Journal of Information Systems Education, Vol. 17(1)

ER Is UML

James Rumbaugh
IBM, 18880 Homestead Rd
Cupertino, CA 95014, USA
rumbaugh@us.ibm.com

ABSTRACT

Entity-Relationship (ER) diagrams are frequently used for data modeling and database design. The Unified Modeling Language (UML) is dominant in the programming area but has not been widely adopted in the database area. I describe the history of UML as inspired by ER diagrams and argue that the use of a suitable variant of UML incorporates the benefits of ER diagrams as well as the advantages of a modeling language used by the programming community.

Keywords: UML, ER diagram, Data modeling

1. UML AND DATA MODELING

UML is not well accepted by many people in the database field, despite a heritage that originally derives from ER diagrams. Both database and programming language people make assumptions about the role of modeling UML and ER diagrams, assumptions that I feel are limiting. Used correctly, UML is an excellent tool for data modeling. In this paper, I describe some of the history of UML and argue for a broader understanding of modeling.

2. ASSUMPTIONS

The prevailing viewpoint among both database and programming language people is that UML is a tool for building applications using object-oriented programming languages, a tool that is not very suitable for database design. Under this viewpoint, UML is identified with the principles of Smalltalk and, by transference since the decline of that language, with C++ and Java. I would summarize those principles as follows:

- identity of objects is the fundamental concept
- data is stored within encapsulated objects
- data in objects is accessed only through methods
- navigation through a network of data is accomplished by a series of method calls
- objects are connected by references stored in objects
- inheritance is of major importance in organizing types and reusing implementation
- models are for designing programs
- the system modeling focus is "data in motion"

These information hiding principles prevent programs from becoming dependent on the detailed data structures and encourage reuse of fragment of programs, at the cost of some rigidity in the organization and accessing of data and the inability to optimize navigation chains.

Relational databases follow a different set of principles:

- serializable ("pure") data values are the fundamental concept
- data is stored in open (unencapsulated) tables
- data in tables is freely available to any program (security is a separate issue)
- navigation through a network of data is accomplished by a series of joins that can be optimized for implementation
- data values are connected by relations (associations, in UML terminology)
- inheritance is a troublesome concept to implement
- keys and indexes are of major importance in specifying and implementing databases
- models are for designing relational tables
- the system modeling focus is "data at rest"

According to these assumptions, there is not much point in mixing UML and ERD, or indeed of much commerce between programming languages and databases. This is based on the misconception that UML requires adoption of the full set of object-oriented programming design principles. On the contrary, UML does not include a built-in design process; it can be used in many different ways. UML has a lot of diagram types, but you don't need to use all of them for any one purpose, such as database design; you don't need to follow Smalltalk principles such as tight encapsulation or heavy use of inheritance. UML class diagrams are excellent for data modeling.

Note that I am talking about relational databases, not object-oriented databases, which have dwindled to a minor niche market. At one time there were over seven major object-

oriented database managers for sale, and their proponents claimed that they would soon displace relational databases. This has not happened. Object databases forced users to know the detailed organization of the data model, while relational databases permit a more fluid approach, for example, by deriving new virtual views. It is also difficult to extract a subset of the fields in an object database, while tables are a natural unit for decomposition of relational databases. Relational databases responded to the threat by incorporating some object-oriented concepts while maintaining their many advantages. Object databases are at risk and are limited to certain kinds of applications.

Much of what I say will no doubt infuriate OO purists as well as many database people. That's okay—I'm a natural iconoclast. There are too many arbitrary boundaries in computing already.

3. HISTORY OF UML

Class diagrams are the oldest part of UML. Most of the basic concepts and notation for UML class diagrams came originally from OMT (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991), which was first described in Rumbaugh, Loomis & Shah (1987). OMT was originally inspired by ER diagrams as described in Ullman's database text (1982), and it was intended for modeling application domains. Some ER extensions already included the concept of generalization, which was not solely an object-oriented concept. OMT was originally applied to describing a class library and to modeling information to be stored in a database, so it had a strong data modeling heritage.

Similar modeling concepts came from Shlaer and Mellor (1988), whose work came from a database background; Coad and Yourdon (1990), who described OO modeling as a merger of semantic data modeling and OO programming languages; and Martin and Odell (1992), with strong roots in previous non-OO modeling notations. Quite a number of other books followed soon after. These books had a strong data modeling aspect.

A different approach was taken by Cox (1986), Wirfs-Brock, Wilkerson, and Wiener (1990), Booch (1991), and others. This approach was based on the Smalltalk principles, was focused more on behavior than data, and dealt more strongly with programming issues. I described the tension between the OO Methodology (OOM) camp and the OO Programming (OOP) camp in an article in 1991 (Rumbaugh, 1991). The OOM camp took an analysis viewpoint and the OOP camp took an implementation viewpoint. To a large extent, the implementation viewpoint has prevailed in the public view of OO modeling. Both viewpoints are necessary for good software development, but much of the community does not even realize there is a difference.

4. LOGICAL DATA MODELING

Many people in both the database field and OO programming field want to rush to implementation. They want models that are little more than visual programs. But modeling is foremost about capturing essential information, not accidents of implementation. Logical models capture real-world knowledge about an application domain as well as requirements of an application. Logical data models are useful for understanding the meaning and constraints on data. Ideally, they avoid implementation and format issues and concentrate on the semantics of the information and the relationships of various kinds of data.

ER diagrams were intended to provide an abstract view of data requirements. It must be remembered that ER diagrams were originally not so well accepted by the database community, because ER diagrams encouraged thinking about data as organized into entities rather than tables. In other words, ER diagrams were object-based. Entities capture the identity of objects independently of the values of their attributes. The concept of identity violates Codd's basic premise about relational databases, namely that they can be represented as relations among pure data values; identity produces groupings of data values that can be distinguished by their identity even if their data values are identical.

Basic UML class diagrams are almost the same as ER diagrams. They both include the concept of things with identity (objects or entities), they attach data values to entities (attributes), and they relate entities (associations or relations). This is not surprising considering that they were originally inspired by ER diagrams.

UML has the ability to specify methods, but you don't need to use them if you are modeling only data. UML has the ability to specify visibility of attributes, but visibility is really a software engineering concept aimed at providing information hiding. I believe that information hiding is a fine programming practice, but it belongs to implementation, not to domain analysis. A logical data model can treat all attributes as freely visible and ignore the whole concept of visibility.

Some (but not all) variants of ER diagram limit multiplicity specification on relations to the binary choices *optional/required* and *single/multiple*. The UML specification of multiplicity as an open integer range is more general and also easy to understand. In any case, some variants of ER diagram also use this kind of specification. An integer range is not the most general form of multiplicity—for example, there is no way to say that it must be an even integer—but I think it is good enough for most software purposes, if not for symbolic evaluation.

5. GENERALIZATION

UML includes generalization, but so do some extended ER variants. It's an important logical concept that is part of ordinary discourse. Its main semantic purpose is to permit polymorphism among data types, something that occurs frequently in many problem domains. If you are modeling semantic information, generalization is a legitimate and

frequently necessary technique, although it was vastly overused in the original Smalltalk-based approaches.
The usual objection to generalization is that database management systems do not support it. This may be true (although that is changing), but models should be used to understand and capture knowledge about the application domain and requirements rather than prematurely constructing the implementation. After all, computers don't implement loops directly either; they are mapped from programming languages by compilers.
Generalization in a data model can be mapped onto SQL tables in a number of different ways; the best approach for a particular problem depends on the details of the problem. In general, there are three ways that generalization can be mapped to tables: push attributes from subclasses to superclasses, push attributes from superclasses to subclasses or factor each class into a number of separate parts.
In the first approach, all the attributes from a set of subclasses are placed in a single superclass. This approach is wasteful of storage—many of the slots will always be empty in any particular instance—but it preserves all information and polymorphism is possible. In fact, this approach permits dynamic reclassification of objects with no difficulty.

In the second approach, all the attributes from a superclass are placed in each of its subclasses. This approach is efficient, but polymorphism is not possible, because the inherited attributes of each sub-class appear to be different. In many applications, however, that is not a problem.

In the third approach, an object is broken into one sub-object for each ancestor class in the generalization hierarchy. This approach is efficient and permits polymorphism. Identity is lost, however, because each object consists of separate pieces. The pieces can be linked together with foreign keys, but this makes queries more complicated.

In all cases, a class maps into a table with a candidate key plus one column per attribute. Associations map into tables with one foreign key per associated class.

A number of books and articles discuss the transformation (Blaha and Premerlani, 1998). An implementation model of SQL tables can be expressed in UML class diagrams, but for such an implementation-level model generalization would not be used (it would have been transformed from the logical model).

6. CANDIDATE KEYS

Perhaps the most serious objection to UML and the main reason it has been rejected in database modeling circles is the lack of candidate keys on classes. (A candidate key is a set of attributes, the values of which uniquely identify a particular object.) OMT originally included candidate keys, but they were removed from UML during the standardization process. Recently, however, some profiles of UML for database design have added candidate keys and indexes back again. Using one of these UML profiles implemented in various modeling tools allows a database modeler to do everything that an ER diagram permits and more.

I think that candidate keys and indexes are useful and desirable for database design, because they are important for efficient implementation. I want to advance a different viewpoint for logical data modeling, however, one that will undoubtedly be controversial with many database modelers.
I don't think candidate keys should be specified at all on classes during logical data modeling. A candidate key is a statement that a particular set of attribute values uniquely identifies a particular object; no other object may have the same set of values. This is an incorrect way to view the world. Nothing is unique in the whole world; keys are always relative to some particular scope. In specifying candidate keys, there is an implicit assumption that the scope of the model is closed in some way and understood as being the scope for the entire model. The assumptions of exactly how it is closed are often unstated and, I suspect, usually not thought about. But making such assumptions works badly if the scope of the application changes, as it usually does eventually. Then things are no longer unique and the former candidate keys cause trouble. This is no idle theoretical speculation; when two companies merge, the assumption that employee IDs, product codes, and internal phone numbers are unique causes a lot of trouble for the integrators.

A better approach is to understand that keys are always relative to some scope, rather than inherent properties of classes. That is, keys are not properties of classes at all but of associations among classes. A scope should be modeled as a class in the model, for example, a company. A key is a set of attributes attached to both a scope class and a target class; with respect to the scope class, the key identifies a unique object within the target class (for example, an employee ID identifying an employee of a particular class). If there is only one instance of the scope (a singleton class), the cost of the scope class is small and it can be optimized out of the implementation. When additional instances appear, however, then the scope objects are important, and if they are included in the original data model, the model need not undergo a radical revision. (Think about an employee with two jobs.) You can't optimize away the scope in the implementation, but that would be true in any case, and if the optimization was generated it can be removed automatically.

I have said that a key is attached to both a scope class and a target class. Something that connects two classes is an association (or relation, in database terms). The concept in UML that identifies unique objects within a scope is called a *qualified association*. A qualified association is an association (such as *employed-by*) between a scope class (such as *company*) and a target class (such as *person*) with a qualifier (such as *employee ID*). A qualifier is a list of attributes owned by the association itself rather than either class. A qualifier is a key in the general sense: an instance of the scope plus an instance of qualifier values maps to a unique instance of the target. Because the qualifier is attached to an association, rather than the scope or target

class, new keys and new scopes can easily be added without conflicting with existing ones. If a merger of scopes occurs in the real world, the old scopes can be preserved in the data world for as long as needed. No single scope need be equated with the whole world.

Qualified associations are perhaps the most neglected important feature in UML. I would recommend that they be taught as a cleaner and more semantically accurate way of modeling keys.

7. OTHER UML MODELS

UML includes other model types mostly intended to specify behavior, such as state machines, interactions, activity graphs, and so on. Most of these are not relevant to data modeling. Nobody says you have to use them (well, some people do say you have to use them, but I'm not one of them).

In addition, the additional UML model types may be useful, even to a database modeler. Many data objects have a life cycle. State machine models can specify the life history of an object. For example, a loan might have the states *pending, accepted, rejected, outstanding,* and *closed.* Some attributes are common to all states, but each state defines additional attributes applicable to that state. For example, the *rejected* state defines the attribute *reason.* Defining a state machine model clarifies the relationship among the data in the various states. Other UML model types can be used to specify other kinds of dynamic behavior.

8. WHY USE UML?

UML class diagrams and ER diagrams are practically the same thing, especially if the right extensions of each one are compared. They use somewhat different notation, but I would hope that we could teach students to look below the surface notation. UML has extra concepts in class diagrams that are not needed in database design, and it has a lot of additional diagrams, but you don't have to use any of them if you don't need them. But ER diagrams are happily in use in the database area. Why even consider using UML at all?

The advantage of UML is that it is widely understood within the computing community, whereas ER diagrams are limited primarily to the database community. But a more important reason is that few models are limited to one community. Programs access databases to read and write data. It would be good if programs and databases were built from the same underlying logical models; using different models for the same underlying information greatly increases the chance of errors and often forces unnecessary translation of formats. Using UML for both data models and program design removes a lot of unnecessary obstacles. In fact, an ER diagram can be regarded as a species of UML diagram.

Note that for data modeling as for programming, UML can be used at two related but different levels: logical modeling and design. Logical modeling focuses on understanding problems at an abstract level; design modeling includes implementation concerns. Both UML class diagrams and ER diagrams are useful for logical modeling. They can also be used for design with the addition of aspects appropriate to the implementation medium: things such as methods, encapsulation, and visibility for programming languages; things such as mappings to tables, candidate keys and indexes, and triggers for databases. Both can be driven from the same logical model.

9. RECOMMENDATIONS FOR TEACHING

I have expressed my viewpoint based on years of experience observing computing practice, which often falls short of what is taught in universities. In fact, recently many universities have focused more on teaching specific programming skills, such as the use of particular languages or tools, rather than the attitudes and best practices that underlay successful software development. I would recommend the following emphases for teaching modeling:

- Emphasize the distinction between logical modeling and implementation. Stress the importance of logical modeling before making a lot of implementation decisions.
- Encourage students to see past superficial features (such as syntax and notation) to the underlying semantic essentials and concepts. For example, UML class diagrams and ER diagrams contain almost the same concepts. Associations and pointers are two sides of the same coin, and so on.
- Develop the ability to abstract. This is the most important skill for computing and one that is lacking in many practitioners. In the future, there will be few jobs in computing for those that lack this ability.

10. CONCLUSIONS

ER diagrams and UML class diagrams are not really very different. In fact, an ER diagram could be viewed as a kind of UML class diagram. UML class diagrams can be used profitably for logical data modeling with no loss of information and with some gain. For database design, additional capabilities such as candidate keys and indexes can be added using UML profiles, some of which already exist.

11. REFERENCES

Blaha, M. and Premerlani, W., *Object-Oriented Modeling and Design for Database Applications.* Prentice Hall, 1998.

Booch, G., *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

Coad, P. and Yourdon, E., *Object-Oriented Analysis.* Yourdon Press, 1990.

Cox, B., *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

Loomis, M., Shah, A. and Rumbaugh, J., "An object modeling technique for conceptual design." *Lecture Notes in Computer Science 276*, 192-202. Springer-Verlag, 1987.

Martin, J. and Odell, J., *Object-Oriented Analysis and Design*. Prentice Hall, 1992.

Rumbaugh, J., Blaha, M., Premerlani, W. Eddy, F., and Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

Rumbaugh, J., "Two cultures: object-oriented programming and object-oriented methodology." *American Programmer 4*, 10, (October 1991) pp. 4-10.

Shlaer, S and Mellor, S., *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, 1988.

Ullman, J., *Principles of Database Systems*. Computer Science Press, 1982.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*. Prentice Hall, 1990.

AUTHOR BIOGRAPHY

James Rumbaugh is a leader in object-oriented modeling and one of the founders of the OMT and UML modeling languages. He has led diverse probjects such as an object-oriented programming language, a generic graphics engine, a VLSI CAD system, algorithms for reconstruction of tomographic images, and helped build one of the first time-sharing operating systems. He worked for 26 years at GE R&D Center, 9 years at Rational Software, and 3 years at IBM. He has a B.S. in physics from MIT, an M.S. in astronomy from Caltech, and a Ph.D. in computer science from MIT.

Information Systems & Computing
Academic Professionals

STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 1055-3096