Object-Oriented Development

Linda M. Northrop

HISTORICAL PERSPECTIVE

The object-oriented model for software development has become exceedingly attractive as the best answer to the increasingly complex needs of the software development community. What was first viewed by many as a research curiosity and an impractical approach to industrial-strength software is now being enthusiastically embraced. Object-oriented versions of most languages have been or are being developed. Numerous object-oriented methodologies have been proposed. Conferences, seminars, and courses on object-oriented topics are extremely popular. New journals and countless special issues of both academic and professional journals have been devoted to the subject. Contracts for software development that specify object-oriented techniques and languages currently have a competitive edge. Object-oriented development is today what structured development was to the 1970s, and the object-oriented movement is still accelerating.

Concepts like "objects" and "attributes of objects" actually date back to the early 1950s when they appeared in early works in artificial intelligence (Berard, 1993). However, the real legacy of the object-oriented movement began in 1966 when Kristen Nygaard and Ole-Johan Dahl moved to higher levels of abstraction and introduced the language Simula. Simula provided encapsulation at a more abstract level than subprograms; data abstraction and classes were introduced in order to simulate a problem. At approximately the same time, Alan Kay was working at the University of Utah on a personal computer that he hoped would be able to support graphics and simulation. Due to both hardware and software limitations, Flex, Kay's computer venture, was unsuccessful. However, his ideas were not lost and surfaced again when he joined Xerox at the Palo Alto Research Center (PARC) in the early 1970s.

At PARC, he was a member of a project that espoused the belief that computer technologies are the key to improving communication channels between people and between people and machines. Based upon this conviction and influenced by the class concept in Simula, the turtle ideas LOGO provided in the Pen classes, the abstract data typing in CLU, and the incremental program execution of LISP, the group developed Smalltalk. In 1972, the first version of Smalltalk was released by PARC. About this time, the term "object-oriented" was coined. Some people credit this to Alan King, who is said to have used the term to characterize Smalltalk. Smalltalk is considered to be the first true object-oriented language (Goldberg & Robson, 1983), and today Smalltalk remains the quintessential object-oriented language. The goal of Smalltalk was to enable the design of software in units that are as autonomous as possible. Everything in the language is an object; that is, an instance of a class. Objects in this nascent Smalltalk world were associated with nouns. The Smalltalk effort supported a highly interactive development environment and prototyping. This original work was not publicized and was viewed with academic interest as being highly experimental.

Smalltalk-80 was the culmination of a number of versions of the PARC Smalltalk, and was released to the non-Xerox world in 1981. The August 1981 issue of *Byte* featured the Smalltalk efforts. On the cover of the issue was a picture of a hot air balloon leaving an isolated island that symbolized the launch of the PARC object-oriented ideas. It was time to start publicizing it to the software development community. The impact was gradual at first but increased to the current level of flurry about object-oriented techniques and products. The balloon was in fact launched and there was an effect. The early Smalltalk research in environments led to window, icon, mouse, and pull-down window environments. The Smalltalk language influenced the development in the early to mid 1980s of other object-oriented languages, most notably Objective-C (1986), C++ (1986), Self (1987), Eiffel (1987), and Flavors (1986). The application of object-orientation was broadened. Objects no longer were associated just with nouns, but also with events and processes. In 1980, Grady Booch pioneered the concept of object-oriented design (Booch, 1982). Since then, others have followed suit, and object-oriented analysis techniques have also begun to be publicized. In 1985, the first commercial object-oriented database system was introduced. The 1990s brought an ongoing investigation of object-oriented domain analysis, testing, metrics, and management. The current new frontiers in object technology are design patterns, distributed object systems, and Web-based object applications.

MOTIVATION

Why has the object-oriented movement gained such momentum? In reality, some of its popularity probably stems from the hope that it, like so many other earlier software development innovations, will address the crying need for greater productivi-

ty, reliability, maintainability, and manageability. However, aside from the hope that object orientation is, in fact, the "silver bullet," there are many other documented arguments to motivate its adoption.

Object-oriented development adds emphasis on direct mapping of concepts in the problem domain to software units and their interfaces. Furthermore, it is felt by some that, based upon recent studies in psychology, viewing the world as objects is more natural since it is closer to the way humans think. Objects are more stable than functions; what most often precipitates software change is change in required functionality, not change in the players or objects. In addition, object-oriented development supports and encourages the software engineering practices of information hiding, data abstraction, and encapsulation. In an object, revisions are localized. Object orientation results in software that is easily modified, extended, and maintained (Berard, 1993).

Object orientation extends across the life cycle in that a consistent object approach is used from analysis through coding. Moreover, this pervading object approach quite naturally spawns prototypes that support rapid application development. The use of object-oriented development encourages the reuse of not only software but also design and analysis models. Furthermore, object technology facilitates interoperability; that is, the degree to which an application running on one node of a network can make use of a resource at a different node of the network. Object-oriented development also supports the concurrency, hierarchy, and complexity present in many of today's software systems. It is currently necessary to build systems, not just black-box applications. These complex systems are often hierarchically composed of different kinds of subsystems. Object-oriented development supports open systems; there is much greater flexibility to integrate software across applications. Finally, use of the object-oriented approach tends to reduce the risk of developing complex systems, primarily because system integration is diffused throughout the life cycle.(Booch, 1994).

OBJECT-ORIENTED MODEL

The object-oriented model is more than a collection of new languages. It is a new way of thinking about what it means to compute and about how information can be structured. In the object-oriented model, systems are viewed as cooperating objects that encapsulate structure and behavior and belong to classes that are hierarchically constructed. All functionality is achieved by messages that are passed to and from objects. The object-oriented model can be viewed as a conceptual framework with the following elements: abstraction, encapsulation, modularity, hierarchy, typing, concurrence, persistence, reusability, and extensibility.

The emergence of the object-oriented model does not mark any sort of computing revolution. Instead, object orientation is the next step in a methodical evolution from both procedural approaches and strictly data-driven approaches. Object orientation is the integration of procedural and data-driven approaches. New approaches to software development have been precipitated by both programming language developments and increased sophistication and breadth in the problem domains for which software systems are being designed. Although, in practice, the analysis and design processes ideally precede implementation, it has been the language innovations that have necessitated new approaches to design and, later, analysis. Language evolution, in turn, has been a natural response to enhanced architecture capabilities and the ever increasingly sophisticated needs of programming systems. The impetus for object-oriented software development has followed this general trend. Figure 1 depicts the many contributing influences.

Perhaps the most significant factors are the advances in programming methodology. Over the last several decades, the support for abstraction in languages has progressed to higher levels. This abstraction progression has gone from address (machine languages), to name (assembly languages), to expression (first-generation languages, e.g., FORTRAN), to control (second-generation languages, e.g., COBOL), to procedure and function (second- and early third-generation languages, e.g., Pascal), to modules and data (late third-generation languages, e.g., Modula 2), and, finally, to objects (object-based and object-oriented languages). The development of Smalltalk and other object-oriented languages as discussed above necessitated the invention of new analysis and design techniques.

These new object-oriented techniques are really the culmination of the structured and database approaches. In the object-oriented approach, the smaller-scale concerns of dataflow orientation, like coupling and cohesion, are very relevant. Similarly, the behavior within objects will ultimately require a function-oriented design approach. The ideas of the entity relationship (ER) approach to data modeling from the database technology are also embodied in the object-oriented model.

Advances in computer architecture, leading to increased capability and decreased cost, and in the introduction of objects into hardware (capability systems and hardware support for operating systems concepts) have likewise affected the object-oriented movement. Object-oriented programming languages are frequently memory and MIPS intensive. They require and are now utilizing added hardware power. Philosophy and cognitive science have also influenced the advancement of the object-oriented model in their hierarchy and classification theories (Booch, 1991). And finally, the ever increasing scale, complexity, and diversity of computer systems have helped both propel and shape object technology.

Because there are many and varied influences on object-oriented development, and because this approach has not reached maturity, there is still some diversity in thinking and terminology. All object-oriented languages are not created equal nor do

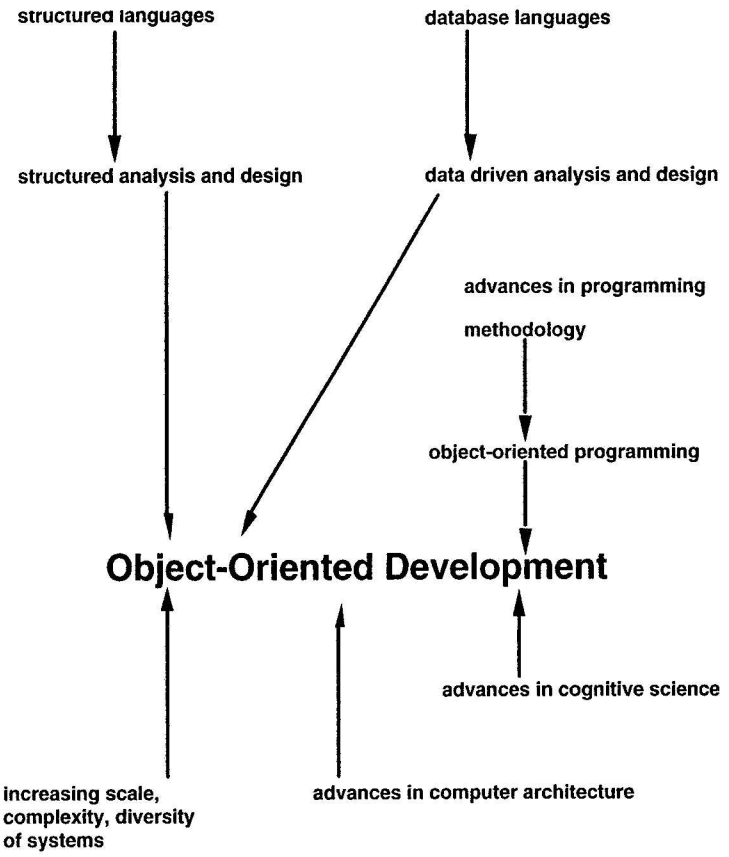

Figure 1. Influences on object-oriented development.

they refer to the same concepts with consistent verbiage across the board. And though there is a movement toward some unification, there is no complete consensus on how to do object-oriented analysis and object-oriented design nor on the symbology to use to depict these activities. Nevertheless, object-oriented development has proven successful in many application areas including air traffic control, animation, banking, business data processing, command and control systems, computer aided design (CAD), computer integrated manufacturing, databases, document preparation, expert systems, hypermedia, image recognition, mathematical analysis, music composition, operating systems, process control, robotics, space station software, telecommunications, telemetry systems, user interface design, and VLSI design. It is unquestionable that object-oriented technology has moved into the mainstream of industrial-strength software development.

OBJECT-ORIENTED PROGRAMMING

Concepts

Since the object-oriented programming efforts predate the other object-oriented development techniques, it is reasonable to focus first on object-oriented programming. In object-oriented programming, programs are organized as cooperating collections of objects, each of which is an instance of some class and whose classes are all members of a hierarchy of classes united via inheritance relations. Object-oriented languages are characterized by the following: object creation facility, message passing capability, class capability, and inheritance. Although these concepts can and have been used individually in other languages, they complement each other in a unique synergistic way in object-oriented languages.

Figure 2 illustrates the procedural programming model. To achieve desired functionality, arguments are passed to a procedure and results are passed back. Object-oriented languages involve a change of perspective. As depicted in Figure 3, functionality is achieved through communication with the interface of an object. An object can be defined as an entity that encapsulates state and behavior; that is, data structures (or attributes) and operations. The state is really the information needed to be stored in order to carry out the behavior. The interface, also called the protocol, of the object is the set of messages to which it will respond. Messaging is the way objects communicate and therefore the way that functionality is achieved. Objects respond to the receipt of messages by either performing an internal operation, also sometimes called a

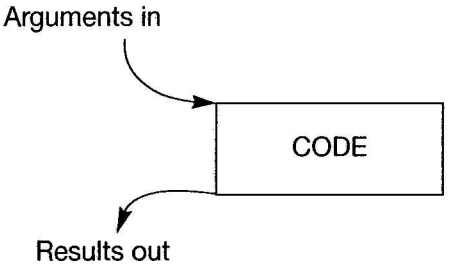

Figure 2. Procedural model.

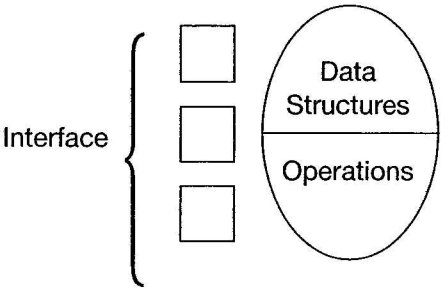

Figure 3. Object-oriented model.

method or routine, or by delegating the operation to be performed to another object. All objects are instances of classes, which are sets of objects with similar characteristics, or, from another viewpoint, a template from which new objects may be created. The method invoked by an object in response to a message is determined by the class of this receiver object. All objects of a given class use the same method in response to similar messages. Figure 4 shows a DOG class and objects instantiated from the dog class. All the DOG objects respond in the same way to the messages sit, bark, and roll. All DOG objects will also have the same state (data structures), though the values contained in what are typically called state variables can vary from DOG object to DOG object.

Classes can be arranged in a hierarchy. A subclass will inherit state and behavior from its superclass higher in the inheritance hierarchy structure. Inheritance can be defined as the transfer of a class' capabilities and characteristics to its subclasses. Figure 5 shows a subclass DOBERMAN of the original DOG class. An object of the DOBERMAN class will have the bark, sit, and roll behavior of the DOG class, but, in addition, it will have the the kill behavior particular to the DOBERMAN class. When a message is sent to an object, the search for the corresponding method begins in the class of the object and will progress up the superclass chain until such a method is found or until the chain has been exhausted (when an error would occur). In some languages, it is possible for a given class to inherit from more than one superclass. This capability is called multiple inheritance.

When dynamic binding is present, inheritance results in polymorphism. Polymorphism essentially describes the phenomenon in which a given message sent to an object will be interpreted differently at execution based upon subclass determination. Figure 6 illustrates a superclass UNMEMBER with its subclasses. If the message "speak" is sent to an object, at execution time it will be determined where the appropriate speak method will be found based upon the current subclass association of the object. Thus, the polymorphism means that the speak capability will vary and in fact will be determined at execution. It is possible for a method to not be actually defined in the superclass but still be included in the interface and, hence, be inherited by subclasses. One calls such a superclass an abstract class. Abstract classes do not have instances and are used only to create subclasses. For example, UNMEMBER would be an abstract class if the method for the message speak were not defined in UNMEMBER. Including speak in the interface of UNMEMBER, however, would dictate that speak would be a message common to all subclasses of UNMEMBER but the exact speak behavior would vary with each subclass. Abstract classes are used to capture commonality without determining idiosyncratic behavior.

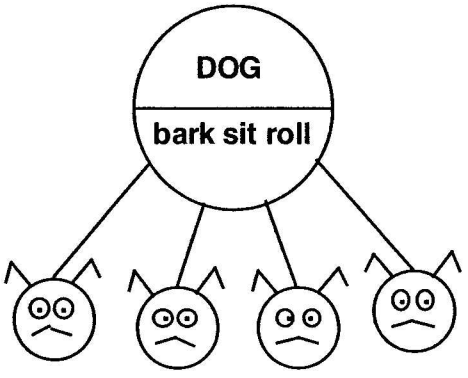

Figure 4. Instantiation of objects from a class.

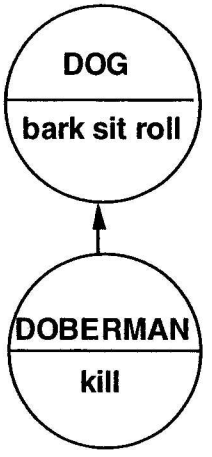

Figure 5. Inheritance.

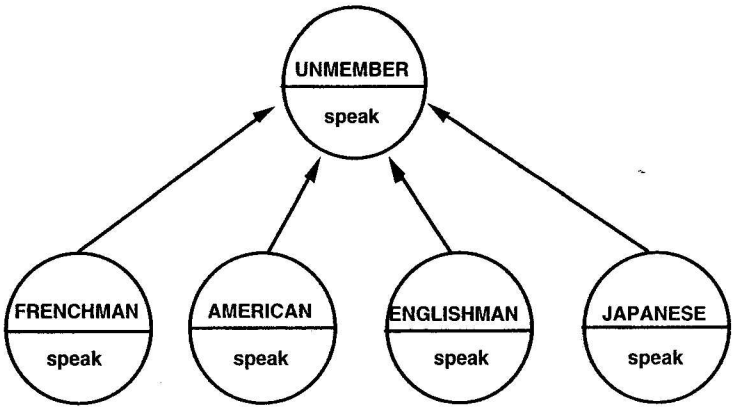

Figure 6. Polymorphism.

Languages

There are essentially four branches of object-oriented languages: Smalltalk-based, C-based, LISP-based, and Pascal-based. Simula is actually the common ancestor of all of these languages. The terminology and capability of the object-oriented languages varies considerably. A sampling of popular object-oriented languages in each branch is given in Table 1. The Smalltalk-based languages include the five versions, including Smalltalk-80, developed at PARC, as well as Digitalk Smalltalk and other such versions. Smalltalk-80 is considered the truest object-oriented language, although it and the others in this group do not have multiple inheritance capability.

In the C-based category are languages that are derived from C. Objective-C was developed by Brad Cox, has an extensive library, and has been used successfully to build large systems. C++ was written by Bjarne Stroustrup of AT&T Bell Labs. C's STRUCT concept is extended in C++ to provide class capability with data hiding. Polymorphism is implemented by virtual functions that deviate from the normal C typing, which is still resolved at compilation. C++ Version 2.0 includes multiple inheritance. C++ is a popular choice in many software areas, especially those in which UNIX is preferred. Similar to C and C++ but much simpler is Java, the latest object-oriented language, which hit the software development scene with great fanfare in 1995. Java, developed at Sun Microsystems, in addition to being object oriented has the capability to compile programs into binary format (applets) that can be executed on many platforms without compilation, providing embedded executable content for Web-based applications. Java is strongly typed and has multithreading and synchronization mechanisms like Ada, yet offers high performance and is portable, like C.

The many dialects, including LOOPS, Flavors, Common LOOPS, and New Flavors, in the LISP-based branch were precipitated by knowledge representation research. The Common LISP Object System (CLOS) was an effort to standardize object-oriented LISP. The Pascal-based languages include, among others, Object Pascal and Turbo Pascal, as well as Eiffel. Object Pascal was developed by Apple and Niklaus Wirth for the Macintosh. The class library for Object Pascal is MacApp. Turbo Pascal, developed by Borland, followed the Object Pascal lead. Eiffel was released by Bertrand Meyer of Interactive Software Engineering, Inc. in 1987. Eiffel is a full object-oriented language that has an Ada-like syntax and operates in a UNIX envi-

Table 1. Object-oriented languages

Smalltalk-80
Objective C
C++
Java
Flavors
XLISP
LOOPS
CLOS
Object Pascal
Turbo Pascal
Eiffel
Ada 95

Table 2. Object-based languages

Alphard
CLU
Euclid
Gypsy
Mesa
Modula
Ada

ronment. Ada, as it was originally conceived in 1983, was not object-oriented in that it did not support inheritance and polymorphism. In 1995, an object-oriented version of Ada was released. Though object-oriented, Ada 95 continues to differ from other object-oriented languages in its definition of a class in terms of types.

There are also languages that are referred to as object-based. A sample of object-based languages appears in Table 2. Object-based languages differ from object-oriented languages ostensibly in their lack of inheritance capability. It should be noted that although Ada 95 is object oriented, its predecessor, Ada, is object based.

OBJECT-ORIENTED SOFTWARE ENGINEERING

Life Cycle

Although the object-oriented languages are exciting developments, coding is not the primary source of problems in software development. Requirements and design problems are much more prevalent and much more costly to correct. The focus on object-oriented development techniques, therefore, should not be strictly on the programming aspects, but more appropriately on the other aspects of software engineering. The promise object-oriented methodologies hold for attacking complexity during analysis and design, and accomplishing analysis and design reuse is truly significant. If it is accepted that object-oriented development is more than object-oriented coding, then a whole new approach, including life cycle, must be adopted (Booch, 1994).

The most widely accepted life cycle to date is the waterfall/structured life cycle (Lorenz, 1993). The waterfall organization came into existence to stem the ad hoc approaches that had led to the software crisis first noted in the late 1960s. A version of the waterfall life cycle is pictured in Figure 7. As shown, the process is sequential; activities flow in primarily one direction. There is little provision for change and the assumption is that the system is quite clearly understood during the initial stages. Unfortunately, any software engineering effort will inherently involve a great deal of iteration, whether it is scheduled or not.

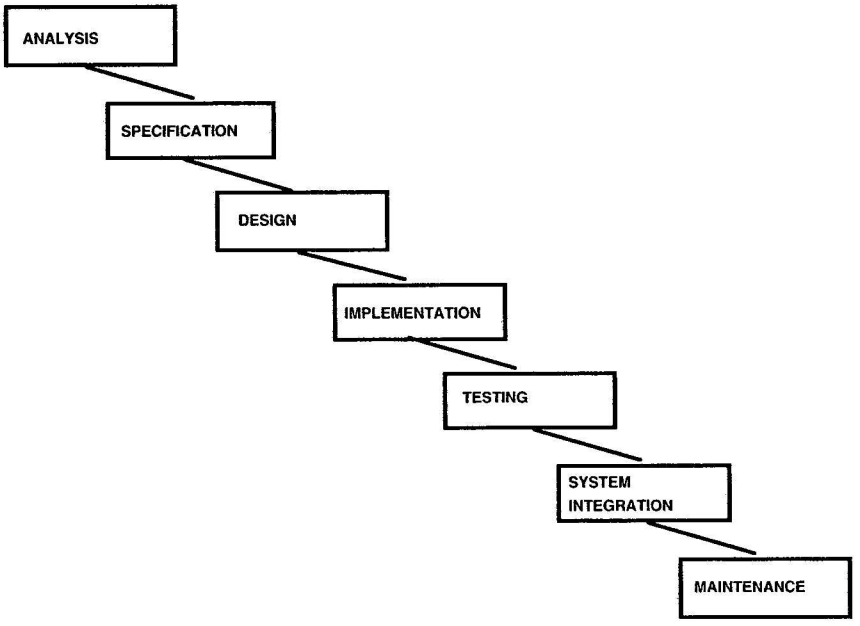

Figure 7. Waterfall life cycle.

Good designers have been described as practitioners who work at several levels of abstraction and detail simultaneously (Curtis, 1989). The waterfall life cycle simply does not accommodate real iteration. Likewise, prototyping, incremental builds, and program families are misfits. The waterfall/structured life cycle is also criticized for placing no emphasis on reuse and having no unifying model to integrate the phases (Korson & McGregor, 1990).

The object-oriented approach begins with a model of the problem and proceeds with continuous object identification and elaboration. It is inherently iterative and inherently incremental. Figure 8 illustrates a version of the water fountain life cycle that has been used to describe the object-oriented development process (Henderson-Sellers & Edwards, 1990). The fountain idea shows that the development is inherently iterative and seamless. The same portion of the system is usually worked on a number of times, with functionality being added to the evolving system with each iteration. Prototyping and feedback loops are standard. The seamlessness is accounted for in the lack of distinct boundaries during the traditional activities of analysis, design, and coding. The reason for removing the boundaries is that the concept of object permeates; objects and their relationships are the medium of expression for analysis, design, and implementation. There is also a switch of effort from coding to analysis and an emphasis on data structure before function. Furthermore, the iterative and seamless nature of object-oriented development makes the inclusion of reuse activities natural.

More recently, a life cycle that has both a macro and a micro view has been proposed to increase the manageability of object-oriented development (Booch, 1994). The macro phases in Figure 9 are analysis, to discover and identify the objects; design, to invent and design objects; and implementation, to create objects. Built into each macro phase is a micro phase depicting the iteration. This life cycle suggests Boehm's Spiral Model (Boehm, 1988).

Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD)

Since object-oriented technology is still relatively new, there are, as noted above, a number of approaches to object-oriented analysis and design. Most of them use graphical representations, an idea that was likely inherited from structured methodologies. Object-oriented analysis builds on previous information modeling techniques, and can be defined as a method of analy-

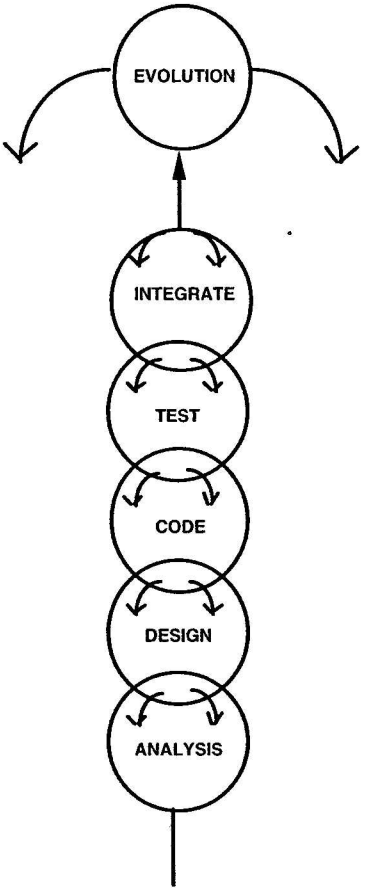

Figure 8. Water fountain life cycle for object-oriented software development.

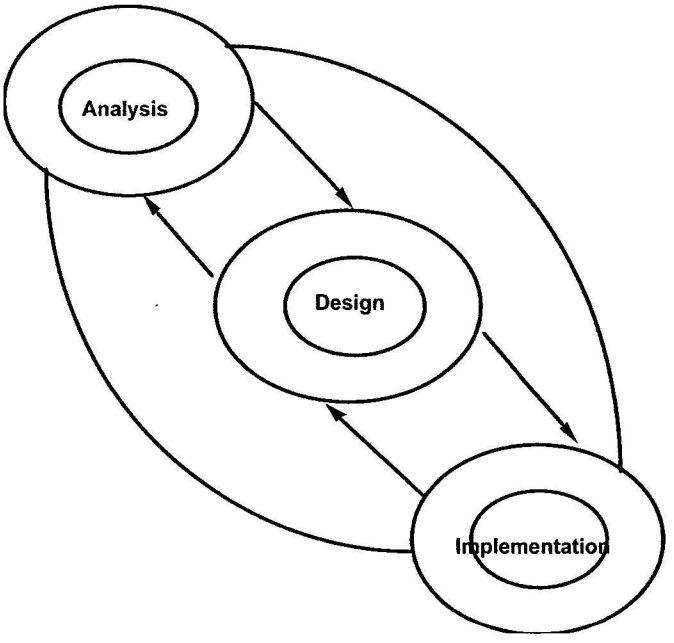

Figure 9. Iterative/incremental life cyle.

sis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain. Analysis activities yield black-box objects that are derived from the problem domain. Scenarios are often used in object-oriented approaches to help determine necessary object behavior. A scenario is a sequence of actions that takes place in the problem domain. Frameworks have become very useful in capturing an object-oriented analysis for a given problem domain and making it reusable for related applications. Basically, a framework is a skeleton of an application or application subsystem implemented by concrete and abstract classes. In other words, a framework is a specialization hierarchy with abstract superclasses that depicts a given problem domain. One of the drawbacks of all current object-oriented analysis techniques is their universal lack of formality.

During object-oriented design, the object focus shifts to the solution domain. Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design (Booch, 1994).

In both analysis and design, there is a strong undercurrent of reuse. Researchers in object technology are now attempting to codify design patterns, which are a kind of reusable asset that can be applied to different domains. Basically, a design pattern is a recurring design structure or solution that when cataloged in a systematic way can be reused and can form the basis of design communication (Gamma, et al., 1995).

OOD techniques were actually defined before OOA techniques were conceived. There is difficulty in identifying and characterizing current OOA and OOD techniques because, as described above, the boundaries between analysis and design activities in the object-oriented model are fuzzy. Given that problem, the following descriptions provide an overview to some of the OOA and OOD techniques being used.

Meyer uses language as a vehicle for expressing design. His approach is really not classifiable as an OOD technique (Meyer, 1988). Booch's OOD techniques extend his previous Ada work. He advocates a "round trip gestalt" process during which objects are identified, semantics of the objects are identified, relationships are identified, implementation is accomplished, and iteration occurs. Originally, he used class diagrams, class category diagrams, class templates, and object diagrams to record design (Booch, 1991). More recently, he has taken ideas from other methods and woven them into his work. Behavior is described with Harel state charts in conjunction with interaction or annotated object diagrams (Booch, 1994).

Wirfs-Brock's OOD technique is driven by delegation of responsibilities. Class responsibility cards (CRCs) are used to record classes responsible for specific functionality and collaborators with the responsible classes. The initial exploration of classes and responsibilities is followed by detailed relationship analysis and implementation of subsystems (Wirfs-Brock, 1990).

Rumbaugh et al. use three kinds of models to describe a system: the object model, which is a static structure of the objects in a system; the dynamic model, which describes the aspects of a system that change over time; and the functional model, which describes the data value transformations within a system. Object diagrams, state diagrams, and data-flow diagrams are used to represent the three models, respectively (Rumbaugh et al., 1991).

In their OOA technique, Coad and Yourdon (1991) advocate the following steps: find classes and objects, identify structures and relationships, determine subjects, define attributes, and define services, to determine a multilayer object-oriented model. The layers correspond to the steps, namely, class and object layer, subject layer, structure layer, attribute layer, and service layer, respectively. Their OOD technique is both multilayer and multicomponent. The layers are the same as those in analysis. The components include problem domain, human interaction, task management, and data management.

Ivar Jacobson offers Objectory, an object-oriented software engineering method developed by Objective Systems in Sweden. Jacobson's method has a strong focus on a particular kind of scenario referred to as a "use case." The use cases become the basis for the analysis model, which gives way to the design model when the use cases are formalized by interaction diagrams. The use cases also drive the testing in a testing phase, which Objectory makes explicit. Objectory is the most complete industrial method to date (Jacobson, 1992).

There are also other published OOA and OOD techniques as well as variations of the above that are not listed here. In recent years as the methods have been evolving, there has been considerable convergence. In late 1995, Booch, Rumbaugh, and Jacobson joined forces and proposed the first draft of a Unified Method, which promises to add some welcome consensus and stability (Booch, 1995).

Management Issues

As organizations begin to shift to object-oriented development techniques, the management activities that support software development also necessarily have to change. A commitment to objects requires a commitment to change processes, resources, and organizational structure (Goldberg & Rubin, 1995). The seamless, iterative, prototyping nature of object-oriented development eliminates traditional milestones. New milestones have to be established. Also, some of the ways in which measurements were made are less appropriate in an object-oriented context. LOC (lines of code) is definitely not helpful. Number of classes reused, inheritance depth, number of class-to-class relationships, coupling between objects, number of classes, and

class size are more meaningful measurements. Most work in object-oriented metrics is relatively new, but references are beginning to surface (Lorenz, 1993).

Resource allocation needs to be reconsidered as does team organization. Smaller development teams are suggested (Booch, 1994), as is cultivation of reuse experts. Incentives should be based on reuse, not LOC. An entirely new mind-set is required if reuse is to really be operative. Libraries and application frameworks have to be supported and built, along with contracted application software. Long-term investment strategies are imperative as well as the processes and commitment to evolve and maintain these reuse assets.

Regarding quality assurance, typical review and testing activities are still essential, but their timing and definition must be changed. For example, a walk-through could involve enacting a scenario of interacting objects proposed to effect some specific functionality. Testing of object-oriented systems is another area that needs to be more completely addressed. Release in terms of a steady stream of prototypes requires a flavor of configuration management that differs from that which is being used to control products generated using structured techniques.

Another management concern ought to be appropriate tool support. An object-oriented development environment is essential. Also needed are a browser for class library, an incremental compiler, debuggers that know about class and object semantics, graphics support for design and analysis notation and reference checking, configuration management and version control tools, and a database application that functions as a class librarian. Tools are now available but need to be evaluated based upon the purpose, organization, and method chosen.

Estimates can also be problematic until there is object-oriented development history to substantiate proposed development estimates of resources and cost. Cost of current and future reuse must be factored into the equation. Finally, management must be aware of the risks involved in moving to an object-oriented approach. There are potential performance risks such as cost of message passing, explosion of message passing, class encumbrance, paging behavior, dynamic allocation, and destruction overhead. There are also start-up risks including acquisition of appropriate tools, strategic and appropriate training, and development of class libraries.

OBJECT-ORIENTED TRANSITION

There are documented success stories, but there are also implicit recommendations. The transition needs to progress through levels of absorption before assimilation into a software development organization actually occurs. This transition period can take considerable time. Training is essential. Pilot projects are recommended. Combination of structured and object-oriented approaches are not recommended. There is growing evidence that success requires a total object-oriented approach for at least the following reasons: traceability improvement, reduction in significant integration problems, improvement in conceptual integrity of process and product, minimization of need for objectification and deobjectification, and maximization of the benefits of object orientation (Berard, 1993).

THE FUTURE

In summary, object-oriented development is a natural outgrowth of previous approaches and has great promise for software development in many application domains. Paraphrasing Maurice Wilkes in his landmark 29 year reprise of his 1967 ACM Turing Lecture, "Objects are the most exciting innovation in software since the 70s" (Wilkes, 1996). Object-oriented development is not, however, a panacea and has not yet reached maturity. The full potential of objects has not been realized. Yet, whereas the future of object-oriented development cannot be defined, the predictions of the early 1990s (Winblad et al., 1990) are already materializing. Class libraries and application frameworks are becoming readily available in the marketplace. Transparent information access across applications and environments is conceivable. Environments in which users can communicate among applications and integrated object-oriented multimedia tool kits are emerging. It is likely that the movement will continue to gain in popularity and techniques will mature significantly as experience increases. It is also likely that object orientation will eventually be replaced or absorbed into an approach that works at an even higher level of abstraction. Of course, these are just predictions. In the not too distant future, talk about objects will no doubt be passé, but for now there is much to generate genuine enthusiasm.

BIBLIOGRAPHY

E. V. Berard, *Essays on Object-Oriented Software Engineering,* Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

B. Boehm, "A Spiral Model of Software Development and Enhancement," in Thayer, R., ed., *Software Engineering Project Management,* IEEE Computer Society Tutorial, Catalog Number EH0263-4, 1988.

G. Booch, "Object-Oriented Design," *Ada Letters, 1,* 3, 64–76 (March–April, 1982).

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

G. Booch, *Object-Oriented Analysis and Design with Applications,* Addison-Wesley, Reading, Massachusetts, 1994.

G. Booch and J. Rumbaugh, *Introduction to the United Method,* OOPSLA '95 Tutorial Notes, Austin, Texas, 1995.

T. Budd, *An Introduction to Object-Oriented Programming,* Addison-Wesley, Reading, Massachusetts, 1991.

P. Coad and J. Nicola, *Object-Oriented Programming,* Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

P. Coad and E. Yourdon, *Object-Oriented Analysis,* 2nd ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

P. Coad and E. Yourdon, *Object-Oriented Design,* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

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

B. Curtis, ". . . But You Have to Understand. This Isn't the Way We Develop Software at Our Company," MCC Technical Report No. STP-203-89, Microelectronics and Computer Technology Corporation, Austin, Texas, 1989.

M. Fowler, *A Comparison of Object-Oriented Analysis and Design Methods,* OOPSLA '95 Tutorial Notes, Austin, Texas, 1995.

I. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns,* Addison-Wesley, Reading, Massachusetts, 1995.

A. Goldberg and P. Robson, *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, Reading, Massachusetts, 1983.

A. Goldberg and K. Rubin, *Succeeding with Objects,* Addison-Wesley, Reading, Massachusetts, 1995.

B. Henderson-Sellers and J. M. Edwards, "The Object-Oriented Systems Life Cycle," *CACM,* 143–159 (Sept. 1990).

I. Jacobson, M. Christerson, P. Jonsson, and G. G. Overgaard, *Object Oriented Software Engineering,* Addison-Wesley, Reading, Massachusetts, 1992.

T. Korson and J. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *CACM,* 41–60 (Sept. 1990).

M. Lorenz, *Object-Oriented Software Development,* Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

B. Meyer, *Object-Oriented Software Construction,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

D. Monarchi and G. Puhr, "A Research Typology for Object-Oriented Analysis and Design," *CACM,* 35–47 (Sept. 1992).

R. Pressman, *Software Engineering: A Practitioner's Approach,* 3rd eEd., McGraw-Hill, New York, 1992.

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

S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data,* Yourdon Press–Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

M. Wilkes, "Computers Then and Now—Part 2," Invited Talk, ACM Computer Science Conference, Philadelphia, Pennsylvania, 1996.

A. L. Winblad, S. D. Edwards, and D. R. King, *Object-Oriented Software,* Addison-Wesley, Reading, Massachusetts, 1990.

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