

Choosing the Right Object-Oriented Method

by Edward Colbert

Absolute Software Co., Inc., 1444 Sapphire Dr., Carlsbad, CA 92009-1200
Voice: (760) 928-0612, FAX: (760) 929-0236, E-mail: colbert@abssw.com

Biography

Ed Colbert has been teaching object-oriented methods and software engineering since 1982. His own Object-Oriented Software Development Method has been praised for addressing real-time issues and for quickly communicating the information developed during analysis and design. He has chaired and vice-chaired Special Interest Groups of the Association for Computing Machinery (ACM). Outside the United States, Ed has recently delivered presentations at ObjectExpo-Europe (England, 1992), LOOK (Denmark, 1992), OOP (Germany, 1992), and SCOOP-Europe (England, 1991). He is a graduate of the University of Michigan (M.S. Computer & Communication Sciences, 1981; B.S. (honors) Chemistry and Biology, 1979).

Abstract

Object-oriented methods can substantially improve achievement of software-engineering and system-building goals. Choosing a method means looking at the support it offers for the object-oriented paradigm, for the application domain of the software developer, and for software-engineering principles, practices, and goals. In this presentation we shall explore how these needs are answered by various current object-oriented methods, including Booch, Buhr, Coad, Colbert, Rumbaugh, and Shlaer-Mellor. Understanding of basic terms such as *object*, *class*, and *inheritance* will be assumed. A book by the presenter, comparing current methods, is scheduled from SIGS Publications this year.

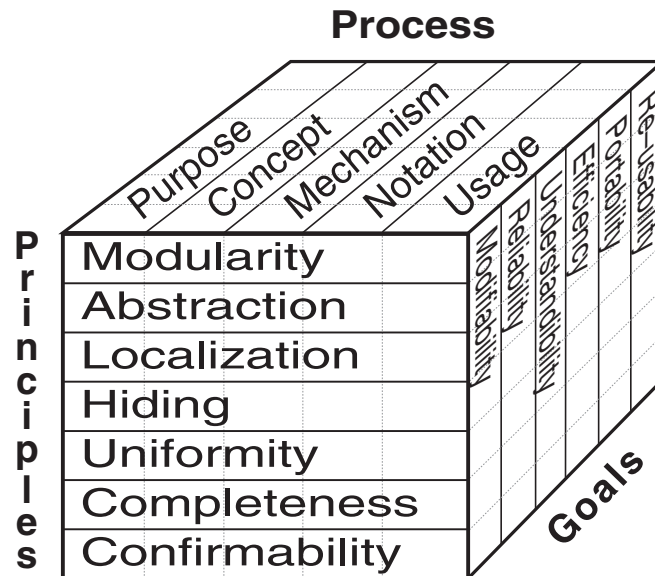
Introduction

The ongoing demand for more, bigger, and better software drives the software industry to improve. A recent development that may qualify as an advance is object-orientation, which has evolved simultaneously in programming languages, databases, artificial intelligence, and development methods. With object-oriented methods, developers are building better software and better systems. The challenge for the project manager, chief scientist, and software engineer is to identify an object-oriented method that can support the full development process and the type of systems to be developed, and can help achieve their software engineering goals.

In 1975, Ross, Goodenough, and Irvine [14] set out fundamental principles, practices, and goals of software engineering. Identifying a “fundamental process” common to the software-engineering activities of requirements analysis, design, implementation, coding and debugging,

and tuning, they applied it to principles and goals,¹ using a cube like the one shown in the figure. For example, as the interaction of *purpose* (part of the “fundamental process”), *confirmability* (a principle), and *modifiability* (a goal) during software design, they gave the formula “The purpose of a design choice may be to structure a system so the effects of a change can be predicted with assurance.”

Because the object–oriented paradigm evolved simultaneously in different areas of software development, the fundamental concepts are not completely standardized. Each object–oriented method developed in a specific software domain (e.g., information systems, real–time systems), although there has been some cross–over. As a result, some methods are best in the development of applications for which the method evolved, while others can be applied more generally. Even object–oriented methods that evolved in the same software domain may differ enough in process and notation that software engineering goals are affected.



Software Engineering Fundamentals

Object–Oriented and Function–Oriented Methodology

Function–oriented methodology focusses on the functions of a system. Function–oriented methods codified software engineering *principles* of abstraction, modularity, localization, and hiding (encapsulation) in terms of the functional abstraction, and defined a *process*, including concept, mechanism, and notation, to support the function abstraction. Such methods also formalized top–down functional decomposition. Data–oriented methods are a specialization of function–oriented methodology. These methods start by analyzing the flow of data between functions (e.g. Yourdon & DeMarco’s Structured Analysis and Design), or the structure of data (e.g. Jackson Structure Programming and System Development, Chen’s Entity–Relation Diagrams), then derive a functional decomposition. Since first and second generation programming languages (e.g., Assembly, FORTRAN, ALGOL) are also function–oriented, function–oriented designs are easy to implement in them. The use of Entity–Relation Diagrams allowed these methods to develop systems that involved relational databases.

¹ They limited their discussion to the “four fundamental goals: modifiability, efficiency, reliability, and understandability”. In my analysis I add re–usability and portability, which if not “fundamental” are important at this level of abstraction for most systems.

Function–oriented methodology sometimes falls short of software engineering *goals*, because function–oriented *processes* and *principles* provide weak support. For example, since the only *mechanism* to support *hiding* is the function, each data item must be either local to a function, or global to all functions in the software. From this reason alone, the software is less *understandable* since a reviewer cannot easily see which functions can read or change a global data item; less *reliable* since a global data item will necessarily be within reach of every function; and less *modifiable* since “the effects of a change [cannot] be predicted with assurance”. Manual reviews and enforcement of procedures can compensate for these shortcomings, but this can be expensive for large systems.

Object–oriented methodology focusses on the objects and classes of objects in a system. An *object* is an abstraction that formalizes fundamental relations between data and operations (e.g. procedures and functions) that manipulate them, and between concurrent processes and operations they perform upon request. A *class* is a re–usable description that can be used to build objects. Object–oriented methods codify software engineering *principles* in terms of the object and class abstractions, and define a *process*, including concept, mechanism, and notation, to support them. *Objects* in this methodology are comparable to *components* in hardware design.

Object–oriented methods typically achieve software engineering *goals* more consistently than function–oriented methods, because object–oriented *principles* and *processes* provide more capabilities. For example, the object–oriented *mechanism* to support *modularity*, *hiding*, and *localization* is based on the abstraction of objects, instead of functions; each object hides its internal structure (component objects), while displaying a set of operations. This object–based mechanism produces systems that are more *understandable* because they look like the physical universe, which we typically see as made of objects, not functions (operations on objects) or data (facts about objects). In object–oriented systems, an object can be reached only by its enclosing objects and by other objects that are components with it of the same enclosing objects, and then only through calling that object’s operations; *understandability* is thus further improved, since a reviewer is shown which objects have access to which other objects. Since an object can be independently verified and tested, and will work correctly when used to build a larger object because its integrity cannot be violated, object–oriented software is more *reliable*. The software is more *modifiable* because the boundaries of objects allow “the effects of change [to] be predicted with assurance”. It is more *portable* because the interaction with an environment–object is localized to a software–object. The class abstraction improves *re–usability* by providing a “blueprint” for building objects of that class, which can then be used in other systems.

Object–Oriented Methods

While object–oriented methodology typically achieves software engineering *goals* more consistently than function–oriented, particular object–oriented methods differ in their support for object–oriented *principles* and *processes* [1, 2, 8-10, 13]. As a result, within this methodology different methods do not achieve software engineering *goals* equally.

The most widely used object-oriented methods, Booch, Buhr, Coad, Colbert, Rumbaugh, Shlaer-Mellor, and Wirfs-Brock [3-7, 15-17, 19], evolved either from the information-processing domain or the real-time domain. Comparing them by their domain of origin reveals fundamental differences².

Object-oriented methods developed in the information-processing domain are convenient for relational database design. Their notation maps directly into database structures, having evolved from data modeling and the use of Entity-Relation Diagrams. During analysis they typically create an “object” model which describes the classes of objects in the system, the attributes of each class, and the relations between classes (essentially the ERD analysis); then describe the state behavior of each object of a class, and a functional decomposition of the work performed in each state using Data Flow Diagrams. The functions are then associated with classes, using method-specific heuristics. During design the object model is refined, processes are identified through event-thread analysis techniques, and a model for implementation in a programming language or database is developed.

These methods tend to fall short of consistent support for object-oriented processes and principles. For example, although the *principle of hiding*, and *mechanisms* to support it, are central to object-oriented methodology, in the early stages of software development these methods typically lack mechanisms to support hiding. They fail to provide that the attributes of an object can be changed only by calling operations of the object, or to prevent improper access to an object that is part of another object. Hiding can be added in the later stages of design, when mechanisms become available, but to do so is laborious because the “analysis” model from the early stages falls short of being *understandable*, as noted above for methods more completely function-oriented. Likewise it is difficult to verify if, when, and how an attribute (or object) is changed, harming *reliability*, *modifiability*, and *re-usability*. For the same reasons, they also do not scale up well for large projects, where hiding becomes increasingly necessary.

The ease of learning these methods that is frequently reported often amounts, on inspection, to little more than the comfort of recognizing them as like data modeling and Structured Analysis and Design, to which they are related conceptually. This comfortable feeling may seem attractive to the many engineers already trained in SA/SD, but it comes at the price of importing, with these derivative methods, many of the limitations that theoretically drove one to object-oriented methodology. For example, substantial confusion between “objects” and attributes, or even between “objects” and functions, is often found after training. Books on these methods often stop at translating a class into a database table, or into the class definition of an object-oriented programming language. They seldom adequately treat thread analysis, or the conversion to operating systems (or in the case of Ada, language) tasks, omitting examples or discussion of implementing the threads in operating systems processes (or in the case of Ada, tasks). They typically ignore such common real-time issues as scheduling and resource-sharing. Learners are left to work out implementation on their own.

² Comparing the methods individually, which is beyond the space limitations of this paper, will be taken up as time permits in the oral presentation.

Object–oriented methods developed in the real–time systems domain better address general system development issues. Their notations, modeling concurrent sequential processes and communications between them, map directly into object–oriented and object–based programming language and operating system constructs, which evolved from a similar conception [11, 12, 18]. During analysis or preliminary design (some of these methods do not treat analysis as a separate activity), they typically create an “object communication” model, which describes the high–level objects in the system and the communications between them, then describe the behavior of each object in terms of the operation requests it receives and issues. These communication and behavior models are then analyzed for “race” conditions, deadlocks, and whether schedules will be met. Classes of objects are abstracted from objects found to share attributes, operations, and behavior. During design the internal structure of each high–level object is defined: component objects are identified, their communications and behavior are established and analyzed, and classes abstracted. The design process is applied recursively until objects cannot be further decomposed. In methods that differentiate language–independent and language–specific notations, language–specific representation is created. Although real–time–system methods should in theory provide good support for database design, so far the leading methods fail to explore this implementation issue.

These methods more consistently support object–oriented processes and principles. To follow the same process and principle as before, they typically do provide mechanisms to support hiding, e.g. requiring that only by calling operations of an object can the components of that object be reached. Thus *understandability* is improved, as noted above in the general discussion of object–oriented methodology; the ease of isolating and verifying changes to objects supports *reliability*, *modifiability*, and *re–usability*; and expansion to large projects is more practical.

What Do You Need?

A method, or a group of methods, may have a characteristic strength or weakness that happens to be irrelevant to project requirements. For example, among methods deriving from each of the data–modeling and real–time–systems domains, some achieve the software engineering goal of reliability particularly well through careful attention to behavior analysis. If this is done early in the process, it can be used to validate that the developer and customer both understand what needs to be developed. Rigorous behavior analysis also detects errors early. However, the price is a higher front–end work load, since time and effort must be invested in detailed behavior descriptions (unless descriptions from previous projects can be re–used), and a delay in the production of code (even though code production will be more efficient when it occurs). What is a worthwhile price for a lower risk of errors? When there is a human–safety or other substantial reliability concern, a method will be wanted that emphasizes behavior analysis; otherwise, a less rigorous method may be adequate, or applying less rigor than a method supports.

Methods that evolved in the information modeling domain are generally most successful when applied to database design and the development of small non–real–time systems. They can also be applied to developing complete information management systems, with about the same

amount of success as achieved by the combination of SA/SD with data modeling methods. Methods that evolved in the real-time systems domain can be more effective for general system development, real-time systems, and large systems. The “real-time” methods are generally less effective for database design, because their notations do not map as directly into database structures, and they typically leave implementation for this purpose inadequately described. The “real-time” methods are generally as effective or more for the development of complete information management systems as the “information modeling” methods, because their support for the development of the total system outweighs their weakness in database design.

Conclusion

Object-oriented methods generally can produce better quality systems than function-oriented methods. However, object-oriented methods differ in their support for the object-oriented paradigm, application domains, and software engineering goals. Choosing a method means looking at the support it offers for the features that make object-oriented methodology valuable, weighed against characteristic strengths that may bear on the project at hand.

References

1. Arnold, P., et al., “An Evaluation of Five Object-Oriented Development Methods”, *Journal of Object-Oriented Programming* (“Focus on Analysis and Design”, 1991), pp. 107-121
2. Berard Software Engineering, *A Comparison of Object-Oriented Development Methods*, Berard Software Engineering, Inc. (1992)
3. Booch, G., *Object-Oriented Design with Applications* (1st ed. 1991)
4. Buhr, R., *Practical Visual Techniques in System Design, with Applications to Ada* (B.W. Kernighan series ed., 1990)
5. Coad, P. and E. Yourdon, *Object-Oriented Analysis* (2nd ed. 1991)
6. Coad, P. and E. Yourdon, *Object-Oriented Design*. (2nd ed. 1991)
7. Colbert, E., “Object-Oriented Software Development: A Practical Approach to Object-Oriented Development”, *TRI-Ada '89 Proceedings* (ACM SIGAda, 1989).
8. Cribbs, J., S. Moon, and C. Roe, *An Evaluation of Object-Oriented and Design Methodologies* (SIGS Books, 1992)
9. De Champeaux, D. and P. Faure, “A Comparative Study of Object-Oriented Analysis Methods”, *JOOP*, vol. 5, no. 1 (1992), pp. 21-32.
10. Fayed, M. and D. de Champeaux. “Object-Oriented Experiences”, *TRI-Ada '92 Proceedings* (ACM, 1992)
11. Goldberg, A. and D. Robson, *Smalltalk-80, The Language* (1989)

12. Ichbiah, J.D., et al., *Rationale for the Design of the Ada Programming Language* (Ada Joint Program Office, 1986)
13. Monarchi, D.E. and P.G. I., “A Research Typology for Object–Oriented Analysis and Design”. *Communications ACM*, vol. 35, no. 9, pp. 35–47 (1992)
14. Ross, D.T., J.B. Goodenough, and C.A. Irvine, “Software Engineering Process, Principles, and Goals”, *Computer* (1975), pp. 17-27
15. Rumbaugh, J., et al., *Object–Oriented Modeling and Design* (1991)
16. Shlaer, S. and S.J. Mellor, *Object Lifecycles, Modeling the World in States* (1992)
17. Shlaer, S. and S.J. Mellor, *Object–Oriented Systems Analysis, Modeling the World in Data* (1988)
18. Sommerville, I., *Software Engineering* (4th ed. 1992)
19. Wirfs–Brock, R., B. Wilkerson, and L. Wiener, *Designing Object–Oriented Software* (1990)