UML Data Models From An ORM Perspective: Part 2

by Dr. Terry Halpin, BSc, DipEd, BA, MLitStud, PhD
Director of Database Strategy, Visio Corporation

This paper appeared in the May 1998 issue of the Journal of Conceptual Modeling
published by Information Conceptual Modeling, Inc. and is reproduced here by permission.

This paper is the second in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 provided some historical background on both approaches, identified several design criteria for modeling languages, and discussed how object reference and single-valued attributes are modeled in both. In Part 2 we compare UML multi-valued attributes with ORM relationship types, including basic constraints on both. As part of this discussion, we also consider how these structures may be instantiated, using UML object diagrams or ORM fact tables.

Multi-valued attributes

Suppose that we are interested in recording the names of employees, as well as the sports they play (if any). In ORM, we might model this situation as shown in

Figure **1**(a). The mandatory role dot indicates that each employee has a name. The absence of mandatory role dot on the first role of the Plays fact type indicates that this role is optional (it is possible that some employee plays no sport). The lack of a mandatory role dot on the roles of EmpName and Sport does not imply that these roles are optional. If in the global schema an object type has only one fact role, this is implied to be mandatory unless the object type has been declared independent. So if EmpName and Sport have no other roles in the complete application, their roles shown here are implicitly mandatory. This is of little importance, since implied constraints are automatically enforced with no additional overhead.

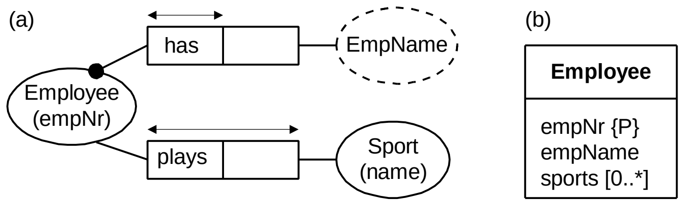

Figure 1: Plays depicted as an ORM m:n fact type (a) and a UML multi-valued attribute (b)

Since an employee may play many sports, and a sport may be played by many employees, Plays is a *many-to-many* (*m:n*) relationship type. This is shown in ORM by making the uniqueness constraint span both roles. Visually, this indicates that for each population of the fact type, only the combination of values for the two roles needs to be unique. In other words, each employee-sport pair can occur on at most one row of the associated fact table. Since it is understood that the population of any fact type is a *set* of rows (not a bag of rows), such a spanning uniqueness constraint always applies. We only show this constraint if no stronger one exists. For example, the uniqueness constraint on the empname fact type is stronger, since it spans just one role; so we don't bother adding the weaker, 2-role uniqueness constraint. Read from left to right, the empname relationship type is *many-to-one* (*n:1*), since employees have at most one name, but the same name may refer to many employees.

One way of modeling the same situation in UML is shown in Figure 1(b). Here the information about who plays what sport is modeled as the *multi-valued attribute* "sports". The "[0..*]" appended to this attribute is a *multiplicity constraint* indicating how many sports may be entered here for each employee. The "0" indicates that it is possible that no sports might be entered for some employee. Unfortunately, the UML standard uses a *null value* for this case, just like the relational model. The presence of nulls in the base UML model exposes users to implementation rather than conceptual issues, and adds considerable complexity to the semantics of queries. By restricting its base structures to elementary fact types, ORM avoids the notion of null values, enabling users to understand models and queries in terms of simple 2-valued logic. The "*" in "[0..*]" indicates there is no upper bound on the number of sports of a single employee. In other words, an employee may play many sports, and we don't care how many. If "*" is used without a lower bound, this is taken as an abbreviation for "0..*".

As mentioned in Part 1, an attribute with no explicit multiplicity constraint is assumed to be mandatory and single-valued (exactly one). This can be depicted explicitly by appending "[1]" to the relevant attribute. For example, to indicate explicitly that each employee has exactly one name, we would use "empName [1]". Although the UML standard [3] specifies that multi-valued attributes are allowed and that "[1]" is the default multiplicity of attributes, some authors of popular UML books appear to be unaware of this (e.g. [2], p. 63). Moreover, some of the principal authors of OMT (the Object Modeling Technique from which UML class diagrams are largely derivative) argue that the default is single-valued with nullability unspecified (i.e. either [1] or [0..1]); for example, see [1], p. 44.

ORM constraints are easily clarified by populating the fact types with sample populations. For example, see Figure 2. The inclusion of all the employees in the EmpName fact table, and the absence of employee 101 in the Plays fact table clearly shows that playing sport is optional. Notice also how the uniqueness constraints mark out which column or column-combination values can occur on at most one row. In the EmpName fact table, the first column values are unique, but the second column includes duplicates. In the Plays table, each column contains duplicates: only the whole rows are unique. Such populations are very useful for checking constraints with the subject matter experts. This validation-via-example feature of ORM holds for all its constraints, not just mandatory roles and uniqueness, since all its constraints are role-based, and each role corresponds to a fact table column.

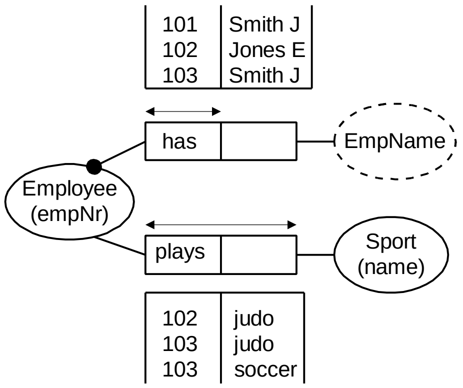

Figure 2: Fact tables with sample populations clarify the constraints

Instead of using fact tables for the purposes of instantiation, UML provides *object diagrams*. These are essentially class diagrams in which each object is shown as a separate class instance, with data values supplied for its attributes. As a simple example, the population of Figure 2 may be displayed in a UML object diagram as shown in Figure 3. For simple cases like this, object diagrams are useful. However, as we see later, they rapidly become extremely unwieldy if we wish to display multiple instances for more complex cases. In contrast, fact tables scale easily to handle large and complex cases.

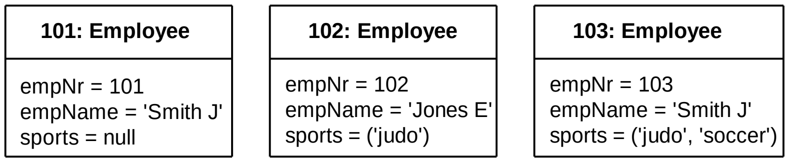

Figure 3: Object diagrams may be used in UML to show sample populations

Let's look at a couple more examples involving multi-valued attributes. At my former university, employees could apply for a permit to park on campus. The parking permit form required one to enter the license plate numbers of those cars (up to three) that one might want to park. A portable sticker was issued that could be transferred from one car to another. Over time, an employee may be issued different permits, and we want to maintain an historical record of this. Suppose that it is also a rule that an employee can be issued at most one parking permit on the same day. One way of modeling this situation in ORM is set out in Figure 4.

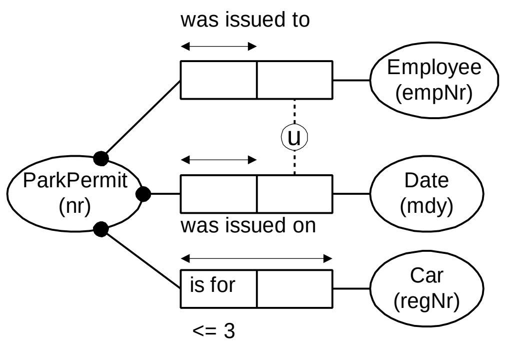

Figure 4: ORM diagram with external uniqueness constraint and frequency constraint

Here the circled "u" depicts an *external uniqueness constraint* (i.e. a uniqueness constraint that spans roles from different predicates). This captures the rule that any given employee on any given date may be issued at most one parking permit. An external uniqueness constraint is equivalent to an internal uniqueness constraint over the same roles in the conceptual join of the predicates. In this case, a join would create the compound fact type: ParkPermit was issued to Employee on Date. If this derived fact type were populated, the Employee-Date combination would be unique, and this is what the constraint means.

Notice also the "<=3" next to the first role of the fact type ParkPermit-is-for-Car. This is a *frequency constraint*, indicating that each permit in the fact column for that role appears there at most three times. In other words, each parking permit allows at most three cars to be parked on campus. In ORM, both uniqueness and frequency constraints may be applied to one or more roles, possibly from different predicates. Frequency constraints place restrictions on the number of times that instances of the role(s) may appear. The frequency might be a single number (e.g. 2), a number range (e.g. 2..5), a list of numbers (e.g. 2, 4) or a combination. The expression "<= *n*" means "at most *n*", but since it applies to entries in the role column (rather than the object type) this is equivalent to "1..*n*", since any entry for the role has already appeared once. The expression ">=*n*" means "at least *n*". A frequency constraint of 1 is simply a uniqueness constraint. However because uniqueness constraints are so common, they are given a special notation of their own.

One way of modeling the same application in UML is shown in Figure 5. In addition to the ParkPermit class, Employee and Car classes are included. The "… " shown here simply indicates that other attributes of these classes exist in the global schema (this use of "… " is not part of the UML notation). For discussion purposes, the attribute domains are displayed. In UML these domains are called "type expressions". Instead of defining a standard syntax for type expressions, UML allows them to be written in any implementation language, assuming the latter provides the relevant parser. For example, one type expression might be a C++ function pointer. To keep our analysis model at least semi-conceptual, we restrict type expressions to simple data types and classes. Data types are sets of pure values (no oids), and include primitive types (e.g. Integer, String) as well as enumeration types (including Boolean and user-defined). In our example, the attribute domains include the data types Integer and Date, as well as the classes Employee and Car.

By using classes as domains in this way, we can at least understand when an attribute corresponds to a association between entities, even it is not displayed as such.

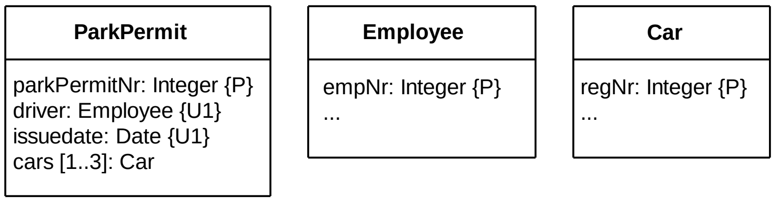

Figure 5: A UML alternative to the ORM model in Figure 4

As discussed in Part 1, constraints not included in the standard notation may be added in braces in some implementation language. In Figure 5 we use "{P}" for "primary identifier", and "{U1}" on both driver and issuedate to indicate that this combination is unique. Taken together, the two "{U1}" annotations correspond to the ORM external uniqueness constraint in Figure 4. The "[1..3]" constraint on the cars attribute indicates that each parking permit is associated with at least one and at most three cars. The "at least one" part of this corresponds to an ORM mandatory role constraint; and the "at most three" corresponds to the "<= 3" ORM frequency constraint. Recall that mandatory role constraints are separated out in ORM, mainly because they have *global* impact (each population instance of that type must play all the roles of that object type in the global schema). In contrast, other ORM constraints (e.g. frequency and uniqueness) are local, applying only to the population of the associated role(s).

As a final example of multi-valued attributes, suppose that we wish to record the nicknames and colors of country flags. Let us agree to record at most two nicknames for any given flag, and that nicknames apply to only one flag. For example, "Old Glory" and perhaps "The Star-spangled Banner" might be used as nicknames for the USA flag. Flags have at least one color. Figure 6(a) shows one way to model this in ORM. For verbalization purposes we identify each flag by its country. Since country is an entity type, the reference scheme is shown explicitly (parenthesized reference modes may abbreviate reference schemes only when the referencing type is a value type). The uniqueness constraint on the role played by Country could be annotated with a "P" for primary reference, but this is implied if Flag has no other reference schemes. The "<= 2" frequency constraint indicates that each flag has at most two nicknames, and the uniqueness constraint on the role of NickName indicates that each nickname refers to at most one flag.

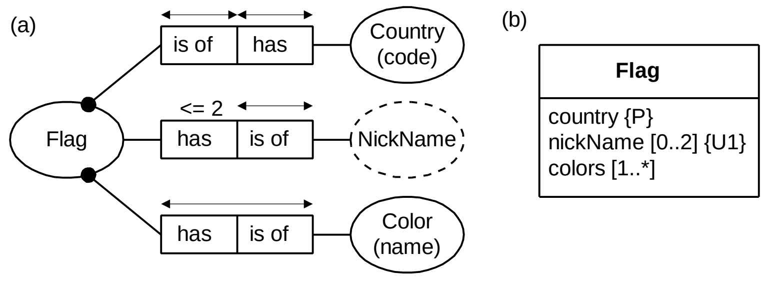

Figure 6

Figure 6(b) shows one way of modeling this in UML. The “[0..2]” indicates that each flag has at most two (from zero to two) nicknames, and we use “{U1}” to indicate that nicknames refer to at most one flag. The [“1..*] declares that a flag has one or more colors. In this case we have omitted the display of attribute domains. NickName would typically have a data type for its domain (e.g. String). If we don’t want to store any information about countries or colors, we might choose String as the domain for country and colors as well (although this is sub-conceptual, because in the real world countries and colors are not character strings). However since we might want to add information about these later, it’s better to use classes for their domains (e.g. Country and Color). If we do this, we need to define the classes as well (cf. our previous example).

As we discuss in the next issue, UML gives us the choice of modeling a feature as an attribute or an association (similar to an ORM relationship type). At least for conceptual analysis and querying, explicit associations usually have many advantages over attributes, especially multi-valued attributes. This choice helps us verbalize, visualize and populate the associations. It also enables us to express various constraints involving the “role played by the attribute” in standard notation, rather than resorting to some non-standard extension (as we did with our braced comments). This applies not only to simple uniqueness constraints (as discussed earlier) but also to other kinds of constraints (frequency, subset, exclusion etc.) over one or more roles that include the role played by the attribute’s domain (in the implicit association corresponding to the attribute). For example, if the association Flag-is-of-Country is explicitly depicted in UML, the constraint that each country has at most one flag can be captured by adding a multiplicity constraint of “0..1” on the left role of this association. Although country and color are naturally conceived as classes, nickname would normally be construed as a data type (e.g. a subtype of String). Although associations in UML may include data types (not just classes), this is somewhat awkward; so in UML, nickname might best be left as a multi-valued attribute. Of course, we could model it cleanly in ORM first.

Another reason for favoring associations over attributes is stability. As we discuss later, if ever we want to talk about a relationship, it is possible in both ORM and UML to make an object out of it, and simply attach the new details to it. If instead we modeled the feature as an attribute, we would not be able to add the new details without first changing our original schema: in effect we would need to first replace the attribute by an association. For example, consider the association Employee-plays-Sport in Figure 1 (a). If we now want to record a skill level for this play, we can simply objectify this association as Play, and attach the fact type: Play-has-SkillLevel. A similar move can be made in UML if the play feature has been modeled as an association. In Figure 1(b) however, this feature has been modeled as the sports attribute; so this attribute needs to be removed and replaced by the equivalent association before we can add the new details about skill level. The notion of objectified relationship types or association classes will be covered in a later issue.

Another problem with multi-valued attributes is that queries on them need some way of extracting the components, and hence complicate the query process for users. As a trivial example, compare queries Q1, Q2 expressed in ConQuer (the ORM query language supported by Visio's ActiveQuery tool) with their counterparts in OQL (the Object Query language proposed by ODMG):

(Q1) **List each** Color **that** is of Flag 'USA'.
(Q2) **List each** Flag **that** has Color 'red'.

(Q1a) **select** x.colors **from** x **in** Flag **where** x.country = "USA"
(Q2a) **select** x.country **from** x **in** Flag **where** "red" **in** x.colors

Although this example is trivial, the use of multi-valued attributes in more complex structures can make it harder for users to express their requirements.

If we choose to avoid multi-valued attributes in our conceptual model, we still have the option of using them in the actual implementation. Both ORM and UML allow schemas to be annotated with instructions to over-ride the default actions of whatever mapper is used to transform the schema to an actual implementation. For example, the ORM schema in Figure 6 can be prepared for mapping by annotating the roles played by NickName and Color to map as sets inside the mapped Flag structure. Such annotations are not a conceptual issue, and can be postponed till mapping. If you ever feel tempted to use multi-valued attributes in UML, you may be thinking about how you want the structure to be implemented rather than first trying to understand how things are related in the real world.

Later issues

The next issue focuses on a detailed comparison between ORM relationship types and UML associations, including related constraints. We then contrast ORM nesting with UML association classes, and ORM co-referencing with UML qualified associations. Later issues discuss more advanced constraints, aggregation, subtyping, derivation rules and queries.

References

1. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.
2. Fowler, M. with Scott, K. 1997, *UML Distilled*, Addison-Wesley.
3. OMG UML Revision Task Force website, http://uml.systemhouse.mci.com/.