December 2001

Experiences with the Unified Modeling Language (UML)

Shouhong Wang
University of Massachusetts Dartmouth

EXPERIENCES WITH THE UNIFIED MODELING LANGUAGE (UML)

Shouhong Wang
Department of Marketing/Business Information Systems
Charlton College of Business
University of Massachusetts Dartmouth
swang@umassd.edu

Abstract

Our experiences with the Unified Modeling Language (UML) indicate that the UML is unnecessarily complicated. It provides an insight of object-oriented modeling by demonstrating that few modeling constructs beyond class, inheritance, messages, and data flows are needed in object-oriented systems modeling.

Introduction

Since the object-oriented (OO) approach became popular in the late 1980s, there have been a variety of methodologies for OO analysis in modeling information systems (Eckert and Golder 1994). The proliferation of methodologies in the OO field has caused confusion in all information technology and computer software related fields. For many reasons, the OO field needs a "standard" language that can be used for software engineers to reconcile and coordinate the needs of various stakeholders. This concern triggered the initiation of the Unified Modeling Language (UML) (Booch et al. 1999, BRJ for short) among other popular modeling languages. The UML is a graphical language for visualizing, specifying, and documenting the artifacts of software-intensive systems. The UML serves as the standard language of blueprints for software in the Object Management Group (OMG), and is becoming an international standard for information technology (Kobryn 1999).

However, there has been no lack of criticism of the UML (e.g., (Henderson-Sellers and Firesmith 1999)). While the issue of whether the UML deserves an international standard for information technology is still under debate, one fact is clear that the UML is overly complex (Siau and Cao 2001). In our view, the major reason behind this fact is the lack of provision of support in the UML at the business information systems analysis level. This paper makes comments on the UML based on our experiences in OO business information systems analysis.

Principles of the Design of a Modeling Language

When design a general modeling language, three principles must be considered: *parsimony*, *compatibility*, and *self-inductiveness*. Parsimony means that the constructs used in a modeling language should be confined to minimal while they are still able to reflect all essential characteristics of the OO paradigm in modeling the world related to computer software. The parsimony principle suggests that the general constructs in a "standard" modeling language must be as simple as possible for the use of the language as a communication tool cross disciplines (e.g., business and computing). Any redundant constructs must be eliminated in a "standard" modeling language.

Compatibility requires that the general constructs used in a "standard" modeling language should be easy to convert to a popular computer OO language(s). Currently, C++ and Java are considered commonly used OO languages for the general purposes of OO programming. Hence, the general constructs of a "standard" modeling language must be harmonious with the syntax of C++ or Java. The principles of parsimony and compatibility require that the design of a modeling language involves a trade-off. Sophisticated modeling languages provide a variety of measures to model the real world; however, the modeling outcomes are not easy to map into the computer world unless a mapping model is available. For instance, constraints (BRJ, p22) represent a type of association between object classes; however, commonly used OO programming languages (C++ and Java) do not support implementation of such association directly. In other words, one must use essential ways (messages and/or inheritance relationships) to implement the constraint association. In fact, the real world is so complicated that the types of associations among the classes are virtually countless. The use of construct "association" does not help to convert the association directly to the computer world. Thus, the need for a distinct notation for an arbitrary association among the classes becomes questionable since few computer programming languages are able to implement an association without further specifications.

A good modeling language is self-inductive; that is, it has a built-in instrument to verify the correctiveness of the resultant model. As shown later in this paper, the specification of messages with the companion data flows in an OO model is a way to check the completeness and consistency of the model results. In our view, the following elementary constructs, which come mostly from (Coad and Yourdon 1991) except for data flow, can represent all the constructs of the UML for business information systems—(1) *Class*; (2) *Attributes*; (3) *Operations* (*Method* or *Message*); (4) *Data flow*; (5) *Inheritance*—as shown in Figure 1.

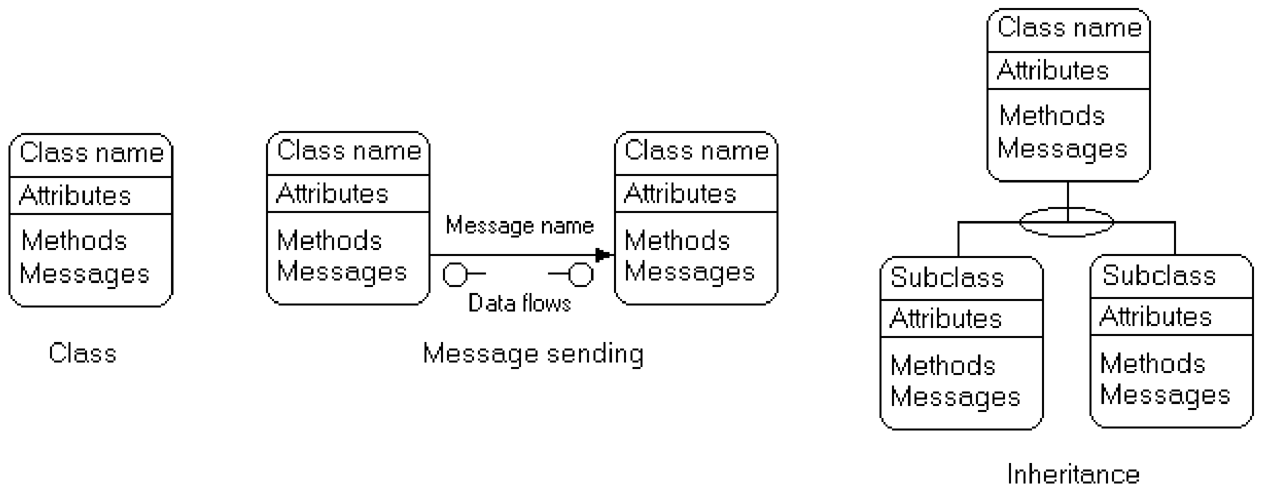

Figure 1. Generic Constructs of Object-Oriented Modeling

General Principles of Object-Oriented Modeling

A particular software-related modeling language is usually associated with a philosophy of computer programming, and a full evaluation of a modeling language should have a much broader coverage than a discussion of its constructs. It is rarely possible to prove one modeling language is better than others in all aspects. Nevertheless, from our point of view, the "goodness" of a modeling language could not be concluded without investigating the general principle of the modeling language at the very top level.

Generally, if a modeling language is OO, any thing of a particular system being modeled can be described as an object by specifying the class in conjunction with the attributes (i.e., data) and methods (i.e., internal operations and messages) that manipulate the object data or request services from other objects. It integrates data, processes, inheritance structures, and dynamic interactions into a single OO diagram. Hence, the systems modeled in an OO modeling language might be easy to maintain and reuse. This is why OO modeling approach is superior over the traditional structured approach in terms of its ability to describe a variety of entities and their relationships in the complex information technology environment.

This principle sounds so basic to the OO field; however, it is not reflected in the UML. Partially because of its historical background of development, the UML employs mixed and redundant constructs. We will have a close look at those questionable constructs used in the UML in the next section.

As realized in the OO community, the OO approach blurs the system analysis stage and logical design stage during the software development life cycle. In the case of using a modeling language, one expects that little expansion is needed to implement the system, provided that the system model defines the object classes properly. To take advantages of information technology for software engineering, people use CASE (Computer Aided Software Engineering) tools. An ineffective "standard" modeling language might build more barriers for the next generation of CASE tools. For instance, given a construct of "association" in the system model specified by the UML, the software designer (or programmer) must figure out what the association exactly means for the computer program. CASE tools can do little work to help the designer (or programmer) in designing the software structure since the model actually tells nothing specific about the "association." The fact is that the system model in the UML demands more creative activities in the later stages of the system development life cycle. The elementary constructs shown in Figure 1 provide a generic instrument for OO modeling. As demonstrated in the next section, many fundamental modeling constructs in UML can be replaced by the elementary constructs shown in Figure 1. The system modeled with these elementary constructs can be less ambiguous and more accurate.

Deficient Constructs in the UML

In this section, we use examples, but not exhaustively, to demonstrate why and how constructs in the UML can be replaced with the elementary constructs shown in Figure 1, and what constructs of systems modeling are missing in the UML.

Responsibility

The UML models responsibilities for a class by using a separate compartment of the class icon for the free-form description of responsibility (BRJ, p53). In the view of systems modeling, this part is not significantly different from ambiguous natural languages. On the other hand, in the view of implementation, no computer language can implement "responsibility" unless the model narrates exactly what the responsibility is. In fact, responsibility is a part of *goal* which should be an object. If the responsibilities are truly important for software systems modeling, they must be modeled in OO terms, as shown in Figure 2.

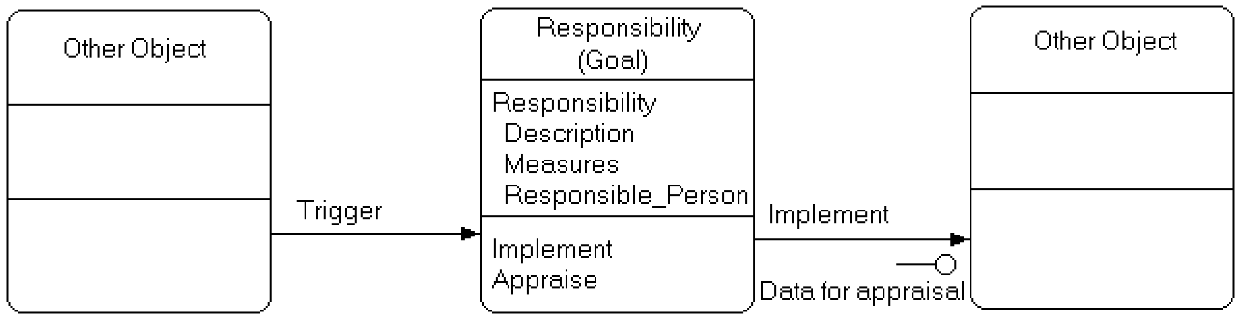

Figure 2. Modeling Responsibility in Object-Oriented Terms

Dependencies

The UML uses dotted lines to describe dependency (BRJ, p63). From the viewpoint of OO modeling and implementation, dependencies can be modeled simply by using messages. As an example, the dependency described in (BRJ, p64) can be modeled in the more general and accurate way of message sending, as shown in Figure 3.

Associations

Association (BRJ, p65) is an ambiguous term, as briefly discussed earlier in this paper. For example, "works for" might be an "association" between Person and Company, but it proves little semantic information. From the viewpoint of OO modeling, such an association in this case can be an inheritance relationship if the association means "is_a_member_of", or a message if the association means, for instance, "get_pay". Unless the association is specified by inheritance and/or message, it can never be mapped onto the computer software world.

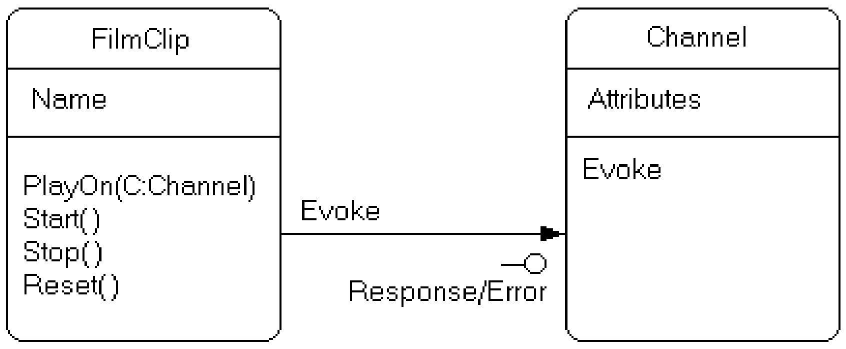

Figure 3. Modeling Dependency in Object-Oriented Terms

Physical Elements

The UML has special treatments on physical elements, such as server nodes (BRJ, p20) and actors (BRJ, p221). Conceptually, these physical elements are objects in the same way as other physical elements, such as customer and employee, for business information systems (Wang 1999). The reason why the notation for server nodes and actors should be different from that for customer and employee is not justified. In fact, for integrated OO systems modeling, any elements related to the information system can have their objects within the computer for various purposes, including information infrastructure management and system evaluation.

Note

In the OO world, a note (BRJ, p76) is an object which can be attached to a host object through an inheritance relationship. A note should be, for instance, an online document, if it is really useful for the user. Again, there is no need for a free-format notation for note.

Interface

The UML defines an interface as "a collection of operations that are used to specify a service of a class or a component" (BRJ, p18). Actually, an interface is generally an independent object. It can have its own attributes in addition to a set of operations, as shown in Figure 4.

Package

The UML defines a package as group elements that are semantically close and that tend to change together (BRJ, p169). The idea of package comes more likely from "folder" in the term of Windows operating system. In OO terms, package is a type of metaclass. All elements related to the package are the attributes of the metaclass. Since OO programming languages normally support multiple inheritance structures, the construct of package is actually redundant. Furthermore, the use of the construct of package might result in incompleteness of modeling because generic operations are usually associated with a package but are unable to be described in the UML. For instance, the construct of a package of business rules (BRJ, p171) does not allow the modeler to specify whether these rules share common operational methods of inference.

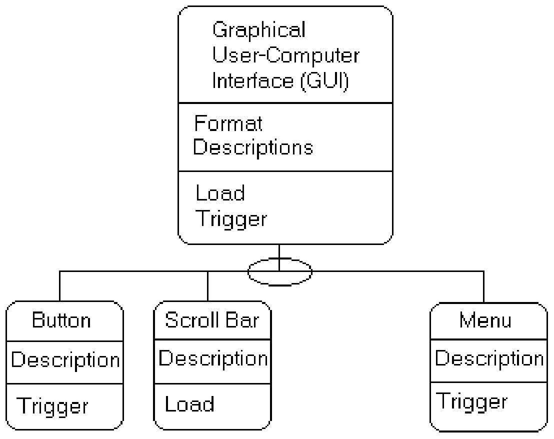

Figure 4. A User-Computer Interface Is an Object

Use Case and Event

A use case (BRJ, p219) is used to identify objects, but is not an object. To model a use case for the sake of systems analysis and design, one must apply an additional modeling process and use non-object-oriented constructs. This inconsistency between the constructs and the object-oriented concept creates great gaps between the modeling outcome and the object world (Pooley and Stevens 1999). Research (e.g., (Meyer 1997)) indicates that use cases are not a good tool for finding classes.

In many cases, a use case is equivalent to an event. However, the UML uses a special graphical representation for events (BRJ, p278). Actually, it is again redundant. Time-dependent aspects of a system can be expressed in the form of event object classes. Events are associated with state transitions of the system and explicitly express the system's dynamic properties. An event object has its attributes (e.g., time and signals) and its operations (e.g., trigger other objects). An event is initiated by a message sent by other objects, except for the very top event object which is often defined as the “system clock” (Wang 1999).

Consistency Control Mechanism

To eliminate the syntactic errors of inconsistency as well as semantic errors in missing representation of the result model, a good modeling language must provide a mechanism to ensure the consistency of the result model. The consistency control mechanism will help the modeler to verify the model, and make it possible for CASE tools to build facilities for consistency analysis. Unfortunately, the UML does not support any means for consistency control. Our solution is that data transmitted by messages between object classes are explicitly annotated and used for consistency control (Figure 1). Data attached to the passing messages in an OO program (C++ and Java) could be those parameters in a message and/or a “return” operation in response to a message. The consistency control mechanism of data flows is similar to what we have used in the data flow diagram, although the concept of data flows in the OO paradigm is quite different from that in the data flow diagram approach. Research (Wang 1999) has experienced the benefit of data flows in OO systems modeling in consistency control. We believe that annotation of data flows along with the host message in OO modeling is useful for checking the completeness and consistency of the modeling results.

Discussions

This paper explains why and how second-order concepts and tedious constructs suggested by UML can be eliminated from a “standard” modeling language. However, it does not stop there. It highlights a view for information systems that messages serve as vehicles conveying data communication between the modules, and suggests the use of data flows to elaborate systems models and check the completeness of the modeling results. We wish the information technology field to invent a concise, truly unified, and pragmatic modeling language as a framework that can be used for effective management information systems modeling.

References

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1999.
Coad, P., and Yourdon, E., *Object-Oriented Analysis,* Englewood Cliffs, NJ: Yourdon Press, 1991.
Eckert, G., and Golder, P., Improving Object-Oriented Analysis, *Information and Software Technology,* (36:2), 1994, pp.67-86.
Henderson-Sellers, B., and Firesmith, D. G., Comparing OPEN and UML: The Two Third-Generation Development Approaches, *Information and Software Technology*, (41), 1999, pp.139-156.
Kobryn, C., UML 2001, *Communications of the ACM*, (42:10), 1999, pp.29-37.
Meyer, B., *Object-Oriented Software Construction*, Upper Saddle River, NJ: Prentice Hall, 1997.
Pooley, R., and Stevens, P., *Using UML,* Reading, MA: Addison-Wesley, 1999.
Siau, K., and Cao, Q., Unified Modeling Language (UML): A Complexity Analysis, *Journal of Database Management*, (12:1), 2001, pp.26-34.
Wang, S., *Analyzing Business Information Systems: An Object-Oriented Approach*, Boca Raton, FL: CRC Press, 1999.