

## Object-Oriented Concepts for Database Design

Michael R. Blaha\* and William J. Premerlani\*\*

\* *OMT Associates Inc., Chesterfield, MO 63017 blaha@acm.org*

\*\* *GE Corporate R&D, Schenectady, NY 12301 premerlani@crd.ge.com*

### Abstract

Object-oriented concepts facilitate both the understanding of requirements for an application and the subsequent design of supporting software. With an object-oriented approach, the software engineer devises a model of the real world, and then seamlessly carries the real-world model forward into design and implementation. Object-oriented models provide a useful abstraction for designing both programming code and database code. This paper focuses on the portion of the OMT methodology that applies to databases. We summarize a process for database design based on our experience.

### 1. Introduction

Software is costly to develop, prone to unexpected errors, and difficult to maintain and gracefully evolve. The goal of software engineering is to ameliorate this software crisis by emphasizing the early conceptual stages of development when software is most malleable and insights can be leveraged for greatest benefit. By placing more emphasis on the initial stages of development, you can save disproportionate effort in subsequent implementation, maintenance, and evolution.

Object-oriented techniques offer a new, more powerful approach to software development than the conventional procedural paradigm. You begin by describing relevant aspects of the real world and gradually elaborating and refining the description until you realize a working system. In contrast, the procedural approach emphasizes the latter portions of the development lifecycle and inhibits the abstract thinking that is so important to software engineering. Object-oriented models facilitate the design of both programming code and database code. This paper extends the Object Modeling Technique (OMT) to address the design of database applications and summarizes some of the major ideas in [1].

The OMT approach to software development uses three distinct, but complementary, models: the object, dynamic, and functional models. These models taken together describe a system. For different kinds of applications, the three models have variable importance. However, the object model dominates for most database applications. The OMT object model is derived from the seminal En-

tity-Relationship model [5] and describes the data structure of a system—objects and their relationships.

A database is a permanent, self-descriptive repository of data that is stored in one or more files. A database management system (DBMS) is the software for managing a database. There are many reasons for formally managing data with a database, such as promoting sharing between users and applications, protecting data from accidental hardware and software failures, and providing security mechanisms for accessing data. The following life cycle applies for most database applications:

1. Model the application.
2. Devise an architecture for coupling the application to a database.
3. Select a specific DBMS.
4. Design the database schema. Write code to define the proper database structures.
5. Write programming language code to compensate for database manager shortcomings, provide a user interface, validate data, and perform application computations. Many DBMSs have productivity tools that simplify routine applications.
6. Run the application(s) and populate the database with information. Query and update the database as needed.

The next two sections of this paper introduce the OMT approach to modeling as a prelude to a detailed discussion of database design (steps 2 through 4 above) for the remainder of the paper. We present a database design process that we have successfully applied to a range of industrial applications.

### 2. Overview of the OMT Methodology

A methodology consists of two major components: process and concepts. A clear process must guide the practitioner through system development. The process is supported by underlying concepts and a notation for expressing the concepts. This section briefly summarizes the OMT process for software development. The next section reviews object modeling concepts and notation.

The OMT software development process consists of the following steps:

- **Conceptualization:** Conceive an application and formulate tentative requirements.
- **Analysis:** Starting from the initial problem requirements, derive a model of the real world. Scrutinize and rigorously restate the requirements.
- **System Design:** Choose an overall architecture. Consider the feasibility, cost, and risk of alternative structures. Also establish policies that will serve as a default for the subsequent, more detailed portions of design.
- **Detailed Design:** Augment and transform the real-world analysis model into a form amenable to computer implementation. Often much of the analysis model is suitable for direct use.
- **Implementation:** Write the actual programming and database language code. This step is straightforward, because much work has already been performed during analysis and design.
- **Maintenance:** Use models to understand system structure and intent. Cogent models stimulate the memories of the original developers and foster communication with other project personnel.

The OMT models seamlessly span software development phases. The models of the real world that are formulated during analysis form the basis for design. During design the analysis models are elaborated with computer implementation constructs. Optimization occurs as analysis constructs are restated for execution efficiency. Throughout the development process the same concepts and notation apply; the only difference is the shift in perspective from an emphasis on describing the real world to a focus on computer resources.

Three kinds of models are used throughout the OMT development process: the object, dynamic, and functional models. The three models describe different aspects of a system and combine together to describe a system fully.

The *object model* characterizes the static structure of things (objects). It looks at structure in terms of groups of analogous objects (classes), their similarities and differences (generalization), and their important relationships with one another (associations). The object model defines the context for software development — the universe of discourse.

The *dynamic model* describes temporal interactions between objects—various stimuli that occur and the response of objects to the stimuli. The objects in a class pass through states with similar responses to events.

There is one state diagram for each class with significant dynamic behavior; the collection of state diagrams constitutes the dynamic model. We use the nested state machine notation of David Harel to express the dynamic model. [7]

The *functional model* defines the computations that objects perform—how output values are computed from input values. The functional model specifies operations that arise within the object and dynamic models. We recommend several notations for expressing the functional model, though we emphasize pseudocode enhanced with a notation for navigating object models.

### 3. Summary of Concepts and Notation for OMT Object Model

The theme of this paper is our process for database design. To a large extent, this process exists apart from the choice of modeling notation. In this section we briefly summarize the highlights of the object modeling notation. We use the UML notation of Booch, Rumbaugh, and Jacobson [3].

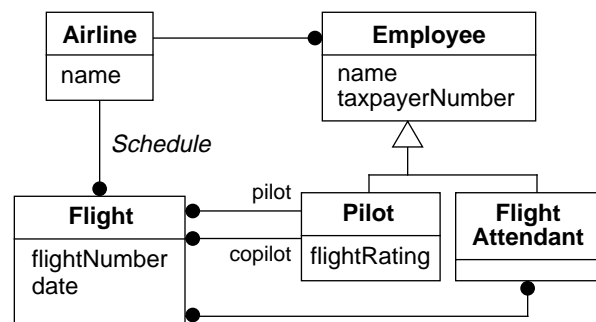


Figure 1 Object model example

Figure 1 presents a fragment of an object model for tracking airline flights. An airline has many employees each of whom has a name and taxpayer number. We consider two categories of employees: pilots and flight attendants. An airline schedules many flights, each of which is staffed with a pilot, copilot, and many flight attendants. Pilots and flight attendants both service many flights.

Object models are built from three basic constructs: classes, associations, and generalizations. A *class* is a description of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. Thus class *Airline* describes objects *United Airlines*, *Lufthansa*, and *Air New Zealand*. A class is denoted by a box with the class name in the top portion of the box. An optional second portion of the class box may list attributes. An *attribute*

is a named property of a class describing data values held by each object in the class. For example, class *Flight* has attributes *flightNumber* and *date*. In general the choice of classes depends on the semantics of an application and the purpose of the object model.

Figure 1 contains five associations. *Associations* are shown as lines connecting class boxes and indicate that the underlying objects are related. For example, the association between *Airline* and *Employee* indicates that employees work for an airline and airlines employ employees. Naming of associations is optional, unless there is ambiguity in the model.

Associations have multiplicity and optional role names. *Multiplicity* specifies how many objects of one class may relate to an object of an associated class. The solid balls in Figure 1 are the notation for “many” multiplicity, meaning zero or more. Thus a flight may be staffed with many flight attendants and each flight attendant services multiple flights. A hollow ball (not shown in Figure 1) means zero or one. The lack of a symbol at the end of an association line means exactly one. (Our notation for multiplicity follows that in [11] and is a departure from the UML [3]. We believe the multiplicity notation in [11] is more readable for large models with many classes.)

A *role* name identifies an end of an association. The pilot and copilot shown in Figure 1 next to the *Pilot* class are role names. A flight has one pilot and one copilot. The same person may serve as pilot on some flights and copilot on other flights.

Generalization is the third basic object modeling construct. *Generalization* organizes classes based on their similarities and differences. In Figure 1 *Employee* is a superclass that refines into two subclasses: *Pilot* and *FlightAttendant*. *Pilot* and *FlightAttendant* inherit *Employee* attributes. Each object in the *Pilot* class is described by both *Employee* attributes and *Pilot* attributes. Each object in the *FlightAttendant* class is described by both *Employee* attributes and *FlightAttendant* attributes. A large hollow arrowhead next to the superclass denotes generalization.

## 4. Overview of the OMT Methodology for Database Applications

### 4.1 Entity-Based Design vs. Attribute-Based Design

The OMT methodology adopts an entity-based approach. In an entity-based approach you note entities from the real world, describe them, and observe relationships among them. In contrast, with an attribute-based approach you list attributes that are meaningful to an application and organize them into groups.

In practice, entity-based development is clearly superior to attribute-based development. First, most applications have an order of magnitude fewer entities than attributes, so an entity-based approach is more tractable. Second, attribute-based development requires that you check normal forms—to ensure that attributes are combined in a reasonable manner so update anomalies are unlikely to occur. These checks can be tedious and confusing. With an entity-based approach attributes are introduced as part of describing real-world objects. To the extent that you directly describe objects and do not intermix different things, the anomalies addressed by normal forms do not arise. This real-world basis also makes an entity-based model more extensible, understandable, and tunable for fast performance. Finally, an entity-based approach is more likely to be compatible with the development approach for programming code and the overall system.

### 4.2 Tailored OMT Process for Database Applications

Database design is consistent with the general OMT framework described in Section 2 but many of the specific design details are peculiar to database applications. During analysis, the focus is on what needs to be done, independent of how it is done. During design decisions are made about how the problem will be solved, first at a high level, then at increasingly detailed levels. We recommend the following steps for database design:

- **System Design:** Devise an application architecture. Choose a data management paradigm and a protocol for interacting with the DBMS.
- **Detailed Design:** Augment and transform the real-world analysis models into a form amenable to computer implementation. Add details to the models such as candidate keys, domains, and storage requirements.
- **Implementation:** Map the resulting object model to the target DBMS. Specify the physical layout of data. Write accompanying programming language code as needed.

## 5. System Design for Database Applications

During system design you make strategic decisions with broad consequences for the software as a whole. The focus of this paper is on the architecture of an individual application running on a single computer. Reference [1] has additional material on distributed databases and integration of applications.

### 5.1 Devising an Architecture

The first task for system design is to devise an architecture. We recommend that your choice of architecture be guided by several principles:

- **Distinguish between operational and decision-support applications.** Operational applications update a database by posting numerous transactions as part of the routine business of an organization. They must access the current data of an organization. Operational applications require fast performance and maximal database availability.

In contrast, decision support applications tend to have few updates and can involve unpredictable and lengthy queries. Decision-support applications have different tradeoffs; they can run more slowly and can tolerate more downtime. However, they must let users express new queries readily.

- **Separate application logic from the user interface.** It is helpful to organize application logic as a library of methods and then invoke the methods from a user interface as appropriate. There may be multiple user interfaces that access the application methods; the user interfaces may differ in presentation format and in the information they present and suppress.
- **Substitute database queries for programming code.** You must strike a proper balance between the role of the DBMS and the role of a programming language. Do not embed all application logic in programming code. Instead, try to offload computation to the DBMS. This offloading can greatly improve performance, increase extensibility, reduce development time, and reduce bugs.

We recommend that you use a formal process in devising an application architecture. First, generate candidate architectures. Second, choose decision criteria and assign weights to them. Next, rate the compliance of the architectures against the criteria. And finally compare scores for each candidate architecture.

### 5.2 Choice of Data Management Paradigm

Given an overall architecture, next choose an appropriate paradigm for managing data. Determine what protection is needed against accidental loss and corruption. You can use flat files and in-memory data structures to store data as well as several kinds of databases—hierarchical, network, relational, and object-oriented.

Applications can directly read and write to sequential or random-access files. Files can provide a simple mechanism for storing data. However file complexity grows rapidly with schema size and all applications must agree on a format. Files can be difficult to maintain, extend, and coordinate for concurrent access.

In-memory data structures can also retain data when augmented with special power supplies or hardware. For example, battery powered memory and EPROMs both re-

tain data. In-memory storage provides high performance, sharing, and is simple to program. However access to data can be inflexible and this approach usually requires custom programming.

Several different kinds of database managers have evolved over the years. Hierarchical and network DBMSs are now obsolete and you should normally avoid them for new applications. We will discuss relational and object-oriented DBMSs.

A relational DBMS (RDBMS) [6] [9] has three major aspects:

- Data that is presented as tables. Tables have a specific number of columns and an arbitrary number of rows.
- Operators for manipulating tables. SQL is the standard language.
- Integrity rules on tables.

Commercial RDBMSs include Oracle, Informix, Sybase, Microsoft Access, and DB2. RDBMSs are intended for applications with simple data types and modest performance demands. Their non-procedural, declarative language is a strength.

Object-oriented DBMSs (OO-DBMSs) [4] [8] are loosely based on concepts embodied in object-oriented programming languages. Versant, ObjectStore, O2, GemStone, and Objectivity are well known OO-DBMSs. OO-DBMSs have been motivated by the deficiencies in RDBMSs. Unlike RDBMSs, OO-DBMSs can quickly navigate data structures, support rich data types, and cleanly integrate with at least one programming language. Unfortunately, OO-DBMSs remedy some RDBMS deficiencies but lose some of the strengths. Unlike OO-DBMSs, RDBMSs have a firm theoretical foundation, mature standards, and a rich declarative language. We regard RDBMSs and OO-DBMSs as both having application niches.

### 5.3 DBMS Interaction Protocols

Next devise a protocol for interaction between the application and DBMS.

An application program can directly access a relational database with embedded SQL code. SQL statements are interspersed with programming language code. This technique is straightforward and well supported by commercial products. However the impedance mismatch between programming languages and SQL complicates development. The declarative nature of SQL clashes with the imperative style of most programming languages. Code generators and fourth generation languages can lessen some drawbacks of this technique.

Another option is to access an object-oriented database directly. By definition there is a clean interface between programming and OO-DBMS capabilities if you use the preferred programming language. However, once again, there are drawbacks to consider. Because of incomplete standards, you must commit to a particular OO-DBMS and would have difficulty changing to another OO-DBMS at a later date.

For several projects [10], we have combined an RDBMS with an in-memory data manager as Figure 2 shows. Applications can directly access the underlying RDBMS for global data that must be shared. Or they can use buffering for extended access to data. Buffered data is checked in and out of memory (object-oriented programming language data structures) by reads and writes to the underlying RDBMS. The layered approach offers high performance. However concurrency is coarse and it is difficult to prevent database corruption by other programs that may circumvent the layer. Several commercial products also provide an OO layer to access an RDBMS.

In a metamodel-based system the application indirectly accesses data: first access the metadata, then formulate the query to access the data. For example an RDBMS processes SQL commands by first accessing the data dictionary and then accessing the actual data. We have implemented a metamodel-driven application using dynamic SQL with an RDBMS [2]. Such a technique is also viable with an OO-DBMS that permits access to metadata and dynamic query construction. Meta applications are complex to develop and understand but are flexible and powerful.

#### 5.4 Object Identity

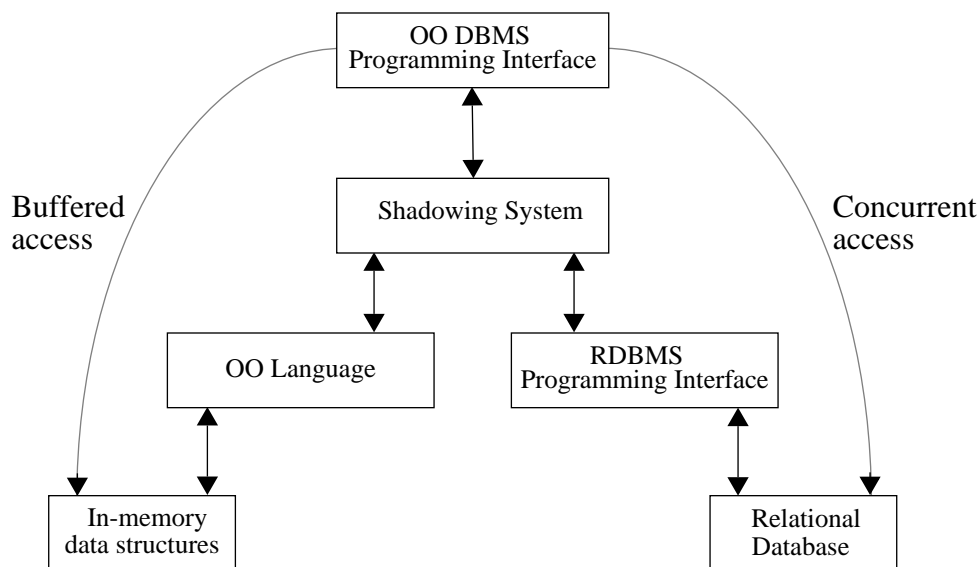
The next step in system design is to choose an approach to object identity. OO-DBMSs give objects existence-based identity, in which a system-generated object identifier (also called an OID, a surrogate, or a pointer) identifies each object. Files and RDBMSs provide the choice of existence-based identity or value-based identity, in which some combination of attributes identifies each object. Furthermore, you can mix the two approaches for an application. Most often we use existence-based identity.

Existence-based identity has several advantages. The resulting primary keys are single attribute, small, and uniform in size. An object identifier can encode information to promote efficient lookup. But existence-based identity also has disadvantages. Identifiers can be troublesome to generate for files and RDBMSs that lack semantic support. Assigning identifiers may require access to global data that can interfere with concurrent locking.

Value-based identity has different advantages. Primary keys have intrinsic meaning to the user, facilitating debugging and maintenance of the database. On the down side, value-based primary keys can be difficult to change. A change to a primary key may require propagation to many foreign keys.

#### 5.5 Miscellaneous Issues

You should consider several additional issues during system design:



**Figure 2** Run-time architecture for RDBMS with in-memory data manager (adapted from reference [10])

- **Schema evolution.** Often the database schema evolves after application development. Then you must formulate a strategy for transitioning data from the old schema to the new schema. (Similarly, you must accommodate the changes in the application programming code.)
- **Historical data.** Some applications require past data and for others the current data suffices. Versions are a related issue. Versions are alternative values for an object that apply under various conditions such as during hypothetical calculations or design studies. Many OO-DBMSs support versions but few DBMSs support historical data.
- **Secondary aspects of data.** Some applications require attribute value metadata such as units of measure, data accuracy, data source, and date of last update. Secondary data describes application data rather than being the actual data. Conventional DBMSs ignore secondary data; if it is important you must plan some workaround.
- **Basic DBMS features.** You must elicit requirements for security, concurrency, and recovery.
- **Distribution.** You must decide if the data should be available at multiple sites. Replication may be needed to achieve performance and reliability objectives. With replication there must be synchronization policies for multiple copies of the same data.
- **Strategy for nulls.** You may use the null value supported by the DBMS or designate a special value to denote null. It is easier to use the null logic from the DBMS, but the standard logic for nulls does not always suffice.

## 6. Detailed Design for Database Applications

The purpose of detailed design is to prepare the analysis model for implementation. You must add intermediate data structures and transform the model for efficient execution. You must also devise algorithms for realizing the dynamic and functional models.

### 6.1 Transformations

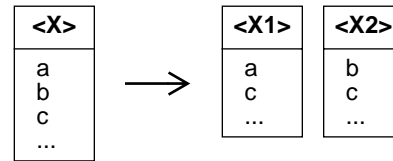
A **transformation** is a mapping from the domain of object models to the range of object models. By applying a series of transformations, you can simplify and optimize a model.

Figure 3 shows two transformations—depending on the direction in which you read the diagram. The angle brackets denote a placeholder for a class that you must instantiate when you use the transformation.

- Read from left to right, Figure 3 shows a transformation for partitioning a class. A designer may split a

class ( $X$ ) into two smaller classes ( $X1$ ,  $X2$ ) by apportioning and replicating information.

- Read from right to left, Figure 3 shows a merge transformation. Under some circumstances a designer can combine  $X1$  and  $X2$  to form  $X$ .

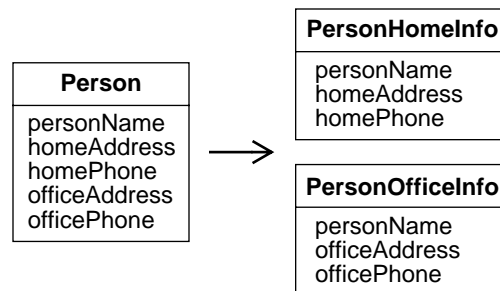


{Each attribute and association for  $X$  is replicated or apportioned. If  $X$  is a subclass in a generalization, both  $X1$  and  $X2$  become subclasses in the generalization. If  $X$  is a superclass, the subclasses multiply inherit from  $X1$  and  $X2$ .}

**Figure 3** Transformations: Partition a class or merge classes

The arrow indicates whether a transformation gains or loses information. A transformation in the direction of an arrow yields a model with equal or less semantic content. A designer need not add information to allow a transformation to occur in the direction of an arrow. Conversely, transformation in a direction without an arrow yields a model with increased semantic content. The designer must assert some additional information to allow such a transformation to proceed. A bidirectional arrow denotes an equivalence transformation.

Figure 4 shows an example for the transformations. A database may store both personal and business information for a person; you can represent this information as one class or split the information into two classes. Both models are correct; the choice between models would depend on the purpose and scope of an application. If you have much personal and business information, it's a good idea to separate them. If there is only a modest amount of information, it may be easier to combine them.



**Figure 4** Example of transformations

## 6.2 Miscellaneous Issues

You must address additional issues during detailed design:

- **Add candidate keys.** You should review your object model and add candidate keys for unique combinations of attributes.
- **Assign domains.** Do not directly use data types. During design you should assign a domain for each attribute; during implementation you can specify a data type for each domain. Domains promote consistency among attributes and reduce implementation decisions.
- **Specify nulls.** Consistent with your general policy from system design, you should specify permissibility of nulls or default values for each attribute.
- **Estimate physical storage.** For each class and many-to-many association you should estimate the number of occurrences, growth rate, and retention period.
- **Performance tuning.** The database design must permit efficient traversal of frequent navigation paths. Often, it is sufficient to build a unique index for each primary and candidate key, and a secondary index for each foreign key.
- **Transaction modeling.** Some applications, such as complex aerospace systems, require that you perform transaction modeling, in which you determine the kinds of queries likely to be executed and obtain data for each query. This data includes how often each query occurs, the number of read and write I/Os, the required response time, the number of concurrent users, and the database bottlenecks. Transaction modeling is intended to increase performance, but the resulting data is difficult to estimate, and the estimates are often wrong. Moreover, commercial information systems seldom need this level of detail. Our experience is that you can often obtain very good performance by merely tuning a database for fast navigation of the object model.
- **Encapsulation versus query optimization.** There is a fundamental, irreconcilable conflict between the goals of encapsulation and the goals of query optimization. This conflict is more prominent for RDBMSs than OO-DBMSs, because RDBMSs emphasize non-proceduralism with the SQL language.

The principle of encapsulation is that an object should only access objects that are directly related (directly connected by an association or generalization). Indirectly related objects should be accessed indirectly via the methods of intervening objects.

Encapsulation increases the resilience of an application; local changes to an application model cause local changes to the application code.

On the other hand, DBMS optimizers take a logical request and generate an efficient execution plan. If queries are broadly stated, the optimizer has greater freedom for devising an efficient plan. For example, RDBMS performance will usually be best if you join multiple tables together in a single SQL statement, rather than disperse logic across multiple SQL statements.

For an OO-DBMS application, you should usually honor encapsulation—writing methods that access only directly related objects. For RDBMSs there is no simple resolution of this conflict and you must make the best tradeoff for an application.

## 7. Implementation with a Relational DBMS

At this point, implementation should be straightforward. You have already addressed most difficult issues during analysis and design.

### 7.1 Mappings

Since you have used transformations to simplify and optimize the model, only a few mappings are needed from object-oriented concepts to tables.

Normally we map each class to a table and each attribute to a column. You may require additional columns for a generated identifier (existence-based identity), buried associations, and generalization discriminators.

You should map each many-to-many association to a distinct table. The primary key of the association is the combination of the primary keys for each class. We recommend that you bury simple one-to-one and one-to-many associations in associated classes as foreign keys. Role names become foreign key attribute names.

You should implement ternary associations and associations with link attributes with distinct tables. Reference [1] provides more explanation for these and other advanced associations.

The simplest implementation for generalization is just to map the superclass and each subclass to a table. The tables share a common primary key. Applications must enforce the partition between subclasses because an RDBMS will not.

You should carefully define referential integrity to enforce the semantics of the object model. If you use existence-based identity, you will not need to propagate the effect of updates. We recommend the following referential integrity guidelines for deletions:

- **Generalization.** Cascade deletions for foreign keys that arise from the implementation of generalization.
- **Buried association, minimum multiplicity of zero.** Normally set the foreign key to null, but sometimes you may forbid deletion.
- **Buried association, minimum multiplicity of one.** You can cascade the effect of a deletion or forbid the deletion.
- **Association table.** Normally we cascade deletions to the records in an association table. However, sometimes we forbid a deletion.

The last step for implementing the object model is to tune the database. Indexes are the primary means for tuning a relational database. Normally, you should define a unique index for each primary and candidate key. (Most RDBMSs create unique indexes as a side effect of SQL primary key and candidate key constraints.) You should also create an index for each foreign key that is not subsumed by a primary key or candidate key constraint.

### 7.2 Miscellaneous Issues

You must consider several additional details for an RDBMS implementation.

- **Functions and constraints as tables.** Sometimes it is helpful to model discrete functions and constraints with tables.
- **Views.** RDBMS views provide a convenient mechanism for reading data. If your RDBMS supports updates, you can define views that consolidate an object across the tables that implement the levels of generalization.
- **Physical layout of data.** Some RDBMSs support collocation of data from different tables. Also you must choose the number of contiguous disk blocks to be assigned to a table or cluster of tables upon allocation of disk space.
- **Style guidelines for SQL code.** Usually it is a good idea to list explicit target attributes when inserting into a table within a program. This isolates the program from minor schema changes, such as reordering or adding attributes. You can help the query optimizer by avoiding nested SQL and using simple queries. You should adopt a naming convention that distinguishes programming variables from database variables.

## 8. Implementation with an OO-DBMS

Implementation with an OO-DBMS should also be straightforward after a thorough analysis and design.

- **Mapping rules.** Mapping rules are straightforward for an OO-DBMS since implementation constructs closely match modeling constructs. The most significant issue is implementation of associations and the options are analogous to those with relational databases. You may promote an association to a class or bury it with a single pointer or dual pointers. Furthermore you can represent “many” multiplicity directly with a linked list, set, or array.
- **Persistent and transient objects.** A transient object may refer to a persistent object, but a persistent object cannot refer to a transient object. Unfortunately persistence is not transparent for many current products. Ideally persistence should be apart from type. Persistent variables should have the same behavior as transient variables aside from referencing constraints and database commit and rollback commands.
- **Third party libraries.** OO-DBMSs can be difficult to combine with third party libraries. Some OO-DBMSs require that classes inherit from a persistent class; this can be disruptive for existing classes. Existing methods often must be modified to include special statements for interfacing to the OO-DBMS. Furthermore the source code for third party libraries is not always available. C++ lacks a standard class hierarchy so C++-based OO-DBMSs and third party libraries may be incompatible. Naming clashes can also occur.
- **Metadata at run time.** Many OO-DBMSs do not permit access to metadata via a program. Many OO-DBMSs do not permit dynamic construction of database queries. These limitations are largely a consequence of language design for OO-DBMSs that are based on C++. C++ is a strongly typed language that resolves references at compile time.
- **Casting rules.** Some OO-DBMSs require confusing and awkward casting changes across generalization levels.

## 9. Conclusion

This paper augments the general OMT methodology for the purpose of database design. You begin by formulating an analysis model of the real world in order to scrutinize and rigorously restate the requirements. During system design you must choose an overall architecture. You must choose a data management paradigm, a DBMS interaction protocol, and a strategy for object identity. During detailed design you prepare the real-world analysis models for implementation. Transformations allow you to optimize and simplify the model. The ultimate implementation with an RDBMS, OO-DBMS, or other data manager is largely straightforward because most diffi-

cult issues have been addressed during analysis and design.

## References

- [1] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, Englewood Cliffs, New Jersey, 1998.
- [2] MR Blaha, WJ Premerlani, AR Bender, RM Salemmme, MM Kornfein, and CK Harkins. Bill-of-material configuration generation. *Sixth International Conference on Data Engineering*, February 5-9, 1990, Los Angeles, California.
- [3] The following books are planned for the Unified Modeling Language:  
 Grady Booch, James Rumbaugh, and Ivar Jacobson. *UML User's Guide*. Addison-Wesley, Reading, Massachusetts.  
 James Rumbaugh, Ivar Jacobson, and Grady Booch. *UML Reference Manual*. Addison-Wesley, Reading, Massachusetts.  
 Ivar Jacobson, Grady Booch, and James Rumbaugh. *UML Process Book*. Addison-Wesley, Reading, Massachusetts.
- [4] RGG Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison-Wesley, 1991.
- [5] PPS Chen. The Entity-Relationship model—toward a unified view of data. *ACM Transactions on Database Systems* 1, 1 (March 1976).
- [6] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. 2nd edition. Benjamin Cummings, Redwood City, California, 1994.
- [7] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8 (1987), 231-274.
- [8] Mary E S Loomis. *Object Databases: The Essentials*. Addison-Wesley, Reading, Massachusetts, 1995.
- [9] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Mateo California, 1993.
- [10] WJ Premerlani, MR Blaha, and JE Rumbaugh. An object-oriented relational database. *Communications ACM* 33, 11 (November 1990).
- [11] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.