# What Is An Object?


This column originally appeared in the June/July '96 issue of Dr. Dobb's Sourcebook

Bjarne Stroustrup once characterized OO programming as "Buzzword-oriented programming", and certainly one of the most abused (or at least misunderstood) buzzwords in the pack is “object” itself. Since the idea of an object is so central, a full discussion of what exactly an “object” is seems in order.

First and foremost, an object is a collection of capabilities. In practical terms, this means that an object is a collection of functions that implement the capabilities, but the emphasis must be on what an object can do—what capabilities does it have—not on how those capabilities are implemented. Of course, most objects will require some data in order to implement their capabilities, but the make up of that data is—or at least should be—irrelevant. The capabilities are exercised by sending the object a “message” that asks the object to do something. You shouldn’t care what fields are inside the class definition. More to the point, you should be able to radically change what’s inside the class definition, and as long as the interface (the message handlers) isn’t affected, the users of the class will be unaffected. Let’s look at a few examples.

Many books use an ATM machine to demonstrate OO concepts, not because any of us will be implementing ATM’s, but because an ATM is a good analog for both OO and client/server architectures. Look at the central bank computer as a server object and an ATM as a client object. Most procedural database programmers would look at the server as a repository of data and the client as a requester of the data. Such a programmer might approach the problem of an ATM transaction as follows:

1. The user walks up to a machine, inserts the card and punches in a PIN.
2. The ATM then formulates a query of the form "give be the PIN associated with this card," and then verifies that it matches the one provided by our user. The ATM sends the PIN to the server as a string, but the returned number is stored in a 16-bit int to make the comparison easier.
3. The user then requests a withdrawal.
4. The ATM formulates another query, this time: "give me the account balance." It then stores the resulting balance in a 32-bit float.
5. If the balance is large enough, the machine would dispense some cash, and then tell the ATM to "update the balance" to the original amount less the amount of the withdrawal.

(This isn’t how real ATM machines work, by the way.) So what's wrong with this picture? A 32-bit float can represent at most 2 gigabucks (assuming no cents and one sign bit). What happens when Bill Gates walks into our bank wanting to open an account? Last time I looked he was worth about 12 gigabucks, and Bill has no intention of giving a penny to charity any time soon, so that won’t help. Similarly, the 16-bit int used for the PIN can hold at most 4 decimal digits. What if you need to change to a 5-digit PIN? The only recourse is to change the ROMs in every ATM in the world (since there's no telling which one Bill will use) to use 64-bit doubles instead of 32-bit floats to hold account balances and to 32-bit longs to hold 5-digit PINs. That’s an enormous maintenance problem, of course.

A similar situation exists in most procedural programs. Change the database schema or the definitions of a few central data types, and virtually every subroutine in the program might have to be rewritten. It’s exactly this sort of maintenance nightmare that OO hopes to solve. To see how, let’s recast the earlier problem in an object-oriented way. Look at the central server as an object that supports two capabilities: verifying a PIN and authorizing a transaction. The handshake now looks like this:

1. The ATM gets the PIN and then asks the server to verify it. That is, it sends the message: "is this the correct PIN for this card?"
2. The server comes back with "yes" or "no."
3. The user then request a withdrawal, and the ATM asks the server to authorize the withdrawal by sending a "Can this person withdraw this amount? Message to the server"
4. The server, again, comes back with "yes" or "no."

What's really going on here is that the server is now an object that implements the "authorization" capability. Rather than requesting the data that we need to authorize a transaction, we ask the "authorizer" (which has the data) to do the work for us. No data (account balances or PINs) is being shipped to the ATM, so there's no need to change the ATM when the server code changes. (Data can be sent to the authorizer object as strings or encapsulated in other classes of objects if absolutely necessary.) The maintenance manager is happily sleeping in the back office instead of running around changing ROMs.

A second example, which I'll discuss in a moment, highlights one of the biggest mistakes made by C programmers when they move from C to an OO language like C++ or Java. C++, in particular, is really just C with a few features added to make it easier to implement an object-oriented design. I, for one, think that there's absolutely no reason to use C++ unless you are indeed implementing an OO design—-the language has so many problems that’s it's not worth messing with it if you aren’t going to take advantage of its strengths. You’re better off just using C. On the other hand, if you are doing an OO design, a language designed to implement OO systems (like C++, or better yet, Java) can make the implementation dramatically easier. Many C programmers try to program in C++ as if they were programming in C, however, implementing procedural systems in C++ rather than OO systems. This practice tends to produce awful code. Java mitigates the situatation a bit because it's more of a "pure" OO language. It's harder, though not impossible, to abuse. A determined inidividual can write garbage code in any language.

One of the most common problems is a misinformed attempt to turn an existing struct definition into a class by making the data private and then providing a “get” and “set” function for each field. This is not object-orientation — it's just a very complicated way to access a field that could be more easily accessed with a dot or arrow. In an OO system, the data wouldn’t be accessed at all by non-members of the class. Rather, messages sent to an object would request that the object exercise some capability, and the message handler might use the data members of the class to do its job. Remember, structs and classes are very different at the design level. A struct is a collection of data, a class is a collection of capabilities.

I saw a good example of this problem is a human-resources package that the programmers thought they were moving to OO. They took and existing "employee record" and tried to

transmogrify it into an "employee" class by making the fields private and providing get and set functions. One problem is that an employee class will almost certainly have a “salary” attribute, and unfortunately, the original employee record had a “salary” field, so the programmers equated to the two. An attribute is not the same as a field, however. An attribute is a defining characteristic of an object. A salary serves to differentiate an employee from a generic “person,” for example. Without the salary, there would be no difference between an employee and a person. Moreover, an attribute does not necessarily map to a field in the class. The salary might be stored in an external database, for example, with the employee storing only the information needed to retrieve the attribute from the database. If the employee does store the salary internally, it might be stored in a float, a double, a binary-coded-decimal array, a string — there's no telling. What, then, could a get-salary function return? One of the main tennents of OO design is that it should be possible to radically modify the private components of a class definition without affecting the users of the class at all. The salary field might be a float today, but there's no guarantee that it will stay that way. Similarly, the matching get-salary function might return a float today, but what if the internal representation changed in such a way that a float wouldn’t work anymore? Say, for example, you needed to return an object of type "money" which worked like a float but handled the round-off-error-on-pennies problem. Letting the function continue to return a float while using the money class internally would defeat the purpose of the money class. Changing the function’s return value would break every subroutine that called the function. It's exactly this rippling effect of a change that OO techniques are meant to avoid.

What the programmers should have done is decided what capabilities that an employee has — what it can do, not what fields it has. Put another way, you should never access an attribute directly, rather you should ask the object to do something with its attributes for you. In the case of an employee, there should be no message of the form "give me your salary." You should not extract the attribute from an object in order to do something with it (like draw it in a window). Rather, you should tell the object itself to do the operation ("print your salary in this window"). The same reasoning would apply to a name: not, “give me your name,” but rather “draw your name here.” This way, the way that the attribute is stored inside the object is utterly irrelevant. As long as the salary gets printed, I don’t care how it's stored. Though the foregoing is the preferred way of doing things, you do sometimes need to extract an attribute from an object. The salary might be needed by a payroll-department class to generate a paycheck, for example, and it might not be appropriate for an "employee" object to control the amount of it's own paycheck. There are several solutions to this problem, probably the best would be for the payroll-department class to ask the employee for its salary. The salary should be encapsulated in a "salary" object, and the "salary" object should implement all the capabilities needed to compute a paycheck (such as "divide yourself by," "subtract this from yourself," and of course, "draw yourself in this window"). This way, the representation of the salary is still hidden. The returned salary object should, of course, be a constant. You don’t want a payroll-department to be able to modify a salary, and you don’t want the exposed salary to be different from the one stored in the employee.

A final example of this structure is a "string" class, which should expose no information about how the characters are stored internally. There should be no “get buffer” function in your string class, for example. In C++, you should have no conversion from string to char* or equivalent. All of the C functions that work on strings should be members of the string class. Not: "give me your buffer so I can print it," but "print yourself." Not: "give me your buffer so I can use strcmp() to compare you with another string" but "compare yourself against this other string." This way you are completely isolated from the way that the string stores

characters internally. A character could be stored in a char, a wchar_t, a multi-byte character string, or something that nobody's thought of yet—it doesn't matter. In the case of compare-yourself, the string could even adjust the comparison algorithms to reflect the language being used (i.e. French, English, Japanese).

This structure implies, of course, that there are no functions in the system that take char* arguments—all strings must be "string" objects. The only place that char* would appear is an argument to a string-class constructor. And that’s another aspect of OO systems—they tend to be all-or-nothing propositions. Everything must be a string object, there's no way to mix string objects with arrays of char and get the maintenance advantage promised by object orientation.

The point of all this is that you can now make changes to the structure of an object without affecting any of the code that uses the object — and that's one of the main strengths of an object- oriented approach. Changes made in one place are highly localized (as are bugs). This localization not only makes code easier to maintain, but also easier to debug and to write initially.