December 2002

IMPLEMENTING DATABASE SOLUTIONS: OBJECT, RELATIONAL, OR BOTH?

Paige Rutner
University of Arkansas

Susan Rebstock-Williams
Georgia Southern University

E. Reed Doke
University of Arkansas

Implementing Database Solutions: Object, Relational, or Both?

Paige S. Rutner
University of Arkansas
prutner@walton.uark.edu

Susan R. Williams
Georgia Southern University
rebstock@gasou.edu

E. Reed Doke
University of Arkansas
rdoke@walton.uark.edu

Abstract

Object –oriented databases have been proposed as both an alternative and as a supplement to relational database technology. This work presents an example of an implementation of both a relational database and an object-oriented database in conjunction with an object oriented system. Key features of the two approaches are presented, as are advantages and disadvantages of both.

Introduction

Dr. Edward Codd of IBM first proposed the relational database model in 1970 and it was a revolutionary solution to a number of concerns, one of which was data storage (Codd 1970). Among those concerns were issues with program-data dependence where files are stored in the context of the application that uses that data. The relational model provides data independence by hiding the storage and retrieval mechanisms for data access and isolating them from applications code (McFadden, Hoffer et al. 1999). Other advantages of relational systems include a sound theoretical basis in relational algebra and easy support for ad hoc query processing. Since their inception in the 1970s, relational systems have dominated the market and currently account for the vast majority of database systems in existence.

Evolving demands for storage, however, have prompted the creation of different types of database systems; chief among them is the object-oriented database (OODB). While it is important to note that there are currently several variations on the OODB theme available commercially, each attempts to implement a means of achieving object persistence. Object-oriented programming has been around since the 1960s. However with the growth of the Internet as a primary driver, the development of object-oriented systems has exploded during the last decade. This dramatic increase in OO systems development has been enabled by the introduction of pure OO languages such as Java, C#, VB.Net and standardized modeling tools such as the Unified Modeling Language (UML).

Accompanying the increase in OO systems development has been the need to store and retrieve the complex objects and data types that are generated and used by these systems. While it is possible to store objects in relational databases, in a situation with any amount of volume or complexity of objects, the additional coding and the need to shift from an OO to a relational system combine to make it a sub optimal solution. The term "impedance mismatch" has been used to describe the problems encountered when mixing object-oriented systems with relational databases (Finn 2001). Object serialization, which is a technique used by object-oriented languages to describe the state of an object in an ordered series of bytes, has been used to make objects persist beyond the process that created them. This effort comprises a large amount of the code for systems combining OO and relational technologies (Jackson 1991; Jackson 1999). Another method for making objects persistent is to decompose them into their attribute values, which are then stored in the database (Doke, Satszinger et al. 2002). When the object is needed again, it is recreated from those values.

While these approaches will certainly suffice in many situations, it is easy to imagine that at some threshold of volume and complexity of objects, they would become unwieldy. Two solutions that have been advanced to deal with these complex objects are object-oriented databases (OODB) and object-relational databases (ORDB). This research presents an overview of these two approaches, provides examples using relational and Object-oriented databases, and suggests future trends for these technologies.

What Is an Object Relational Database?

Object relational databases (ORDB) represent a compromise between the traditional relational model and the new object-oriented systems. Also called, hybrid databases, extended relational databases and universal servers, they offer the ability to handle such OO features as extended data types, complex objects and inheritance (McFadden, Hoffer et al. 1999). For example, Oracle 9 allows for the creation of abstract data types from one or more atomic attributes. A user could create an address data type as a combination of columns:

Street	VARCHAR2(50)
City	VARCHAR2(25)
State	CHAR(2)
Zip	NUMBER

This approach promotes reuse of the data type once it is constructed as well as adherence to standards for the data (Koch and Loney 1997).

Proponents of the object-relational model suggest that there are certain features that should be present in such systems (Jackson 1999):

- Support for base type extensions;
- Support for complex objects;
- Support for inheritance;
- Support for production rules.

Who Is Using Object-Relational Databases?

Whether they realize it or not, just about everyone who is using a current relational database product is using an ORDB. Most of the major vendors have incorporated some type of extender technology into their systems to allow for object handling (McFadden, Hoffer et al. 1999). Like any compromise approach however, there is a cost to this combination of technologies. Critics of ORDBs argue that queries to the systems will become increasingly more difficult to optimize as the volume of user defined functions and data types increases (Jackson 1999).

What Is an Object-Oriented Database?

A clear definition of the term "object-oriented database" is somewhat elusive. Essentially, it is a database that is centered around an object model as opposed to one that is built on a relational model. There are several OODB vendors, each of whom seems to have implemented a slightly different solution to the object storage problem. OODBs are generally seen as an alternative to relational systems rather than an extension of them (Jackson 1999). The Object-Oriented Database System Manifesto includes a list of features that are considered to be mandatory for a system to be termed an OODB (Atkinson, Bancilhon et al.).

According to Jackson, "an OODB is first and foremost a database" (1991). The Manifesto recognizes this by including several features that are not limited solely to the object-oriented arena. Following is a brief discussion of these features.

- Persistence
  — One of the main functions of a database is to retain the data after it has been processed. The data should remain even when the system is not in use.

- Secondary storage management
  - This feature includes functions that are invisible to the user, such as index management and query optimization. These functions should not require explicit action by the user.

- Concurrency
  - Concurrency controls contain a mechanism to protect the integrity of the data when multiple users are accessing the system. The management of multiple users should meet the standards set by current database management systems.

- Recovery
  - There are many mechanisms in place that allow a database to return to its original state after a hardware or software failure. Again, this feature in OODBs should measure up to the benchmarks set by current DBMSs.

- Ad hoc query facilities
  - The specific form in which users perform ad hoc queries is not specified, rather the Manifesto provides descriptions of the functionality that is required. The user should be able to concisely and efficiently query the database in a manner that is independent of the specific application being used.

The preceding features are those that are considered DBMS elements of the OODB. In addition, there should be specifically object-oriented features as summarized below.

- Complex objects
  - The need for complex objects is one of the driving forces behind OODBs. Constructors such as sets, tuples, lists, bags, and arrays are used to assemble simpler objects into more complex ones. In contrast with the relational model, the constructors in the object model should be applicable for any object.

- Object identity
  - An object should possess a unique and immutable identifier that is independent of the objects contents or value. So, for example, if a system includes a Person object, that person could change his recorded name, address, telephone number, etc. and still retain the same identity in the system.

- Encapsulation
  - The encapsulation of data and behavior within an object is a useful characteristic of OO systems. It allows for information and implementation hiding.

- Types and classes
  - Classes are the main building blocks of an OO system. Classes in an OODB system will typically contain some way to create instances of the class as well as some type of "warehouse" function that maintains a listing of all instances of that class. The set of all instances of a class is referred to as its extent or extension and it allows, among other things, for the manipulation of all instances of the class in a single operation.

- Class or type hierarchies
  - Hierarchies express inheritance in an OO system. This feature allows for increased reuse of code by allowing the maximum possible number of classes in the inheritance hierarchy to share common methods.

- Overriding, overloading and late binding
  - Overriding and overloading are both mechanisms whereby methods with the same names may be assigned different behaviors. Overriding exists between methods in different classes in an inheritance hierarchy. Overloading consists of methods with the same name but different return types or parameter lists contained within a single class. Late binding is the mechanism that allows the specific method function to be determined at run time rather than compile time.

- Computational completeness
  - The ability to express computable functions in the data manipulation language of the database is something that is not present in SQL, the standard for relational databases. In OODBs, this feature is typically enabled by using a standard OO programming language for data manipulation.

- Extensibility
  - The user should be able to define new data types that will function in the same manner as system defined data types.

Table 1

Data type	Description	Example applications
Image	Bit-mapped representation	Documents, photographs, medical images, fingerprints
Graphic	Geometric objects	CAD, CAM, CASE, presentation graphics
Spatial	Geographic objects	Maps
Recursive	Nested objects	Bills of materials
Audio	Sound clips	Music, animation, games
Array	Subscripted variables	Time series data, multidimensional data
Computationally intensive	Data requiring extensive computations	Data mining, financial instruments (e.g. derivatives)

Adapted from McFadden, Hoffer, Prescott, Modern Database Management 5th Edition

Who Is Using OODBs?

Object-oriented databases are well described as a niche technology. The strengths of OODBs include the ability to accurately model complex objects and complex relationships, which makes them well suited to applications such as geographic information systems (GIS), computer integrated manufacturing (CIM), CAD/CAM systems, and hospital patient care tracking systems. OODBs have also been used to model complex financial instruments such as derivatives. Inheritance allows for the easy creation of new instruments and the OO approach helps to capture the instruments' structure and behavior (Leavitt 2000). Table 1 lists some of the data types and applications for which OODBs may be useful.

A Simple Example

This example uses a simple scenario designed to compare and contrast the use of an OODB and of an RDB in an object-oriented system. The example is designed based on the three-tier architecture recommended by Doke, et al. in which the classes in the system are separated into three distinct categories: problem domain classes which represent the user's environment, graphical user interface (GUI) classes which interact with users, and data access classes which interact with the database (2001). The main benefit of this approach is that it allows modification to various components of the system with a minimum of disruption to other elements. In our example, the user interface and problem domain classes used in the OODB and RDB situations are virtually identical. The difference between two approaches is found in the structure of the data access classes. These classes were written in Java and the code for all of them are contained in Appendix A.

This example uses two problem domain classes: Customer and Order represented in the class diagram in Figure 1. Each problem domain class contains standard constructor and accessor methods, in addition to a method named tellABoutSelf() that returns formatted attribute values. A Tester class is used to replace GUI classes that would ordinarily be in place to allow users to interact with the system. The Tester class instantiates customers and orders to be added to the database, retrieves individual entries for customers and orders from the database, and retrieves all orders for a given customer from the database.

The data access classes provide data storage and retrieval services and are responsible for all interactions with the database. These classes reflect the specific data storage technology being used. The relational database implementation here uses Microsoft Access to store the data in two tables, Customer and Order, linked by the customer's phone number. In order to store object data in the Access database, the objects are decomposed into their component attributes and then the individual attribute values are stored. When the objects are retrieved, the data access class queries the database for the appropriate attributes and then reassembles them into a new instantiation of the object.

The object-oriented database implementation illustrated for this example uses the FastObjects database produced by Poet Software. This is a pure object-oriented database that maps the objects created by the problem domain classes onto objects in the database. The objects are never disassembled and reassembled, as is the case when they are stored in a relational database. Rather, they are simply saved with their attributes and relationships intact.

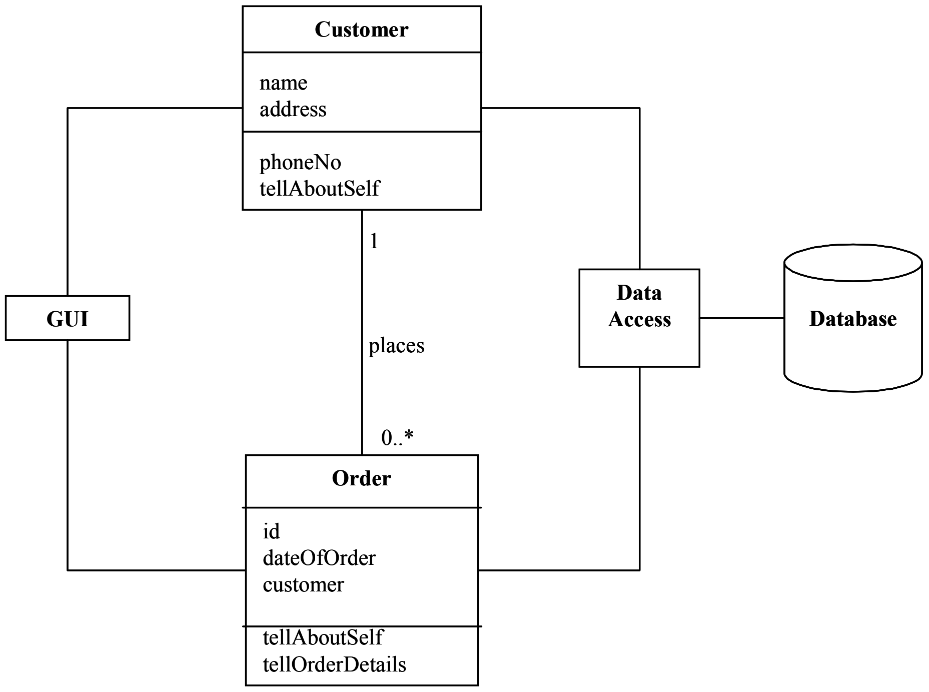

Figure 1

A comparison of the two data access classes needed to work with the two different databases illustrates the relative ease of using an object-oriented database in an object-oriented system. Consider for example, the Data Access (DA) method named findCustomer that queries the database for information about a customer who is identified by their phone number. This method contains code to define and execute an SQL query. If the query returns results, the relevant customer attributes are extracted from the result set, Customer is instantiated, and the instance is then returned to whichever class requested it. The findCustomer method in the DA class using the relational database requires ten non-comment lines of code to accomplish its task plus additional code for error checking. Figure 2 illustrates the sequence diagram for this process.

Contrast this with the same method using the FastObjects database. It requires only two non-comment, non-error checking lines of code to retrieve the customer information: One line queries the database and the other returns the object to the calling problem domain class. Methods performing other functions in the data access classes are also more efficiently coded in the OODB example as opposed to the RDB example. Figure 3 shows the sequence diagram for these events.

The obvious advantage of the OODB approach in conjunction with an object-oriented system is illustrated in this very simple example. The difference in an industrial strength situation where the complexity of the objects is increased by orders of magnitude would be even more pronounced.

Given this increase in coding efficiency, it is not unreasonable to ask why any organization with a need to store complex objects would not opt to use an OODB. The answer is that OODBs still represent a small niche of the database market. The relational database model has been the dominant paradigm for such a length of time that most organizations have a considerable investment of time and resources in them, as well as large amounts of stored data in those systems. The costs of converting all of that existing data to a new type of object storage system would be staggering.

The benefit of the three-tiered architecture described previously is that the only classes affected by a change in the underlying data storage mechanism are the data access classes. One likely scenario is that an organization might choose to use both an existing relational database system for their more traditional data storage and supplement it with an object-oriented system for the storage of complex objects. Under the three-tiered architecture, the GUI and problem domain classes in the system would be unaffected by these choices. Only the data access classes would be changed. In fact, the Object Data Management Group (ODMG), a consortium of OODB vendors, recommends the "pure ODMG approach" in which product specific references are isolated in a single file, allowing a switch from one OODB product to another with a minimum of disruption to other classes in the system.

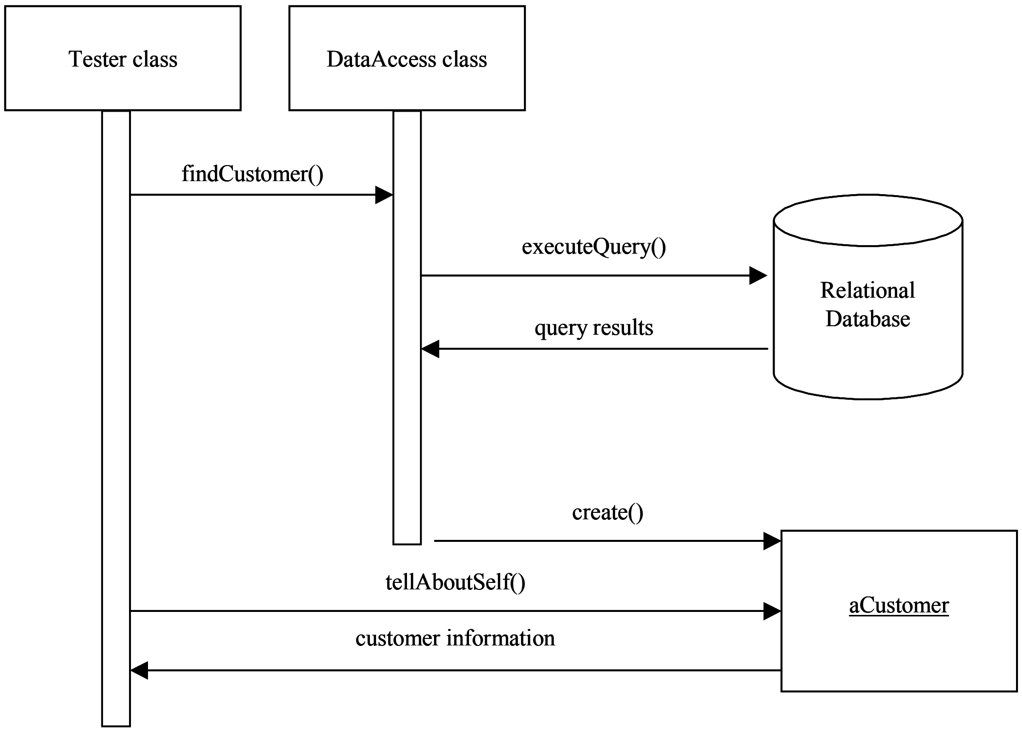

Figure 2. Sequence Diagram for RDB Example

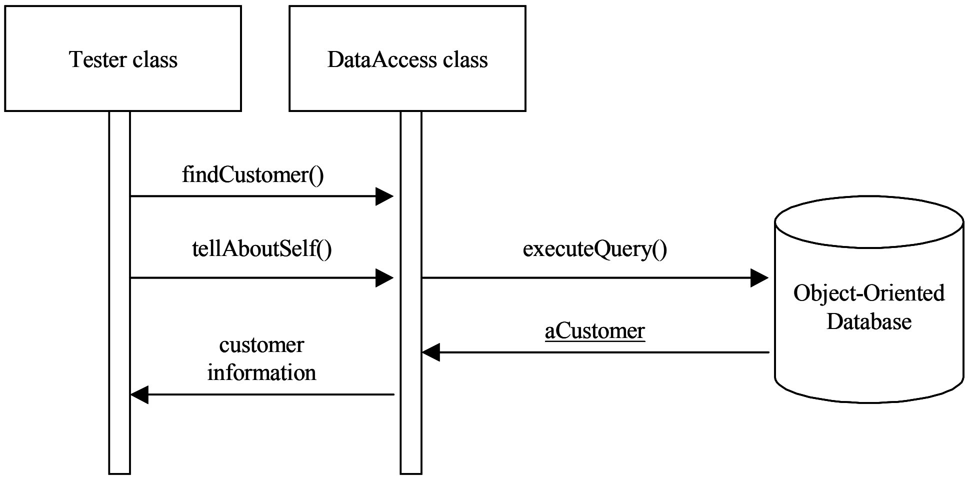

Figure 3. Sequence Diagram for OODB Example

Summary

Object-oriented databases are a solution to a very specific problem – the need for storage of complex objects. As we have demonstrated here, the OODB allows for far more efficient coding of the classes in an OO system that interact with the database. There are some proponents of the object-oriented approach who would argue that if an organization is going to use object-oriented technology, they should do so consistently through all aspects of the system from the user interface through to the data storage system. Our recommendation is not so sweeping; rather, we would advocate using the appropriate technology for the situation at hand with an understanding of the trade-offs involved. While the pure OO approach might allow for more efficient coding and less impedance mismatch, it is only achieved at the cost of sacrificing those existing systems that the organizations has in place.

The quantification of this trade-off is a subject of interest both to academics and practitioners. Our simple example provides some insight into that trade-off in terms of the volume of code required to implement object storage in an relational database versus an object-oriented database. Obviously the factors affecting the choice between an OODB and an RDB go beyond simply the length and complexity of the coding required. What existing systems must be converted? What type of investment of resources will be required to make that conversion? Will the OODB solution chosen remain stable? Or will it become obsolete and be replaced by another product in the near future? We hope to use the knowledge gained in preparing the comparison presented in this research to form a basis for moving forward to find answers to some of these questions.

References

Atkinson, M., F. Bancilhon, et al. "The object-oriented database systems manifesto."
Doke, R., J. Satszinger, et al. (2002). Object-oriented Application Development Using Java, Course Technologies.
Finn, M. (2001). "Impedance mismatch in databases." Database and Network Journal **31**(6): 8-10.
Jackson, M. S. (1991). "Tutorial on object-oriented databases." Information and Software Technology **33**(1): 4-12.
Jackson, M. S. (1999). "Thirty years (and more) of databases." Information and Software Technology **41**(14): 969-978.
Koch, G. and K. Loney (1997). Oracle 8: The Complete Reference, Osborne/mcGraw-Hill.
Leavitt, N. (2000). "Whatever happened to object-oriented databases?" Computer: 16-19.
McFadden, F., J. Hoffer, et al. (1999). Modern Database Management, Addison-Wesley.

Appendix A

Customer class

```
// Customer

public class Customer
{
    // attribute definitions
    private String name;
    private String address;
    private String phoneNo;

    // constructor with parameters
    public Customer(String aName, String anAddress, String aPhoneNo)
    {   // invoke accessors to populate attributes
        setName(aName);
        setAddress(anAddress);
        setPhoneNo(aPhoneNo);
    }

    // get accessors
    public String getName()
        { return name;}
    public String getAddress()
        { return address;}
    public String getPhoneNo()
        { return phoneNo;}

    // set accessors
    public void setName(String newName)
        { name = newName;}
```

```
	public void setAddress(String newAddress)
		{ address = newAddress;}
	public void setPhoneNo(String newPhoneNo)
		{ phoneNo = newPhoneNo;}
	public String tellAboutSelf()
		{ return (getName() + ", " + getAddress() + ", " + getPhoneNo());}
}
```

Order class

```
// Order -- Order class with Customer reference variable

public class Order
{
	// attribute definitions
	private String id;
	private String dateOfOrder;
	private Customer customer; //reference variable for Customer

	// constructor with parameters
	public Order(String anId, String aDateOfOrder, Customer aCustomer)
	{	// invoke accessors to populate attributes
		setId(anId);
		setDateOfOrder(aDateOfOrder);
		setCustomer(aCustomer);
	}

	// get accessors
	public String getId()
		{ return id;}
	public String getDateOfOrder()
		{ return dateOfOrder;}
	public Customer getCustomer()
		{ return customer;}

	// set accessors
	public void setId(String newId)
		{ id = newId;}
	public void setDateOfOrder(String newDateOfOrder)
		{ dateOfOrder = newDateOfOrder;}
	public void setCustomer(Customer newCustomer)
		{ customer = newCustomer;}
	public String tellAboutSelf()
		{ return (getId() + ", " + getDateOfOrder() + ", " +
						getCustomer().tellAboutSelf());}
	public String tellOrderDetails()
		{ return (getId() + ", " + getDateOfOrder()); }
}
```

Data Access Class for RDB example

```
// Data Access class provides DA services
import java.util.Vector;
import java.io.*;                    // needed for file i/o
import java.sql.*;              // needed for SQL

public class DA
{
      static Customer aCustomer;
      static Order anOrder;

      // The Data Source name is "CustomerDBMS"
      static String url = "jdbc:odbc:CustomerDBMS";
      static Connection aConnection;
      static Statement aStatement;

      // declare variables for Customer and order attribute values
      static String name;
      static String address;
      static String phoneNumber;
      static String id;
      static String dateOfOrder;

      // establish the database connection
      public static void initialize()
      {
            try
            {     // load the jdbc - odbc bridge driver for Windows
                  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                  // create connection instance
                  aConnection = DriverManager.getConnection(url, "", "");
                  // create statement object instance for this connection
                  aStatement = aConnection.createStatement();
            }
            catch (ClassNotFoundException e)
                  {System.out.println(e);}
            catch (SQLException e)
                  { System.out.println(e);     }
      }

      // find a customer in the database
      public static Customer findCustomer(String key) throws NotFoundException
      {
            aCustomer = null;

            // define the SQL query statement using the phone number key
            String sqlQuery = "SELECT Name, Address, PhoneNo " +
                                          "FROM CustomerTable " +
                                          "WHERE PhoneNo = '" + key +"'";

            // execute the SQL query statement
            try
            {
```

```
            ResultSet rs = aStatement.executeQuery(sqlQuery);

            // next method sets cursor & returns true if there is data
            boolean gotIt = rs.next();
            if (gotIt)
                {
                    // extract the data
                    String name = rs.getString(1);
                    String address = rs.getString(2);
                    String phoneNumber = rs.getString(3);

                    // create Customer instance
                    aCustomer = new Customer(name, address, phoneNumber);
                }
            else
                {
                    // nothing was retrieved
                    throw (new NotFoundException("not found "));
                }
            rs.close();
        }

        catch (SQLException e)
            { System.out.println(e);}

        return aCustomer;
    }

    // find an order in the database
    public static Order findOrder(String key) throws NotFoundException
    {
        anOrder = null;

        // define the SQL query statement using the order Id key
        String sqlQuery = "SELECT Id, orderDate, Name, Address, PhoneNo " +
                                        "FROM OrderTable, CustomerTable " +
                                        "WHERE Id = '" + key +"' " +
                                        "AND PhoneNo = customerPhoneNo";

        // execute the SQL query statement
        try
        {
            ResultSet rs = aStatement.executeQuery(sqlQuery);

            // next method sets cursor & returns true if there is data
            boolean gotIt = rs.next();
            if (gotIt)
                {
                    // extract the data
                    String id = rs.getString(1);
                    String orderDate = rs.getString(2);
                    String name = rs.getString(3);
                    String address = rs.getString(4);
                    String phone = rs.getString(5);

                    // create Customer instance
                    aCustomer = new Customer(name, address, phone);

                    // create Order instance
                    anOrder = new Order(id, orderDate, aCustomer);
                }
            else
                {
                    // nothing was retrieved
                    throw (new NotFoundException("not found "));
                }
            rs.close();
        }

        catch (SQLException e)
            { System.out.println(e);}

        return anOrder;
    }

    // find all orders for a customer
    public static Vector findOrdersForCustomer(Customer aCustomer)
    {
        Vector orders = new Vector();
        anOrder = null;
        String key = aCustomer.getPhoneNo();
        // define the SQL query statement
        String sqlQuery = "SELECT Id, orderDate FROM OrderTable " +
                        " WHERE OrderTable.customerPhoneNo = '" + key + "'";
        // execute the SQL query statement
        try
        {
            ResultSet rs = aStatement.executeQuery(sqlQuery);

            // next method sets cursor & returns true if there is data
            boolean more = rs.next();
            while(more)         // loop for each row of result set
          {  // extract the data
                String id = rs.getString(1);
                String orderDate = rs.getString(2);

                anOrder = new Order(id, orderDate, aCustomer); // create Order instance
                orders.addElement(anOrder);
                more = rs.next();
            }

            rs.close();
        }

        catch (SQLException e)
            { System.out.println(e);}

        return orders;
    }

    // add new order to database
    public static void addOrder(Order anOrder) throws DuplicateException
```

```
    {
        // retrieve the order attribute values
        id = anOrder.getId();
        dateOfOrder = anOrder.getDateOfOrder();
        phoneNumber = anOrder.getCustomer().getPhoneNo();

        // create the SQL insert statement using attribute values
        String sqlInsert = "INSERT INTO OrderTable " +
                                    "(Id, orderDate, customerPhoneNo)" +
                                    "     VALUES ('" +
                                    id        + "', '" +
                                    dateOfOrder     + "', '" +
                                    phoneNumber  + "')";

        // see if this order already exists in the database
        try
        {
            Order o = findOrder(id);
            throw (new DuplicateException("Order Exists "));
        }

        // if NotFoundException, add order to database
        catch(NotFoundException e)
        {
            try
            {
                // execute the SQL update statement, a 1 return good
                int result = aStatement.executeUpdate(sqlInsert);
            }

            catch (SQLException ee)
                { System.out.println(ee);   }
        }
    }

// add new customer to database
    public static void addCustomer(Customer aCustomer) throws DuplicateException
    {
        // retrieve the customer attribute values
        name = aCustomer.getName();
        address = aCustomer.getAddress();
        phoneNumber = aCustomer.getPhoneNo();

        // create the SQL insert statement using attribute values
        String sqlInsert = "INSERT INTO CustomerTable " +
                                    "(Name, Address, PhoneNo)" +
                                    "     VALUES ('" +
                                    name        + "', '" +
                                    address     + "', '" +
                                    phoneNumber  + "')";

        // see if this order already exists in the database
        try
        {
            Customer c = findCustomer(phoneNumber);
            throw (new DuplicateException("Customer Exists "));
        }

        // if NotFoundException, add customer to database
        catch(NotFoundException e)
        {
            try
            {
                // execute the SQL update statement, a 1 return good
                int result = aStatement.executeUpdate(sqlInsert);
            }

            catch (SQLException ee)
                { System.out.println(ee);   }
        }
    }

    // close the database connection
    public static void terminate()
    {
        try
        {   // close everything
            aStatement.close();
            aConnection.close();
        }
        catch (SQLException e)
            { System.out.println(e);    }
    }

}
```

Data Access class for OODB example

```
//  AIS Demo
//  DA class using a pure ODMG approach (where possible) with POET

import org.odmg.*; //import the ODMG classes
import java.util.*;
import com.poet.odmg.Extent;
import com.poet.odmg.util.CollectionOfObject;

public class DA
{
       static Database db; //declare the db variable
       static Transaction txn; //declare the transaction variable

    public static void initialize()
    {
             try
             {
              // establish the connection to the database
              db = ODMG.get().newDatabase();
        db.open("FastObjects://LOCAL/AISBase", Database.OPEN_READ_WRITE);

       //create a transaction object
       txn = ODMG.get().newTransaction();
```

```
         txn.begin();
        }
        catch (ODMGException e)
        { System.out.println("unable to initialize"); }
    }

    public static void addCustomer(Customer aCustomer) throws DuplicateException
    {

        try // bind the Customer instance to the db using phone as the object name
        {
          db.bind(aCustomer, aCustomer.getPhoneNo());
          txn.checkpoint();
        }

      catch (ObjectNameNotUniqueException exc)
        {
             throw (new DuplicateException("Customer Exists"));
        }

        catch (ODMGRuntimeException exc)
        {
             txn.abort();
             throw exc;
      }
    }

    public static void addOrder(Order anOrder) throws DuplicateException
    {
      try //create a new Order instance and bind it to the db
    {
      db.bind(anOrder, anOrder.getId());
      txn.checkpoint();
    }

    catch (ObjectNameNotUniqueException exc)
    {
      throw (new DuplicateException("Order Exists"));
    }

    catch (ODMGRuntimeException exc)
    {
      txn.abort();
      throw exc;
    }
    }

    public static Customer findCustomer(String phoneNo) throws NotFoundException
    {
          try //lookup the customer
          {
                Customer c = (Customer)db.lookup(phoneNo);
             return c;
          }

          catch (ObjectNameNotFoundException exc)
```

```
            {
               throw (new NotFoundException("Order not found"));
            }

            catch (ODMGRuntimeException exc)
            {
               txn.abort();
               throw exc;
            }
        }

        public static Order findOrder(String order) throws NotFoundException
        {
        try //lookup the order and a customer
            {
                    Order ord = (Order)db.lookup(order);
               return ord;
            }

            catch (ObjectNameNotFoundException exc)
            {
               throw (new NotFoundException("Order not found"));
            }

            catch (ODMGRuntimeException exc)
            {
               txn.abort();
               throw exc;
            }
    }

       // find all orders for a customer
       public static Vector findOrdersForCustomer(Customer aCustomer)
       {
            Vector orders = new Vector();
            Order anOrder = null;

            try
            {
              Extent allOrders = new Extent("Order");
              allOrders.setIndex("CustIndex");

              while (allOrders.hasNext())
              {
                    Order o = (Order)allOrders.next();
                    if (o.getCustomer().equals(aCustomer))
                          orders.addElement(o);
          }
            }
            catch (ODMGRuntimeException exc)
            {
                  txn.abort();
                  throw exc;
            }
         return orders;
       }
```

```
// find all orders for a customer using OQL query
    public static Vector findOrdersForCustomerUsingOQL(Customer aCustomer)
    {
        Vector orders = new Vector();
        Order anOrder = null;

        try
        {

          String query = "select x from x in OrderExtent " +
                    "where x.customer =$1";
          com.poet.odmg.OQLQuery test = new com.poet.odmg.OQLQuery(query);
          test.bind(aCustomer);
          Object result = test.execute();

          if ( result == null )
              System.out.println( "result is null" );
          if ( result instanceof CollectionOfObject )
          {
              Iterator iterator = ((CollectionOfObject) result).iterator();
              while ( iterator.hasNext() )
              {
                  Order o = (Order)iterator.next();
                  orders.addElement(o);
              }
          }
        }
        catch (ODMGException e)
        { System.out.println("Error");}

        txn.checkpoint();


      return orders;
    }

  public static void terminate() throws ODMGException
  {
          txn.commit(); //commit the transaction
        db.close();
    }
}
```