Issues in Information Systems

Volume XII, No. 1, pp. 328-338, 2011

OBJECT REUSE AND INTEGRATION IN OBJECT-RELATIONAL DATABASE DEVELOPMENT

Ming Wang, California State University, ming.wang@calstatela.edu

ABSTRACT

One of the most important features of the Object-relational database (ORDB) is information reuse and integration. The feature provides standard data structure, data portability, and maintainability for database applications development. Despite the undeniable object reuse and integration features of ORDB, very little research has been published to address its importance in the database application development in the real-world scenario. This paper presents a case study to investigate object reuse and integration in design and implementation of ORDB.

Keywords: *Relational database, Object-relational database, object-oriented technology*

INTRODUCTION

With rapidly increasing volumes of digital information and broadening range of applications development, applications have become more and more complex and the software development costs have increased. This tremendous challenge has led to the idea of object reuse and integration in software development. The Object-relational database management system (ORDBMS) provides a way to solve the problem. ORDBMS enhances object-oriented technology into the relational database management systems (RDBMS) and extends traditional RDBMS to ORDBMS. As an evolutionary technology, ORDBMS allows users to take advantage of reuse features in object-oriented technology, to integrate objects to relations and to maintain a consistent data structure in the existing RDBMS. If multiple applications use the same set of database objects in ORDBMS, the standard for the database objects are created, and these objects can be reused and integrated to ORDB applications [4].

In response to the evolutional change of ORDBMSs, SQL:1999 started supporting object-relational data modeling features in database management standardization and SQL:2003 continues this evolution. Currently, all the major database vendors have extended to their relational database to object-relational database products to reflect the consensus of SQL the standards [2]. Using ORDBMS to develop applications can enforce use of standard data structure, provide object-oriented programmers' integrated view of data and allow multiple applications to operate cooperatively. Ultimately, this can result in improved operational efficiency for the IT department, increase object-oriented programmers' productivity, lower development efforts, decrease maintenance cost, reduce risk of defect rate, and raise applications' reliability.

Although ORDB technology is already available for use in all the major database vendors' products, its adoption rate of database application developers is relatively low. One of the major criticisms of ORDBMS is that its complexity results in the loss of the essential simplicity and purity of the relational database model. This paper presents a case study to investigate information reuse and integration in design and implementation of ORDBMSs. The UML class diagram is used to model ORDB design. Oracle SQL statement script is used to illustrate ORDB implementation. The purpose of the paper is to present a framework of object reuse and integration features in ORDBMS for database researchers and industrial application developers.

Issues in Information Systems

Volume XII, No. 1, pp. 328-338, 2011

CASE STUDY: ORDB DESIGN

The Pacific Bike Traders Co. assembles and sells bikes to customers. The company currently accepts customer orders online and wants to be able to track orders and bike inventory. The new ORDB system will be created to handle the current transaction volume generated by employees processing incoming sales orders. The system must update the available quantity on hand to reflect that the bike has been sold and produce customer sales orders, invoices and reports showing inventory levels.

Business Rules

The following business rules are defined for the Pacific Bike Traders ORDB scenario.

One customer may originate many orders.
One order must be originated from one customer.

One order must contain one or more bikes.
One bike may or may not be in many orders.

One employee may or may not place many orders.
One order must be placed by one employee.

One bike is composed of a front wheel, rear wheel, crank, and stem.

One employee must be either a full-time or part-time.

Reuse and Integration in UML Design

Base on the scenario and its business rules, a UML class diagram in Figure 1 is developed to model the Pacific Trader ORDB design. Each class is displayed as rectangle that includes three sections: the top section gives the class name; the middle section displays the attributes of the class; and the last section displays methods. Associations between classes are indicated with multiplicity ("min..max." notation). Inheritance is indicated with an empty triangle. Aggregation is marked with an empty diamond, whereas composition is marked with a solid diamond. Aggregation models a whole-part relationship. A sales order is made of line items (bikes). The dotted line links to the associative class generated from the many-to-many relationship. Composition models a closer whole-part relationship than aggregation.

Inheritance shows attributes in the Employee super class are shared by Fulltime and Part-time subclasses. Composition shows that bike parts can be integrated in the bike class. Name, address, and phone objects are reused in customer and employee classes.

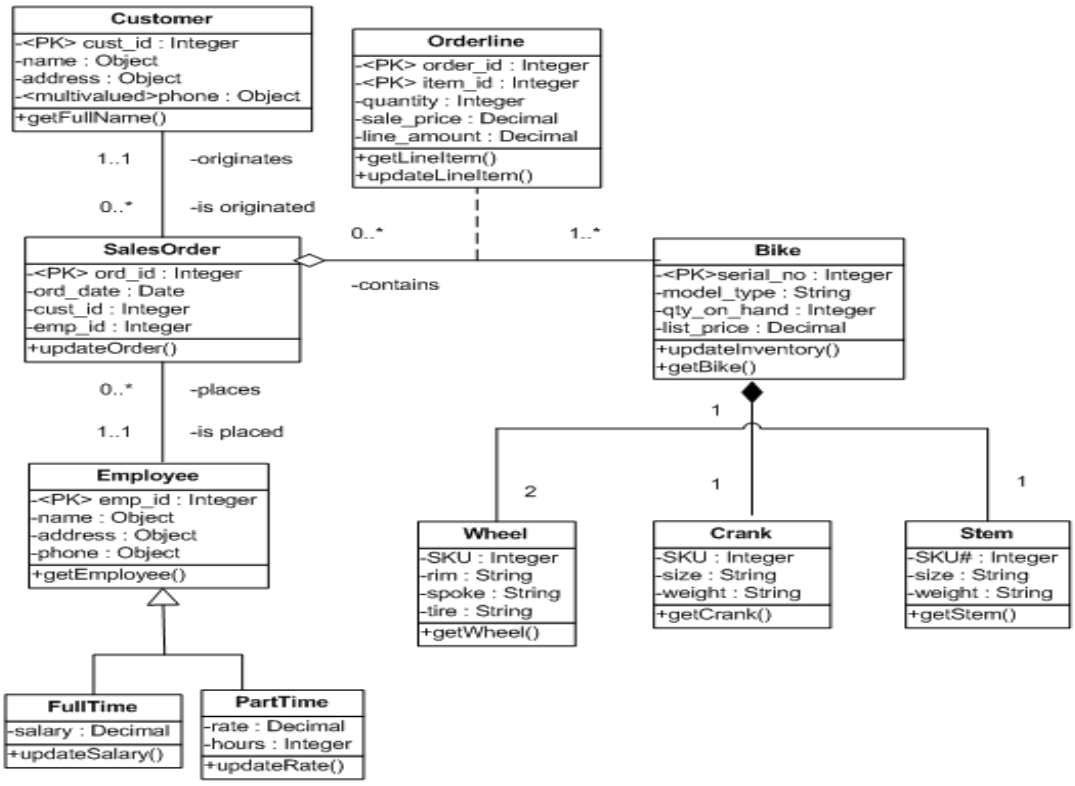


Figure 1 Pacific Bike Trader's UML Class Diagram [5]

CASE STUDY: ORDB IMPLEMENTATION

Based on the Pacific Trader's UML class diagram in Figure 1, the six object reuse and integration features of ORDBMSs are identified and implemented with Oracle SQL Scripts. The implementation shows how the UML class diagram maps and supports ORDBMS. For the sake of simplicity, it is assumed that referential integrity constraints will be added later. This section will focus on discussion of the six object reuse and integration features: 1) Object type reuse for data standardization; 2) Encapsulated user-defined methods for standard data access; 3) Object type inheritance for data reuse; 4) Integration of nested table data; 5) Object views for reuse of relational data 6) Integration ORDB applications with object type interface.

Object types for data reuse

Object type is user-defined data type (UDF) or abstract data type (ADT) that is used in ORDB creation. Commonly used objects such as address and name should be defined with object types. Once object types are defined they are stored in the database permanently and can be used repetitively to create any new columns and tables in the database. Reuse of object types can standardize data stored in ORDBs. The following SQL statements define Address_ type and Name_type as object types and varray_phone_type as a VArray type for reuse in the ORDB.

```
CREATE TYPE address_type AS OBJECT
   (street VARCHAR2(30),
    city   VARCHAR2(25),
    state  CHAR(2),
    zip    NUMBER(10));

CREATE TYPE name_type AS OBJECT
 (f_name VARCHAR2(25),
  l_name VARCHAR2(25),
  initial      CHAR(2));

CREATE TYPE varray_phone_type AS VARRAY(3)
OF VARCHAR2(14);
```

The above Address_type, Name_type and varray_phone_type can be used to define columns in the customer table below.

```
CREATE TABLE Customer(
  Cust_ID          NUMBER(5),
  CustName         name_type,
  CustAddress      address_type,
  CustPhones       varray_phone_type);
```

Object tables can also be entirely defined by an object type, instead of using relational tables consisting of one or more object columns. The employee object table can be created by the employee_type in the following statements.

```
CREATE TYPE employee_type AS OBJECT
      (emp_id     NUMBER(10),
       SSN        NUMBER(9),
       name       name_type,
       dob        DATE,
       address    address_type,
       phones              varray_phone_type);

CREATE TABLE Employee of employee_type;
```

Defined methods for standard data access

Once attributes of an object type are defined, the user can define methods for each object type. Methods describe the behavior of attributes. For each object type, the user can define the methods that operate on attributes in the object type and encapsulate the methods with the attributes in the object_type. The following statements add a method to the Name_type object type interface defined in Section 3.1. The first statement adds the method header to the object type interface. The second statement adds the method body the object type body:

```
ALTER TYPE name_ty   ADD MEMBER FUNCTION full_name RETURN VARCHAR2;

CREATE TYPE BODY name_ty AS
```

```
MEMBER FUNCTION full_name
RETURN VARCHAR2 IS
BEGIN
          RETURN(l_name ||  ’  ‘  || f_name);
        END full_name;
END;
```

The following SELECT statement calls the method defined in the Customer table.

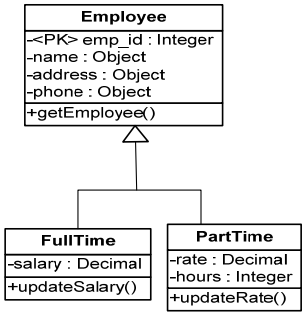
```
SELECT c.custName.full_name ( ),
c.custAddress.City FROM customer c;
```

C.CUSTNAME.FULL_NAME()	CUSTADDRESS.CITY
Tommy Ford	Des Moines

The name_ty object type is associated with the full_name ( ) method, which concatenates the first and last names together. If this functionality is embedded in the server, it allows the functionality to be shared and reused by all the applications. The specified methods are privately encapsulated in the object body. Reusability of methods comes from the ability to store persistent standard data type and functions/methods together on the server, rather than having them coded in each application.

Object Type Inheritance for Data Reuse

ORDBMSs allow users to define hierarchies of data types. With this feature, users can build subtypes in hierarchies of database types. If users create standard data types to use for all employees, then all of the employees in your database will use the same internal format. Users might want to define a full time employee object type and have that type inherit existing attributes from employee_ty. The full_time_ty type can extend employee_ty with attributes to store the full time employee’s salary. The part_time_ty type can extend employee_ty with attributes to store the part-time employee’s hourly rates and wages. Inheritance allows for the reuse of the employee_ty object data type. The details are illustrated in the following class diagram:



Object type inheritance was one of new features of Oracle 9i/10g. For employee_ty to be inherited from, it must be defined using the NOT FINAL clause because the default is FINAL, meaning that object type cannot be inherited. Oracle 9i can also mark an object type as NOT INSTANTIABLE; this prevents objects of that type derived. Users

can mark an object type as NOT INSTANTIABLE when they use the type only as part of another type or as a super_type with NOT FINAL. The following example marks address type as NOT INSTANTIABLE:

```
CREATE TYPE employee_ty AS OBJECT (
  emp_id       NUMBER,
  SSN          NUMBER,
  name         name_type,
    dob        DATE,
  phone        varray_phone_type,
  address      address_type
) NOT FINAL NOT INSTANTIABLE;
```

To define a new subtype full_time_ty inheriting attributes and methods from existing types, users need to use the UNDER clause. Users can then use full_time_ty to define column objects or table objects. For example, the following statement creates an object table named FullTimeEmp.

```
CREATE TYPE full_time_ty UNDER employee_ty (Salary  NUMBER(8,2));

CREATE TABLE FullTimeEmp of full_time_ty;
```

The preceding statement creates full_time_typ as a subtype of employee_typ. As a subtype of employee_ty, full_time_ty inherits all the attributes declared in employee_ty and any methods declared in employee_ty. The statement that defines full_time_ty specializes employee_ty by adding a new attribute "salary". New attributes declared in a subtype must have names that are different from the names of any attributes or methods declared in any of its supertypes, higher up in its type hierarchy. The following example inserts row into the FullTimeEmp table. Notice that the additional salary attribute is supplied.

A supertype can have multiple child subtypes called siblings, and these can also have subtypes. The following statement creates another subtype part_time_ty under Employee_ty.

```
CREATE OR REPLACE TYPE part_time_ty UNDER employee_ty (
rate Number(7,2),
hours Number(3))NOT FINAL;

CREATE TABLE PartTimeEmp of part_time_ty;
```

A subtype can be defined under another subtype. Again, the new subtype inherits all the attributes and methods that its parent type has, both declared and inherited. For example, the following statement defines a new subtype student_part_time _ty under part_time_ty. The new subtype inherits all the attributes and methods of student_part_time _ty and adds two attributes.

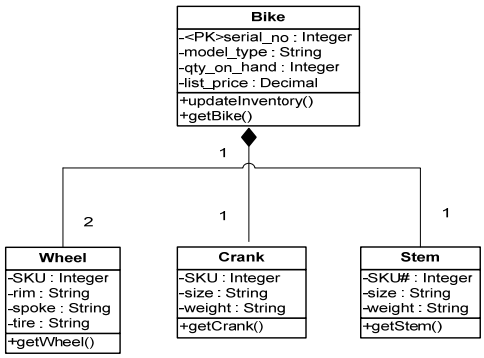
```
CREATE TYPE student_part_time_ty UNDER part_time_ty(school  VARCHAR2(20),
    year VARCHAR2(10));
```

Issues in Information Systems

Volume XII, No. 1, pp. 328-338, 2011

Integration of Nested Table Data

A nested table is a table that can be stored within another table. With a nested table, a collection of multiple columns from one table can be placed into a single column in another table. Nested tables allow users to embed multi-valued attributes into a table, thus forming an object.




```
CREATE TYPE wheel_type AS OBJECT(
SKU       VARCHAR2(15),
rim       VARCHAR2(30),
spoke     VARCHAR2(30),
tire      VARCHAR2(30));

CREATE TYPE crank_type AS OBJECT
(SKU       VARCHAR2(15),
crank_size  VARCHAR2(15),
crank_weight  VARCHAR2(15) );

CREATE TYPE stem_type AS OBJECT(
SKU       VARCHAR2(15),
stem_size     VARCHAR2(15),
stem_weightVARCHAR2(15)
);
```

The following statement creates nested table types: wheel_type, crank_type and stem_type:

```
CREATE TYPE nested_table_wheel_type AS TABLE OF wheel_type;
CREATE TYPE nested_table_crank_type AS TABLE OF crank_type;
CREATE TYPE nested_table_stem_type AS TABLE OF stem_type;
```

The following example creates the table named Bike with that contains four nested tables:

```
CREATE TABLE bike
( serial_no      INTEGER PRIMARY KEY,
  model_typeVARCHAR2(20),
```

```
    front_wheel nested_table_wheel_type,
      rear_wheel nested_table_wheel_type,
      crank nested_table_crank_type,
      stem nested_table_stem_type
)
 NESTED TABLE front_wheel
   STORE AS front_wheel,
 NESTED TABLE rear_wheel
   STORE AS rear_wheel,
 NESTED TABLE crank
   STORE AS nested_crank,
   NESTED TABLE stem
STORE AS nested_stem;
```

The following statement shows output of the nested tables created in the table Bike.

DESC Bike;

Name	Type
SERIAL_NO	NUMBER(38)
MODEL_TYPE	VARCHAR2(20)
FRONT_WHEEL	NESTED_TABLE_WHEEL_TYPE
REAR_WHEEL	NESTED_TABLE_WHEEL_TYPE
CRANK	NESTED_TABLE_CRANK_TYPE
STEM	NESTED_TABLE_STEM_TYPE

Object Views for Reuse of Relational Data

Object view allows users to develop object structures on the top of the existing relational tables. Object view creates a layer on the relational database so that the database can be viewed in terms of objects. The object view is a bridge that can be used to create object-oriented applications without modifying existing relational database schemas [3]. This enables you to develop OO features with existing relational data. It is a bridge between the relational database and OO programming. The object view is a bridge that can be used to create object-oriented applications without modifying existing relational database schemas. By calling object views, relational data can be retrieved, updated, inserted, and deleted as if such data were stored as objects. The following statement can retrieve Analysts as object data from the relational SalesOrder table. Using object views to group logically-related data can lead to better database performance. The following statements show how the object view reuses existing relational data.

Relational table: SalesOrder

ORD_ID	ORD_DATE	CUST_ID	EMP_ID
100	05-SEP-05	1	1000
101	05-OCT-05	1	1001

The following statements show how to create an object view on the SalesOrder relational table:

```
CREATE TYPE SalesOrder_type AS OBJECT(
sales_ord_idNUMBER(10),
ord_date DATE,
cust_id NUMBER(10),
emp_id NUMBER(10));

CREATE VIEW customer_order_view
OF SalesOrder_type
WITH OBJECT IDENTIFIER (sales_ord_id) AS
SELECT  o.ord_id, o.ord_date, o.cust_id
FROM salesOrder o
WHERE o.cust_id = 1;
```

The following SQL statement generates the output of the object view:

```
SELECT * FROM customer_order_view;
```

SALES_ORD_ID	ORD_DATE	CUST_ID
100	05-SEP-05	1
101	01-SEP-05	1

Integration with Object Interface

The beauty of ORDBMSs is reusability and sharing. Reusability mainly comes from storing methods in object types and performing their functionality on the ORDBMS server, rather than have it coded in each application. Sharing comes from using user-defined standard data types and methods to make database structure more standardized [1]. An object integration solution provides integrated view of data, regardless of where that data is actually located. It provides improved operational efficiency by allowing multiple applications to operate cooperatively. Ultimately, this can result in improved operational efficiency for the IT department, as well, by improving communication and cooperation between applications.

The structure of object type includes an interface and a body. The public interface declares the data structure and the method header shows how to access the data. This public interface serves as an interface to applications. The private implementation fully defines the specified methods.

Public Interface

Specification:
Attribute declarations
Method specifications

Private Implementation

Body:
Method implementations

The following statement displays the public interface of the object type name_type. The output of the name_type public interface shows attributes and method headers as follows:

DESC name_ty;

Name	Type
F_NAME	VARCHAR2(25)
L_NAME	VARCHAR2(25)
INITIALS	CHAR(2)

METHOD
MEMBER FUNCTION **FULL_NAME** RETURNS **VARCHAR2**

Although the user-defined methods are defined with object data within the object type, they can be shared and reused in multiple database application programs. This can result in improved operational efficiency for the IT department, as well, by improving communication and cooperation between applications.

CONCLUSION

The main contribution of this paper is to identify, present and implement object reuse and integration features of ORDBMSs in a real-world scenario. The presented case will promote awareness and recognition of object reuse and integration features of ORDBMS.

The significance of the paper is to provide readers with guidelines on how to design and implement ORDBMSs with object reuse and integration features. The use of ORDBMS to develop applications can enforce the reuse of varying user-defined object types, provide programmers' an integrated view of data and allow multiple database applications to operate cooperatively. Ultimately, this can result in improved operational efficiency for the IT department, increase programmers' productivity, lower development effort, decrease maintenance cost, reduce the defect rate, and raise the applications' reliability. With object reuse and integration, and a standard adherence access path, database application developers can create a de facto standard for database objects and multiple database applications to make database application development more productive and efficient.

The solution to the presented case can be generalized in either the projects of advanced database courses or industrial database application development. Major relational database vendors have upgraded their products to Object-relational database management systems (ORDBMSs) and ready to be used by industrial practitioners. Practically, ORDBMSs allows the users to take advantages of OODBMS and to maintain a consistent data structure in an existing relational database. Theoretically, as Stonebraker [4] predicted in his four-quadrant view of the database world more than ten years ago, ORDBMS may be the most appropriate DBMS that processes complex data and complex queries.

REFERENCES

1. Begg, C and Connolly, T. (2010). *Database systems: A Practical Approach to Design, Implementation, and Management*, 5th Ed. Addison Wesley.
2. Hoffer, J. A., Prescott, M. B. & McFadden F. R. (2009). *Modern Database Management*, 9th Ed. Prentice Hall.
3. Loney, K. & Koch, G. (2002), *Oracle 9i: The Complete Reference*, Oracle Press/McGraw-Hill/Osborne,
4. Stonebraker M. and Moore, D. (1996). *Object-relational DBMSs: the Next Great Wave*. San Francisco, CA: Morgan Kaufmann Publishers, Inc.
5. Wang, M. (2006). Teaching ORDB with UML Class Diagram in an Advanced Database Course, *Journal of Information Systems Education.* 17(1); pp.73-83.