Object-Oriented Software Systems

David Robson
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

This article describes a general class of tools for manipulating information called object-oriented software systems. It defines a series of terms, including software system *and* object-oriented. *The description is greatly influenced by a series of object-oriented programming environments developed in the last ten years by the Learning Research Group of Xerox's Palo Alto Research Center, the latest being the Smalltalk-80 system. The article describes object-oriented software systems in general, instead of the Smalltalk-80 system in particular, in order to focus attention on the fundamental property that sets the Smalltalk-80 system apart from most other programming environments. The words "object-oriented" mean different things to different people. Although the definition given in this article may exclude systems that should rightfully be called object-oriented, it is a useful abstraction of the idea behind many software systems.*

Many people who have no idea how a computer works find the idea of object-oriented systems quite natural. In contrast, many people who have experience with computers initially think there is something strange about object-oriented systems. (I don't mean to imply that computer-naive users can create complex systems in an object-oriented environment more easily than experienced programmers can. Creating complex systems involves many techniques more familiar to the programmer than the novice, regardless of whether or not an object-oriented environment is used. But the basic idea about how to create a software system in an object-oriented fashion comes more naturally to those without a preconception about the nature of software systems.) I had had some programming experience when I first encountered an object-oriented system and the idea certainly seemed strange to me. I am assuming that most of you also have some experience with software systems and their creation. So instead of introducing the object-oriented point of view as if it were completely natural, I'll try to explain what makes it seem strange compared to the point of view found in other programming systems.

Software Systems

A software system is a tool for manipulating information. For the purposes of this article, I'm using a very broad definition of information.

> **Information:** *A representation or description of something.*

There are many types of information that describe different things in different ways. One of the great insights in computer science was the fact that information can (among other things) describe the manipulation of information. This type of information is called *software*.

> **Software:** *Information describing the manipulation of information.*

Software has the interesting recursive property of describing how to manipulate things like itself. Software is used to describe a particular type of information-manipulation tool called a *software system*.

> **Software system:** *An information-manipulation tool in which the manipulation is described by software.*

A distinction is made in information-manipulation tools between *hardware* systems and *software* systems. A hardware system is a physical device like a typewriter, pen, copier, or television set. The type of manipulation performed by a hardware system is built in and can only be changed by physical modification. The type of manipulation performed by a software system is not built in—it is determined by information, which can be manipulated.

The virtue of software systems is that the mechanism developed for manipulating information can be used to manipulate the mechanism itself. Software systems that actually manipulate other software systems are called *programming environments*.

> **Programming environment:** *A software system that manipulates software systems. An environment for the design, production, and use of software systems.*

Thus, a programming environment is also recursive: it *is* what it manipulates. The fact that software systems can be manipulated is both good news and bad news. Since a text editor is a software system, it is not "cast in concrete" and you can change it to conform to your style of interacting with text more closely than it does now (using a programming environment). However, you also may reduce it to the proverbial "pile of bits" (not a text editor at all).

Data/Procedure-Oriented Software

The traditional view of software systems is that they are composed of a collection of *data* that represents some information and a set of *procedures* that manipulates the data.

> **Data:** *The information manipulated by software.*
>
> **Procedure:** *A unit of software.*

Things happen in the system by invoking a procedure and giving it some data to manipulate.

As an example of a software system, consider a system for managing windows that occupy rectangular areas on a display screen. The windows contain text and have titles. They can be moved around the screen, sometimes overlapping each other. (The details of this system are not important. Its main purpose is to point out the differences between the structure of a data/procedure system and an object-oriented system.)

A window-management system implemented as a data/procedure system would include data representing the location, size, text contents, and title of each window on the screen. It would also include procedures that move a window, create a window, tell whether a window overlaps another window, replace the text or title of a window, and perform other manipulations of windows on a display. To move a window, a programmer would call the procedure that moves windows and pass to it the data representing the window and its new location.

A problem with the data/procedure point of view is that data and procedures are treated as if they were independent when, in fact, they are not. All procedures make assumptions about the form of the data they manipulate. The procedure to move a window should be presented with data representing a window to be moved and its new location. If the procedure were presented with data representing the text contents of a window, the system would behave strangely.

In a properly functioning system, the appropriate choice of procedure and data is always made. However, in an improperly functioning system (eg: one in the process of being developed or encountering an untested situation), the data being manipulated by a procedure may be of an entirely different form from that expected. Even in a properly functioning system, the choice of the appropriate procedure and data must always be made by the programmer.

These two problems have been addressed in the context of the data/procedure point of view by adding several features to programming systems. Data typing has been added to languages to let the programmer know that the appropriate choice of data has been made for a particular procedure. In a typed system, the programmer is notified when a procedure call is written using the wrong type of data. Variant records allow the system to choose the appropriate procedure and data in some situations.

Object-Oriented Software

Instead of two types of entity that represent information and its manipulation independently, an object-oriented system has a single type of entity, the *object*, that represents both. Like pieces of data, objects can be manipulated. However, like procedures, objects describe manipulation as well. Information is manipulated by sending a *message* to the object representing the information.

> **Object:** *A package of information* and *descriptions of its manipulation.*
>
> **Message:** *A specification of one of an object's manipulations.*

When an object receives a message, it determines how

to manipulate itself. The object to be manipulated is called the *receiver* of the message. A message includes a symbolic name that describes the type of manipulation desired. This name is called the message *selector*. The crucial feature of messages is that the selector is only a *name* for the desired manipulation; it describes *what* the programmer wants to happen, not *how* it should happen. The message receiver contains the description of how the actual manipulation should be performed. The programmer of an object-oriented system sends a message to invoke a manipulation, instead of calling a procedure. A message names the manipulation; a procedure describes the details of the manipulation.

Of course, procedures have names as well, and their names are used in procedure calls. However, there is only one procedure for a name, so a procedure name specifies the exact procedure to be called and exactly what should happen. A message, however, may be interpreted in different ways by different receivers. So, a message does not determine exactly what will happen; the receiver of the message does.

If the earlier example of the window-management system were implemented as an object-oriented system, it would contain a set of objects representing windows. Each object would describe a window on the screen. Each object would also describe the manipulations of the window it represents—for example, how to move it, how to determine whether it overlaps another window, or how to display it. Each of these manipulations would correspond to a selector of a message. The selectors could include move, overlap, display, delete, width, or height. (In this article, an alternate typeface is used for words that refer to specific elements in example systems.)

In addition to a selector, a message may contain other objects that take part in the manipulation. These are called the message *arguments*. For example, to move a

window, the programmer might send the object representing the window a message with the selector move. The message would also contain an argument representing the new location. Since this is an object-oriented system, the selector and argument are objects: the selector representing a symbolic name and the argument representing a location or point.

The description of a single type of manipulation of an object's information (the response to a single type of message) is a procedure-like entity called a *method*. A method, like a procedure, is the description of a sequence of actions to be performed by a processor. However, unlike a procedure, a method cannot directly call another method. Instead, it must send a message. *The important thing is that methods cannot be separated from objects.* When a message is sent, the receiver determines the method to execute on the basis of the message selector. A different kind of window could be added to the system with a different representation and different methods to respond to the messages move, overlap, display, delete, width, and height. Places where messages are sent to windows do not have to be changed in order to refer to the new kind of window; whichever window receives the message will use the method appropriate to its representation.

Objects look different from the outside than they do from the inside. By the outside of an object, I mean what it looks like to other objects with which it interacts (eg: what rectangles look like to other rectangles or to windows). From the outside, you can only ask an object to do something (send it a message). By the inside of an object, I mean what it looks like to the programmer implementing its behavior. From the inside, you can tell an object how to do something (in a method). For example, a window can respond to messages having the selectors move, overlap, display, delete, width, or height.

However, nothing is known outside the window about how it responds to these messages. (It is known that a window will move when asked to, but it is not known how it accomplishes the move.)

The set of messages an object can respond to is called its *protocol*. The external view of an object is nothing more than its protocol; the internal view of an object is something like a data/procedure system. An object has a set of variables that refers to other objects. These are called its *private variables*. It also has a set of methods that describes what to do when a message is received. The values of the private variables play the role of data and the methods play the role of procedures. This distinction between data and procedures is strictly localized to the inside of the object.

Methods, like other procedures, must know about the form of the data they directly manipulate. Part of the data a method can manipulate are the values of its object's private variables. For example, we might imagine three ways that a window represents its location and size (internally). The private variables might contain:

- four numbers representing the x and y location of the center, the width, and the height
- two points representing opposite corners of the window
- a single rectangle whose location and size are the same as the window's

The method that moves a window (the response to messages with the selector move) assumes that a particular representation is used. If the representation were changed, the method would also have to be changed. Only the methods in the object whose representation changed need be changed. All other methods must manipulate the window by sending it messages.

A message must be sent to an object to find out anything about it (ie: our concept of manipulation includes inquiring about information, as well as changing information). This is needed because we don't want the form of an object's inside known outside of it. The response to a message may return a value. For example, a window's response to the message width returns an object that represents its width on the display (a number). The method for determining what to return depends on the form of the window's private variables. If they are represented as the first alternative listed above (four numbers), the response would simply return the value of the appropriate private variable. If the second alternative is used (two points), the method would have to determine the width from the x coordinates of the two corners. If the third alternative is used (one rectangle), the width message would simply be passed on to the rectangle and the rectangle's response would become the window's response.

Classes and Instances

Most object-oriented systems make a distinction between the description of an object and the object itself. Many similar objects can be described by the same general description. The description of an object is called a *class* since the class can describe a whole set of related objects. Each object described by a class is called an *instance* of that class.

> **Class:** *A description of one or more similar objects.*
>
> **Instance:** *An object described by a particular class.*

Every object is an instance of a class. The class describes all the similarities of its instances. Each instance contains the information that distinguishes it from the other instances. This information is a subset of its private variables called *instance variables*. All instances of a class have the same *number* of instance variables. The *values* of the instance variables are different from instance to instance. An object's software (ie: the methods that describe its response to messages) is found in its class. All instances of a class use the same method to respond to a particular type of message (ie: a message with a particular selector). The difference in response by two different instances is a result of their different instance variables. The methods in a class use a set of names to refer to the set of instance variables. When a message is sent, those names in the invoked method refer to the instance variables of the message receiver. Some of an object's private

variables are shared by all other instances of its class. These variables are called *class variables* and are part of the class.

The programmer developing a new system creates the classes that describe the objects that make up the system. The programmer of the window-management system would create a class that contained methods corresponding to the message selectors move, display, delete, width, and height. This class would also include the names of the instance variables referred to in those methods. These names might be frame, text, and title, where:

> frame is a rectangle defining the area on the screen,
> text is the string of characters displayed in the window, and
> title is the string of characters representing the window's name

The classes representing rectangles and strings of characters are included in most systems, so they don't need to be defined.

In a system that is uniformly object-oriented, a class is an object itself. A class serves several purposes. In a running system, it provides the description of how objects behave in response to messages. The processor running an object-oriented system looks at the receiver's class when a message is sent to determine the appropriate method to execute. For this use of classes, it is not necessary that they be represented as objects since the processor does not interact with them through messages (preventing a nasty recursion). In a system under development, a class provides an interface for the programmer to interact with the definition of objects. For this use of classes, it is extremely useful for them to be objects, so they can be manipulated in the same way as all other descriptions. Classes also are the source of new objects in a running system. Here again, it is useful for the class to be an object, so object creation can be accomplished with a simple message. For example, the message new might be sent to a class to create a new instance.

Inheritance

Another mechanism used for implicit sharing in object-oriented systems is called *inheritance*. One object inherits the attributes of another object, changing any attributes that distinguish the two. Some object-oriented systems provide for inheritance between all objects, but most provide it only between classes. A class may be modified to create another class. In such a relationship, the first class is called the *superclass* and the second is called the *subclass*. A subclass inherits everything about its superclass. The following modifications can be made to a subclass:

- adding instance variables
- providing new methods for some of the messages understood by the superclass
- providing methods for new messages (messages not understood by the superclass)
- adding class variables

As an example, the window-management system might contain windows that have a minimum size. These would be instances of a subclass of the ordinary class of windows that added an instance variable to represent the minimum size and provided a new method for the message that changes a window's size.

Conclusion

The realization that information can describe the manipulation of information is largely responsible for the great utility of computers today. However, that discovery is also partially responsible for the failure of computers to reach the utility of some predictions made in earlier times. On the one hand, it can be seen as a unification between the manipulator and the manipulated. However, in practice, it has been seen as a distinction between software and the information it manipulates. For small systems, this distinction is harmless. But for large systems, the distinction becomes a major source of complexity. The object-oriented point of view is a way to reduce the complexity of large systems without placing additional overhead on the construction of small systems. ■