



An Object-Oriented System Development Primer

Before discussing specific object-oriented development activities, we must understand the basic object-oriented concepts. In addition, we must be able to differentiate the object-oriented approach from the procedural approach.

Procedural Programming vs. Object-Oriented Programming

There are primarily two methods of programming in use today: *procedural*, and *object-oriented*. The earliest programming languages were procedural. This means that a program is made of one or more *procedures*. A procedure is a set of programming language statements that are executed by the computer, one after the other. The statements might gather input from the user, manipulate data stored in the computer's memory, and perform calculations or any other operation necessary to complete its task. For example, suppose we want the computer to calculate someone's gross pay. Here is a list of things the computer should do:

1. Display a message on the screen asking "How many hours did you work?"
2. Allow the user to enter the number of hours worked. Once the user enters a number, store it in memory.
3. Display a message on the screen asking "How much do you get paid per hour?"
4. Allow the user to enter their hourly pay rate. Once the user enters a number, store it in memory.
5. Once both the number of hours worked, and the hourly pay rate are entered, multiply the two numbers and store the result in memory.
6. Display a message on the screen that tells the amount of money earned. The message must include the result of the calculation performed in step 5.

If the algorithm's six steps are performed in order, one after the other, it will succeed in calculating and displaying the user's gross pay.

Procedural programming was the standard when users were interacting with text-based computer terminals. For example, Figure C-1 illustrates the screen of an older MS-DOS computer running a program that performs the pay-calculating algorithm. The user has entered the numbers shown in bold.

Figure C-1

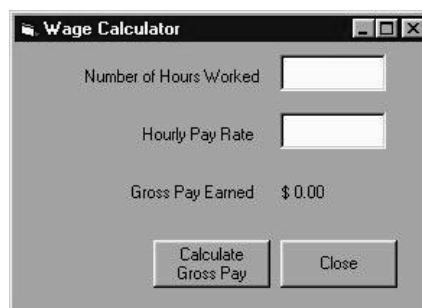
```
How many hours did you work? 10  
How much are you paid per hour? 15  
You have earned $150.00  
C>_
```

In text-based environments using procedural programs, the user responds to the program. Modern operating systems, however, such as Windows 2000 and Windows XP use a graphical user interface, or *GUI* (pronounced “gooey”). Although GUIs have made programs friendlier and easier to interact with, they have not simplified the task of programming. GUIs make it necessary for the programmer to create a variety of on-screen elements such as windows, dialog boxes, buttons, menus, and other items. Furthermore, the programmer must write statements that handle the user’s interactions with these on-screen elements, in any order they might occur. No longer does the user respond to the program, but the program responds to the user.

This has helped influence the shift from procedural programming to object-oriented programming. Whereas procedural programming is centered on creating procedures, object-oriented programming is centered on creating *objects*. An object is a programming entity that contains data and actions. The data contained in an object is known as the object’s *attributes*. The actions, or behaviors, that an object performs are known as the object’s *methods*. The object is, conceptually, a self-contained unit consisting of data (attributes) and actions (methods).

Object-oriented programming (OOP) has revolutionized GUI software development. For instance, in a GUI environment, the pay-calculating program might appear as the window shown in Figure C-2.

Figure C-2



This window can be thought of as an object. It contains other objects as well, such as text input boxes, and command buttons. Each object has attributes that determine the object’s

appearance. For example, look at the command buttons. One has the caption “Calculate Gross Pay” and the other reads “Close”. These captions, as well as the buttons’ sizes and positions, are attributes of the command button objects. Objects can also hold data that has been entered by the user. For example, one of the text input boxes allows the user to enter the number of hours worked. When this data is entered, it is stored as an attribute of the text input box.

The objects also have actions, or methods. For example, when the user clicks the “Calculate Gross Pay” button with the mouse, one would expect the program to display the amount of gross pay. The “Calculate Gross Pay” button performs this action.

The Benefits of OOP in Non-GUI Environments

The complexity of GUI software development was not the first difficult challenge that procedural programmers faced. Long before Windows and other GUIs, programmers were wrestling with the problems of code/data separation. In procedural programming, there is a distinct separation between data and program code. Data is kept in variables of specific data types, as well as programmer-defined data structures. The program code passes the data to modules that are designed to receive and manipulate it. But, what happens if the format of the data is altered? Quite often, a program’s specifications change, resulting in redesigned data structures, changed data types, and additional variables being added to the program. When the structure of the data changes, the modules that operate on the data must also be changed to accept the new format. This results in added work for programmers and a greater opportunity for bugs to appear in the code.

OOP addresses the problem of code/data separation through *encapsulation* and *data hiding*. Encapsulation refers to the combining of data and procedures into a single object. Data hiding refers to an object’s ability to hide its data from code that is outside the object. When an object’s data is hidden, it can only be accessed through the object’s procedures, so an object’s procedures provide an interface for programming statements outside the object to access the object’s data. This means that programming statements outside the object do not need to know about the type or internal structure of an object’s data. They only need to know how to interact with the object’s procedures. When a programmer changes the type or structure of an object’s internal data, he or she also modifies the object’s procedures that provide the interface to the data.

Component Reusability

Another trend in software development that has encouraged the use of OOP is *component reusability*. A component is a software object that performs a specific, well-defined operation or that provides a particular service. The component is not a stand-alone program, but can be used by programs that need the component’s service. For example, Sharon is a programmer who has developed a component for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her component is coded to perform all the necessary 3D mathematical operations and handle the computer’s video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline, and does not possess a great deal of knowledge about computer graphics, he can use Sharon’s component to perform the 3D rendering (for a small fee, of course!).

Component reusability and object-oriented programming technology set the stage for large-scale computer applications to become systems of unique collaborating entities (components).

An Everyday Example of an Object

Think of your alarm clock as an object. It has the following attributes:

- The current second (a value in the range of 0–59)
- The current minute (a value in the range of 0–59)
- The current hour (a value in the range of 1–12)
- The time the alarm is set for (a valid hour and minute)
- Whether the alarm is on or off (“on” or “off”)

As you can see, the attributes are merely data values that define the alarm clock’s “state.” The alarm clock also has the following methods:

- Increment the current second
- Increment the current minute
- Increment the current hour
- Sound alarm
- Set time
- Set alarm time
- Turn alarm on
- Turn alarm off

The methods are all actions that the clock performs. Each method manipulates one or more of the attributes. For example, every second the “Increment the current second” method executes. This changes the value of the current second attribute. If the current second attribute is set to 59 when this method executes, the method is programmed to reset the current second to 0, and then cause the “Increment current minute” method to execute. This method adds 1 to the current minute, unless it is set to 59. In that case, it resets the current minute to 0 and causes the “Increment current hour” method to execute. (It might also be noted that the “Increment current minute” method compares the new time to the alarm time. If the two times match, and the alarm is turned on, the “Sound alarm” method is executed.)

The methods described in the previous paragraph are part of the alarm clock object’s private, internal workings. External entities (such as yourself) do not have direct access to the alarm clock’s attributes, but these methods do. The object is designed to execute these methods automatically and hide the details from you, the user. These methods, along with the object’s attributes, are part of the alarm clock’s *private persona*.

Some of the alarm clock’s methods are publicly available to you, however. For example, the “Set time” method allows you to set the alarm clock’s time. You activate the method by pressing a set of buttons on top of the clock. By using another set of buttons, you can activate the “Set alarm time” method. In addition, another button allows you to execute the “Turn alarm on” and “Turn alarm off” methods. These methods are part of the alarm clock’s *public persona*. They define an interface that external entities may use to interact with the object.

Classes and Objects

In general terms, a *class* is a type, or category of object. A class specifies the attributes and methods that objects of that class possess. A class is not an object, however. An object is a specific instance of a class.

For example, Jessica is an entomologist (someone who studies insects) and enjoys writing computer programs. She designs a program to catalog different types of insects. In the program, she creates a class named `Insect`, which specifies variables and methods for holding and manipulating data common to all types of insects. The `Insect` class is not an object, but a data type that specific objects may be created from. Next, she defines a `housefly` object, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about houseflies. It contains the attributes and methods specified by the `Insect` class.

Object Relationships

Special relationships may exist between objects. The possible relationships may be formally stated as

- Access
- Ownership
- Inheritance

Informally, these three relationships may be described as

- Uses a
- Has a
- Is a

The access relationship

The first relationship, access, allows an object to modify the attributes of another object. Normally, an object has private attributes, which are not accessible to parts of the program outside the object. An access relationship between two objects means that one object has access to the other object's private attributes. When this relationship exists, it can be said that one object *uses* the other.

The ownership relationship

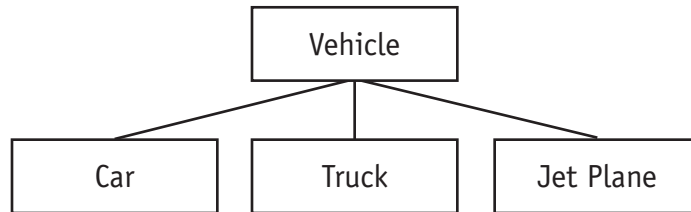
The second relationship, ownership, means that one object is an attribute of another object. For example, a human resources system might have an object that represents an employee. That object might have, as an attribute, another object that holds the employee's name. It can be said that the employee object has a name object. In OOP terminology, this type of relationship is known as a *whole-part hierarchy*, or *composition*.

The inheritance relationship

The third type of relationship is inheritance. Sometimes an object class is based on another class. This means that one class is a specialized case of the other. For example, consider a program that uses classes representing cars, trucks, and jet planes. Although those three types of objects in the real world are very different, they have some common characteristics: They

are all modes of transportation, and they all carry some number of passengers. So, each of the three classes could be based on a Vehicle class that has the attributes and behaviors common to all of the classes. This is illustrated in Figure C-3.

Figure C-3



In OOP terminology, the Vehicle class is the *base class*. The Car, Truck, and Jet Plane classes are *derived classes*. All of the attributes and behaviors of the Vehicle class are inherited by the Car, Truck, and Jet Plane classes. The relationship implies that a Car is a Vehicle, a Truck is a Vehicle, and a Jet Plane is a Vehicle. Inheritance is discussed in detail in Chapter 15.

In addition to inheriting the attributes and behaviors of the base class, derived classes add their own. For example, the Car class might have attributes and behaviors that set and indicate whether it is a sedan or coupe, and the type of engine it has. The Truck class might have attributes and behaviors that set and indicate the maximum amount of weight it can carry, and the number of miles it can travel between refueling. The Jet Plane class might have attributes and behaviors that set and indicate the plane's altitude and heading. These added components make the derived classes more specialized than the base class. For that reason, this type of relationship is often called a *generalization/specialization hierarchy*.

Messages

We have established the fact that objects have methods, which are behaviors. But how do we make an object perform a behavior? Quite simply, we send the object a *message*, requesting that it execute a particular named method. In C++, messages are actually member function calls. For example, suppose a program has a class named `Shape`, which has a public member function named `calcArea`. The program defines an instance of the `Shape` class named `square`, and contains the following statement:

```
square.calcArea();
```

This statement sends the `calcArea` message to the `square` object.

Developing an Object-Oriented System

Object-oriented systems development generally consists of two phases: object-oriented analysis, and object-oriented design. A *systems analyst* usually conducts these phases. This section briefly discusses the analyst's activities in each phase.

Object-Oriented Analysis

During object-oriented analysis, the analyst creates a logical design of the system. The logical design specifies what the system is to do, but does not specify how the system should do it. In general, the object-oriented analysis phase consists of the following steps:

1. Examine the problem domain and model the system from within the context of that perspective.

First, you should understand the nature of the problem you are trying to solve. This involves examining the following parameters:

- The significant events that are part of the problem.
- The internal parties that will interact with the system.
- The external parties that will interact with the system.
- The inputs and outputs required by the system.

Once you have identified the system's major events, internal and external parties, inputs, and outputs, you can create a model. The *Unified Modeling Language (UML)* is a tool that is widely used to graphically depict and document system designs. UML specifies a number of diagrams for modeling the various parts of a system. One such diagram is the *use case diagram*. A use case diagram shows the sequence of steps that take place in performing a task and identifies the parties (actors) involved in the task.

2. Identify objects (entities) that exist within the boundaries of the system.

In order to determine which classes will appear in a program, the analyst should think of the real-world objects and data elements that are present in the problem. One popular method of discovering the objects within a problem is to write a description of the problem with its major events and parties, then identify all the nouns in the description. Each noun is candidate to become a class. Here are examples of items that may be candidates for classes:

- The result of a business event, such as a customer order
- Physical objects, such as vehicles, machines, or manufactured products
- Record keeping items, such as customer histories and payroll records
- Any role played by a human (employee, client, teacher, student, and so forth)

In object-oriented analysis, classes are first modeled with a *class diagram*. The class diagram, which is another UML tool, shows all the classes involved in a use case.

3. Identify the necessary object relationships and interactions for the system.

The next step is to identify the relationships that exist between and among the classes. You identify these relationships by looking for the natural associations that exist between the objects. Once identified, you can add the appropriate relationship to the class diagram.

4. Understand that there is no "right" solution to these tasks.

For each problem, there may be many solutions. The role of the analyst is to understand the problem and design a system that efficiently solves it or improves the existing process.

Object-Oriented Design

During object-oriented design, the analyst determines how the objects and system requirements identified in the analysis phase will be implemented. It usually involves the following activities:

1. Determine the correct hierarchical relationship between objects in the system.

In this step, the analyst determines how to implement the object relationships identified in the analysis phase, and verifies that those relationships are correct. If the relationships are not correct, the models produced during the analysis phase are modified.

2. Determine the correct ownership of the attributes and behaviors in the individual object-to-object interactions.

When relationships exist between classes, especially the inheritance relationship, a “spectrum” of classes emerges. The analyst must carefully consider which class in the spectrum is the appropriate owner of each attribute and method.

3. Implement and test the object/system interaction.

Finally, the analyst (or a programmer/developer) implements the design using an object-oriented programming language. The language must support the features specified in the object-oriented design. In addition, the system’s interactions with the operating system and the user interface are implemented. The system is then tested and further refined.

Design Issues

The analyst must address many reusability and design issues that impact the complexity and efficiency of an object-oriented system. Among them are the following:

- Are the classes too specific in design to be used generically, or too general in design to be used without modification or extension?
- To change a class’s capabilities, should the class be modified, or should a child class be designed as a specialization? If the class is to be modified, will the changes create unwanted side effects because of other relationships that exist?
- Is the definition of a class such that objects of the class can stand on their own in multiple scenarios?
- Is there an overuse of inheritance or specialization? Inheritance can be taken to an extreme, where one class is the child of another class, which is the child of another class, and so on. Excessive generalization/specialization relationships might be necessary in some cases, but they can result in overly complex systems.
- Does it make sense to implement an entity as an object? Are there other data structures that make more sense?