AN OBJECT MODELING TECHNIQUE FOR CONCEPTUAL DESIGN

M.E.S. Loomis, A.V. Shah, J.E. Rumbaugh

GE/Calma Company
9805 Scranton Rd.
San Diego, CA 92121

General Electric CR&D
P.O. Box 8, Bldg. K-1
Schenectady, NY 12301

Abstract

Our Object Modeling Technique is a software engineering technique for collecting and representing information about requirements and designs. It enforces the object-oriented notions of modularity, separation of implementation details from external behavior, and abstract data types. We have successfully applied the Object Modeling Technique to support conceptual design, followed by implementation using both conventional programming languages and an object-oriented language.

Introduction

The term "object-oriented" has become a buzzword that means different things to different people. It has been used synonymously with such terms as modularity, information hiding, encapsulation, and abstract data types. "Object-oriented" also refers to a system design philosophy, that makes objects active components of a system, responsible for manipulating their own internal states. By contrast, a procedure-oriented design philosophy makes data passive and manipulated by active procedures, whose definition is external to the data.

The Object Modeling Technique discussed here is a software engineering technique that we have developed for collecting and representing information about requirements and specifications. We have applied the technique to establish the scope of problem areas, to communicate precisely system specifications, to show clearly the dependencies between subsystems, to help in planning and scheduling projects, to structure databases, and to drive the coding and testing of software.

The object-oriented approach can permeate a system to various degrees. We have used the Object Modeling Technique in Calma product architecture work for communicating high-level interfaces and functionality. We have used it to specify applications that will be implemented using conventional programming languages (FORTRAN, C). These languages, however, implicitly encourage a procedure-oriented design philosophy. We have also used the Object Modeling Technique in conjunction with an object-oriented programming language.

The benefits of taking an object-oriented approach stem primarily from the modularity of the resultant specifications and codes. This modularity provides for well-specified interfaces, and encapsulates the data structure and operations of the objects that make up a system. As a result, the objects become potentially reusable code modules. Each object is responsible for its own actions; side-effects are not invisibly strewn throughout the system.

An object model is a model diagram, method descriptions, and a glossary of class definitions. We refer to an object model diagram constructed using our Object Modeling Technique as an OMT diagram. This paper discusses the major concepts and representation of object classes, attributes, methods, and relationships, and outlines a modeling procedure.

Object Classes

An *object class* describes a set of things (physical or abstract) that have the same behavior and characteristics. Every object class has a name, a set of attributes, and a set of methods. The attributes of an object class are also known as its fields or variables; the methods are also known as operators. Object classes are commonly referred to simply as classes, and we now begin using that term.

For example, "Chair" is the name of a class; the "chair in the corner of the room" is one object in the class Chair, the "chair in the center of the room" is another.

An *object instance* has specific values for each of the attributes defined in its object class. For example, the attribute SeatingMaterial of the class Chair may have the value "cane" for a specific chair object.

One object instance can be associated with other object instances through relationships defined for their classes. For example, a Chair object may be "next to" a Wall object, and may be "contained in" a Room object.

Note that the term "object" is often used in the literature and common discussion to refer to both an object class and to a specific object instance, the reader having to derive the distinction from context. The same confusion between "entity class," "entity" and "entity instance" plagues other data modeling techniques. Here we will consistently use the term "class" to mean object class, and "object" to mean an instance of a class.

A class is indicated in an OMT diagram by a box containing three sections, separated by horizontal lines (Fig. 1). The first section contains the class name; the second section contains a list of the class attributes; the third section contains a list of the class methods. The attribute and method sections can be omitted during early development or if it is unimportant to show them for a particular purpose.

An important characteristic of an object-based system is separation of external and internal specifications. Internals should be hidden from users so that implementation considerations do not influence the logical use of a system. For example, the developer of a graphics class Circle has to be concerned about data structures to store information about circles in a database and about algorithms to scale and move circles. But a user of the class Circle need only know about the attributes and behavior of the class.

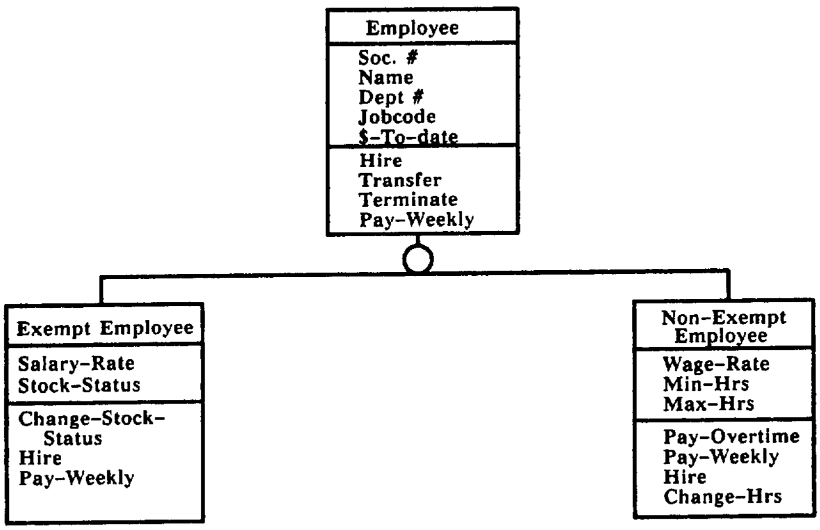

Fig. 1 Example generalization relationships.

The Object Modeling Technique is used first for defining the class' external specifications, which are visible to users of the objects, then for defining internal representations. Algorithms are specified with other techniques that show flow, timing, and so forth. Successive refinement of the data definition and structure can drive all the way to code.

Attributes

A class has a set of attribute specifications. Every attribute has a name and an optional type and annotation. A class cannot have more than one attribute with the same name. An attribute is indicated in an OMT diagram by listing its name in the second section of the appropriate class box.

The attribute name is followed by a type, which is the name of the class that determines the valid set of values for the attribute. The type can be of any class, including a user-defined class. The familiar types include character, integer, float, double, short, and so forth, e.g., Color:character, X:float, BirthDate:date. The type is optional in an object model reflecting a conceptual design. However, it must be specified when the model is used to reflect a detailed design.

Methods

A class has a set of methods, each with a name and behavior. A class cannot have two methods with the same name. A method is indicated in an OMT diagram by listing the name of the method in the third section of the class box.

A method's external effects are its behavior. The method's behavior is described in text that accompanies the OMT diagram. This behavioral description specifies the method's input and output parameters, effects on the attributes of the classes involved, the names of other methods to be invoked by this method, and the sequence of their invocation.

For example, consider the class Array, with its method Append. Append's behavior can be specified as follows:

Append (Array, Object) -> Array

Inputs are an Array and another object.

Increments the Size of the Array by 1.

Copies the object into the Array.

The algorithm for the method is not detailed in the conceptual object model, but rather would appear elsewhere in the internals specification.

Another example is the method NewVersion in the class Design:

NewVersion (Design, Time, Date) -> NewDesign

Inputs are a Design object, the current Time and Date.

Increments the Design's VersionNumber by 1;
records Time and Date in the VersionTime and
VersionDate attributes respectively.

Makes this Design object the current Design.

The algorithm for versioning is not detailed in the conceptual object model. Forward differences, backward differences, or complete copies may be saved, but this is internal detail that is hidden from the user and is not external to the class.

Methods to simply set, retrieve, or change attribute values typically are not shown in a conceptual object model, but are assumed.

Relationships: General Classification

A relationship is a logical association between classes. Our Object Modeling Technique requires that relationships be between two classes. A relationship is shown in an OMT diagram by a line connecting the pair of classes. Each relationship is defined by a pair of class names and a relationship type.

There are three types of relationships, each of which has its own distinct semantics: generalization, aggregation, and association. It is in this area of relationships that our Object Modeling Technique differs most from the conventional object-oriented paradigm of Smalltalk80 [1], which directly supports only the relationship notions of generalization. We need to represent aggregation and association, especially in developing CAD systems. Others (e.g. [2,3]) include some of these notions in their object-oriented languages, but without sufficient details of semantics or representation.

A notion that pertains to several types of relationships is cardinality, which indicates the number of objects of one class that are related to objects of the other class. The cardinality of a class in a relationship is shown in an OMT diagram by an annotation on the line at the point where it meets a class box. Although the Object Modeling Technique supports more precise specification, in practice the most important distinction is between a case of finite cardinality, usually {0,1} or {1}, and one of infinite cardinality, usually {0+}. These cases are so common that they are denoted by special shapes on the relationship line: {0,1} is a small open circle, {1} is a simple line, and {0+} is a small solid circle.

Generalization Relationships

Generalization is used to show families of similar objects. Generalizations are also called "a kind of" or "is a" or "category" or "specialization" relationships. The notion of generalization appears in the database world in semantic data models [e.g., 4,5], but is not well supported by commercial database management systems [6]. By contrast, generalization is the primary type of relationship that is supported directly by object-oriented programming languages such as C++ [7], Objective-C [8], and Smalltalk80 [1].

In a generalization, one class has the role of \fIsuperclass\fP and the other has the role of *subclass*. The subclass is a specific case of the superclass. For example, the superclass Student could have subclass GraduateStudent. It might also have a specialization with subclass UndergraduateStudent.

Subclasses are shown in an OMT diagram as connected to the superclass by a line ending in a large circle, on top of a line which sprouts one line for each of its subclasses (Fig. 1). A generalization can be read:

<subclass> is a kind of <superclass>.

<superclass> can be a <subclass1> or a <subclass2> ...

For example, in Fig. 1,

ExemptEmployee is a kind of Employee.

NonExemptEmployee is a kind of Employee.

Employee can be an ExemptEmployee or a NonExemptEmployee.

If all the pertinent subclasses of the superclass are not shown, then a "floating" sprout is drawn (Fig. 2). A Student can be a GraduateStudent or an UndergraduateStudent or neither. UnclassifiedStudent is not shown as a class because it has neither its own attributes nor methods in this example.

The attributes, methods and relationships of the superclass are inherited by its subclass through the generalization. That is, whatever semantics apply to the superclass automatically apply to the subclass as well. For example, in Fig. 1, each ExemptEmployee has attributes SalaryRate, StockStatus, and those inherited from Employee: Soc#, Name, Dept#, JobCode, and $toDate. Its methods are ChangeStockStatus, Hire, PayWeekly, and two inherited from Employee: Transfer and Terminate. Inheritance leads to reusability of superclass attribute, method and relationship specifications and implementations.

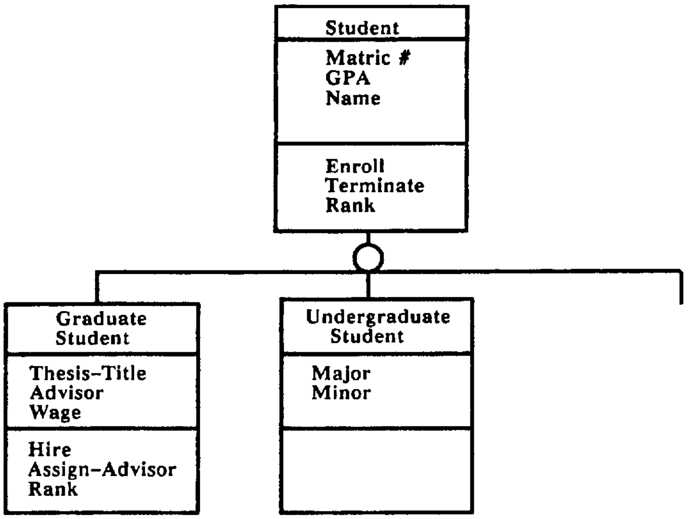

Fig. 2. Example generalization relationships

The search for a subclass' attribute or method proceeds from the subclass "up to" its superclass. When the desired name is found, the value or method there is used. For example, the JobCode value for a particular NonExemptEmployee object would be implicitly found in its superclass Employee object. Inheritance can proceed up any number of levels of generalization.

A subclass can specialize the actions of an inherited method by including the name of that method in the subclass definition. Two forms of method specialization are *override* and *augmentation*. With override, the subclass' method is found first and executed instead of the superclass' method. With augmentation, the subclass' method is found and executed first, then the same-named method in the superclass is executed. The distinction between override and augmentation is specified in the behavioral descriptions of the methods.

Aggregation Relationships

Aggregations are also called "a part of" relationships, and are used to show groupings of objects. They appear in the database world in a restricted way as bill-of-material structures, which are conveniently represented by network data models, and are less obvious in hierarchic and relational databases. Aggregation is ignored by the popular object-oriented programming languages, e.g., [1,7,8].

In an aggregation, one class has the role of assembly and the other has the role of *component*. Each assembly instance is composed of a set of component instances. For example, the assembly Car could have a component Door. Car might also have an aggregation with component class Hood.

The number of objects of the component class that are allowable for an assembly object may be any valid cardinality, depending on the real world being modeled. For example, the aggregation between assembly Car and component Door has cardinality 1-to-many; the aggregation between assembly Car and component Hood is 1-to-1.

Aggregations are shown in an OMT diagram by a line, terminated at the assembly class box by an arrow and at the component class box by the appropriate cardinality annotation. Several examples are shown in Fig. 3. An aggregation can be read:

<component> is part of <assembly>.

Going from the assembly to the components:

<assembly> contains <cardinality1> <component1>
and <cardinality2> <component2> ...

For example in Fig. 3, the relationships are:

Door is part of Car.
Hood is part of Car.
Car contains many Doors and a Hood.
Bicycle contains two Wheels and a Frame.
Shipment contains many Cars and many Trucks.

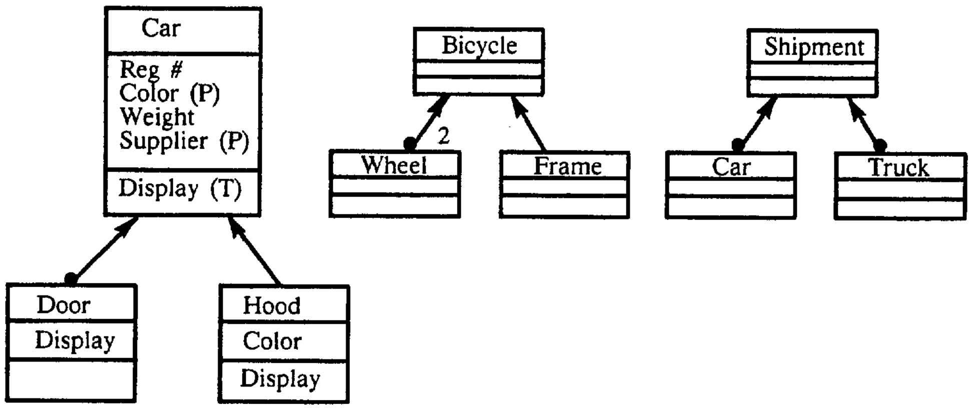

Fig. 3. Example aggregation relationships

The attribute values of an assembly class can be selectively propagated to each of its component classes. In contrast to inheritance, which applies to all the attributes of the superclass, propagation is done on an attribute by attribute basis. There are two kinds of attributes: transitive and intransitive. Transitive attributes (e.g., Color, Material, Supplier) apply to the entire assembly and all its components. Intransitive attributes (e.g., Weight, Cost, Length, Strength) apply only to the assembly as a whole.

An attribute to be propagated to component classes is shown on an OMT diagram by appending (P) to the attribute name in the assembly class. The search for a component object's attribute value ends in the component object if the desired attribute is found with a non-null value, or proceeds "up" any number of levels of aggregation, stopping when the target (P)-annotated attribute is found with a non-null value. For example (Fig. 3), the Supplier value propagates to both Door and Hood. Neither Reg# nor Weight is marked for propagation. Color always propagates to Door, but propagates to Hood only if the value for Color in Hood is null.

The methods of a component class can be triggered selectively by invoking the methods of its assembly class. A triggering method is shown on an OMT diagram by appending (T) to the method name in the assembly. The triggering method iterates over the members of the assembly, invoking the same-named methods in its component objects, according to a specified sequence. For example, invoking the Display(T) method of the assembly Car object invokes the Display method of each of its component objects.

Association Relationships

Associations are user-defined relationships. The relationship has a specific, user-defined role relative to each of the associated classes. For example, in an association between Person and Car, the relationship could have the role Owner relative to Person and the role Asset relative to Car.

Associations appear in the database world as inter-record relationships (i.e., set types in CODASYL databases, parent-child relationships in hierarchic databases, and foreign-key relationships in relational databases [6]). By contrast, associations are not directly supported by the popular object-oriented programming languages (e.g., [1,7,8]).

An association is shown in an OMT diagram by a solid line connecting two class boxes. The role of the association relative to a class is named on the line near the class box (Fig. 4). Appropriate cardinality annotations or symbols are affixed to the ends of the line. An association can be read:

<role1> of <class1> is <cardinality1> <class2>.

For example, in Fig. 4, the associations are:

Responsibility of Employee is many Projects.
Manager of Project is one Employee.
Input of Machine is many Materials.
Output of Machine is many Materials.
Consumer of Material is a Machine.
Producer of Material is a Machine.

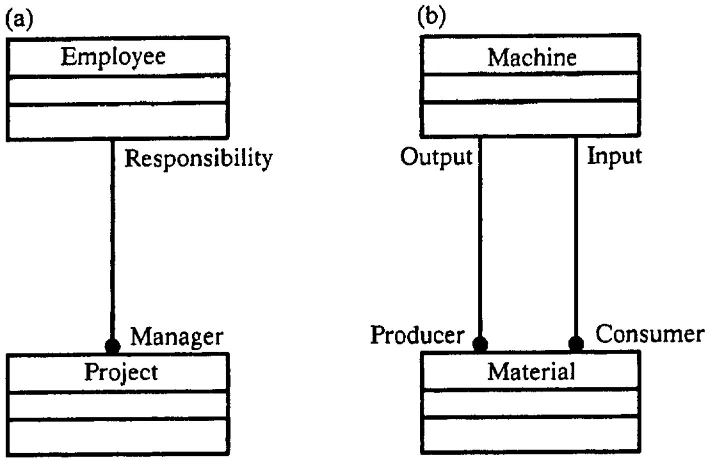

Fig. 4. Example association relationships

There is direct interchangeability between associations and attributes. This allows programming languages, that do not support associations directly, to support them indirectly. For example,Project in Fig. 4a could be implemented as a class with attribute Manager, whose type is the Employee class. The Employee class could have an attribute Responsibility, whose type is a set of keys for the Project class. Similarly, Machine in Fig. 4b could be implemented as a class with attributes Input and Output, each with its type the Material class. The Material class could have attributes Consumer and Producer, each with its type a set of Machine objects.

The role name of an association defaults to the pertinent class name if a role name is not specified explicitly. When default role names are used, an association may be given a verb name to help explain its meaning. For example, the association between Person and Car could be given the verb name "owns / is owned by". The relationship would be read:

Person owns Car; Car is owned by Person.

Qualified Relationships

Qualified relationships are special cases of associations, in which an object selection mechanism is added.

In Fig. 5a there is a 1-to-many association between classes Company and Person, with the role of Emp relative to Company. Fig. 5b further specifies this association. The small box appended to Company's class box contains Emp#. The combination of a Company object and an Emp# value uniquely identifies a Person object. The qualifier, Emp#, is relative to the Company class;

in itself it is not necessarily a unique identifier in the sense of a primary key in a relational database.

The class Person may have different identifiers in the context of various usages. For example, Fig. 5b adds to Fig. 5a a qualified relationship to further specify the Airline – Person association. The combination of an Airline object and a FrequentFlyer# identifies a Person object.

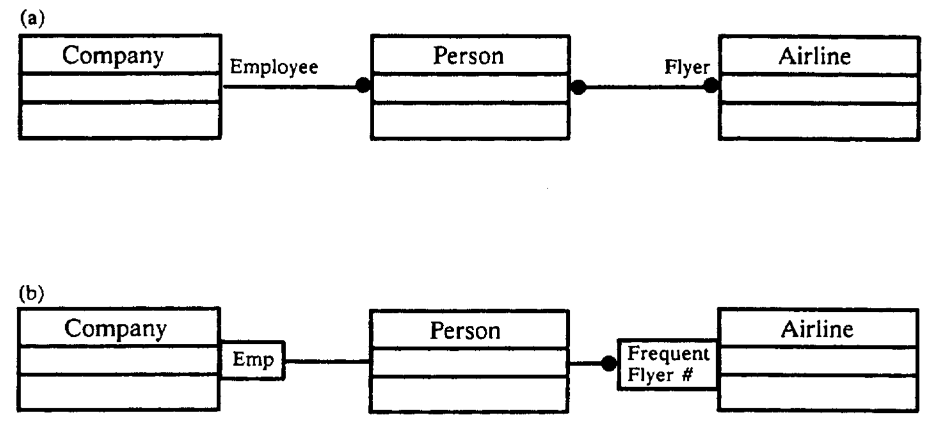

Fig. 5. Association relationships, and corresponding qualified relationships

Modeling Procedure

The fundamental steps to using the Object Modeling Technique to prepare a conceptual design include the following:

1. Talk to experts in the subject area to be modeled and write a statement of the scope of the effort.
2. Create an initial list of classes, whose names are commonly the nouns in a description of the area being modeled. Write a definition for each class.
3. Find the relationships between the classes and begin using the OMT notation. Typically relationships are most easily found in the following order: associations, generalizations, aggregations, qualified relationships.
4. Annotate the aggregations and associations with their correct cardinalities. These often change later when the problem is better analyzed.
5. Review the model with additional subject-area experts.
6. List the methods for each class.

7. Trace the path in the model needed to evaluate each method. If paths or classes are missing, add them. If a path is ambiguous, add information to make it unambiguous.
8. List the attributes of each class.
9. Collapse relationships where possible.
10. Review the model with additional subject-area experts.

There is a great deal of feedback and iteration among these steps, especially steps 3 through 9. The process of identifying methods may cause the modeler to introduce additional generalizations, or the process of identifying attributes may cause the modeler to combine classes. The increased level of semantic knowledge captured at each step may cause a modification of the structure already developed. Each iteration of model development makes the specification increasingly explicit. The hard parts are performing the correct abstractions to simplify things.

References

[1] Goldberg, A. and D. Robson. Smalltalk-80: The Language and its Implementation, (Addison-Wesley, 1983).

[2] Atwood, T.M. "An Object Oriented DBMS for Design Support Applications," CompInt 85 Computer Aided Technologies, IEEE, Montreal, (Sept. 8-12, 1985), pp. 299-307.

[3] Woelk, D., W. Kim and W. Luther. "An object-oriented approach to multimedia databases," Proc. SIGMOD '86, ACM, 1986, pp. 311-325.

[4] Smith, J. and D.C.P. Smith. "Database abstractions: aggregation and generalization," ACM Transactions on Database Systems, Vol. 2, No. 2., 1977, pp. 103-133.

[5] Elmasri, R.A., J. Weeldreyer, and A. Hevner. "The category concept: An extension to the entity-relationship model," in Data & Knowledge-Engineering, (North-Holland, 1985), pp. 75-116.

[6] Loomis, M.E.S. The Database Book, (New York NY: Macmillan Publ. Co., 1987).

[7] Stroustrup, B. The C++ Programming Language, (Reading MA: Addison-Wesley Publ. Co., 1986).

[8] Cox, B.J. Object Oriented Programming: An Evolutionary Approach, (Reading MA: Addison-Wesley Publ. Co., 1986).