated for multiple subclass arrows to show that each of these subclasses belong to the same classification scheme. This repetition can be avoided by merging the arrow shafts to end in a single arrowhead, as in Figure 3.

The UML standard states that "the discriminator must be unique among the attributes and association roles of the given superclass" but I'm unsure what this means. If it implies that an attribute or association role can't be used as a discriminator, then that would seem to be bizarre. If it doesn't imply this, then the notation is open to inconsistency. As a trivial example, consider the Patient subtyping in Figure 4, where the sex attribute is used as a discriminator. This attribute is based on the enumerated type Sexcode, which has been defined using the stereotype «enumeration», and listing its values as attributes.

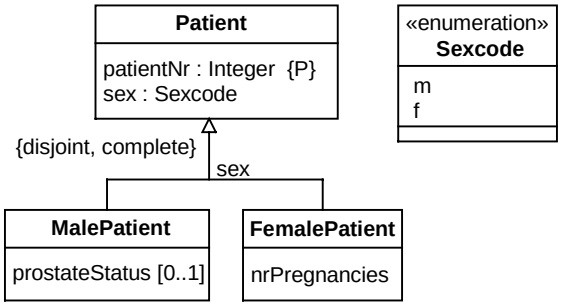

Figure 4

By itself, this model fails to ensure that instances populating these subtypes have the correct sex. For example, there is nothing to stop us populating MalePatient with some patients that have the value 'f' for their sexcode. This problem is best explained using ORM, where it's easy to display populations. Figure 5 shows an equivalent ORM schema, with prostate status being measured for a female, and pregnancies being recorded for one of the males. This kind of nonsense is allowed because the model hasn't formally related the subtypes back to their precise sex.

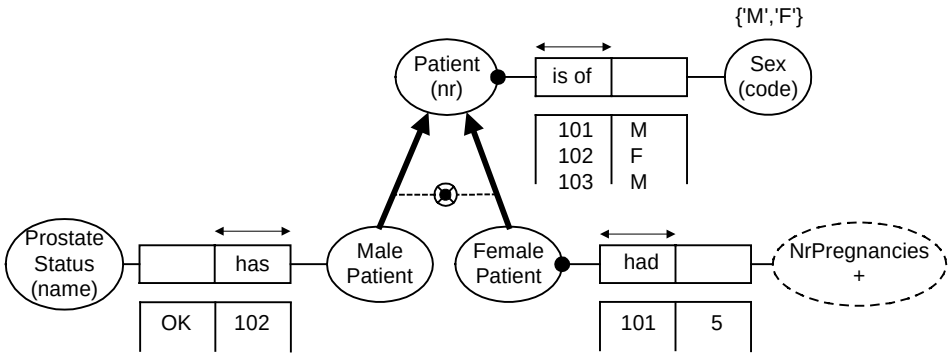

Figure 5: What is the problem here?

ORM overcomes this problem by requiring that *formal subtype definitions* be declared for all subtypes. These definitions must refer to roles played by the supertype(s). The correct schema is shown in Figure 6, together with a satisfying population. Note that the ORM partition (exclusion and totality) constraint has been removed from the diagram since it is now implied by the combination of the subtype definitions and the three constraints on the fact type Patient-is-of-Sex. Though long part of ORM, formal subtype definitions are not yet supported by VisioModeler, which allows them to be entered only as comments. However the conceptual query technology underlying ActiveQuery potentially provides one way of formally defining and mapping subtypes, and the related formal theory is mature.

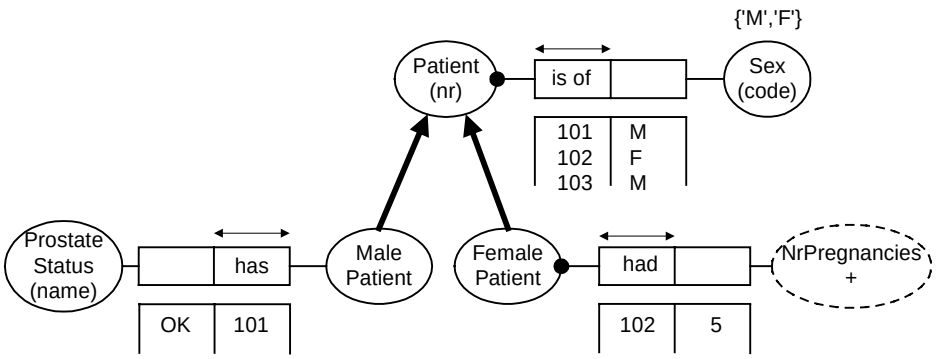

Figure 6: Formal subtype definitions are needed, and subtype partition constraints are implied

While the subtype definitions in Figure 6 are trivial, in practice more complicated subtype definitions are sometimes required. As a basic example, consider a schema with the fact types City-is-in-Country, City-has-Population, and now define LargeUScity as follows:

each LargeUScity **is a** City **that** is in Country 'US' **and** has Population > 1000000

There does not seem to be any convenient way of doing this in UML, at least not with discriminators. One could perhaps add a derived Boolean isLarge attribute, with an associated derivation rule in OCL, and then add a final subtype definition in OCL, but this would be less readable than the ORM definition above.

This article has ignored various subtyping issues such as mapping and context-dependent reference. For an ORM perspective on these and related issues see [4, 5, 6].

Later issues

Later issues will discuss other advanced graphic constraints in ORM and UML (ring, join, aggregation etc.), derivation rules and queries.

References

1. Booch, G., Rumbaugh, J. & Jacobson, I. 1999, The Unified Modeling Language User Guide, Addison-Wesley, Reading MA, USA.

2. Blaha, M. & Premerlani, W. 1998, Object-Oriented Modeling and Design for Database Applications, Prentice Hall, New Jersey.

3. Fowler, M. with Scott, K. 1997, UML Distilled, Addison-Wesley.

4. Halpin, T. 1995, Conceptual Schema and Relational Database Design, 2nd edn, Prentice Hall Australia.

5. Halpin. T.A. 1995, 'Subtyping: conceptual and logical issues', Database Newsletter, ed. R.G. Ross, Database Research Group Inc., vol. 23, no. 6, pp. 3-9, reproduced by permission on www.orm.net.

6. Halpin, T. & Proper, H. 1995, 'Subtyping and polymorphism in object-role modelling', Data & Knowledge Engineering 15, 3 (June), 251-281, reproduced by permission on www.orm.net.

7. OMG-UML v1.2, OMG UML Revision Task Force website, http://uml.systemhouse.mci.com/.