ACHIEVING BUZZWORD COMPLIANCE IN OBJECT ORIENTATION

David C. Hay
Essential Strategies, Inc
13 Hilshire Grove Lane
Houston, TX 77055
(713) 622-7400

[The first of two articles.]

The data processing industry in the nineties is sitting at the confluence of several important trends. Information engineering represents the integration of the first three: (1) the development of structured system development techniques, (2) the change in orientation from program processes to data structure, and (3) the development of relational database systems. It remains to take full advantage of the insights gained from a fourth: object oriented programming and systems analysis.

Contrary to the publicity they have received, the techniques of object orientation do not represent as much of a departure from the way things have been done in the past as their proponents would suggest. As with many "revolutionary" changes in our industry, a large part of the changes have simply been in vocabulary. This paper is a discussion of those object oriented concepts which represent simply renamed extensions of information engineering and relational database design, and those that represent radical departures from them.

About information engineering

Information engineering was a logical extension of the structured techniques that were developed during the 1970's. Structured programming led to structured design, which in turn led to structured systems analysis. These techniques were characterized by their use of drawings (structure charts for structured design, and data flow diagrams for structured analysis), both to aid in communication between users and developers, and to improve the analyst's and the designer's discipline. Until about 1980, though, the drawings had to be done by hand — significantly reducing their appeal to those who had to do them, and reducing also their responsiveness to change. During the 1980's, tools began to appear which both automated the drawing of the diagrams, and kept track of the things drawn in a data dictionary.

After the example of computer-aided design and computer-aided manufacturing (CAD/CAM), the use of these tools was named computer-aided systems engineering (CASE).

Even as CASE tools made it easier to create data flow diagrams to represent business processes, it was clear that this didn't really address what was needed to model a business. Among other things, business processes had a tendency to change frequently. Systems based on data flow diagrams were vulnerable to frequent changes. It became apparent, however, that the structure of business *data* did not change as much, and systems whose architecture was based on this data structure tended to be more stable.

At the same time that data structure began attracting interest, *relational* theory and the technology for implementing relational database management systems became practical. The relational approach provided great flexibility and allowed systems to be developed that would respond to future requirements not anticipated when the system was built.

In 1976 Peter Chen developed the first version of data ("entity/relationship") modeling,[1] and numerous alternative versions of data modeling have followed. Data modeling soon replaced data flow modeling as the technique of choice for understanding a business.

One of these versions of data modeling techniques is the Oracle Method (previously called "CASE*Method"). Figure 1 is an example of a data model using this notation. It asserts that a PURCHASE ORDER must be <u>either</u> *to* a PERSON <u>or</u> *to* an ORGANIZATION, and may be *composed of* one or more LINE ITEMS, each of which must be *for* an ITEM FOR SALE. An ITEM FOR SALE must be either a PRODUCT or a SERVICE.

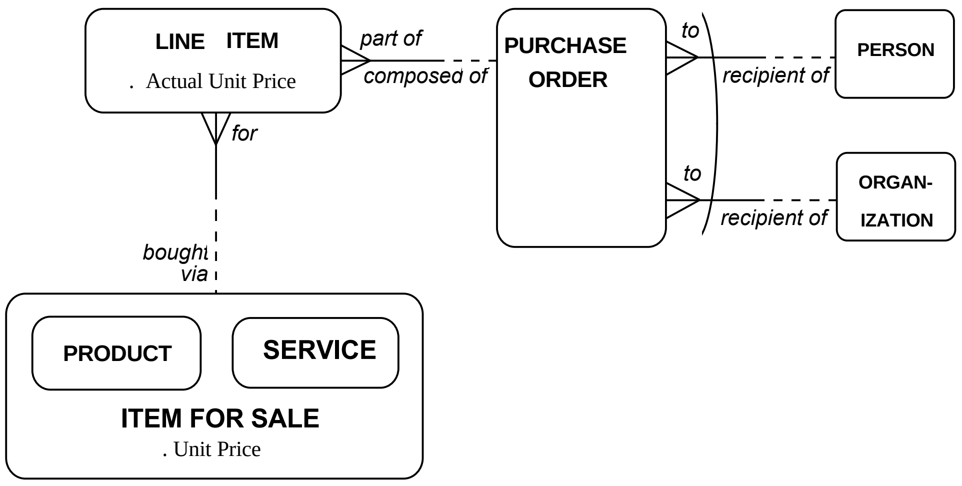


Figure 1: A Data Model

Data modeling supported the relational approach, and information engineering supported data modeling.

One problem, however, is that, as defined in information engineering, the functional side does not have the thoroughness and rigor of the data side. Information engineering and relational theory provide clear guidance as to how to translate a data model into a database design, but, aside from suggesting that modules are derived from business functions, it provides little guidance as to how to translate the function model into

[1] Chen, P. P. "The Entity-relationship Model — Toward a Unified View of Data," *ACM Transactions on Database Systems,* 1,(1), 1976, pp. 9-36.

program design. In the 1970's Ed Yourdon's and Larry Constantine's principles of structured design[2] provided real guidance in defining modules, with rules for defining each module's use of data. Unfortunately, however, the rules were defined in terms of third-generation programming languages, such as COBOL, FORTRAN, and Pascal, and the rules don't quite apply to fourth generation and other new application languages.

To some extent the Yourdon and Constantine structured guidelines are less necessary now as well — at least as they were originally formulated. Where third generation systems consisted of complex programs to be executed in sequence, modern systems consist of databases that are hit asynchronously by many different kinds of (often simple) transactions. Moreover, the code that used to make up the bulk of FORTRAN and COBOL programs was concerned with the mechanics of input and output (now handled by the database management system), so the remaining code to be defined in terms of the business aspects of each transaction is relatively small.

The biggest use of the function hierarchy has turned out to be as a guide for designing menu structures.

Still, programs multiply, even in this decade, and the lack of clear guidelines for defining program structure has kept us from being as effective as designers as we should be.

Object Orientation

Enter object orientation. While part of the information systems community had been pursuing systems analysis via entity/relationship diagrams, the programmers were chugging along with structured design and confronted with the limitations it presented. Somewhere along the line they picked up on some programming techniques developed back in the 1960's in the computer simulation world: They discovered that in programming, if you organized the problem around the data being manipulated, instead of around the processes being performed, your life became much easier. Languages such as SmallTalk and C++ were developed specifically to support this.

As the name suggests, this approach treats the programming world as being composed of objects, and objects of the same type may be grouped into classes. Program code is then organized around these classes as class "behavior".

In the last ten years or so, the object oriented programming world stuck its head up and realized that perhaps they had something to say to the people who were analyzing requirements. The requirements analysts could look at the world in terms of objects as well, and perhaps their lives would also be made easier by virtue of this approach.[3]

2 Yourdon, E, and L. Constantine. *Structured Design* (Englewood Cliffs, New Jersey: Prentice-Hall, 1979).

3 For example, Rumbah, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design* (Englewood Cliffs, New Jersey: Prentice-Hall, 1991).

The problem is that the information engineers had already been doing this all along.

Object orientation's classes, when applied to the business world, look suspiciously like entities — defining things of significance about which we wish to hold information. Object models, then, differ from data models only in their terminology and syntax. An object class is equivalent to an entity, and an entity occurrence is an object.

An "object model", then, is really an entity/relationship (data) model in disguise. The advent of object orientation took the array of a dozen or so data modeling notations that were available and added half a dozen more. Fortunately, Unified Modeling Language (UML) has arrived on the scene to at least reduce the number of object modeling notations. It remains to be seen whether it will replace data modeling overall.[4]

Now it should be said that while many concepts are similar with merely a change of names, and some concepts in object orientation represent mere manipulations or constraints on information engineering techniques, it is the case that some concepts and approaches are truly new and different. Moreover, there is a difference in attitude between the practitioners of information engineering and the more object oriented practitioners. The rest of this article describes the superficial and serious changes, plus that difference in attitude.

Terminology change

So, the ***object*** in the object oriented approach turns out to be not new at all. A model of objects in a business is nothing other than a data model in new clothes. As described above, where object orientation speaks of an object as a thing in the world (tangible or intangible), and groups these things into ***classes***, information engineering calls such a class an ***entity*** (or, as some would have it, "entity type"). What the object oriented crowd calls objects, information engineering calls ***occurrences*** of an entity. Entities and object classes are equivalent. Entity occurrences and objects are the same thing.

Entity types and object classes both have ***attributes***. The object oriented literature points out that an attribute is *defined* for its class, whereas its *value* is assigned for each occurrence (object). A particular object's use of an attribute is also called an ***instance variable***.[5]

Data/Object model variations

Some aspects of object orientation reflect not changes to entity modeling itself but simply additional constraints applied to it. Some of these constraints are already reflected in the practices of many data modelers.

---

[4] Hay, D. "UML Misses the Boat", http://www.essentialstrategies.com/publications/modeling/uml.htm.

[5] The terminology of the two approaches is brought together well in Schmidt, Bob. *Data Modeling for Information Professionals* (Englewood Cliffs, New Jersey: Prentice-Hall PTR, 1999).

First, while data modelers and relational database designers have argued for some time about the virtue of system-generated unique keys, object oriented designers come down firmly on the side of using them. *All* objects are identified uniquely by an internal key that is not visible to the user. The database manager should be able to manage the data in such a way as to allow the user to refer to objects in whatever way he or she chooses, without having to know the identifier that maintains the internal integrity of the database. Relational theory calls such an identifier artificial (a so-called "surrogate key") and would have all things identified by visible attributes, but the usefulness of system-generated keys has been widely demonstrated, and the object oriented folks have made it official.

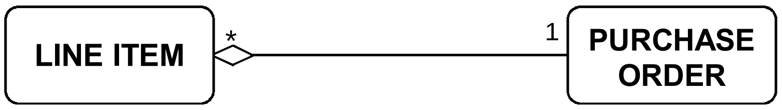
Object orientation extends data modeling with the concept of a "class variable". Where the value of an attribute of STUDENT is called by object orientation an "instance variable", the definition of a class object allows specification of a "***class variable***", which describes all students in the class. Information engineers have to put such an attribute into an explicitly defined parent entity. In the STUDENT example, such an entity might be SCHOOL.

Certain patterns widely recognized in data models have special significance in the object oriented world. Among them is the idea of ***aggregation***. That is, each group must be composed of one or more members. Many relationships are of this form, and the object oriented theory has given it a name. Figure 2 shows the UML[6] notation for aggregation, as compared with that described by the Oracle Method.[7] Aggregation presumes a "nullify" referential integrity constraint. That is, if the parent is deleted, the child records continue to exist. The "restricted" constraint, not allowing the parent to be deleted if there are children, constitutes ***composition***, also shown in Figure 2. The information engineering approach treats referential integrity constraints as a separate topic, that may or may not be shown explicitly on the model.

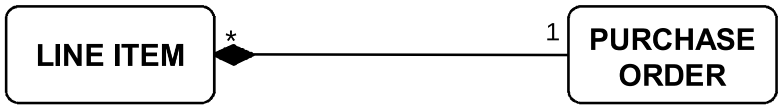
---

6 For example, as described in Fowler, M. *UML Distilled: Applying the Standard Object Modeling Language* (Englewood Cliffs, NJ: Prentice-Hall, 1997), pp. 80–82.

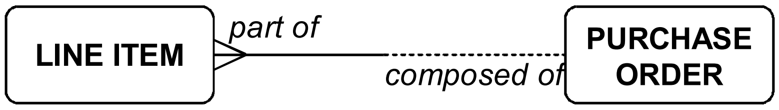
7 Barker, R. *CASE Method: Entity Relationship Modeling* (Harlow, England: Addison-Wesley, 1990).



Aggregation in UML



Composition in UML



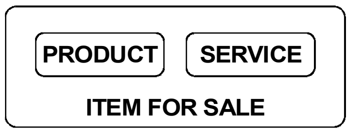
Entity/relationship model

Figure 2: Aggregation

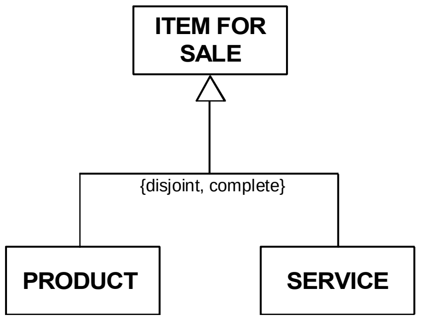
The concept of super-type and sub-type is important in object orientation, especially the notion that entities ***inherit*** attribute and relationship values from their super-type. Object models (like other kinds of data models) do not show this as overlapping symbols the way the Oracle notation does, but as separate boxes with one to one "isa" relationships connecting them. (See Figure 3.)

Whatever the notation, the attributes of ITEM FOR SALE would be inherited by PRODUCT and by SERVICE, but the attributes of PRODUCT and SERVICE would not apply either to ITEM FOR SALE, or to each other.

In UML, the reference to "complete" shows that, in this case, every occurrence of ITEM FOR SALE is either an occurrence of PRODUCT or SERVICE. There are no other kinds of ITEM FOR SALE. The "disjoint" means that PRODUCT and SERVICE are mutually exclusive. That is, no ITEM FOR SALE can be both an PRODUCT and a SERVICE. These are the constraints that always apply in the Oracle Method, but UML allows for other alternatives to be possible. That is, the set of sub-types might not be exhaustive, and they may overlap.



Oracle Notation



UML Notation

Figure 3: Inheritance

Object oriented literature adds to the language the concept of ***generalization*** and ***specialization***. If you have an entity which you subsequently decide to break into sub-entities, you have "specialized" it. If you start with a group of entities which you decide to group into a super-type, you have "generalized" it. It is not clear what adding these terms to our vocabulary contributes to the discourse.

In this, by the way, the entity modeling of information engineering is closer to object orientation than it is to the relational system developed from the models. While a data model can represent the object oriented concept of sub- and super-types, a relational database cannot. One way or another, a database designer must make compromises to convert these hierarchical structures to relational tables: he or she must decide whether to convert these to one or several tables.

On the other hand, object oriented programs, in theory at least, deal with inheritance from super-types directly. That is "in theory", since object oriented programming doesn't actually address database design issues at all. (More on this, below.)

One issue which is still controversial in the object oriented world is whether it is appropriate to describe a sub-type as being a member of more than one super-type ("multiple inheritance"). To do so is to assert that an object which is a member of the class may be a member of more than one other class, potentially inheriting attributes from each. Figure 4 shows that PERSON and ORGANIZATION are sub-classes of both CUSTOMER and VENDOR.

The Oracle Method — as well as some object oriented theorists — does not permit multiple inheritance. This is claimed by practitioners of the method to be unnecessary if the problem is simply stated differently.

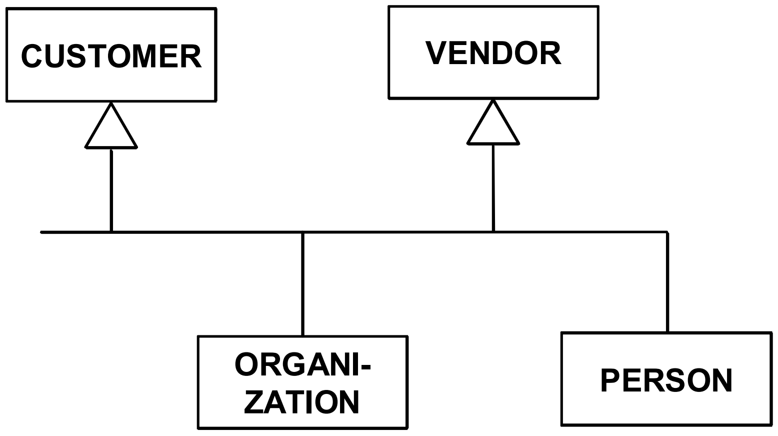


Figure 4: Multiple Inheritance

For example, the situation shown in Figure 4 can be handled with better definitions of entities and relationships: ORGANIZATION and PERSON are sub-types, but of something called PARTY. PARTY, however, says nothing about the roles they play. A PARTY may be both a VENDOR and a CUSTOMER: If it is a *vendor in* a PURCHASE ORDER, it is a vendor. If it is also a *customer in* a SALES ORDER, it is also a customer. (See Figure 5.) It turns out that "vendorness" and "customerness" are not characteristics of the entity itself, but of its *relationship* to other entities.

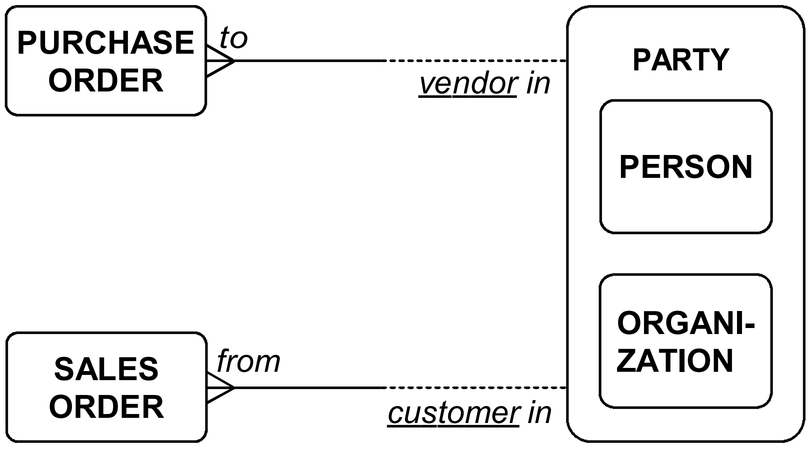


Figure 5: An Alternative to Multiple Inheritance

The attributes which in the multiple inheritance example are inherited by ORGANIZATION from its being a VENDOR are more properly attributes of the PURCHASE ORDER, not of the organization itself.

If need be, a SUPPLY ROLE, linking one or more PRODUCTS to a company, may be defined where the "vendorship" exists without any purchase orders.

Some object oriented languages support multiple inheritance, while others do not. Those which do are very complex. It is easier to redefine the problem so that multiple inheritance is not necessary.

This raises a point that is implicit in the way some authors address object orientation: inheritance can be viewed as being not just from super-types but from any entity to which the entity in question is related. For example, in Figure 6, a PURCHASE ORDER is described not only by its own attributes but also by the attributes of the PERSON or ORGANIZATION to which it is related. This is particularly important when dealing with default values. For example, the "(standard) unit price" of a product is inherited by LINE ITEM, even though it may be overridden by an "actual unit price".

In general, object orientation presumes that the classification of objects into sub-types and super-types is dynamic, and the facility for changing them should be available. In relational databases, once a data structure has been defined to accommodate this structure, it is not easy to change.

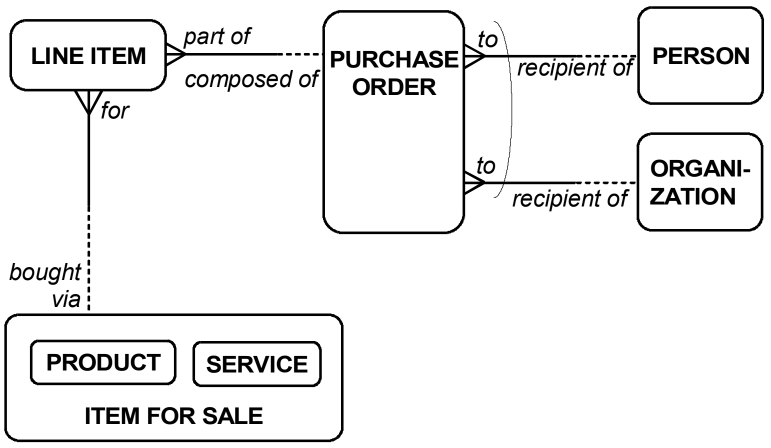


Figure 6: Inheritance Across Entities

[Next month: New Concepts, Attitudes, and a Bibliography.]