IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4513

UML Dialect for Designing Object-Relational Databases

Leszek A. Maciaszek
Macquarie University, Sydney, Australia, leszek@mpce.mq.edu.au

Kin-Shing Wong
Tower Technology Pty Ltd, Sydney, Australia, ksw@towertechnology.com.au

ABSTRACT

The market trends indicate that the next generation database technology will be dominated by object-relational systems. *This shift to the new technology calls for* visual modeling techniques *to facilitate the design of object-relational database systems. Even though* Unified Modeling Language (UML) *is not currently equiped to manage this task, it can be extended for it.The paper defines design constructs needed for the development of an object-relational database system. The constructs include those that assist in the migration process from a relational to an object-relational database. Both data and procedural constructs are considered. Many UML extensions in the proposed UML dialect are derived by* stereotyping *existing UML elements. New classes are created to model object-relational constructs, and they are assigned their own distinct* icons. *Any special constraints on relationships between concepts in the extended UML are explained through practical examples. The mappings from design models to an object-relational implementation are exemplified.*

INTRODUCTION

The ***design*** is a low-level model of system's architecture and its internal workings. As opposed to systems *analysis*, the design is constrained by software/hardware platform on which the system is to be implemented. This means that to claim support for design phase of software lifecycle, a visual modeling language must understand the underlying *implementation model*.

"The ***Unified Modeling Language (UML)*** is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system" (Rumbaugh *et al*., 1999). As such, UML provides rather generic concepts that do not support well the *design* of systems. This includes "pure" object-oriented systems such as those that use C++, Java or Smalltalk as a programming language and an object database system to manage persistent objects. Each of these programming languages and databases has its own peculiarities that a generic UML does not accommodate.

The *generic UML* is also not expressive enough for designing relational and object-relational databases. As far as *relational database design* is concerned, UML is often not even supported by lower-engineering visual modeling tools and code generators. If UML is to be a player in the design of *object-relational databases* it will need to be extended to support the forthcoming *SQL3 standard* (Melton, 1996; Melton, 1998) and to capture the peculiarities of various *object-relational database implementations* (Oracle8, Informix Dynamic Server, DB2 UDB (Universal Data Base)).

UML has static, dynamic and architectural parts (Booch *et al.,* 1999; Rumbaugh *et al.,* 1999). The architectural constructs are used to arrange models into modules that partition a large system into workable components. These constructs are used for ***architectural design***, which determines solution strategies for the client and server aspects of a database system. UML contains generic constructs for representing such architectural decisions, in particular for organizing modules into packages and run-time elements into components.

The description of internal workings of each architectural module is called ***detailed design***. The detailed design is responsible for specification of algorithms and data structures for each module. These algorithms and data structures must be tailored to reinforcing and obstructive constraints of the underlying implementation platform.

UML captures information about the *static* (data) and *dynamic* (procedural) aspects of a system. A static view of the system is mostly captured in class diagrams. Dynamic aspects are expressed in use cases, state diagrams, activity diagrams and interaction diagrams. UML constructs and views support high-level modeling of object-oriented systems.

When it comes to capturing low-level design issues, UML offers special ***extensibility mechanisms*** - stereotypes, constraints and tagged values. A visual modeling tool can additionally allow for easy introduction of new graphical icons to represent stereotyped, constrained and tagged constructs. If such extensions target a particular implementation platform, such as an object-relational database, a new ***UML dialect*** is created. The ultimate aim of a UML dialect for designing object-relational databases (thereafter called the ***UML/ORDBS dialect***) is to be sufficiently expressive to support *lower-engineering activities*, including data and code generation, and reverse-engineering from existing databases.

BACKGROUND

Stonebraker and Brown (1998) predict that by the year 2005 the object-relational database market will be 50% larger than the relational database market. The major market forces in favor of object-relational databases are software requirements of new multimedia applications and the growing need of business systems to support querying of complex data for decision-making functions.

The ***object-relational database (ORDB)*** model extends the relational database (RDB) model by providing a richer object-oriented type system and by adding constructs to SQL for complex queries. A forthcoming SQL standard (SQL3) provides the direction for object-relational database implementations. Apart from vendors specifically targeting this market (eg. UniSQL and Omniscience), major relational database vendors - like Oracle, IBM and Informix - already ship their ORDB products to the market.

A practical implication for any ORDB vendor is to be compatible with relational technology, even though this may conflict with some SQL3 guidelines. As a result, ORDB products are significantly more complex than SQL-3 standard may stipulate. For example, Oracle8 (Koch and Loney, 1997) supports three different (yet compatible) database solution strategies:

- a ***typical relational*** solution based on Oracle's *built-in datatypes*,
- a *"**pure" object-relational*** solution based on *object tables*,
- an ***"evolutionary" object-relational*** solution based on *abstract datatypes* and *object views* defined on an existing relational schema.

The need to accommodate relational and object-relational constructs in a single model leads not only to some peculiar solutions, but it forces (at least initially) an abandonment of some important object-oriented features (eg. inheritance). There are significant differences between ORDB products (such as Oracle8, Informix Dynamic Server and UDB), and between ORDB products and the forthcoming SQL3 standard (Muller, 1999).

Needless to say that ***design of object-relational databases*** is a challenging task, even for an experienced database designer familiar with object-oriented modeling. The initial difficulty lies in structuring of complex objects. A multi-level structure of abstract datatypes needs to be established before the stored objects (*object tables*) can be defined. But the ORDB vendors are incompatible on even such fundamental issues.

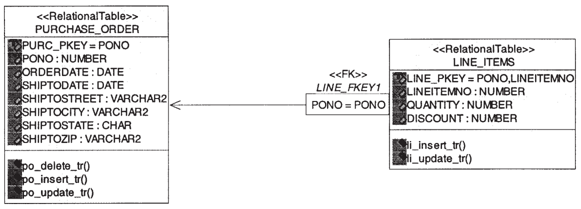
Even though SQL3 includes *generalization* of types into an inheritance structure, Oracle8 and UDB do not support inheritance as yet. Most ORDB vendors support *multimedia data types* (cartridges in Oracle, datablades in Informix, extenders in DB2 UDB), but they have their own unique ways of integrating these extensions into the SQL typing systems. The approaches to the *relaxation* of the *first normal form* also vary in ORDB products. SQL3 allows *arrays* as column values but it does not allow *nested tables*. Oracle8 has nested table and varying arrays. Collection data types supported by Informix include sets, multisets, and lists. The solutions to linking tables through *pointers* or *references* are not "standardized" either.

The *incompatibilities*, as defined above, make it difficult to propose a single set of UML extensions to cater for ORDB design. This paper takes an approach of relying on a *specific technology* rather than on vague and contradictory principles. An advantage is that we can offer a UML/ORDBS dialect that is coherent and has sufficient technical depth. We believe that this is a better strategy and the same UML extensibility mechanism can be readily used to provide variations to our UML/ORDBS dialect so that other products can be accommodated. The UML/ORDBS dialect presented in this paper targets ***Oracle8***.

As mentioned, UML offers few *extensibility mechanisms* that give ability to tailor the modeling language to a database software platform, yet still share the concepts that are generic and common to other components of an application (such as the client application programming interface). The major extensibility mechanism in UML is ***stereotype***. "A stereotype represents a variation of an existing model element with the same form (such as attributes and relationships) but with a different intent. ... A stereotyped element may have additional constraints, beyond those of the base element, as well as a distinct visual image." (Rumbaugh *et al.*, 1999).

Figure 1 is an example of a simple class diagram where classes are stereotyped as relational tables (the stereotyped labels are placed within matched guillemets, which are the quotation marks used in French and some other languages). The constraint placed on the relationship is also stereotyped as FK (foreign key).

Figure 1. Stereotypes as UML extensibility mechanism



An intent of stereotypes and other lesser extensibility mechanisms of UML is that a generic modeling element is still an ordinary element but with some differences in semantics. This is a correct approach if a new UML dialect is to retain its genericity and commonality to all domains, while at same time enabling the definition of specific constructs. Vendors of visual modeling (CASE) already offer some *add-ins* that utilize stereotypes to target specialized domains. In particular, Rational Corporation provides an add-in for Oracle8 in its ***Rational Rose*** visual modeling tool (Rational, 1998). The UML/ORDBS dialect, that we propose in this paper, uses Rational Rose / Oracle8 add-in as a starting point.

UML EXTENSIONS FOR DESIGNING OBJECT-RELATIONAL DATABASES

A built-in knowledge of the ORDBS constructs is a prerequisite for our UML/ORDBS dialect. Static, dynamic and architectural constructs have to be identified. The most important among these constructs is the *abstract data type*, which is known as the **user-defined type** (UDT) in SQL3, the *distinct type* in DB2 UDB and Informix Dynamic Server, and the *object type* in Oracle8.

Object type is the basis for defining object tables. It gives meaning to data, it defines operations on these data, it tells you how to compare and convert its own objects, it can be used for defining nested tables and for referencing objects in other tables. However, inheritance and encapsulation are not supported in Oracle8 and, therefore, not considered in our UML/ORDBS dialect (this limitation of the dialect is not consequential because the generic UML defines inheritance and encapsulation constructs).

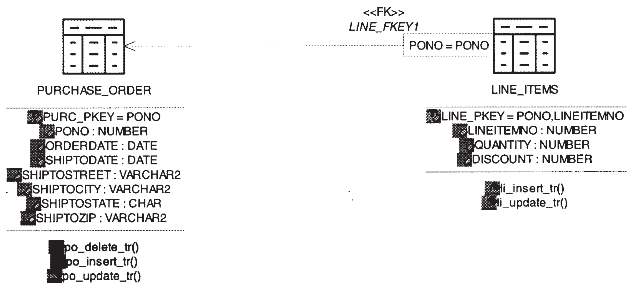
The constructs supported by our UML/ORDBS dialect are classified in six groups and discussed in the remainder of this section. The examples that illustrate the usage of these constructs are drawn from the Oracle8 Purchase Order tutorial (Oracle, 1998). When needed, the tutorial is appropriately extended. The groups of constructs are:

- conventional relational constructs (tables, views, triggers, etc.)
- packages, stored procedures and functions
- object types and object tables
- collections (varying arrays and nested tables)
- object views and INSTEAD OF triggers
- clients and transactions

Conventional relational constructs

The UML/ORDBS support for conventional relational constructs is based on the stereotypes available in the Rational Rose / Oracle8 tool. The major difference is the introduction in UML/ORDBS of new graphical icons. Figure 2 illustrates two *relational tables*: PURCHASE_ORDER and LINE_ITEMS. The *referential integrity* is based on the purchase order number (PONO) columns, and it is enforced by a number of *triggers* (such as po_delete_tr). Other aspects of the example should be self-explanatory.

Figure 2. Conventional relational constructs in UML/ORDBS



Packages, stored procedure and functions

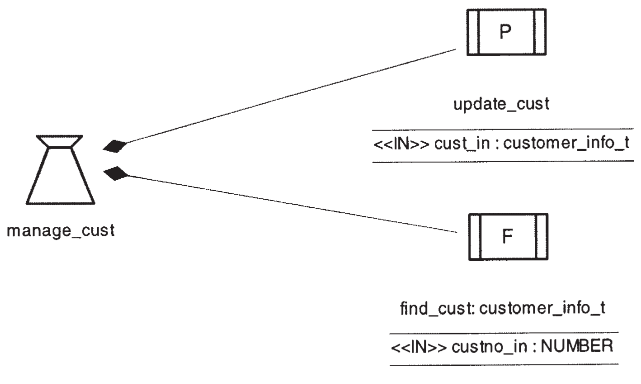
Relational and object-relational databases allow storing programs in the database. These programs are called stored procedures or functions. *Stored procedures* cannot return values to the calling program, but *functions* can. *Packages* are larger program units that can contain procedures, functions, variables, and SQL statements (Koch and Loney, 1997). The dot notation is used to execute a procedure or a function within a package (package_name.procedure_name).

Below is an example of package specification with one function and one procedure. The function find_cust allows to find a customer object given customer number (custno_in) as input parameter. The procedure update_cust enables to update customer information in the database. It expects customer object (cust_in) as input parameter.

```
CREATE OR REPLACE PACKAGE manage_cust AS
        FUNCTION find_cust(custno_in IN NUMBER) RETURN customer_info_t;
        PROCEDURE update_cust(cust_in IN customer_info_t);
END manage_cust;
/
```

Figure 3 shows a design of package manage_cust. There are three graphical icons to represent a package, procedure and function. The package contains procedure update_cust and function find_cust. The containment is expressed with UML *aggregation by value* relationship.

Figure 3. Packages, stored procedures and functions in UML/ORDBS


P
update_cust
<<IN>> cust_in : customer_info_t
manage_cust
F
find_cust: customer_info_t
<<IN>> custno_in : NUMBER

Object types and object tables

As explained earlier, the *object type* in Oracle8 corresponds (loosely speaking) to the concept of *class* and *abstract data type*. It defines a data structure (*attributes*) and operations (*methods*) that act on these attributes. Object type is just a template for object creation; it itself does not hold any data. *Objects* can be stored *persistently* in *object tables* (each row is an object). *Transient* objects are stored in programming language variables (Feuerstein, 1997).

The code below defines an object type (purchase_order_t) and an object table (purchase_tab). The object type contains three *scalar attributes* (pono, orderdate and shipdate). The attribute custref is a *reference* to an object of type customer_info_t. The types of line_item_list and shiptoaddr are other user-defined object types.

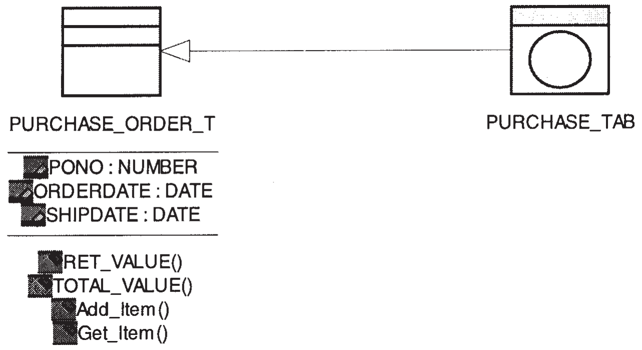
The purchase_order_t houses three functions and one procedure. The function ret_value is a MAP function that specifies (in the function body that is not shown here) how to translate or "map" a purchase_order_t object into a scalar datatype that the ORDBS knows how to compare. The function total_value returns the total amount of a purchase order. The procedure add_item adds a new order item to the purchase order, and the function get_item finds an order_item. The PRAGMA RESTRICT_REFERENCES clauses control the ability of methods (functions or procedures) to modify the database.

The next statement defines the object table purchase_tab. The statement places the scope on the custref reference column. The scope says that the references can refer only to the customer_tab objects. The object table also has a nested column line_item_list. The line_item_list objects are stored in a separate table po_line_tab.

```
CREATE OR REPLACE TYPE purchase_order_t AS OBJECT (
   pono                          NUMBER,
   custref                       REF customer_info_t,
   orderdate                     DATE,
   shipdate                      DATE,
   line_item_list                line_item_list_t,
   shiptoaddr                    address_t,

   MAP MEMBER FUNCTION ret_value RETURN NUMBER, PRAGMA
   RESTRICT_REFERENCES (ret_value, WNDS, WNPS, RNPS, RNDS),

   MEMBER FUNCTION total_value RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES (total_value, WNDS, WNPS),

   MEMBER PROCEDURE add_item(stock_ref REF stock_info_t, q NUMBER,
   d NUMBER), PRAGMA RESTRICT_REFERENCES (add_item, WNDS,
   WNPS, RNPS, RNDS),

   MEMBER FUNCTION get_item(i BINARY_INTEGER) RETURN
   line_item_t, PRAGMA RESTRICT_REFERENCES (get_item, WNDS, WNPS,
   RNPS, RNDS),
);

CREATE TABLE purchase_tab OF purchase_order_t (
   PRIMARY KEY (pono),
   SCOPE FOR (custref) IS customer_tab
   )
   NESTED TABLE line_item_list STORE AS po_line_tab ;
```

Our UML/ORDBS dialect offers two new icons to represent object types (such as purchase_order_t) and object tables (such as purchase_tab). We use *generalization relationship* to say that purchase_tab is a kind of purchase_order_t. The methods are physically contained in the type, but we could alternatively use the *aggregation by value* relationships (as in Figure 3) to express the same semantics.

Figure 4. Object types and object tables in UML/ORDBS



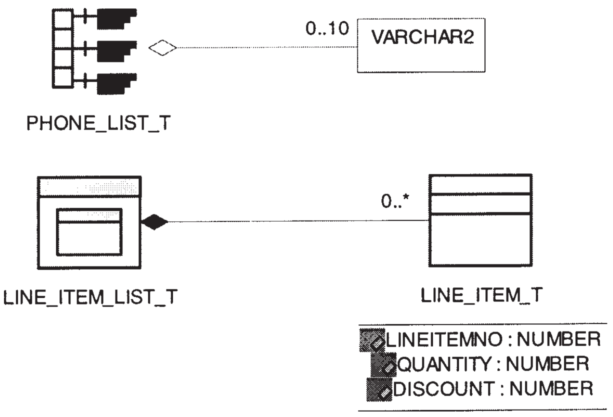
Collections (varying arrays and nested tables)

Collections are used to store (and retrieve) nonatomic data in a single column of an object table. A varying array *VARRAY* is an ordered set of data elements and it has a predefined maximum size. *A nested table* is an unordered set of data elements, stored in a special auxiliary table called a *store table*.

The VARRAY phone_list_t defined below is a varying array of maximum ten elements of scalar datatype varchar2. The nested table line_item_list_t contains elements of the object type line_item_t.

```
CREATE TYPE phone_list_t AS VARRAY(10) OF VARCHAR2(20) ;
/

CREATE TYPE line_item_t AS OBJECT (
   lineitemno NUMBER,
   stockref REF stock_info_t,
   quantity NUMBER,
   discount NUMBER
   ) ;
/

CREATE TYPE line_item_list_t AS TABLE OF line_item_t;
/
```

Figure 5. Collections (varying arrays and nested tables) in UML/ ORDBS



Graphical icons for a VARRAY and a nested table in the UML/ORDBS dialect are shown in Figure 5. *Aggregation by reference* is used to link the *scalar* datatype (note the icon) - the same scalar datatype can be a component of many different collections. The line_item_t objects, on the other hand, are *aggregated by value* with the line_item_list_t nested table because each component object can belong only to one collection.

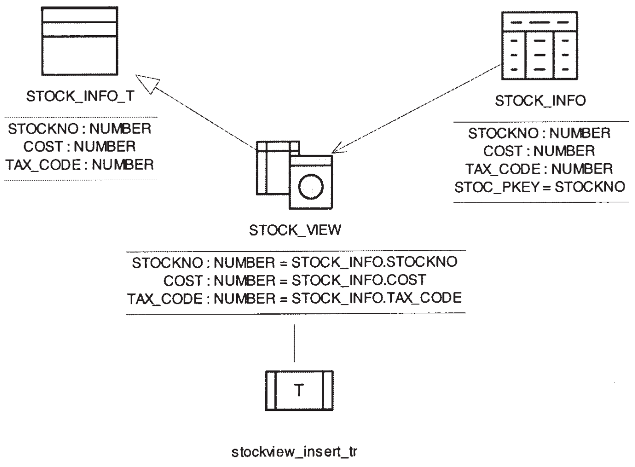
Object views and INSTEAD OF triggers

An *object view* is a virtual object table. Object views facilitate transition of applications from a relational database to an object-relational database. An object view is an updatable (*INSTEAD OF triggers*) named query on relational and/or object tables.

The code below presents a simple example of an object view (stock_view) that contains relational table stock_info. An INSTEAD OF trigger instructs the ORDBS that any inserts on object view need to be executed as programmed in insert statement in the trigger body.

```
CREATE OR REPLACE VIEW
stock_view OF stock_info_t WITH OBJECT OID(stockno) AS
   SELECT *
    FROM stock_info ;

CREATE OR REPLACE TRIGGER
stockview_insert_tr INSTEAD OF INSERT ON stock_view
BEGIN
   INSERT INTO stock_info VALUES (
```

Figure 6. Object views and INSTEAD OF triggers in UML/ORDBS



```
        :NEW.stockno,
        :NEW.cost,
        :NEW.tax_code );
END ;
/
```

Figure 6 shows two new icons: for object view (stock_view) and for trigger (stockview_insert_tr). Attribute part of object view defines how a query retrieves data from relational table (an arrowed line indicates *retrieves relationship*). A *generalization relationship* is used to specify that stock_view is a kind of stock_info_t. An *association relationship* links INSTEAD OF trigger with its object view.

Clients and transactions

An object-relational database application consists of inter-communicating objects that perform business transactions (normally initiated from a client application). Therefore, we need to have at least two more UML constructs to model the flow-of-control in client/server programs: *client* and *transaction*.

A *client* construct is an abstract concept. In reality, client objects represent various view and control objects as available in a particular GUI framework (such as primary window, dialog box, menu item, etc.). As we do not attempt in this paper to define a UML dialect for GUI part of a system, we will combine all these objects under a single "umbrella" object called *client*.

A *transaction* construct represents the notion of the database transaction as a unit of the database consistency. A transaction either completes satisfactorily all its operations and it then commits the changes to the database, or the changes must be rolled back.

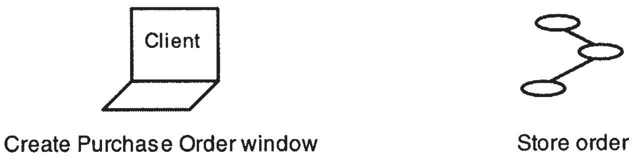
Figure 7 shows graphical icons for a client (Create Purchase Order window) and for a transaction (Store order) in our UML/ORDBS dialect. The usage of these icons in UML/ORDBS is illustrated in the case study in the next section (Figure 10).

CASE STUDY - PUTTING IT ALL TOGETHER

In the previous section, we defined the primitive constructs in our UML/ORDBS dialect. In this section, we put it all together and show how these constructs can be used on a small case study. The case study extends the examples in the tutorial provided in Oracle8 documentation (Oracle8, 1998).

The application domain refers to a purchase order (PO) application handling customers, stock of products for sale and purchase orders. Customers place orders on stock items. A stock item can appear on many purchase orders. A purchase order can have any number of line items, but each line item refers to a single stock item.

Figure 7. Clients and transactions in UML/ORDBS



Designing ORDB schema with UML dialect

Figure 8 shows a static model for the PO application. It shows object types, object tables, a nested table and a VARRAY. These graphical objects are linked by aggregation, generalization and reference relationships.

Aggregation relationships are used to capture the containment of object types. For example, purchase_order_t contains an attribute which type is defined as address_t. *Reference relationships* (supported by Oracle8 REF operator) are shown as arrowed lines pointing to the referenced row objects. Object tables are derived as subclasses of object types and are therefore linked by *generalization relationships*. For example, each row of customer_tab is a customer_info_t object.

Designing ORDB programs with UML dialect

A typical functionality of a program interacting with a database is best captured by the acronym *CRUD* - create, read, update, delete persistent objects. A question arises how to manipulate *persistent objects* in object-relational database. Feuerstein (1997) describes four approaches (in the context of Oracle8 database):

1. Handle persistent objects from client application by sending SQL statements (SELECT, INSERT, UPDATE, and DELETE) to server database for execution. This approach does not take advantage of object types (and no methods have to be defined).
2. Still handle persistent objects from client application but invoke constructor methods to insert data and various update methods to update data. Use SELECT and DELETE as in point 1.
3. Implement all data manipulations via object methods. This method limits reuse because (in Oracle8 context) it effectively ties each object type to a single object table.
4. Design object methods to avoid direct references to persistent object tables, instead acting only on its own object and on data exchanged via method arguments. Packages can be constructed to introduce a level of indirection between method invocation and persistent storage, thus allowing for type reuse.

The last approach seems to be the most attractive from object-oriented perspective. Below is a specification of a package (manage_po) that uses this approach to handle the CRUD operations on purchase order objects.

```
CREATE OR REPLACE PACKAGE manage_po AS
    /* This package depends on the following
     * SEQUENCE                 po_seq
     * OBJECT TYPE              customer_info_t
     *                          purchase_order_t
     * OBJECT TABLE             purchase_tab
     */
```

Figure 8. UML/ORDBS schema design

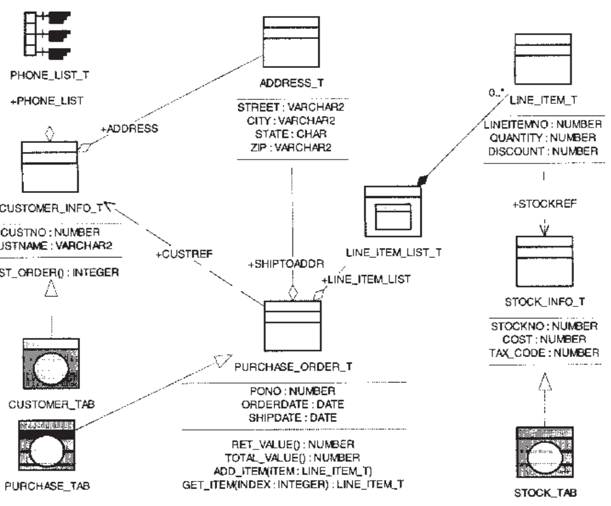
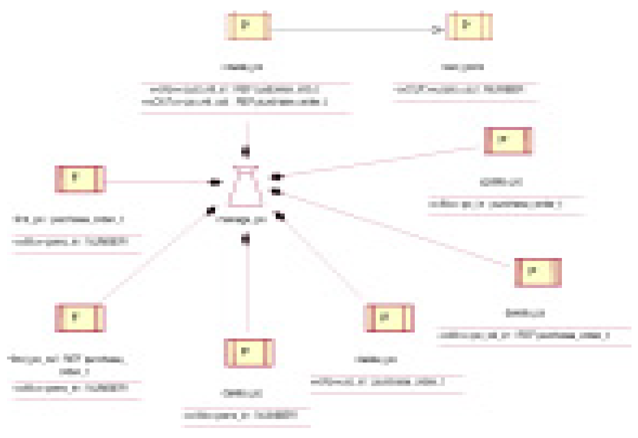


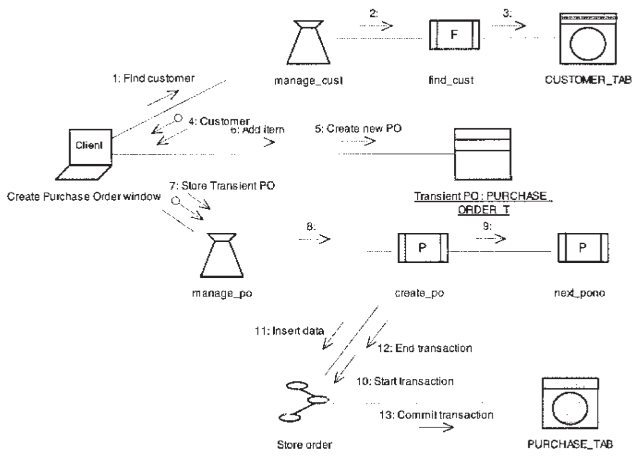
Figure 9. UML/ORDBS program design



```
/* return the next new purchase order number */
PROCEDURE next_pono(pono_out OUT NUMBER);

/* create a new purchase order
 * a new purchase order number will be assigned automatically
 */
PROCEDURE create_po
(   cust_ref_in                     IN     REF customer_info_t,
    po_ref_out                      OUT  REF purchase_order_t
);

/*
 * find purchase orders when provided with pono_in
 * return an object or a reference to the object
 */
FUNCTION find_po(pono_in IN NUMBER) RETURN purchase_order_t;
FUNCTION find_po_ref(pono_in IN NUMBER) RETURN REF
purchase_order_t;

/* update the given purchase order */
PROCEDURE update_po(po_in IN purchase_order_t);

/*
 * overloaded delete purchase order functions
 */
PROCEDURE delete_po(pono_in IN NUMBER);
PROCEDURE delete_po(po_in IN purchase_order_t);
PROCEDURE delete_po(po_ref_in IN REF purchase_order_t);

END manage_po;
/
```

The above package can be designed using our UML/ORDBS dialect as shown in Figure 9. *Aggregation by value* relationships are used to identify all component procedures and functions within the package. The only exception is the procedure next_pono, which is "indirectly" contained in the package because it is invoked by the procedure create_po whenever a new purchase order number needs to be generated. The arrowed line signifies *invocation relationship*.

Figure 10. UML/ORDBS client/server flow-of-control design



Designing client/server flow-of-control with UML dialect

In the previous two sections, we showed how the static structure and the architectural design of each program could be visualized in the UML/ORDBS dialect. However, a *flow-of-control* between client and server objects to accomplish a *business transaction* has not been shown yet.

Figure 10 shows an *object collaboration diagram* that represents such a flow-of-control for the creation of a new purchase order. A client object named Create Purchase Order window activates the process. The first task is to retrieve a customer for whom the purchase order is to be created. This is initiated by a message sent to *package* manage_cust, and more specifically to *function* find_cust that returns a customer object from *object table* customer_tab.

Once the customer is displayed on *client window*, a *transient object* of type purchase_order_t is instantiated and line items are added to it. After purchase order is created (still in the program's memory and visible in a client window), a *client object* requests that the object be persistently stored in object-relational database. The client passes Transient PO as an argument of *procedure* manage_po.create_po. This procedure requests that another procedure (next_pono) generates next purchase order number under which PO can be stored. The *database transaction* is only started now and, if successful, it commits changes to object table purchase_tab.

CONCLUSIONS

A new UML dialect for designing object-relational databases was described in this paper. The dialect supports directly object-relational constructs of Oracle8, but it could be easily customized for other ORDBS-s. The UML extensions, including the introduction of graphical images, were implemented on top of Rational Rose. The UML/ORDBS dialect has been integrated into Rational Rose add-in for Oracle8 and it can provide a limited capability for Oracle8 code generation.

As yet, the capabilities of more extensive *forward and reverse engineering* with Oracle8 have not been built into our UML/ORDBS dialect. This is because of inherent limitations in Rational Rose (and most other visual modeling tools). For example, Rational Rose does not fully support *stereotyping* outside of class models. Stereotyping UML elements in models that do not visualize classes is difficult and occasionally impossible (eg. in sequence diagrams, collaboration diagrams or state diagrams, where classes are not the modeling elements).

More importantly, to create a sufficiently expressive UML dialect, a tool has to allow for appropriate extensions of its own internal *metamodel* (and a *scripting language* to program such extensions needs to be provided by a tool vendor). Rational Rose does not give this capability. As a result, the stereotyped elements are relatively "mindless" and tasks such as code generation are not completely obtainable. Unless vendors of visual modeling tools "open up" the metamodels to system developers, so that they can fully customize the tools to their needs, the industry adoption of UML dialects extended for system design (as opposed to system analysis) will be sluggish.

REFERENCES

Booch, G. Rumbaugh, J. and Jacobson, I. (1999): *The Unified Modeling Language User Guide*, Addison-Wesley, 482p.

Feuerstein, S with Pribyl, B. (1997): *Oracle PL/SQL Programming*, 2nd ed., O'Reilly & Associates, Inc., 987p.

Koch, G. and Loney, K. (1997): *ORACLE8. The Complete Reference*, Osborne McGraw-Hill, 1300p.

Melton, J. (1996): Assessing SQL3's New Objects Directions. A Shift in the Landscape, *Database Prog. & Design*, Aug., pp.51-54.

Melton, J. (1998): SQL3 Moves Forward, *Database Prog. & Design*, 6., pp.63-66.

Muller, R.J. (1999): *Database Design for Smarties*, Morgan Kaufmann, 442p.

Oracle (1998): *Oracle8 Server Application Developer's Guide*, on-line documentation.

Rational (1998): *Rational Rose 98. Using Rational Rose / Oracle8*, Rational Corp., 100p.

Rumbaugh,J. Jacobson, I. and Booch, G. (1999): *The Unified Modeling Language Reference Manual*, Addison-Wesley, 550p.

Stonebraker, M. Brown, P. with Moore, D. (1998): *Object-Relational DBMSs Tracking the Next Great Wave*, 2nd ed., Morgan Kaufmann, 266p.