



Database Modelling in UML - Part 3

By Geoffrey Sparks, www.sparxsystems.com.au

5. For each class add a unique object identifier

In both the relational and the object world, there is the need to uniquely identify an object or entity.

In the object model, non-persistent objects at run-time are typically identified by direct reference or by a pointer to the object. Once an object is created, we can refer to it by its run-time identity. However, if we write out an object to storage, the problem is how to retrieve the exact same instance on demand.

The most convenient method is to define an OID (object identifier) that is guaranteed to be unique in the namespace of interest. This may be at the class, package or system level, depending on actual requirements.

An example of a system level OID might be a GUID (globally unique identifier) created with Microsoft's 'guidgen' tool; eg. {A1A68E8E-CD92-420b-BDA7-118F847B71EB}. A class level OID might be implemented using a simple numeric (eg. 32 bit counter).

If an object holds references to other objects, it may do so using their OID. A complete run-time scenario can then be loaded from storage reasonably efficiently.

An important point about the OID values above is that they have no inherent meaning beyond simple identity. They are only logical pointers and nothing more. In the relational model, the situation is often quite different.

Identity in the relational model is normally implemented with a primary key. A primary key is a set of columns in a table that together uniquely identify a row. For example, name and address may uniquely identify a 'Customer'. Where other entities, such as a 'Salesperson', reference the 'Customer', they implement a foreign key based on the 'Customer' primary key.

The problem with this approach for our purposes is the impact of having business information (such as customer name and address) embedded in the identifier. Imagine three or four tables all have foreign keys based on the customer primary key, and a system change requires the customer primary key to change (for example to include 'customer type'). The work required to modify both the 'customer' table and the entities related by foreign key is quite large.

On the other hand, if an OID was implemented as the primary key and formed the foreign key for other tables, the scope of the change is limited to the primary table and the impact of the change is therefore much less.



Also, in practice, a primary key based on business data may be subject to change. For example a customer may change address or name. In this case the changes must be propagated correctly to all other related entities, not to mention the difficulty of changing information that is part of the primary key.

An OID always refers to the same entity - no matter what other information changes. In the above example, a customer may change name or address and the related tables require no change.

When mapping object models into relational tables, it is often more convenient to implement absolute identity using OID's rather than business related primary keys. The OID as primary and foreign key approach will usually give better load and update times for objects and minimise maintenance effort. In practice, a business related primary key might be replaced with:

1. A uniqueness constraint or index on the columns concerned;
2. Business rules embedded in the class behaviour;
3. A combination of 1 and 2.

Again, the decision to use meaningful keys or OID's will depend on the exact requirements of the system being developed.

6. Map attributes to columns

In general we will map the simple data attributes of a class to columns in the relational table. For example a text and number field may represent a person's name and age respectively. This sort of direct mapping should pose no problem - simply select the appropriate data type in the vendor's relational model to host your class attribute. For complex attributes (ie. attributes that are other objects) use the approach detailed below for handling associations and aggregation.

7. Map associations to foreign keys

More complex class attributes (ie. those which represent other classes), are usually modelled as associations. An association is a structural relationship between objects. For example, a Person may live at an Address. While this could be modelled as a Person has City, Street and Zip attributes, in both the object and the relational world we are inclined to structure this information as a separate entity, an Address.

In the object domain an address represents a unique physical object, possibly with a unique OID. In the relational, an address may be a row in an Address table, with other entities having a foreign key to the Address primary key.



In both models then, there is the tendency to move the address information into a separate entity. This helps to avoid redundant data and improves maintainability.

So for each association in the class model, consider creating a foreign key from the child to the parent table.

8. Map Aggregation and Composition

Aggregation and composition relationships are similar to the association relationship and map to tables related by primary-foreign key pairs. There are however, some points to bear in mind.

Ordinary aggregation (the weak form) models relationships such as a Person resides at one or more Addresses. In this instance, more than one person could live at the same address, and if the Person ceased to exist, the Addresses associated with them would still exist. This example parallels the many-to-many relationship in relational terminology, and is usually implemented as a separate table containing a mapping of primary keys from one table to the primary keys of another.

A second example of the weak form of aggregation is where an entity has use or exclusive ownership of another. For example, a Person entity aggregates a set of shares. This implies a Person may be associated with zero or more shares from a Share table, but each Share may be associated with zero or one Person. If the Person ceases to exist, the Shares become un-owned or are passed to another Person. In the relational world, this could be implemented as each Share having an 'owner' column which stored a Person ID (or OID) .

The strong form of aggregation, however, has important integrity constraints associated with it. Composition, implies that an entity is composed of parts, and those parts have a dependent relationship to the whole. For example, a Person may have identifying documents such as a Passport, Birth Certificate, Driver's License & etc. A Person entity may be composed of the set of such identifying documents. If the Person is deleted from the system, then the identifying documents must be deleted also, as they are mapped to a unique individual.

If we ignore the OID issue for the moment, a weak aggregation could be implemented using either an intermediate table (for the many-to-many case) or with a foreign key in the aggregated class/table (one-to-many case). In the case of the many-to-many relationship, if the parent is deleted, the entries in the intermediate table for that entity must also be deleted also. In the case of the one-to-many relationship, if the parent is deleted, the foreign key entry (ie. 'owner') must be cleared.



In the case of composition, the use of a foreign key is mandatory, with the added constraint that on deletion of the parent the part must be deleted also. Logically there is also the implication with composition that the primary key of the part forms part of the primary key of the whole - for example, a Person's primary key may be composed of their identifying documents ID's. In practice this would be cumbersome, but the logical relationship holds true.

9. Define relationship roles

For each association type relationship, each end of the relationship may be further specified with role information. Typically, you will include the Primary Key constraint name and the Foreign Key Constraint name. Figure 6 illustrates this concept. This logically defines the relationship between the two classes.

In addition, you may specify additional constraints (eg. {Not NULL}) on the role and cardinality constraints (eg. 0..n).

10. Model behaviour

We now come to another difficult issue: whether to map some or all class behaviour to the functional capabilities provided by database vendors in the form of triggers, stored procedures, uniqueness and data constraints, and relational integrity.

A non-persistent object model would typically implement all the behaviour required in one or more programming languages (eg. Java or C++). Each class will be given its required behaviour and responsibilities in the form of public, protected and private methods.

Relational databases from different vendors typically include some form of programmable SQL based scripting language to implement data manipulation. The two common examples are triggers and stored procedures.

When we mix the object and relational models, the decision is usually whether to implement all the business logic in the class model, or to move some to the often more efficient triggers and stored procedures implemented in the relational DBMS.

From a purely object-oriented point of view, the answer is obviously to avoid triggers and stored procedures and place all behaviour in the classes. This localises behaviour, provides for a cleaner design, simplifies maintenance and provides good portability between DBMS vendors.

In the real world, the bottom line may be scaling to 100's or 1000's of transactions per second, something stored procedures and triggers are purpose designed for.



If purity of design, portability, maintenance and flexibility are the main drivers, localise all behaviour in the object methods.

If performance is an over-riding concern, consider delegating some behaviour to the more efficient DBMS scripting languages. Be aware though that the extra time taken to integrate the object model with the stored procedures in a safe way, including issues with remote effects and debugging, may cost more in development time than simply deploying to more capable hardware.

As mentioned earlier, the UML Data Profile provides the following extensions (stereotyped operations) with which you can model DBMS behaviour:

- Primary key constraint (PK);
- Foreign key constraint (FK);
- Index constraint (Index);
- Trigger (Trigger);
- Uniqueness constraint (Unique);
- Validity check (Check).

11. Produce a physical model

In UML, the physical model describes how something will be deployed into the real world - the hardware platform, network connectivity, software, operating system, dll's and other components. You produce a physical model to complete the cycle - from an initial use case or domain model, through the class model and data models and finally the deployment model.

Typically for this model you will create one or more nodes that will host the database(s) and place DBMS software components on them. If the database is split over more than one DBMS instance, you can assign packages («schema») of tables to a single DBMS component to indicate where the data will reside.

Conclusion

This concludes this short article on database modelling using the UML. As you can see, there are quite a few issues to consider when mapping from the object world to the relational. The UML provides support for bridging the gap between both domains, and together with extensions such as the UML Data Profile is a good language for successfully integrating both worlds.

References

Muller, Robert J., *Database Design for Smarties*, Morgan Kaufman, 1999.

Rational Software, *The UML and Data Modelling*, Rational Software



Ahoo Engineering Group

Ambler, Scott W., *Mapping Objects to Relational Databases*, AmbySoft inc, 1999