# OBJECT-ORIENTED PROGRAMMING

In the past decade or so, object-oriented (OO) technology has become one of the dominant technologies in the computing industry (1,2). In fact, the 1990s have been widely known as the "decade of object-orientation (OO)" from the software development point of view. The abbreviation OO is commonly used to refer to both object-oriented, an adjective, and, object-orientation, a noun, depending on the context in which it is used. In a survey conducted in the early 1990s (3), it was reported that over 75% of the Fortune 100 companies have adopted OO technology to some degree for their computing needs. A recent follow-up survey indicates that the companies are beyond looking at OO technology on the drawing board and that they are using objects as part of their main software development technologies.

Many computer science disciplines have also successfully integrated OO technology as a new approach to problem solving into their respective areas of research and development. This is evident from the use of the burgeoning OO technology in the areas of programming (4–6), database management systems (7,8), and systems analysis and design (9–12), just to name a few.

The popularity of OO can be further demonstrated by looking at the number of computer programmers that has turned to OO. It has been reported that for C++, a popular OO programming language, "there's is a doubling of C++ programmers every seven months (13)!" Also to echo the ubiquity of OO, the keynote speaker of a recently held OOPSLA conference (14), the biggest OO conference in the world, stated that "OO is now in the air we breath" (in reference to the popularity of OO).

The push for OO can be attributed to recognition by the software development community that as software becomes more and more complicated, there needs to be a better means of tackling the overwhelming amount of software backlog problems and software engineering inefficiencies. (These problems and inefficiencies are primarily ascribed by many to the traditional procedural approach to software development.) The OO approach is touted as a means of handling the aforementioned problems.

So, what is the OO approach and how does it differ from the traditional approach? Swiftly onset, the OO approach differs in that instead of focusing on just the procedural aspects of software design, which is *not* how users perceive their requirements, the OO approach packages the data and the programs that manipulate the data into a single unit called an *object*. This is a drastic departure from the procedural approach where the data and program are very much separated.

To model the real-world objects more closely, the OO approach attempts to mimic how real-world objects are perceived. Imagine if you were asked to describe an airplane to someone. Your answer might be that an airplane is an object that supports a form of transportation by air; namely, it flies. Alternatively, your answer might be that airplane allows one to get from point A to point B relatively quickly, that is, you emphasize the speed by which it transports. As can be seen in this simple example, an object can typically be described by its *behavior* (e.g., an airplane flies) and its *data* (e.g., the speed of an airplane).

With the OO approach, an airplane has both the data and behavior packaged into a single unit. The behavior (also commonly called the *services* or *operations*) allows the users of the objects to manipulate the data indirectly, that is, via a well-defined protocol in the behavior (e.g., request a change on the current cruising altitude of an airplane object). With the traditional approach, a data structure is defined to capture the data. Then the data structure needs to be passed from function to function to manipulate the data. Because no behavior is associated with the data, there is no well-defined protocol for data manipulation, and this can be problematic for the integrity of the data.

At the macro level, an OO system exemplifies the way real-world objects interact with each other by having the objects in the system communicate by sending *messages.* The messages represent requests for the objects to exhibit their behavior (e.g., to fly an airplane at a certain speed). The autonomous manner in which individual objects interact via message passing is analogous to a traffic control system (15). Each object (e.g., car, bus, pedestrian) adheres to certain internalized rules that govern how the object should behave vis-a-vis the other objects when they are in their respective states. For example, when a traffic light object changes its state from green to red, then a car should stop, and a pedestrian going in a different direction should proceed. If all of the objects behave correctly, then the collaborating objects go through the traffic system safely without any accidents!

That is in contrast to a train control system, a metaphor for a traditional computer program, where a central control facility is established to monitor the movements of all trains. Each train's state or status is controlled by the master facility at any given time. The objects are not autonomous and the system resembles a top-down, functional decomposition approach to software development.

Another example that illustrates the change of mind-set when comparing OO with traditional computations is the comparison between making a peanut butter jelly sandwich and running a luncheonette (16). Here, when addressing the former, one would concentrate on the procedural (i.e., algorithmic) aspect of the problem, namely, first take two slices of bread, then a jar of peanut butter, and then spread some peanut butter on one slice, etc. This model of computation is concerned very much with how to tell the computer what to do. Now imagine running a luncheonette and the tasks involved. Clearly, the emphasis here is on how the community of objects involved interacts, that is, what the responsibilities of the waiter, cook, manager, busboy, and so on, are and how these objects collaborate. The details of how a cook may prepare a dish, that is, the recipe, is only part of the overall system of interacting objects.

In summary, a twist on a well-known quote, given here, provides the essence of OO thinking: "Ask not what you can do *to* your data structures, but what your data structures can do *for* you (17)."

This quotation clearly shows that the OO approach is fundamentally different from the procedural approach at both the micro and macro levels. The more natural way of modeling system requirements in OO allows software designers to communicate better with users, and as a result better quality software can be built. This is the great potential that the new OO paradigm offers. However, because of the paradigm's newness, it is generally a challenge for someone who has been preexposed to the traditional paradigm to shift to the OO way of thinking. Some studies have reported that the paradigm shift might take as long as six months.

The remainder of this article is organized as follows. The second section gives a brief historical tour of OO programming. The fundamentals, that is, the "nuts and bolts" of OO are discussed in the following section. The next section describes the defining characteristics of OO, polymorphism, inheritance, and encapsulation, and provides examples and benefits for each of the characteristics. The state of the art of object technology is examined in the last section.

## OO PROGRAMMING: A BRIEF LOOK AT THE HISTORY

Even though OO was not known to the software community at large until the 1980s, almost all of the major concepts of OO were developed in the 1960s by Dahl and Nygaard in the Simula 67 programming language (18).

As the name Simula suggests, the Simula 67 language was inspired by problems involving the simulation of real-life systems. Thus, the general software community was not aware of the general appeal of the language design in the early years. In fact, the importance of the language constructs was recognized only slowly, even by the original developers of the language (19).

It was not until Alan Kay, considered by many the father of OO, organized a research group at Xerox PARC in the 1970s and developed a language known as Smalltalk that the Simula language appeared again. Kay was concerned with discovering a programming language that would be understandable to noncomputer professionals, and he found the notion of computation by simulation a metaphor that novice users easily understand. As a result, Smalltalk was developed with Simula as its strongest influence. The Smalltalk language evolved through a number of iterations within Xerox PARC in the 1970s and Smalltalk-80, the end product, was presented to the world. In a widely read issue of *Byte Magazine* in 1981 (2), Smalltalk-80 was showcased, and the software community (not the world just yet) took notice.

In almost the same time period but slightly later, Bjarne Stroustrup at AT&T Bell Laboratory was working on an extension to the C programming language. Again, much like Smalltalk, the extension was heavily influenced by Simula 67. This extension eventually evolved into the C++ programming language (20). But unlike Smalltalk, which is commonly called a "pure" OO programming language, C++ is more commonly called a hybrid language because C++ represents an extension to C, that is C++ is basically a better C plus object extensions. Thus a C++ program is program that is strictly procedural if none of the OO features is used.

Because of the installed base of the C language, when C++ first became available to the general public, it quickly became a popular language, first as a better C and later as

BusinessWeek
SOFTWARE MADE SIMPLE
It's called object-oriented programming— a way to make computers a lot easier to use. Here's what it can do for you.

**Figure 1.** This issue of *Business Week* features OO technology as the cover story.

an OO programming language. In fact, it has been reported that there was a doubling of C++ programmers every seven months during its initial years!

With the momentum generated from the Smalltalk and C++ projects, the OO industry was suddenly bombarded with many new OO programming languages in the late 1980s, including Eiffel, Objective-C, Object Pascal, Actor, and Common Lisp Object System (CLOS), to name a few. For better or worse, these languages came and went for the most part, and the industry settled more or less on two major OO programming languages, Smalltalk and C++. Smalltalk was able to survive the OO programming language competition partly because when non-C programmers (e.g., COBOL programmers) jump on the OO bandwagon, they find that C++ is too cryptic a language to learn and as a result, most flock to Smalltalk. In fact, in a keynote speech in 1997, Alan Kay said "I was the one who coined the term (object-oriented), and C++ was not what I had in mind!"

A number of milestones were also set in the late 1980s with respect to OO programming. One was the formation of the Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) international conference. The first conference was held in 1986, and it represents the one and only forum that allows researchers and practitioners in the field to come together and share their research and experience. Thousands of people attend the conference annually.

Another milestone was the publication of a cover story entitled "Software Made Simple" in *Business Week* magazine, September 1991 (1) (see Fig. 1). (The sidebar says "It's called object-oriented programming—a way to make computers a lot easier to use. Here's is what it can do for you.") This article detailed the OO paradigm and explained how OO makes software development easier than the traditional approach. The article introduced object technology to the world (not just the software community) and made OO recognizable to not only the "geeks" or "techies" but also to upper management, a necessary condition to succeed in adopting OO in an organization.

The evolution of OO programming continued with the introduction of Java in the early 1990s. James Gosling, a researcher at Sun Microsystems, envisioned a world of consumer electronic devices, such as VCRs, microwave ovens, TVs, and computers, all linked together via a computer network. To serve the need for an easily and reliably programmable system, Gosling and his team designed the programming language Oak later called Java. The name Java is not an acronym. It is for coffee, which is what the designers were drinking when the name change was discussed. (Oak, named after an Oak tree outside of Gosling's office, has already been used for an existing language.)

There were a number of design criteria on which Java must be based for it to serve the embedded consumer electronic market. One very important criterion is that it must be reliable. When a program written for a general-purpose computer fails, one reboots the computer. That is "to be expected" in the world of computer systems. But one should not expect a novice user of, say, a VCR, to have to reboot the VCR again and again because of some program's failure. Another important consideration is the ability to run a program in a platform-independent manner. This permits swapping the underlying chip/platform so that the program still runs under the new environment. This "write once, run anywhere" characteristic is now widely used as a marketing slogan for the Java language. Other characteristics describing Java include simple, OO, network, secure, portable, high-performance, multithreaded, and dynamic. It is interesting to note that when Gosling gave his keynote speech in the 1996 OOPSLA conference, he did not list OO as one of the characteristics. When asked, the reason given is that to say Java is OO is like saying here is a person and by the way, he/she also breathes.

Java as a language for embedded consumer electronics did not succeed, but Java as a language for the web took the industry by storm. When the original idea fell apart, the World Wide Web was just around the corner for Gosling and his team to regroup and retarget the language. The concept of an applet, a program that is embedded inside a web page, was devised and the rest is history.

The remainder of this article uses Java as the language of illustration. For illustrations of concepts in other languages, consult the appropriate language manuals or textbooks.

## OO: THE FUNDAMENTALS

At the heart of OO systems is the notion of a *class,* from which *objects* are instantiated. Once a class, which is an analysis-time or *design-time* concept, has been developed, an object, which is a *run-time* concept, can be created by instantiating the class. This is why a class is often called an object factory or a blueprint/template for object creation.

A class allows modeling both the *data* and *behavioral* aspects of an entity. Its *attributes* capture the data or the state, and its *methods* capture the behavior or services that the class provides. The code snippet (in Java) following illustrates how a class may be defined, how objects may be instantiated from the class, and how messages may be sent to the objects to request that certain services be performed.

```
 1. class Employee {
 2.     private String name;
 3.     private int salary;
 4.     public void setName (String n) {
 5.        name = n;
 6.     }
```

```
7.      public String getName () {
8.        return name;
9.      }
10.     public void setSalary (int sal) {
11.       if (sal >= 0)
12.         salary = sal;
13.       else
14.         System.out.println("Salary must be nonnegative!
            Salary not set.");
15.     }
16.     public int getSalary () {
17.       return salary;
18.     }
19.     public int getAnnualSalary () {
20.       return salary * 12;
21.     }
22.     public void print () {
23.       System.out.println("Name is"+name);
24.       System.out.println("Salary is"+salary);
25.     }
26. }
27. class TestEmployee {
28.     public static void main (String argv[]) {
29.       Employee e1 = new Employee();
30.       Employee e2 = new Employee();
31.       e1.setName("Smith");
32.       e1.setSalary(5000);
33.       System.out.println("Annual sal of"+e1.getNa-
          me()+"is"+e1.getAnnualSalary());
34.       e2.setName("Adams");
35.       e2.setSalary(−1000);
36. e2.print();
37.     }
38. }
```

Let us dissect the previous code. First, two classes are defined, Employee and TestEmployee. The keyword `class` (in lines 1 and 27) defines a class in Java. (All keywords are bold in the program listing shown.) The Employee class models the real-world Employee entity. It contains two attributes (lines 2 and 3) to store the name and salary of an employee object. The keyword `private` deals with the accessibility of the data, and it is covered in the next section. The class also contains six methods (lines 4–25) to define the services of the class. The get() and set() methods are typically provided to get and set each of the attributes defined for a class. They are sometimes called the *accessors* of a class.

The TestEmployee class is a dummy class created merely to test the functionality of the Employee class, that is it serves as a client, using services provided by the Employee class. (Java requires that all code be defined inside a class. Thus, no stand-alone code can be created to test the functionality of Employee.) The main method represents the point of entry for all Java applications. Java applets, on the other hand, have a different entry point. The method init() represents the entry point for applets. Upon entering this method, two employee objects are created (lines 29 and 30). This is done using the `new` operator. The statements can be read as "Declare an object reference of type Employee, call it e1 (for line 29), and assign it the object that is created from the Employee class using the new operator." At this point e1 and e2 reference two Employee objects, as depicted in Fig. 2. Name defaults to empty string, and salary defaults to zero.

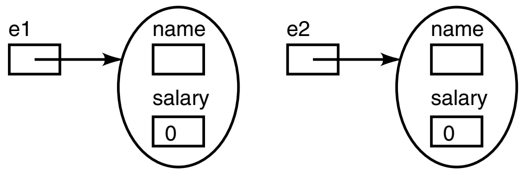

**Figure 2.** e1 and e2 are object *references* (or simply *pointers* in some languages). They point to memory locations that have been allocated for the objects. The objects take on default values initially.

In lines 31 and 32, the attributes for the first employee object are set by sending setName() and setSalary() messages to e1, the object reference. Because the object understands these messages (it is instantiated from the Employee class and the methods setName() and setSalary() are defined in the class), the requests are honored. Following that, more messages are sent to the object to retrieve its name and annual salary. The retrieved data are concatenated together into one string and sent to the standard output stream.

Similarly, the messages setName() and setSalary() are sent to e2 (lines 34 and 35). But because the salary argument sent is negative, the method setSalary() rejects the request, and the salary attribute is not set in this case. This illustrates how a class designer can provide the necessary integrity checks to make sure that an object of the class carries only valid data. This topic is discussed further in the next section. Lastly, a message is sent to e2 to request that the object prints itself. The output of the program is given below.

```
Annual salary of Smith is 60000
Salary must be non-negative! Salary not set.
Name is Adams
Salary is 0
```

The new states of the objects are also depicted in Fig. 3. String is not a primitive data type in Java, and thus the diagram shown is a simplification of what actually happens. The *name* attribute should be a reference to a String object.

## OO DEFINING CHARACTERISTICS: THE PIE

The defining characteristics of OO can be summarized simply as PIE, polymorphism, inheritance, and encapsulation. Each of the characteristics is presented following, in reverse order, E, I, and then P because P relies on the notion of I.

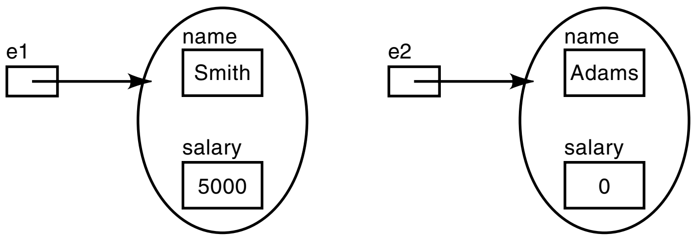

**Figure 3.** e1 and e2 references two objects that have their values set via the set() methods. Set() methods are the well-defined protocol for changing the contents of objects.

### Encapsulation

The term encapsulation means hiding the internal implementations of objects from their clients to abstract the underlying complexities and details. To understand what encapsulation is good for, all that you need to do is to imagine that a student object representing you contains a GPA of −2.5 or an employee object representing you contains a salary of −$50,000.

The key idea here is that an object's client should not be allowed to access the object's internal details directly. That is, if a client attempts to execute the statement

```
anEmployee.salary = −50000
```

the attempt should fail. In fact, if the salary attribute of the Employee class is declared `private`, a Java compiler will produce an error message that resembles "Variable salary in class Employee not accessible from class TestEmployee." Some OO programming languages (e.g., C++, Java) require that you specify the accessibility of the attributes to get the desired effect (i.e., private, public, protected) whereas others simply set all attributes to private (e.g., Smalltalk). On the other hand, if the attribute is `public`, then it is accessible by all, and it can take on any value set by the client.

So, how does one access the attributes if they are inaccessible by the clients? This is accomplished by having the clients communicate with the object via a standard protocol, that is the object interface. In the previous, the method setSalary(int aSalary) is needed to request a change to the salary attribute. Then the method can perform any sort of constraint checking against the parameter when a change is requested. This ensures that the clients are prohibited from accessing the attributes directly, and also are hidden from the implementation of the attributes.

Communication by only public interfaces also means that method implementations are hidden from the clients. Thus, an object's client cannot rely on a certain implementation used in the method (e.g., linear search rather than binary search) or on the attributes in terms of data structures (e.g., array rather than list implementation). This permits better program maintenance because the class designer can change the implementation details without affecting the client's code.

Encapsulation also allows building more complex systems because much of the complexities of a system can be hidden away from its clients via an appropriate interface. As long as the client and server understand the "contract" between them, hence the notion of *design by contract* (21), the two parties can go about using one another's services happily. This is how class libraries have been developed to support OO systems development.

### Inheritance

In real life, inheriting an estate from one's ancestor is typically considered a good thing. One gets something for nothing! Similarly, in OO software development, one can inherit free from others. Specifically, a class can inherit from one or more classes depending on the OO programming language used. In languages, such as Smalltalk and Java, a class B may inherit only from another class A. This is called *single inheritance.* Class B is called the *subclass,* and class A is called the *superclass.* (In C++, the term *derived* class is used instead of subclass, and the term *base* class is used instead of superclass.)

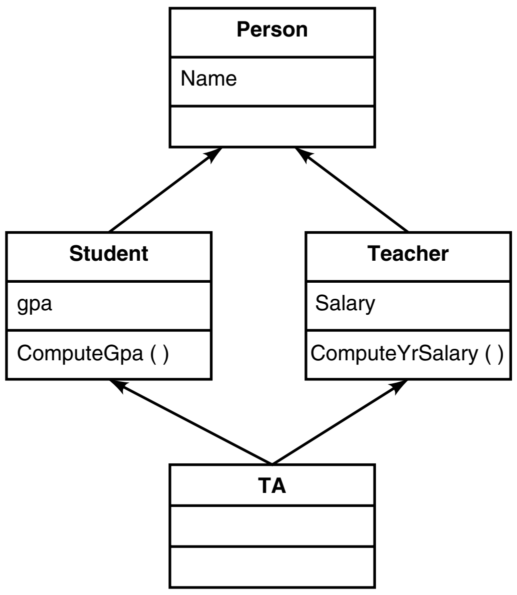

**Figure 4.** Class hierarchy showing inheritance of classes.

In languages, such as C++, a class may inherit from multiple classes. This notion is known as *multiple inheritance.*

Figure 4 shows a class hierarchy, illustrating the various inheritance possibilities. The diagram uses a standard notation known as Unified Modeling Language (UML), described later in this article. Each box represents a class. The top pane of a box represents the class name, the middle pane shows the attributes of the class, and the bottom pane models the methods. The arrows represent inheritance relationships.

When a class inherits from another class, the subclass inherits all of the attributes and methods from the superclass. For example, when Student inherits from Person, it gets the name attribute. Thus, an object of the Student class has both the name and gpa attributes and the method computeGpa(). If multiple inheritance is supported, TA can inherit from both the Student and Teacher classes. As a result, a TA object consists of name, gpa, and salary as attributes and computeGpa() and computerYrSalary() as methods. This behavior gives the desired effects because a TA *is a* Student and a Teacher. For this reason, inheritance is sometimes called an *is-a* relationship and is-a is oftentimes used to test whether an inheritance relationship is appropriate.

The previous gives a rather simplistic view of inheritance in general. Some languages support *selective inheritance* (e.g., Eiffel), that is a class can choose to inherit only a selective set of attributes/methods from another class whereas others provide features, such as *private inheritance* (e.g., C++), to hide inherited attributes/methods from the clients of the subclass. In the previous example, will a TA object also get two copies of the name attribute because each of Student and Teacher gets one? The answer in C++ is "yes" unless *virtual inheritance* is used. To learn more about the intricacies of inheritance in the respective languages, consult the appropriate language manuals.

With inheritance, the development time shortens, and also the maintenance of programs is easier because if changes are needed in certain classes, the subclasses get the changes au-

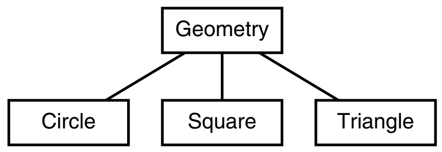

**Figure 5.** A hierarchy of geometric objects.

tomatically by virtue of inheritance. Thus, maintenance needs to be made only at the appropriate places.

Other benefits of inheritance include rapid prototyping, consistency of interface, increased software reliability, and classification of objects.

## Polymorphism

The term polymorphism means many forms. In the context of OO, it means that a message sent to an object can be interpreted in many forms depending on what the receiver object is. A message *X* when sent to object1 may invoke method1. Now when the *same* message *X* is sent to object2, which is instantiated from a different class, method2 may be invoked instead.

As an example, consider the task of printing geometric objects stored in a container. These objects may come from different classes, for example, circle, square, and triangle (see Fig. 5).

Given the above hierarchy shown, a container object may be created as follows:

```
Geometry myGeoContainer [] = new Geometry [10];
myGeoContainer [0] = new Circle();
myGeoContainer [1] = new Square();
myGeoContainer [2] = new Triangle();
...
```

The notation [] represents arrays in Java. Thus, the previous states that myGeoContainer is an array of object references, and each is a reference to an object of a type of Geometry. It is perfectly legal to assign a circle, square, or triangle object to an element of myGeoContainer. Each array element expects a Geometry object and each of circle, square, or triangle *is a* Geometry object. (Recall the is-a relationship discussed in the earlier section.)

Now back to the task at hand. To print the objects in the container myGeoContainer, one can use a code segment that resembles the following:

```
for (int j=0; j<myGeoContainer.length; j++){
  switch(myGeoContainer[j].tag){
    case CIRCLE: printCircle(myGeoContainer[j]); break;
    case SQUARE: printSquare(myGeoContainer[j]); break;
    case TRIANGLE: printTriangle(myGeoContainer[j]);
      break;
    default: error(); break;
  }
}
```

Although the previous code segment carries out the task, the solution is not extensible. Consider the scenario that a new type of object, say, Rectangle, is to be added to the container. Now, the switch construct shown previously must be changed to accommodate this new case, or an error results. When such as a change is frequent and it propagates to numerous other segments of codes, the maintenance cost is prohibitive.

An alternate solution that is more dynamic and extensible uses the concept of polymorphism. Consider the following segment of code.

```
for (int j=0; j<myGeoContainer.length; j++){
  myGeoContainer[j].print();
}
```

Instead of writing a switch construct to handle the various types of objects, one needs simply to send a generic print message to the object being examined. Assuming that all the subclasses of Geometry implement their own print() methods which can be ensured by making Geometry an *abstract class* and print() an *abstract method.* This forces the subclasses to implement the print() method or they cannot be instantiated. Then the previous code iterates over the container, sends the print message to the first object, that is, a Circle object, then to the second, a Square object, and lastly to a Triangle object. In each of the three cases, the print method from the appropriate class is invoked to handle the print request.

When a new object is added to the container (e.g., a Rectangle object), the OO code presended previously need not be changed at all. This illustrates how polymorphism makes program maintenance easier. In general, polymorphism allows writing more generic code and that increases the reusability of the software.

## OBJECT TECHNOLOGY: THE STATE OF THE ART

Object technology has come a long way since the introduction of Simula 67. It has evolved from a technology (primarily just the programming language component in the early years) that targeted only simulation-related applications to one that is used in major sectors of the economy—banking, defense, manufacturing, retail, to name a few. When used in these sectors, the programming language component is merely a piece of a much larger set of complementary technologies (e.g., OO modeling tools, rapid application development (OO RAD) tools, class libraries, components, and OO databases).

The advancement of object technology can be observed further from looking at the standardization efforts surrounding many facets of the technology. They include the standardization of a number of OO programming languages, OO modeling language, and OO databases. Each of these is briefly elaborated following.

With respect to programming language standardization, the C++ programming language has finally been approved by the C++ Committee of the International Standards Organization (ISO) after eight years of deliberation on the language features. The standard covers both the C++ language itself and its standard library. The final ratification by two dozen countries was expected by March 1998. With respect to Java, Sun Microsystems Inc. has won ISO approval to become a Publicly Approved Submitter of standards for the programming language. This means that Sun wins the control of the Java trademark and the specification's maintenance. These two events signify the shakeup and maturity of the technology.

All of the previous discussion deals with objects from an OO programming perspective. But before implementing an

OO system, one first needs to perform numerous activities in the system development life cycle (SDLC) including feasibility study, requirements specification, analysis, and design of the system. Collectively, the notations used to document the artifacts of each of phase in the SDLC and the process that prescribes how the various phases are to be carried out is called a methodology.

The state of OO methodology has evolved from panel discussions in past OOSPLA conferences that are entitled "Which Method Is Best? Shoot Out at the OO Corral" and "OO Methodology Standard: Help or Hindrance" to the approval of Unified Modeling Language (UML) as the standard OO modeling language by the Object Management Group (OMG), a consortium of software technology companies. UML can be defined as a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system (22). This work is the collaboration of many from the software industry led by Grady Booch, James Rumbaugh, and Ivar Jacobson (the "Three Amigos"). The amigos had their own methodologies before the consolidation of methodologies began in the mid-1990s. This, again, shows the maturity of object technology.

Although the third leg of the three standardization efforts, the OO database standard, is lagging in terms of overall acceptance, it is a standard defined for object databases by the Object Database Management Group (ODMG), a subgroup of OMG. ODMG is also working with American National Standards Institute (ANSI) and ISO to define the next standard for Structured Query Language (SQL), called SQL3, that includes object extensions.

While the various database standards are being worked on by the respective bodies, many tool vendors are already capitalizing on object-to-relational mapping tools to leverage existing relational systems. This permits the users to keep their investments in their relational databases and yet take advantage of the power of objects. However, it has been shown that this technique can quickly "hit the relational wall (23)."

In addition to the standardization work previously overviewed, there are many other efforts that show the advancement and continued growth of object technology. Two of these efforts, distributed object computing and patterns, are discussed here.

With the advent of the Web, distributed object computing is receiving more and more attention from the software community. The idea is that in the era of network computing, where everyone is globally connected, inexpensively, to each other, objects can be distributed and accessed across the network easily because the framework for distribution is readily available via the Web. This area of research has been active for a number of years, ever since object technology became popular in the mid to late 1980s. But without a distribution channel as economical and as popular as the Web, this research area has not been fully explored by the software community.

Now, as expected, when the potential for the distributed object computing market is so enormous, there are bound to be a number of players in the field. Indeed, there are a few major architectures available, including OMG's Common Object Request Broker Architecture (CORBA)/Internet Inter-ORB Protocol (IIOP), Microsoft's Distributed Common Object Model (DCOM), and Java's Remote Method Invocation (RMI). These architectures spell out the protocols of how an object on one machine can communicate easily across machine boundaries to other objects. An object can invoke methods on remote objects almost as easily as a local object method invocation. Many object visionaries are predicting that object interfaces will become as ubiquitous as Web interfaces (24). If this happens, the dream of object enthusiasts to have ubiquitous objects will be realized sooner than later.

Another current trend in the OO community is realizing the power of reuse and how object technology is very well suited for reuse. The phenomenon occurs in many forms, one of which is the study of patterns, specifically design patterns. The idea is that like most complex structures, good computer programs can (and should) mimic the structure of similar, proven effective, older programs. By imitating these older programs, one need not start the analysis and design efforts over again. One can reuse existing analysis and design solutions. Thus, a *design pattern* is an attempt to capture and formalize the process of imitation (17).

One of the first attempts to describe the concepts of design pattern is the work by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, commonly referred to as "The Gang of Four" (25). Their work was heavily influenced by the architect Christopher Alexander, who described the patterns of building livable homes (26). Since then, numerous other pattern-related projects have surfaced (27,28), and the software community is again attempting to digest the voluminous amount of research on the topic.

The two technologies showcased previously represent some of the advances in object technology in recent years. They continue to be topics of interest in terms of where objects are heading. Some of the other notable topics include objects for small-scale devices and objects for the large.

Ironically, Java, now represents the hottest object technology and is being heavily used in embedded devices, such as TV set top boxes, cellular phones, and personal digital assistants (PDAs). (Recall that Java's birthplace was consumer electronics.) Many software and electronics industry giants (e.g., Sun Microsystems, SONY, TCI) are collaborating to develop intelligent small devices using Java.

On the other end of the extreme in scale, companies like IBM are also capitalizing on the popularity of objects in their large-scale systems (e.g., IBM OS 390). IBM's newest operating system, OS 390, which runs on mainframes, is touted to be the superserver for its clients via the Web or the traditional means. With its Domino web server and Java for OS390, now IBM mainframe users can take advantage of the power of objects and the Web that for many years have been accessible only at the PC/workstation level.

## BIBLIOGRAPHY

1. Software made simple: Object-oriented programming, *Business Week,* September 30, 1991.
2. Outlook '92, *Byte Magazine,* October, 1991.
3. Executive Summary, *Object Mag.,* **2** (2): 1992.
4. M. Ellis and B. Stroustrup, *The C++ Annotated Reference Manual,* Reading, MA: Addison-Wesley, 1990.
5. A. Goldberg and D. Robson, *Smalltalk-80: The Language,* Reading, MA: Addison-Wesley, 1989.

6. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification,* Reading, MA: Addison-Wesley, 1996.
7. R. Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems,* Reading, MA: Addison-Wesley, 1991.
8. D. Maier and S. Zdonik, *Readings in Object-Oriented Database Systems,* San Mateo, CA: Morgan-Kaufmann, 1990.
9. G. Booch, *Object-Oriented Design with Applications,* 2nd ed., Redwood City, CA: Benjamin-Cummings, 1994.
10. M. Fowler and K. Scott, *UML Distilled,* Reading, MA: Addison-Wesley, 1997.
11. I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach,* Reading, MA: Addison-Wesley, 1992.
12. J. Rumbaugh et al., *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
13. Interview with Barjne Stroustrup, in the videotape *The World of Objects,* Borland International, 1991.
14. *Proc. Int. Conf. Object-Oriented Programming: Syst., Languages, and Applications,* Portland, OR, October 1994.
15. E. Anderson, What the Hell is OOPS, Anyway? Harvard Business School, **9**: 192–104, 1992.
16. L. A. Stein, *Interactive Programming in Java: A Non-Standard Introduction,* Tutorial #14, OOPSLA, Atlanta, GA, October 1997.
17. T. Budd, *Understanding Object-Oriented Programming Using Java,* Reading, MA: Addison-Wesley, 1998.
18. O-J. Dahl and K. Nygaard, Simula, An algol-based simulation language, *Commun. ACM,* **9** (9): 671–678, 1966.
19. K. Nygaard and O-J. Dahl, The development of the Simula languages, in R. Wexelblat, (ed.), *History of Programming Language,* New York: Academic Press, 1981.
20. B. Stroustrup, *The Design and Evolution of C++,* Reading, MA: Addison-Wesley, 1994.
21. B. Meyer, *Object-Oriented Software Construction,* Englewood Cliffs, NJ: Prentice-Hall, 1988.
22. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language,* OOPSLA Tutorial Notes, Rational Software Corp., 1997.
23. M. Loomis, Hitting the relational wall, *J. Object-Oriented Programming,* January, 1994.
24. D. Orchard, Java component and distributed object technologies, *Object Mag.,* **7** (11): 1988.
25. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley, 1995.
26. C. Alexander et al., *Pattern Language,* New York: Oxford Univ. Press, 1977.
27. J. Coplien and D. Schmidt, *Pattern Languages of Program Design,* Reading, MA: Addison-Wesley, 1995.
28. M. Fowler, *Analysis Patterns,* Reading, MA: Addison-Wesley, 1997.

BILLY B. L. LIM
Illinois State University