A Mechanism For Converting A Relational Database Into An Object-Oriented Model: An AIS Application

Hui Du, (E-mail: huidu@panam.edu), University of Texas, Pan American
Ting J. (TJ) Wang, (E-mail: wang@rmu.edu), Robert Morris University

ABSTRACT

The object-oriented (OO) approach in system design and development is gaining popularity. In the management information systems literature, OO system development is viewed as superior to conventional systems development because of advantages such as easier modeling, more efficient model reuse, and more convenient maintenance (Booch 1994; Briand, et al. 1999; Coleman, et al. 1994; Cockburn 1999). Several studies have explored the applicability of the OO paradigm for the design and implementation of accounting information systems (AIS) and the advantages of OO design for this purpose (Adamson and Dilts 1995, Chu 1992a, 1992b; Kandelin and Lin 1992; Murthy and Wiggins 1993; Verdaasdonk 2003). Nevertheless, OO techniques are often applied only to front-end applications while a relational database is generally used to store data at the back-end. Based on an existing relational database model for a retail enterprise, this paper contributes to the AIS literature by providing a mechanism for transforming a relational database into an OO data model.

I. INTRODUCTION

A significant amount of management information systems literature describes the superior performance and the advantages of object-oriented (OO) models as they relate to system design and development. Indeed, OO technology has moved into the mainstream of industrial-strength software development. A nationwide Internet survey of 150 developers, most with practical experience in object-oriented systems development (OOSD), revealed that both OO and non-OO developers view OOSD as superior to conventional systems development (Johnson 2000). Many system development practitioners assert that as compared to conventional systems development, OOSD has advantages such as easier modeling, more efficient model reuse, higher system quality, more convenient maintenance, and better communication between database and applications (Booch 1994; Briand et al. 1999; Coleman et al. 1994; Cockburn 1999). The OO approach has also proven helpful in the analysis and design phases of information modeling when developing a decision support system for maintenance management (Nagarur and Kaewplang 1999), and for capturing *ex ante* accounting data for operations management decisions (Verdaasdonk 2003).

Conventional system development decomposes a system into applications for processing data and database for storing data (Fichman and Kemerer 1993). Applications that process data are sometimes called the information systems' application level or front-end, while the database that stores data is referred to as the information systems' database level or back-end. OO techniques can be applied at the application level, the database level, or both. At the application level, OO techniques are currently well utilized due to the popularity and power of OO programming languages such as JAVA and C++. At the database level, however, when an OO database – rather than a relational database – might work better at the back-end, OO design has drawn much less attention because of the dominance of relational databases in use by companies. Although applications programmed with the OO techniques can connect to a relational database at the back-end, using the OO approach at both the front and back-ends might actually work

more smoothly. In fact, many companies that need a large database for storing their customer information now use an OO database, e.g., Delta Airlines, Amazon.com, and Deutsche Bank[1].

The integration of relational database technology with accounting information systems (AIS) has been discussed in accounting literature for decades (McCarthy 1979; 1982; Denna, et al. 1994; Wang, et al. 2002). From an entity-relationship (E-R) view of an accounting model to a more generalized framework, i.e., the REA (Resource-Event-Agent) model, relational database approaches and more general domain ontology usually are presented when designing accounting information systems. However, despite its market dominance, the relational data model has proven to be an ineffective tool for modeling financial accounting systems (Chu, 1992a). In contrast, OO features, such as encapsulation and generalizing abstractions and complex data structure, promote a design that integrates accounting structures and procedures and helps create a system that is friendly, modular, and reliable (Chu, 1992a). Further, Chu (1992b) illustrated how the ability of OO techniques to represent complex data structures coupled with polymorphism can enhance the ease with which accounting systems are built and used. Kandelin and Lin (1992) demonstrated the use of OO techniques in the construction of an accounting system and the benefits of using OO techniques, which include powerful generalization hierarchies and code reuse. Murthy and Wiggins (1993) explored the applicability of the OO paradigm for the design and implementation of AIS. Adamson and Dilts (1995) presented an OO methodology that produces a measurably less complex accounting information system than traditional E-R modeling.

In this paper, we migrate a relational database model into an OO database model. As mentioned above, while substantial research has been done in the design and development of AIS using both the relational database modeling and the OO technologies, this paper contributes to the AIS literature by providing a mechanism to convert a relational database schema to an OO database schema. It is important to bridge the relational database model and OO database model for two reasons. First, since the relational database approach dominates the information systems of most companies, the initial phase of relational database design (analyzing business activities and identifying entity/relationship sets or objects) can still be used in developing an OO database. Second, providing a comparison of a relational database model and an OO database model is helpful for enhancing readers' understanding of OO design. Even though theories and algorithms on the conversion from relational model to OO model are well developed in computer science and information systems research disciplines (Elmasri and Navathe, 2000), this paper is the first to implement the conversion mechanism in the domain of AIS. We also discuss some advantages of the OO approach in terms of its ease in modeling and efficiency in maintenance.

The paper begins with a brief review of some fundamentals of an OO model, followed by a mechanism of converting a relational to an OO database model using McCarthy's (1979) commonly known relational database accounting model. We then use a sale transaction example to illustrate the data operations in an OO system. We conclude by highlighting the advantages of the OO design.

II. A BRIEF REVIEW OF OOSD

OOSD creates objects that encapsulate both data and the operations that process data. In contrast, conventional system development decomposes the procedures and data in application programs and database, respectively. Table I lists four different scenarios in which types of application techniques used in the front-end are applied to types of databases in the back-end. As shown in Table I, with a relational database in the back-end, front-end applications can be developed by non-OO techniques (e.g., C, Visual Basic), or OO techniques (e.g., JAVA, C++), or both (Scenario 1, 2, and 3, respectively). When the back-end database is an OO database, the front-end applications usually are developed by OO techniques (Scenario 4). This paper is focused in Scenario 4 with an emphasis on an OO database.

[1] For a list of companies that use an OO database, visit the website of OO database vendor, *Objectstore*, at: www.objectstore.net/our_customers/index.ssp.

TABLE I. Relational vs. OO Database Systems

Scenario	Front-end (Applications developed by)	Back-end (Database)
1	Non-OO techniques: C, VB	Relational
2	OO techniques: Java, C++	
3	Non-OO and OO techniques	
4	OO techniques	OO

The basics of OO programming and the architectural constructs of OO databases are available in many publications (Andleigh and Gretzinger 1992; Booch 1994; Lippman 1998). However, the OO approach contains two primary features for designing systems: encapsulation i.e., storing data and related operations together within objects, and inheritance i.e., sharing commonalities between similar classes of objects. For example, "order", an object in an OO system, can contain all of the information about an order (e.g., customer, item, delivery date, payment) as well as the "methods" required to compute the amount of the order.

Table II contrasts terms used in relational and OO databases. Although some of the terms are different concepts (e.g., table and class), they may be similar in certain aspects. We compare them to aid the reader's understanding.

TABLE II Comparison of Terms Used in Relational Databases and OO Databases

Relational Databases	OO Databases
Record	Object
Table	Class
Attribute/Field	Attribute/Data Member
Function	Method

Below is a summary of the features and the terminology used in the OO system.

- Object: Object contains attributes (data members) and methods (operations) that manipulate data. Its attributes form a structure that is similar to those of a record in a table. An object is different from a record in that an object contains methods that manipulate the attributes of the object. For instance, our previous example of an object, "order", contains characteristics of a relational database record. In addition, "order" can also include an operation that calculates the amount of the order.
- Class: Class is a user-defined data type. While an object exists as an entity, a class is a programmed definition of a collection of objects that have the same characteristics. Thus, a class definition is only an abstraction. An object is an instance of a class.
- Attributes/Data member: Data members are needed to characterize objects--similar to fields or attributes in relational database. A data member must carry its data type. A class can also become a data member of other classes with the data type being the class. For example, "customer" can be an attribute in a relational database table "order"; similarly, in OO design, the object "order" has *custome*r, which contains all customer information, as one of its data members. The data type of *customer* in a table may be a *string*. Interestingly, the data type of *customer* in OO design can be the class of "Customer".
- Method: Method (most often called "function" in a non-OO environment) is a user-defined operation. Method is encapsulated within an object and manipulates the data members of the object. A method definition usually includes arguments and return types. For example, "inventory" is an object that contains specific information about the product items, such as ID number, description, and sales price. A method can also be contained in "inventory", *int updateQty(int qty)*, which, when triggered (see "Message" below), will return an integer number indicating the quantity of items available in the database. Data member and method can be private or public. In database operation, a private data member or method can be accessed only by methods within the same class. A public data member or method can be accessed by any method.

- Message: A message is a communication that invokes a method. It is also similar to a function in a non-OO environment. In an OO environment, messages are sent to the target objects. A message is sent with the name of the method, and with parameters binding to the arguments of the method.
- Inheritance: Classes can be viewed as the basic building blocks for reusable programs. Inheritance provides the means for building new classes based on existing classes, as opposed to redesigning the new classes from scratch. A new class inherits both data members and methods from its parent class. The new class can provide additional data members and methods that are not available to its parent. Methods inherited from a parent can also be redefined to adapt to new situations.

We will also describe pointer and array because they are important data types that play critical roles in OO design, even though they have been used in many non-OO settings.

- Pointer: A pointer is a data type, which holds values that are the addresses of objects in memory. Through a pointer, objects being pointed can be accessed and manipulated.
- Array: An array is a collection of objects of a single data type. Each element of an array is accessed by the element's position in the array.

Figure 1 is an example of a class "Account" coded in C++ [*] with descriptions in the right column. This class has several private data members: *accountNumber* with *short* integer as its data type; *accountType* with *char* as its data type; *accountName* with *char array* of size 20 as its data type; and *balance* with *double* as its data type. A char array of size 20 has 20 elements with each element holding a char type value. This "Account" class also has several public methods: *Account(...)* is a method for creating an object, which is a specific Account. *~Account()* is a method required in C++ to delete the object from the memory. *getBalance()* is a method to obtain account balance stored in the data member *balance*. Similarly, *getAccountType()* returns the account type stored in data member *accountType*. *getAccountName()* returns a constant char pointer, which points to the char array that holds the account name. The account name can be accessed through the pointer. *debit()* and *credit()* are methods to decrease and increase, respectively, the account balance by a certain amount. These two methods do not need to return anything since both have return types of *void.*

Figure 1 Example of Class "*Account*"

Class		Description
class Account {		A class named "Account". The class starts with "{".
private:		
	short accountNumber; char accountType; char accountName[20]; double balance;	All of them are private data members with different data types. For example, the data type for data member "accountNumber" is "short" for short integer.
public		
	Account(short accountNumber, char accountType, const char *accountName);	A method called "Account (…)" for creating objects.
	~Account();	A method for deleting objects.
	double getBalance();	A method for obtaining balance.
	char getAccountType();	A method for returning account type.
	const char *getAccountName();	A method for pointing to account name.
	void debit(double amount);	A method for decreasing account balance.
	void credit(double amount);	A method for increasing account balance.

[*] According to OO convention, class names begin with capital letters. Object names begin with lower case letters. We italicize methods, data members, and their data types, all of which start with lower case letters. We sometimes put classes and objects between quotation marks.

III. A CONVERSION MECHANISM

To migrate a relational database model to an OO database model, we start from an E-R model in a relational database and follow two major steps in the conversion process: A) Mapping entities into classes, and B) Mapping (or remodel) relationships.

A. Entities Classes

First, all entities in a relational database design are converted into classes in an OO database. This is the most straightforward part of mapping. Data members with data types are also included in the classes by taking the attributes directly from the entities.

B. Re-Model Relationships

Second, after entities are converted into classes, relationships must be mapped into the OO design. There are three cases for relationship mapping. For a one-to-one relationship, add one class as a data member to the other class. If one class has partial participation in the E-R model before mapping, it should be included as a data member in the other class that has total participation. If both entities have equal participation, it does not matter which one is included to the other as a data member. For a one-to-many (1-to-n) relationship, add the class as a data member at the one-side to the class at n-side. An alternative approach is to add the class at the n-side to the class at the one-side. In this approach, the class at the n-side must become an array so that it will be able to hold multiple instances. For a many-to-many (m-to-n) relationship, include either class as a data member of an array to another class. To summarize, the mapping methods include the following: 1) add the class at the one-side as a data member to the class at the other side regardless of the relationship at other side (i.e., one or n); 2) add the class at the n-side as *a data member of an array* to the class at the other side regardless the other side is one or n; and 3) add the side of partial participation to the side of total participation. (Note: this summary does not generalize the mapping process for all cases.)

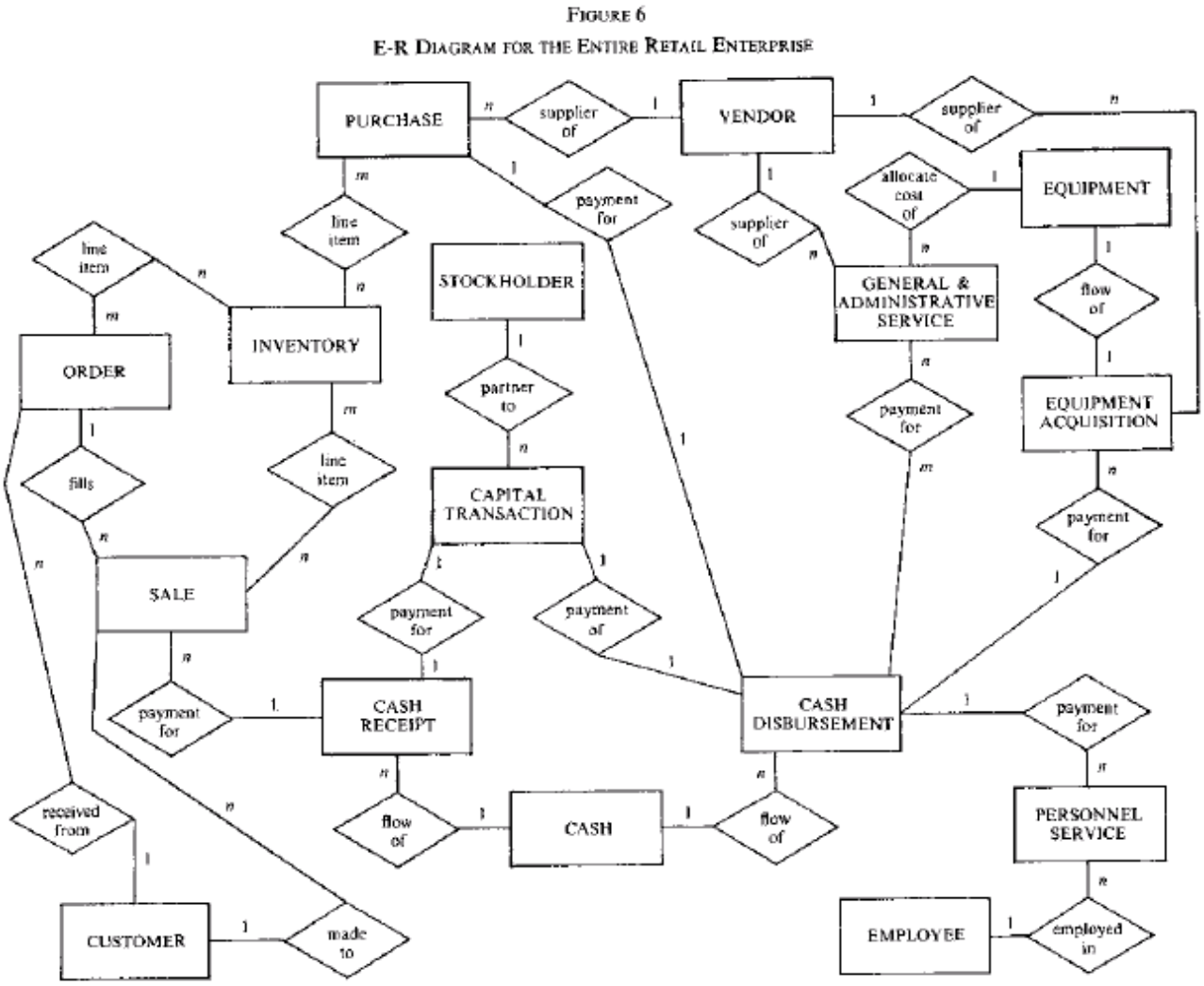
A traditional E-R model can be mapped easily into an OO design. By adding classes to each other as data members (as well as carrying methods), the OO design is able to encapsulate both data and methods in one object. The use of array eliminates the need for creating linked tables in the relational database to handle many-to-many relationship (an array is a collection of objects of a single data type). Because of the atomicity of attribute in a relational database, one record can only be linked to a single record in another table. In a many-to-many relationship between two tables, their relationship must go through a newly created linked table. However, in an OO design, no additional class is needed since one object can be linked to *a collection* of objects of another class via an array.

We will demonstrate the conversion by using a retail enterprise as an example. Figure 2 is an E-R diagram for a retail enterprise (adopted from Figure 6, page 675, in McCarthy 1979). We map the E-R diagram in Figure 2 into an OO database model in Figure 3. According to the first step of the conversion mechanism (i.e., mapping entities into classes), sixteen entities in Figure 2 are converted into 16 classes in Figure 3.

Figure 3 illustrates the OO data model for the retail enterprise. Following OO convention, class names are shown in the top section, data members with their data types in the middle section, and methods that manipulate data in the bottom section of the rectangles. Following the second step of the conversion mechanism (i.e., mapping relationships) for the cases of one-to-one and one-to-many relationships, classes at one-side relations are added to the related classes as their data members shown as small empty circles. For example, the relationship between "ORDER" and "SALE" in Figure 2 is one-to-many. "Order" as the class at one-side is added to the related class "Sale" as its data member. Thus, "*order*" is a data member in the class "Sale". In the case of many-to-many relationships, classes at either side of relationships can be added into the related classes as data members using array as the data type, shown as small filled circles. For example, the relationship between "PURCHASE" and "INVENTORY" in Figure 2 is many-to-many. We added "Inventory" class into the related class "Purchase" as a

data member "*items*". The data type for "*items*" is an inventory array notated as "*Inventory[]*". All data members data member "customer" is "Customer" class.

Figure 2 Relational Database E-R Diagram of a Merchandise Enterprise
(Adopted from McCarthy 1979, Figure 6, E-R Diagram for the Entire Retail Enterprise, Page 675)



All data members that result from the conversion rules follow the same pattern (e.g. data member: customer – data type: Customer; data member: cash – data type: Cash, etc.). Due to space limitations, we are unable to provide a complete presentation for all 16 classes that are shown in Figure 3; thus, we leave some with only a class name (e.g. Vendor, Cash, etc.).

Figure 3 An OO Diagram for the Merchandise Enterprise

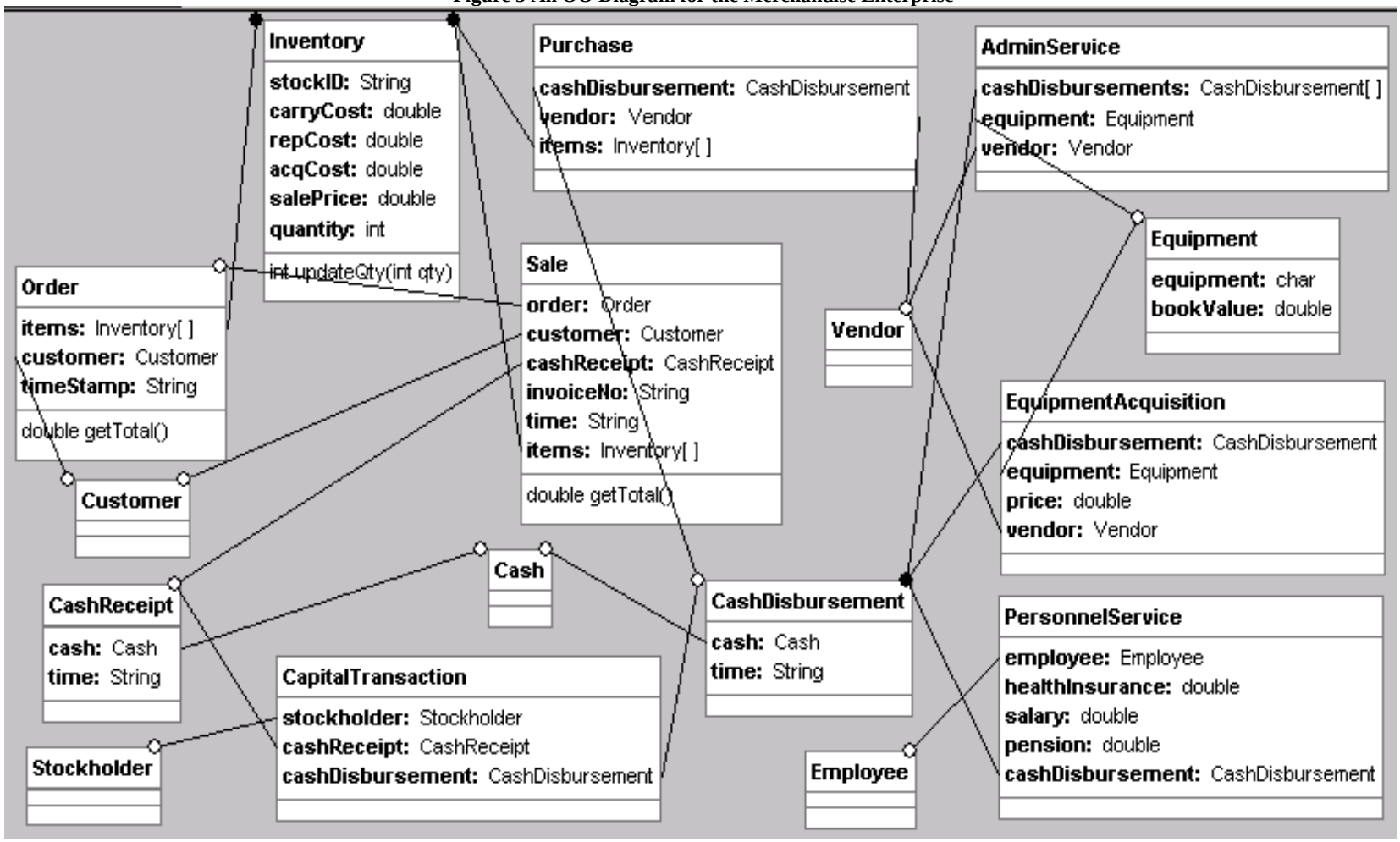


Table III lists the mapping result from the first step in the first two columns on the left, contrasting the tables used in the relational database and the classes as well as new features created (e.g., data types & methods) in the OO database. Table III also shows the pointers, arrays, and methods created after the conversion.

TABLE III Comparison of Changes Made from Relational Databases to OO Databases

Entity Under the Relational Database	**Class Under the OO Database**	**Features under the OO Database**		
Table Name	**Class Name**	**Pointer (To)**	**Array**	**Method**
Order	Order	customer (Class of Customer) & items (Class of Inventory)	items	getTotal()
Sale	Sale	order (Class of Order), customer (Class of Customer), cash receipt (CashReceipt), & items (Class of Inventory)	items	getTotal()
Customer	Customer	-	-	-
Inventory	Inventory	-	-	updateQty(int qty)
Purchase	Purchase	cashDisbursement (Class of CashDisbursement), vendor (Class of Vendor), & items (Class of Inventory)	items	-
Cash Receipt	CashReceipt	cash (Class of Cash)	-	-
Stock Holder	StockHolder	-	-	-
Capital Transaction	CapitalTransaction	stockholder (Class of Stockholder), cashReceipt (Class of CashReceipt), & cashDisbursement (Class of CashDisbursement)	-	-
Cash	Cash	-	-	-
Vender	Vender	-	-	-
Cash Disbursement	CashDisbursement	cash (Class of Cash)	-	-
Employee	Employee	-	-	-
Gen. & Adm. Service	AdminService	cashDisbursement (Class of CashDisbursement)	cashDisbursements	-
Equipment	Equipment	-	-	-
Equipment Acquisition	EquipmentAcquisition	cashDisbursement (Class of CashDisbursement), equipment (Class of Equipment), & vendor (Class of Vendor)	-	-
Personnel Service	PersonnelService	employee (Class of Employee) & cashDisbursement (Class of CashDisbursement)	-	-

IV. AN EXAMPLE OF OO DATA OPERATIONS

We use a sale transaction example to describe the data operations in the converted OO database. We will not fully populate the OO system with data because it would be too long a process for this paper. Figure 4 describes a sale transaction in the OO data operations. When a customer places an order with payment, an "order" object, a "cashReceipt" object, and a "sale" object are created. A "customer" object is also created if it is not already in the database. These objects are stored in the OO database.

In Figure 4, a sale object, $sale_i$, can be created in two ways: 1) through an order and a cashReceipt object if the sale comes from an order; 2) through a customer and an item object if the sale comes from a walk-in customer. The sale object has the following data members: $order_i$, $customer_m$, $cashReceipt_i$, *invoiceNo*, *time*, and $items_j$. Among those data members, $order_i$ and $cashReceipt_i$ are pointers, pointing to an order and a cashReceipt object, respectively. Data member $customer_m$ and $items_j$ are also pointers, pointing to a customer and an item object, respectively[2]. The other data members contain only text data. For instance, *invoiceNo* is "I46" and time is "12/24/02 13:51". The $cashReceipt_i$ object contains two data members: $cash_i$ and *time*. *time* is simply a text data, "12/24/02 13:51". $cash_i$ is a pointer to a cash object. The $cash_i$ object has one numerical data member, *amount*, "$2,916.00". The $order_i$ object has data members: $items_i$, $customer_m$, *timeStamp*. Similar to the data members of the sale object, *timeStamp* is text data, "12/24/02 13:49"; $items_i$ and $customer_m$ are pointers to objects of their classes. All data members in the object $customer_m$ are text data: *fname* "John", *lname* "Smith", *address* "1 1st st, City, ST", *phone* "123-456-7890". $items_i$ is a pointer to an *inventory array* that includes two object elements: one stores the first ordered item with *quantity* of "102" and another, the second, with *quantity* of "90". In addition to the difference in quantity, these two elements differ only in the values of their data members because they are different objects. In another words, they have different values in *stockID*, *acqCost*, *repCost*, *carryCost*, and *salePrice*. $items_j$ in this example is the same as $items_i$. We let $items_i$ in $order_i$ point to $items_i$ while making $items_j$ in $sale_i$ point to $items_j$. The reason is to show that what is sold can be different from what was ordered, though in this particular example, they are the same.

To compute the total sale value of an order (which is the amount of cash to be paid by the customer and, eventually, the value of the data member *amount* in $cash_i$ object), the method *getTotal* is included in both order object and sale object. This method is provided in the sale object in case the sale object is not created through order object. Method *getTotal* would do the following for the above transaction: a) find subtotal for the first item, $18 (Price) x 102 (Quantity) = $1,836; b) find subtotal for the second item, $12X90=$1,080; c) find total sale for the order, $2,916 ($1,836+$1,080). In our example, we assume the customer paid in full with cash. Next, a cash object, $cash_i$, is created with the amount paid, $2,916. When the transaction is completed, messages are sent to the "items" objects (assuming they already exist in the database), to invoke method "*updateQty*", which updates the objects by subtracting the quantities of sold items in "sale" or "order".

V. SUMMARY OF OO APPROACH ADVANTAGES

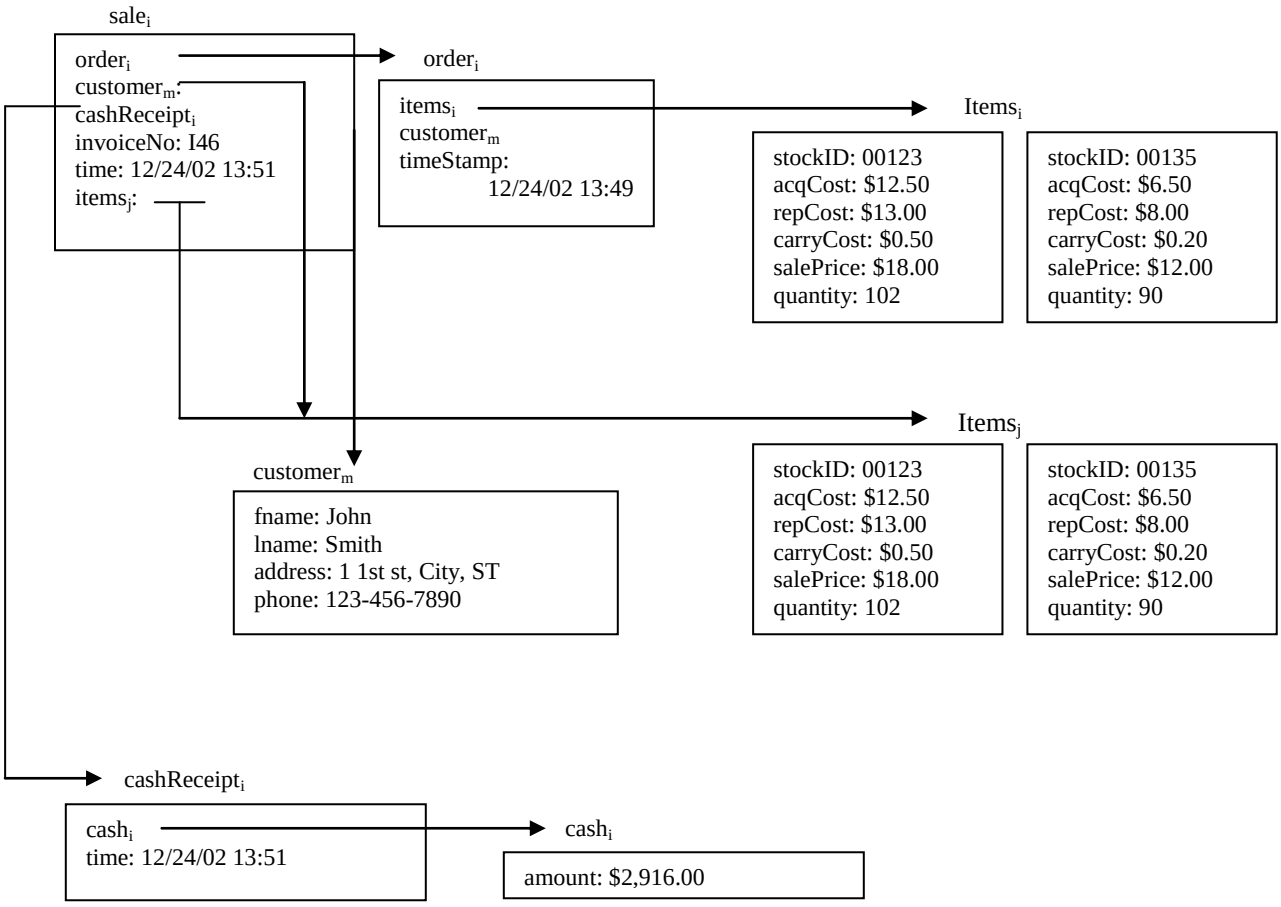
We have completed the conversion for a retail enterprise from a relational database into an OO system based on the given conversion mechanism. We outlined the steps in the conversion mechanism and illustrated each step using a retail enterprise as an example. In conclusion, we will summarize the advantages of the OO approach in the system design and development.

Easy Database Modeling

In an OO database, entities are represented by classes and records are represented by objects. Data members of an object correspond to attributes of a record or an instance in a relational database. Data members of an object can be of any data type (e.g., class, pointer, array, or array of pointers), whereas in a relational database, attributes of records can only be atomic values and cannot be pointers or arrays. In an OO database, data members can be pointers that can point to arrays; therefore, data members do not represent atomic values only. As a result, linking tables are eliminated in the OO system, but are required in a relational database to handle many-to-many relationships. In summary, data members of an object in the OO approach can be another object or a pointer pointing to another object, making modeling easier in the OO system. The ease of database modeling in the OO design has been demonstrated in AIS literature (Adamson and Dilts 1995; Chu 1992a and 1992b).

[2] An alternative is to set data member *customer* and *items* to be null since this information can be retrieved from the data member $order_i$. In case that a sale object is not created through order, there are text data in *customer* and *items*.

Figure 4 Illustration of a Sale Transaction in the OO Operations



Efficient Model Reuse

One of the key characterizations of OO design is inheritance. Inheritance provides the mechanism for creating new classes based on existing classes, rather than having to build new classes from the beginning. A child class inherits all the features of its parent class, namely its data members and methods. Hence, the design and the program of the parent class can be directly used in the child class while the child class can still add new features to itself. Occasionally, the child class may need to modify some methods inherited from the parent class for new situations; however, the methods in the parent class remain unchanged despite modifications of these same methods in the child class. An important benefit of inheritance is the ability to adjust objects to different or new requirements. The vendor class in Figure 3 is further illustrated in Figure 5 to show the detailed class design with inheritance. The company uses merchandise vendors and service vendors, both sharing common attributes, such as *name, address, telephone, and contactPerson*, yet also possessing distinguishing attributes, such as *payMethod, deliveryMethod, rate, and billUnit*. We create a parent class Vendor to handle these common attributes and two child classes, MechandiseVendor and ServiceVendor, to manage their own distinguishing attributes, while both child classes are able to inherit all the attributes from their parent class, Vendor. In this way, the Vendor class can be reused no matter how many different vendors are added in the future. To add a new type of vendor in the system, for example, a supply vendor, we only need to create a child class of the Vendor class for the distinguishing features of the supply vendor – there is no need to create the supply vendor class from scratch. The efficiency of reusing models is greatly enhanced. Figure 6 shows two samples of "Object" vendor. Systems controls can also benefit

from the similar inheritances: that is, child class automatically inherits system controls that are defined for parent class. The efficient model reuse and the benefit of inheritances are consistent with the OO literature in accounting information systems (Chu 1992a; Kandelin and Lin 1992).

Figure 5 Vendor Inheritance

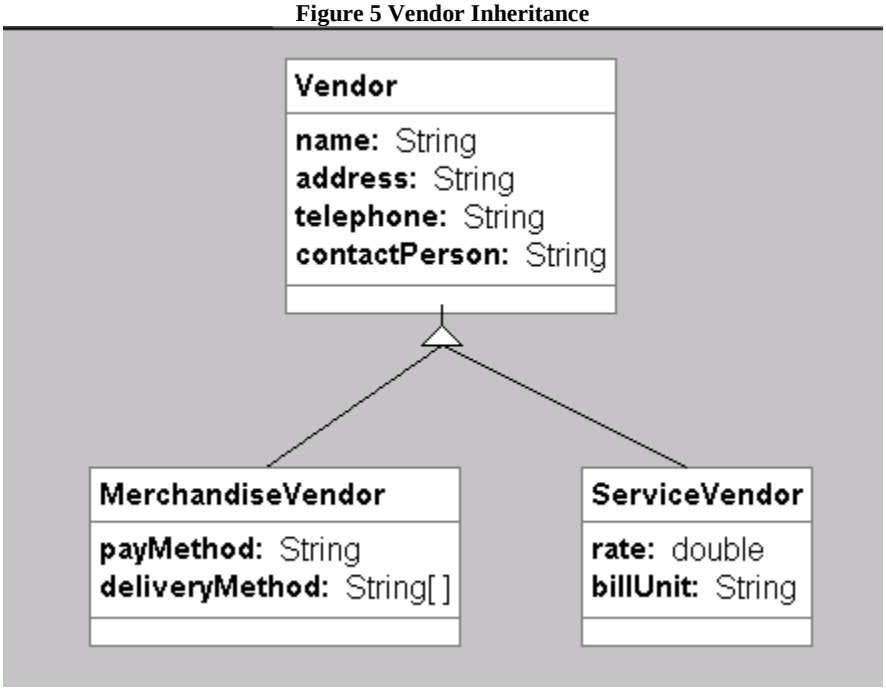


Figure 6 Samples of Creating Vendor Objects

$MerchandiseVendor_i$

name: Nicevendor
address: 1 Main Street, City, State12345
phone: 123-456-7890
contactPerson: John Smith
payMethod: Upfront
deliveryMethod: FOB Destination

$ServiceVendor_j$

name: Goodvendor
address: 2 Main Street, City, State12345
phone: 213-456-7890
contactPerson: Mary Taylor
rate: 100
billUnit: day

Smooth Communication Between Database And Applications

Given the reality that relational databases still dominate the database part of information systems, applications developed by OO always require data mapping between the OO applications and the relational database. While data are represented as attributes in tables in relational database, both data and their operations can be encapsulated in objects in the OO. Even though we sometimes compare classes and tables for some similar aspects, classes and tables are totally different concepts and have different set of rules to follow. A class object and a table record may contain the same data members/attributes, but with a completely different data structure. The process of matching between class data members and table attributes takes a big toll on program development and program performance. At the program development stage, two different sets of designs and programming (the relational database and the OO applications) must be carried out. When the program is in use, the data have to be mapped each time a process is needed. As a result, systems may suffer relatively low performance because of increased computation time.

In comparison, our converted accounting information system uses an OO database[3]. Objects containing data and methods are read and stored in the database. All objects stored in the OO database possess the same structure and features as the objects used in application programs. Therefore, there is no need to map data in the database and in the application. The OO class design and programming are able to handle both the database and the applications. The time needed to develop applications can be reduced and the system performance is also improved.

Convenient maintenance

Existing systems with application programs and database programs need constant, ongoing adjustment and maintenance. Although the basic concepts of a relational database are straightforward, the implementation rules and the detailed procedures for data manipulation are rather complex. Maintenance usually requires well-trained database programmers. Moreover, the development of applications requires a different set of knowledge and skills, typically involving the talents of an application programmer. For instance, to compute FIFO or LIFO inventory balances in relational database systems, application programmers and database developers must work together to understand the stored procedures (e.g., nested stored procedures, cursors, and loops). Thus, maintenance for applications based on relational database systems usually requires a team of application programmers and database programmers. Since OO database and OO applications use the same design techniques, OO application programmers are able to maintain the OO system without the need of OO database developers.

This paper introduced the popular OO design through a conversion of a relational database design into an OO model. It also offered helpful insight to accounting professionals who are interested in the OO techniques.

REFERENCES

1. Adamson, I., and D. Dilts. 1995. "Development of an accounting object model from accounting transactions". *Journal of Information Systems*. 9 (1) (spring): 43-64.
2. Andleigh, P. and M. Gretzinger. 1992. "Distributed object-oriented data-systems design." Englewood Cliffs, NJ: Prentice-Hall.
3. Booch, G. 1994. "Object-oriented analysis and design with applications." 2nd ed. Redwood City, CA: Benjamin/Cummings Pub. Co.
4. Briand, L., E. Arisholm, S. Counsell. 1999. "Empirical studies of object-oriented artifacts, methods, and processes: state of the art and future directions." *Empirical Software Engineering: An International Journal*. 4 (4) (December): 387-404.
5. Chu, P. 1992a. "An object-oriented approach to modeling financial accounting systems." *Accounting, Management and Information Technologies*. 2 (1): 39-56.
6. _____. 1992b. "Applying object-oriented concepts to developing financial systems." *Journal of Systems Management*. 43 (5) (May): 28-34.
7. Cockburn, A. 1999. "The impact of object-orientation on application development." *IBM Systems Journal*. 38 (2/3): 308-332.
8. Coleman, D., P. Arnold, S. Bodoff. 1994. *Object-oriented development: the fusion method.* Englewood Cliffs, NJ: Prentice-Hall.
9. Denna, E., J. Jasperson, K. Fong, and D. Middleman. 1994. "Modeling conversion process events." *Journal of Information Systems*. 8 (1) (spring): 43-54.
10. Elmasri, R. and S. Navathe. 2000. *Fundamentals of database systems*. 3rd ed. Reading, MA: Addison-Wesley.
11. Fichman, R. and C. Kemerer. 1993. "Adoption of software engineering process innovations: the case of object orientation." *Sloan Management Review*. 34 (2): 7-22.
12. Johnson, R. 2000. "The ups and downs of object-oriented systems development." *Association for Computing Machinery. Communications of the ACM*. 43 (10) (October): 68-73.

[3] There are many OO database systems in marketplace. To implement our example, we use a free OO database system downloaded from *ObjectStore, a Division of Progress Software*. More information about this database is available at: http://www.objectstore.net. Another OO database system is available at: http://www.fastobjects.com/us/

13. Kandelin, N., and T. Lin. 1992. "A computational model of an events-based object-oriented accounting information system for inventory management." *Journal of Information Systems*. 6 (1) (spring): 47-62.
14. Lippman, S. and J. LaJoie. 1998. *C++ primer*. 3rd ed. Reading, MA: Addison-Wesley.
15. McCarthy, W. 1979. "An entity-relationship view of accounting models." *The Accounting Review* 54 (4) (October): 667-686.
16. _____. 1982. "The REA accounting model: A generalized framework for accounting systems in a shared data environment." *The Accounting Review*. 57 (3) (July): 554-578.
17. Murthy, U. and C. Wiggins. 1993. "Object-oriented modeling approaches for designing accounting information systems." *Journal of Information Systems*. 7 (2) (fall): 97-111.
18. Nagarur, N. and J. Kaewplang. 1999. "An object-oriented decision support system for maintenance management." *Journal of Quality in Maintenance Engineering*. 5 (3): 248-257.
19. Verdaasdonk, P. 2003. "An object-oriented model for *ex ante* accounting information." *Journal of Information Systems.* 17 (1) (spring): 43-61.
20. Wang, T., H. Du, and H. Lee. 2002. "A user-oriented approach to data modeling: A blueprint for generating financial statements and other accounting-related documents and reports." *Review of Business Information Systems*. 6 (1): 77-92.

Notes

Notes