Object-Oriented Methods

Early object-oriented methods with class models and behavioral models include OMT (Rumbaugh *et al.*, 1991), Booch (1994), Fusion (Coleman *et al.*, 1994), Coad–Yourdon, Shlaer–Mellor, and Martin–Odell. These models are similar, although there are slight differences in semantics and major differences in notation. Other popular methods with more behavioral focus include Objectory (Jacobson, 1992), ROOM (Selic *et al.*, 1994), and RDD (Wirfs-Brock *et al.*, 1990). More recent work has focused particularly on the development process.

Unified Modeling Language

The Unified Modeling Language (UML) began in 1994 as an attempt to unify the Booch and OMT models but quickly developed into a broadly based effort to standardize object-oriented modeling concepts, terminology, and notation. Ultimately researchers from 17 companies submitted a proposal to the Object Management Group (OMG) that was unanimously adopted as a standard in 1997 after extensive feedback from the general public. The UML specification contains a metamodel (a model of legal models) of modeling constructs, constraints on well-formed models, definitions of semantics of the constructs, and notation for expressing models visually. UML does not standardize the development process; it is intended to support many current and future processes. UML has been widely accepted by the object-oriented community to replace the plethora of earlier notations. The Object Management Group has the responsibility for future evolution of the UML.

Support Tools

Large system development requires the assistance of automated editing tools to construct, verify, and maintain large models. Such tools are available from a number of vendors. A typical full-featured model editing tool permits the interactive drawing of models, the ability to browse the model both graphically and textually, code generation and reverse engineering of existing code, and facilities for organizing models into modules and for coordinating the work of multideveloper teams. Tools perform bookkeeping that would otherwise be tedious and error-prone; they also edit and display model diagrams interactively and present different views of a model on user request. Different tools support a variety of target implementation media, including various programming languages, object-oriented and relational databases (*q.v.*), fourth generation language (4GL) systems, and certain specialized environments. Models can also be constructed without producing any target application, for example, to understand the structure of a business organization.

Bibliography

1990. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. *Designing Object-Oriented Software.* Upper Saddle River, NJ: Prentice Hall.

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

1992. Jacobson, I. *Object-Oriented Software Engineering.* Reading, MA: Addison-Wesley.

1994. Booch, G. *Object-Oriented Analysis and Design with Applications,* 2nd Ed. Redwood City, CA: Benjamin/Cummings.

1994. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. *Object-Oriented Development: The Fusion Method.* Upper Saddle River, NJ: Prentice Hall.

1994. Selic, B., Gullekson, G., and Ward, P. T. *Real-Time Object-Oriented Modeling.* New York: John Wiley.

1995. Firesmith, D., and Eykholt, E. *Dictionary of Object Technology.* New York: SIGS Books.

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

1999. Rumbaugh, J., Booch, G., and Jacobson, I. *The Unified Modeling Language Reference Manual.* Reading, MA: Addison-Wesley.

James Rumbaugh

OBJECT-ORIENTED PROGRAMMING (OOP)

For articles on related subjects *see* ABSTRACT DATA TYPE; C++; CLASS; ENCAPSULATION; INFORMATION HIDING; JAVA; OBJECT-ORIENTED ANALYSIS AND DESIGN; PACKAGE; SIMULA; SOFTWARE ENGINEERING; SOFTWARE DESIGN PATTERNS; and STRUCTURED PROGRAMMING.

Introduction

The essence of object-oriented programming is to model systems of real entities with the goal of separating their internal structure from their external, visible interactions. It emphasizes the hiding or *encapsulation* of the "inner" state of entities and the specification of interactive properties of entities by an interface of operations (the events in which they may participate). This separates the inner functioning of entities like banks, aircraft, or people from their external behavior in interacting with other entities. The separation is realized by partitioning the state of a system of entities into chunks associated with objects so that each chunk is responsible for its own protection against access by unauthorized operations. In a concurrent environment, objects protect themselves against asynchronous access, removing the synchronization burden from processes that access the object's data.

A programming language is said to be *object-based* if it supports objects as a language feature, and is said to be *object-oriented* if, additionally, objects are required to belong to *classes* that can be incrementally modified through *inheritance* (whereby a class may *inherit* the *capabilities* of a base class and also extend or modify these capabilities). Among the object-oriented languages are Simula, Smalltalk, C++, Eiffel, Ada (*q.v.*), and Java, but not Fortran, C, or Pascal. Originally Ada was object-based, supporting the functionality of objects, but was not object-oriented, since it did not support the management of objects through classes and inheritance. The current Ada 95 standard, however, provides a type-extension mechanism to implement inheritance, and may thus be considered object-oriented.

Early programmers thought of programs as instruction sequences. Procedure-oriented languages (*q.v.*) introduced procedural abstractions that encapsulate sequences of actions into procedures. Object-oriented languages encapsulate data as well as sequences of actions, providing a stronger encapsulation mechanism than procedures and, consequently, a more powerful modeling tool. Both procedures and objects are server modules that may be called by clients to determine a stimulus/response behavior in interacting with their environment (*see* CLIENT–SERVER COMPUTING. The role of procedures is to transform input data specified by parameters into values, while the role of objects is to serve as a repository of data (the current system state) and to respond in a manner determined by the current system state. For example, the response of a bank to a withdrawal request depends on the value of the current balance. Object-oriented programming is a modeling paradigm that models objects of the real world by collections of interacting objects of a programming system.

The procedure-oriented paradigm has strong organizing principles for managing actions and algorithms, but has weak organizing principles for managing shared data, while object-oriented systems organize data by restricting applicable operations to those associated with a specific object or class. Inheritance provides a second layer of structure by structuring classes into hierarchies. We can think of classes as a mechanism for classifying objects into categories with similar interface behavior, and inheritance as a mechanism for classifying classes by factoring out properties common to several subclasses into a superclass.

Functional, logic-based, and procedure-oriented paradigms execute algorithms whose semantics are described by computable functions, while objects provide persistent services to clients over time that cannot be entirely described by computable functions. Objects determine interactive "marriage contracts" with clients over the lifetime of the object that cannot be entirely described by algorithmic "sales contracts" (Wegner, 1997). Objects can better model embedded systems, graphical user interfaces, and distributed systems than procedures and algorithms, supporting more powerful forms of problem solving, like distributed ATM banking systems and airline reservation systems.

Objects

Objects in programming languages are collections of operations that share a state. The operations determine the messages (calls) to which the object can respond, while the shared state is hidden from the outside world and is accessible only to the object's operations (*see* Fig. 1). Variables representing the internal state of an object are called *instance variables* and its operations are called *methods*. The collection of methods of an object determines its *interface* and its behavior:

```
name:object
  local instance variables (shared state)
  operations or methods (interface of
      message patterns to which the
      object may respond)
```

An object named `point` with instance variables `x`, `y` and methods for reading and changing them may be defined as follows:

```
point:object
    x:=0; y:=0;
    read-x: ↑x; -return value of x
    read-y: ↑y; -return value of y
    change-x(dx):x:=x + dx;
    change-y(dy):y:=y + dy;
```

The object `point` protects its instance variables `x,y` against arbitrary access, allowing access only through messages to `read` and `change` operations. The object's behavior is entirely determined by its responses to acceptable messages and is independent of the data representation of its instance variables. Moreover, the object's knowledge of its callers is entirely determined by its messages. Object-oriented message-passing facilitates two-way abstraction: senders have an abstract view of receivers and receivers have an abstract view of senders.

An object's interface of operations (methods) can be represented by a record:

```
untyped object interface: (op1,op2,..,opN)
```

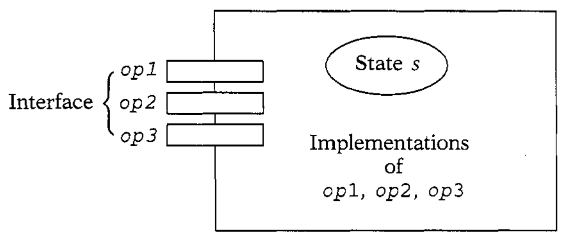

Figure 1. Object modules.

Objects whose operations `opi` have type `Ti` have an interface that is like a typed record, but differs from records in that fields of object records may be interdependent because of the shared state. Typed record interfaces are called *signatures.*

```
Typed Object Interface (Signature):
                  (op1:T1,op2:T2,..,opN:TN)
```

The `point` object has the following signature:

```
point-interface =
     (read-x: Real,
     read-y: Real,
     change-x: Real → Real,
     change-y: Real → Real)
```

The parameterless operations `read-x` and `read-y` both return a **`Real`** number as their value, while `change-x` and `change-y` expect a **`Real`** number as their argument and return a **`Real`** result.

The operations of an object share its state so that state changes by one operation may be seen by subsequently executed operations. Operations access the state by references to the object's instance variables. For example, `read-x` and `change-x` share the instance variable `x`, which is nonlocal to these operations, although local to the object.

Nonlocal references in functions and procedures are generally considered harmful, but they are essential for operations within objects, since they are the only mechanism by which an object's operations can access its internal state. Sharing unprotected data within an object is combined with strong protection (encapsulation) against external access. The strong encapsulation at the object interface is realized at the expense of modularity (and reusability) of component operations. This captures the distinction within any organization or organism between closely integrated internal subsystems and contractually specified interfaces to the outside world.

Classes

We distinguish between object-based languages that support objects as a language primitive and object-oriented languages that additionally support the management of objects through classes (*q.v*) and inheritance. In object-oriented languages, the behavior of objects is specified by classes, which are like the types of traditional languages, but serve additionally to classify objects into hierarchies through the inheritance mechanism.

Classes serve as templates from which objects can be created. They are record-structured templates whose instantiation creates objects whose interfaces are record-structured. The class `point` has precisely the same instance variables and operations as the object `point`, but their interpretation is different. Whereas the instance variables of a `point` object represent `actual` variables, class instance variables are *potential,* being instantiated only when an *object* is created:

```
point:class
 local instance variables
   (private copy for each object of the class)
 operations or methods (shared by all
   objects of the class)
```

Instances of a class can be created by a `make-instance` operation, which creates a copy of the class instance variables that may be acted on by the class operations:

```
p:= make-instance point; -- create a new
     instance of the class point, call it p
```

Instance variables in class definitions may be initialized as part of object creation:

```
p1:= make-instance point (0,0); -- create
     point initialized to (0,0), call it p1
p2:= make-instance point (1,1); -- create
     point initialized to (1,1), call it p2
```

The two points `p1`, `p2` each have private copies of the class instance variables and share the operations specified in the class definition. When an object receives a message to execute a method, it looks for the method in its class definition. We may think of a class as specifying a behavior common to all objects of the class. The instance variables specify a data structure (*q.v.*) for realizing the behavior. The public operations of a class determine its behavior, while the private instance variables determine its structure.

Inheritance

Inheritance is a mechanism for sharing code and behavior. It allows reuse of the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new instance variables.

Fig. 2 describes mammals by an inheritance hierarchy of classes (representing behaviors). The class of mammals has persons and elephants as its subclasses. The class of persons has mammals as its superclass and students and females as its subclasses. The instances John, Joan, Bill, Mary, and Dumbo each have a unique base class. In *single inheritance,* illustrated here, membership of an instance in more than one base class, such as Joan being both a student and a female, cannot be expressed. We discuss *multiple inheritance* below.

Why does inheritance play such an important role in object-oriented programming? Inheritance can express relations among behaviors such as classification,

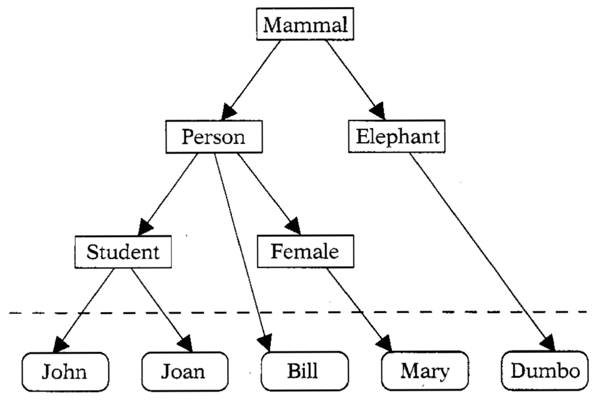

Figure 2. Example of an inheritance hierarchy.

specialization, generalization, approximation, and evolution. Thus, in Fig. 2 we classify mammals into persons and elephants. Elephants specialize the properties of mammals, and conversely mammals generalize the properties of elephants. The properties of mammals approximate those of elephants. Moreover, elephants evolved from early species of mammals.

Inheritance classifies classes in much the same way that classes classify values. The ability to classify classes provides greater classification power and conceptual modeling power. Classification of classes may be referred to as second-order classification. Inheritance provides second-order sharing, management, and manipulation of behavior that complements first-order management of objects by classes.

Inheritance can be expressed by record extension. If a class is specified by a record ⟨`op1:T1, op2:T2`⟩ then inheritance can extend this record by horizontal extension that adds one or more new operations and by vertical extension that modifies existing operations. The Oberon language, a successor to Pascal (*q.v.*) and Modula-2, uses record extension to implement inheritance, as does Modula-3. Languages like Self abandon inheritance at the class level in favor of *delegation*, which is a form of inheritance at the object level that is more flexible than inheritance in handling objects that may change their class, like students who become professors, or lawyers who become judges. In delegation, if an object of a subclass is called upon to execute an operation defined in a parent class, it delegates that operation to the parent class, e.g. by sending it a message. If an object changes its class, it can then delegate such an operation to the new parent.

Virtual classes are incomplete behavior specifications that require subclasses to complete their behavior specification before they can be instantiated. The class of mammals in Fig. 2 is a virtual class. It specifies behavioral attributes common to all mammals and must be supplemented by behavioral attributes of specific mammals (persons or elephants) before instances like Joan and Dumbo can be created. Summarizing:

- *virtual class*: incomplete behavior specification, cannot be directly instantiated (mammals)
- *subclass*: completes virtual behavior specification (persons or elephants)

Incomplete behaviors are natural building blocks in constructing composite behavior specifications. Composition of incomplete behaviors during program development may be contrasted with modification of already complete behaviors during maintenance and enhancement.

Tree (*q.v.*) structure is a general mechanism for sharing of the properties of ancestors by descendants. Just as block structure facilitates the sharing of data declared in ancestor blocks by descendant blocks, inheritance hierarchies facilitate the sharing of code and behavior of superclasses by subclasses.

Multiple inheritance, which supports subclasses that share the behavior of several superclasses, gives rise to more complex structures, such as directed graphs (*see* GRAPH THEORY), since a subclass can inherit from several parents. It is natural in some contexts, but is conceptually complex in part because classes mix specification and implication in a single language construct. C++ and Eiffel have multiple inheritance; Simula and Smalltalk have single inheritance. Java separates the notion of interfaces as behavior specifications from the notion of classes as implementations, and permits multiple inheritance of interfaces to be specified in a cleaner way than is possible for multiple inheritance of classes.

Implementation of Class Inheritance

The following implementation of inheritance is simple, though not the most efficient. Consider a class A with instance a and a subclass B with instance b, as in Fig. 3. Both A and B define behavior by operations shared by their instances, and have instance variables that cause a private copy to be created for each instance of the class or subclass. The instance a of A has a copy of A's instance variables and a pointer to its base class. The instance b of B has a copy of the instance variables of both B and its superclass A and a pointer to the base class of B. The class representation of B has a pointer to its superclass A, while A has no superclass pointer, since it is assumed to have no superclass.

When b receives a message to execute a method, it looks first in the methods of B. If found, the method is executed using the instance variables of b as data. Otherwise, the pointer to its superclass is followed. If

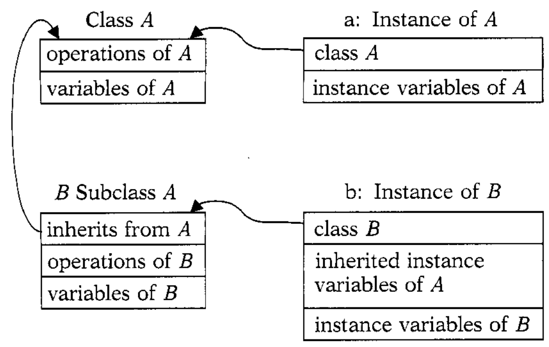

Figure 3. Implementation of inheritance.

it finds the method in *A*, it executes it on the data of *b*. Otherwise, it searches *A*'s superclass if there is one. If *A* has no superclass and the method has not been found, it reports failure. This search algorithm may be defined by the following procedure:

```
procedure search (name, class)
    if (name = localname) then
        do localaction
    else if (inherited-module = nil)
        then undefinedname
    else
        search (name, inherited-module)
```

Actual implementations avoid the run-time search for the appropriate method. When a program is compiled, the compiler can determine in what class or superclass any method is declared, and can generate the code needed to invoke it at run time. There are two possible complications. If the language provides multiple inheritance, then any call of a multiply-inherited method must specify which version is meant, which may be done as in C++ by attaching the parent class name to the method name: `parent_class::method_name`.

The other complication arises because it is possible for a method binding to change dynamically, as when a function in a virtual class is bound to a specific operation. A common technique is to maintain a *virtual methods table* (VMT) for each class. An object has a pointer to its VMT, and if a run-time assignment changes the binding (e.g. if an elephant becomes a circus-elephant), only the VMT pointer must change. The VMT requires an extra step to look up each function, but no run-time searching.

Evolution of Object-Oriented Programming

Simula 67 (Dahl *et al.*, 1968) was the first object-oriented language. Its language primitives included objects, classes, and inheritance, and it was used extensively for simulation and other applications, primarily in Europe. Smalltalk (Goldberg and Robson, 1983), developed by the software concepts group at Xerox PARC in the 1970s and embodied in a stable implementation in Smalltalk 80, caught the public imagination because of its implementation on personal computers and its interactive graphical interface that permitted browsing and the display of objects in multiple windows. Smalltalk implementations were initially too slow to be commercially viable, but in the 1990s became increasingly competitive with traditional languages.

The US Department of Defense language Ada (*q.v.*) included the notion of packages, which are like objects, but did not have a notion of classes or inheritance in its 1983 version. Starting in the mid-1980s, object-oriented programming became a popular term, and object-oriented dialects of existing programming languages began to appear, like Object-Pascal, Objective C, and C++. C++ has proven to be a popular object-oriented language because it is upward compatible with C.

The preeminence of C++ as the dominant object-oriented programming language is being challenged by Java, which has a cleaner design, built-in threads that support concurrency (*see* CONCURRENT PROGRAMMING), an impressive collection of class libraries for graphical user interfaces and system software, and excellent documentation through textbooks written by the language designers. Java is increasingly being adopted as a language for first courses in computing and has a chance of displacing C++ as the dominant object-oriented language.

As object-oriented technology is being adopted by the software engineering community, attention is shifting from object-oriented languages to object-oriented design, epitomized by design methods like OMT, UML and OOAD (*see* OBJECT-ORIENTED ANALYSIS AND DESIGN), and component software (*q.v.*) systems like CORBA/OpenDoc, COM/OLE/ActiveX, and Java/JavaBeans. There is also much work on computer-aided software engineering (CASE—*q.v.*) tools for object-oriented programming. There is no doubt that object-oriented technology is a permanent part of the computer landscape. Object-oriented systems are not merely a more scalable technology for design; they are actually more expressive than procedure-oriented technology and allow systems that provide services over time to be designed in an integrated fashion that could not be designed in the procedure-oriented paradigm without introducing *ad hoc* shared data structures.

Is Object-Oriented Programming Fundamental?

The popularity of object-oriented programming in the late 1980s and 1990s rivals the fashionability of structured programming (*q.v.*) in the 1970s. It provides high-level structure in objects, classes, and class hierarchies, complementing structured programming techniques for microstructure at the level of statements and expressions. Object-oriented programming is more specific and comprehensive in its prescription for problem solving than structured programming is. The latter is concerned with "structure" in general, while object-oriented programming focuses on a specific form of structure: that associated with objects.

Modeling entities by their behavior (their response to messages) is a central principle of scientific method in many disciplines: behaviorism in psychology, operationalism in physics, and Platonic ideals in philosophy. Objects are a canonical form of description for any discipline or domain of discourse. Its universality as a representation, modeling, and abstraction technique supports the view that the object-oriented paradigm is conceptually and computationally fundamental.

Bibliography

1968. Dahl, O. J., Myrhaag, B., and Nygaard, K. *Simula 67 Common Base Language.* Norwegian Computing Center. Revised in 1970, 1972, and 1984.
1983. *Reference Manual for the Ada Programming Language.* US Dept of Defense.
1983. Goldberg, A., and Robson, D. *Smalltalk 80: The Language and its Implementation.* Reading, MA: Addison-Wesley.
1990. Wegner, P. "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger,* **1**, *1* (August), 7–87.
1991. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Upper Saddle River, NJ: Prentice Hall.
1997. Wegner P. "Why Interaction is More Powerful than Algorithms," *Comm. of the ACM,* **40**, *5* (May), 80–91.
1998. Arnold, K., and Gosling, J. *The Java Programming Language,* 2nd Ed. Reading, MA: Addison-Wesley.

Peter Wegner

OBJECT PROGRAM

For articles on related subjects *see* LANGUAGE PROCESSORS; LINKERS AND LOADERS; PROCEDURE-ORIENTED LANGUAGES; and SOURCE PROGRAM.

An *object program* is the output of a translating program, such as an assembler or a compiler, which converts a *source program* written in one language into another language, such as machine language, capable of being executed on a given computer.

This output may be in one of several forms: It may be in an intermediate language (*q.v.*), needing further translating; it may be *relocatable,* in which data and program references are still expressed relative to a base address; or it may be *absolute,* in which all linkages between program elements have been made and absolute address assignments established so that the program is ready to be loaded and executed. Usage varies as to which of these may be called the *object program.* In some sense, any output of a translating program is the object of that step, and hence is an object program, but the term is most often used to denote a binary file that, after *linking* to other binary files, is ready for direct execution (*see* LINKERS AND LOADERS).

Charles H. Davidson

OCR

See OPTICAL CHARACTER RECOGNITION.

OFFICE AUTOMATION

See ELECTRONIC OFFICE.

ONLINE CONVERSATION

For articles on related subjects *see* BULLETIN BOARD; COMPUTER CONFERENCING; ELECTRONIC MAIL; GROUPWARE; INTERNET; and WORLD WIDE WEB.

Online conversation is communication between two or more participants in which there is little or no perceived delay between sending a message and it being received and read. Whereas electronic mail may be compared to sending or receiving a letter by post, online communications are very much like conversations carried out in person or on a telephone.

Two online conversation methods whose popularity and use grew explosively in the late 1990s are Instant Messaging (IM) and Chat. IM extends a service to the Internet that had been available on many time-shared computer systems for years. This service would typically allow a user to determine whether another user was logged in and, if so, to send a message directly to his or her terminal, or to open up a two-way "talk" session with split screens for the two sides of the conversation. IM services make this possible on the Internet via a "Buddy List" of the set of online users with whom a user may exchange instant messages. In order to use this service, an Internet user must run an IM program which displays the Buddy List. These programs connect to specialized servers which keep track of everyone currently running that IM program as well as which of these users have "buddied" with which other users. When a user on someone's list comes online (and runs a copy of the IM program), the first user's list will indicate that this user is online (and vice versa) and will allow instant messaging. When a message is sent it usually pops up in a window on the recipient's computer screen. If the recipient chooses to reply, a conversation is begun.

IM was first popularized by America Online and brought to the Internet largely by Mirabalis' ICQ