Dan Shafer on LiveCode and OOP, Part I

Dan Shafer returns to revJournal with an exclusive two-part article comparing traditional Object-Oriented Programming with Object-Like Programming in LiveCode. In Part I, Dan explores and explains the basics of traditional OOP. Don't miss it!

Publisher's Note: *As many of you are aware, Dan Shafer has launched his own online LiveCode resource at LiveCode Pros, and has just published Volume One of his three-volume LiveCode masterwork, "LiveCode: Software at the Speed of Thought." Here, however, is a* ***revJournal*** *exclusive: the following two-part series is based on material that didn't make it into Dan's sensational books. Think of them as deleted scenes and outtakes that didn't, for whatever reasons, make it into Dan's final cut. Read and enjoy ...*

Overview

In this two-part article, you will learn:

- the important role being played in software development by concepts grouped under the rubric "Object-Oriented Programming" (OOP)
- how Runtime LiveCode parallels some of those concepts and goes its own way in others
- how an understanding of OOP can help you be a better LiveCode programmer

Runtime LiveCode: Object-Like Programming

Let me be clear at the outset of this discussion: LiveCode is not object-oriented programming in the "traditional" sense. I know it lacks some essential features of true OOP. But my interest in this article is how closely some key ideas in Runtime LiveCode resemble those in OOP. The objective is to see what the world of OOP has to offer would-be LiveCode gurus.

To describe LiveCode and its somewhat loose ties to OOP, I have adopted the phrase Object-Like Programming. LiveCode is object-like in that it uses some of the same terminology, adopts some of the same methods and adapts others, and in many ways looks and "feels" like OOP.

Before you can appreciate the validity and utility of all this, though, you need a basic understanding of OOP. The next section presents some of the fundamental ideas in OOP, but it is not an exhaustive treatment of the subject. If you are already familiar with OOP, you might want to skim or skip the next section.

OOP Fundamentals

Object-oriented programming is a way of looking at programming tasks that differs from the traditional approach. In procedural programming with C and other similar languages, you describe functions and procedures that operate on certain types of data. The data is separate from the functions that operate on it. In OOP, data and procedures that operate on the data are together, packaged in something called an object.

There are five central ideas in OOP: objects, messages, methods, classes, and inheritance. Although we explain each of these ideas briefly, they are so intertwined that an understanding of each depends on an understanding of the others.

What are objects?

Viewed abstractly, an object is a single programming entity that combines data with the procedures or functions that operate on that data. Viewed from a programming standpoint, objects are the elements of an OOP system that send and receive messages. I discuss messages in greater detail in the next section.

If you write a procedure to invert something in Pascal, you have to know in advance what kind of data the procedure will operate on. For example, inverting text might mean changing it from black letters on a white background to white letters on a black background. Inverting a matrix, however, is a complex mathematical operation unrelated to text color display. Similarly, inverting a graphic object like a pyramid is different from inverting text or numeric matrices. If you want a program to be able to invert any of these types of data, you would write a separately named procedure for each type of data, check in your program for the type of data to be manipulated, and then call the appropriate procedure. This process is depicted in Figure 2-1.

```
PROCEDURE invertjext
PROCEDURE invert_pyramid
Some text to be inverted
PROCEDURE invert matrix
I 9 8 4 3 87 2.04
II 240.1 009.0 -9.0031.25158
PROGRAM do_something CASE datajype of
text
invert_text
matrix
```

```
invert_matrix
pyramid
invert_pyramid
```

Figure 2-1. **Data, procedures, and programs are separate in traditional programming**

Object-oriented programming, however, permits the designer to say, in essence, "I want to invert whatever object I?m working with, so I'll just use an invert command and let the system take care of the problem." In OOP parlance, this command is referred to as a message. An object called, for example, matrix receives a message called invert and carries out its own processing in response to the message. There are still three separate invert routines in the program, but the part of the program that inverts an object doesn't need to be aware of them. This situation is represented in Figure 2-2.

You can probably see at least one big advantage to the OOP approach. If you want to add a new type of object to a procedural language application?for example, a list of items where invert means "reverse the order of"?you have to define a new procedure and add a new case to the main program. In other words, everything will change. Thanks to the well-known ripple effect, the consequences of this could take a long time to resolve if a bug gets introduced in the process. In an OOP world, though, you simply create a new object and add to it the ability to invert itself. Any other objects that send this new object an invert message do not need to be modified. The change is isolated and, therefore, manageable.

```
Object:Matrlx
I 9 8 4 3 87 2.04
II 240.1 009.0 -9.0031.25158
Method: Invert Method: Other(s)
invert message
Object: Pyramid
Method: Invert Method: Other(s)
Object: ProceMor-1
Object: Text
Some text to be inverted
Method: Invert Method: Other(s)
invert message
Need to invert an object
invert message
```

Figure 2-2. **Data, messages, and methods in OOP**

What are messages?

I have spoken glibly of messages as if it were obvious what they are and what they do. Although they may constitute a new programming idea, the concept of messages is not novel. When you call a friend across town and he or she answers the phone, you are sending messages. When you mail a letter to someone and the person on the other end opens it and reads it, you're sending messages. In fact, you probably do most of your work by sending messages of one kind or another to other people and to machinery or computers.

A message in the OOP world corresponds to a procedure or function call in a procedural language. Everything in OOP is accomplished only one way: One object sends a message to another object and the receiving object reacts. There are no alternate ways to get things done. The simple elegance of this model makes programs written for an OOP environment potentially easy to understand.

What are methods?

The idea of a method is the easiest of the five basic OOP concepts to explain. A method corresponds almost exactly to a function or a procedure. A method is the code contained in an object that tells the object how to react when it receives a message with the same name as the method.

In the previous example, each type of object had an invert method. When an object receives an invert message from another object, it simply carries out the instructions in that method. Sometimes, it sends a message back to the sending object indicating it has completed the instruction. Other times, it might trigger another method in another object, perhaps even a different invert method in another object, to accomplish its goals. To do so, it sends a message to that object.

If a message is sent to an object that does not have a method of that name defined, the system generally handles the problem with an error message such as "Object pyramid doesn't understand how to invert." Objects in OOP can be designed to understand and respond to any theoretically large number of methods.

What are classes?

Groups of objects with sets of common characteristics are called classes. The most important thing objects in a class have in common is the way they react to messages. If we had to write the same method for every single object, some of the advantages of OOP discussed in the next

section would be lost in a mass of code. But if we define a class, each object we create as an instance, or example, of that class will already know how to behave in response to the messages the class contains.
The class, like everything else in an object-oriented world, is itself an object. Individual objects are referred to as instances of a class. Because a class is also an object, an object can be both a class and an instance of another class. The concept that makes classes significant is the fifth central OOP idea: inheritance.

What is inheritance?

In a true object-oriented world, objects inherit behavior from their ancestors in an ever-expanding and descending chain of heredity. All objects in OOP have at least one ancestor. The closer an object is to the root object class from which all other objects and classes spring, the fewer ancestors it has. This structure resembles an outline or a classification scheme.
Figure 2-3 depicts a classification structure for a class called Furniture. As you can see, this class has subclasses called Seating, Table, and Lamp. The subclass Seating, in turn, has other subclasses, and so on.
The key idea in inheritance is that if the class Furniture has a method called, for example, movelt, every member of every subclass can use that method in the same way without any special code in the subclass. In other words, if you send the message movelt to a love seat, the love seat object need not have a method called movelt. It simply passes the message up the hierarchy to its immediate ancestor (in this case, Sofa), which reacts if it has a method named movelt or passes the message on if it doesn't have a corresponding method.

Class: Furniture

Class: Seating

Class: Chair

Instance: Instance: Instance: Class: Stool Instance:
Instance: Instance:
Class: Sofa Instance: Instance: Instance:

Class: Table

Class: Lamp

armchair easy chair secretarial chair
milking stool
bar stool
snack counter stool
sofa bed sofa love seat

***Figure 2-3.* Typical classification scheme**

Generally, you can ?override? a method defined in a class so that an individual instance can react differently to that message. If you find yourself doing this too often, the method may be one that isn't really a good one around which to build a class.

Object-oriented programming summary

Let's see if we can summarize this sketchy look at object-oriented programming. Everything in an OOP environment is an object. Each object (except the one central object from which all others are descended) has at least one ancestor. An object inherits methods from all its ancestors in the chain that tell it how to respond to messages. Everything in an OOP environment is accomplished by objects sending messages to other objects.

It really is just that simple.

Why Object-Oriented Programming?

Why has OOP become such an important idea in the past few years? It really seems to be just a new way of looking at programs and data. So what's all the excitement about?

The characteristics inherent in OOP create numerous advantages in soft-ware design and development. Let's take a look at the three main ones often singled out by proponents of the OOP approach to computer programming. These are:

- the natural "feel" of the OOP model of the problem
- the high degree of code reusability
- the ease of maintenance and modification

OOP is "natural"

The world in which we live is composed of objects. And as we saw earlier, we accomplish much of what we do by sending messages to other objects in our world and reacting to their messages. Furthermore, we generally do things by telling other objects what we want done rather than by explaining in great detail how to do it. The how describes the procedure and is part of the proce-dural programming model. The what describes the task, the problem, and its solution in descriptive, or declarative, terms, and is part of the declarative programming model of which OOP is a prime example. When you give your computer a print message, you don't tell it, "Now I want you to take this document that I've just finished

creating and analyze its bit map structure. Got it?" You just tell it to print and expect its behavior to follow.
Similarly, if you give an assignment to a subordinate, you generally say, "I need the quarterly objectives report on my desk by 3:00, Jim." You don't say, "Jim, I want you to sit down at your desk. Take out a piece of paper and a pencil. Now, put at the top of the paper...."
But these descriptions ? simplified for illustration ? are good summaries of the differences between procedural programming and OOP. The world just doesn't work procedurally. Consequently, it is much easier to write programs designed to emulate or simulate reality and intelligence in OOP environments than in more procedure-oriented environments.

Code reusability

If you can define one object that is usable in several different systems, you can move it from one system to another with great ease in an OOP environment. There is nothing new to declare in the second system, no data structures to worry about, no other objects or procedures to modify. Simply pick up the object from program A and plop it down in program B and run it.
Consequently, if a programmer is proficient in and comfortable with OOP design concepts, he or she spends a great deal of time building reusable tools and objects. After that, a large percentage of programming time is spent simply assembling the appropriate objects into new "worlds," or systems. This means OO programmers tend to spend comparatively little time reinventing wheels.

Ease of maintenance

As we saw earlier, the ripple effect that causes so many software maintenance headaches all but disappears in an OOP environment. If the object behaves in a certain way in system A, it is guaranteed to work the same way in system B. Debugging is effectively (though not totally) reduced to finding messages sent to inappropriate objects, messages sent with the wrong number of arguments, and missing or undefined objects and methods.
Join us next week for Part II of this article, in which we'll consider OOP in the context of what it all means to you, the LiveCode developer.

In Part I of this article, we covered the basics of traditional OOP, and took a look at what it's all about and why it's favored by many professional programmers. This week, we'll consider OOP in the context of LiveCode's object-based programming model...

OOP and LiveCode

So what does all of this have to do with LiveCode and LiveCode? After all, we've already pointed out that LiveCode is not an object-oriented program-ming language.

There are some strong parallels between LiveCode and true OOP systems, though, and these parallels are neither accidental nor incidental. Although the parallels are not exact and don't hold up throughout the architecture of LiveCode, they are interesting and important enough to merit our attention. My hope is that by seeing the aspects of design and programming that LiveCode and more traditional OOP languages have in common, you will see how to take advantage of OOP concepts in designing stacks.

Objects in LiveCode

There are 12 types of objects in LiveCode: stacks, cards, buttons, fields, scrollbars, images, graphics, players, EPS (Encapsulated Postscript), audio clips, video clips, and groups. Like OOP objects, each of these can send and receive messages. Each type of object can be associated with a script that contains handlers, which correspond to methods (as we'll see in a moment). So the object and the program code that enables it to respond in a specific and predictable way to a message are packaged together, exactly like objects in an OOP environment.

Messages in LiveCode

The parallel between OOP systems and LiveCode continues when we examine the subject of messages. LiveCode uses the same term to describe the communications that take place between objects. LiveCode includes system messages that are sent as a result of events triggered by stack users. Each type of message can be addressed to one or more of the types of objects encompassed by LiveCode.

When an event takes place, a system message is generated and sent to the object of which the event is the target. That object reacts as called for in the handlers contained in its script. The parallel with OOP is quite strong.

It is important to differentiate system messages, which are sent in response to user actions, from built-in commands and functions in LiveCode. The latter are always handled directly by LiveCode. In fact, you cannot intercept these built-in messages or change their behavior in any way. For example, the built-in command add that you use to add numbers is always handled the same way and only by LiveCode. You cannot put a handler or function called add in your application and expect to treat the add operation differently.

Methods in LiveCode

As we have pointed out, each type of object in the LiveCode hierarchy can have a script associated with it. In each script there can be one or more handlers. These handlers correspond closely to OOP methods. A handler is associated with each type of message the object can receive. There are two types of handlers in LiveCode scripts: function handlers and event handlers. The latter derive their name from the fact that they are typically triggered by an event, as described in the preceding section. All LiveCode objects also have properties. Proper-ties are an important concept; they are discussed in depth in Chapter 18. Some properties bear a close resemblance to methods as well. For example, a button can have a property of being automatically highlighted when it is pressed. This is a character trait, or behavior, of the object, and so it corresponds at least roughly to a method. LiveCode objects come with a set of built-in properties and you can add custom properties as needed to add characteristics to those objects.

Classes in LiveCode

There is no strong analog in LiveCode to OOP's concept of classes. The hierarchical form of inheritance (see the next section) used in LiveCode is not precisely parallel to that of object-oriented programming, due in part to the lack of classes for objects. For example, there is no class called a button class to which all buttons belong and which has individual instances of buttons. Although there is some commonality of behavior among buttons ? they all, for example, cause something to happen when they are pressed ? there is really no classification scheme resembling OOP classes.

The concept of groups in LiveCode bears some semblance of connection to the notion of an object in OOP, when groups are set up in such a way that they appear on multiple cards and both appear and behave identically on all of those cards. Creating a new card which contains one or more groups will, under the proper circumstances, place those groups on the newly created cards. The group object is not replicated, but rather exists in such a way that its script is executed regardless of the card on which it appears.

But groups are not, strictly speaking, classes. They simply exhibit some of the characteristics of classes in an OOP environment.

Inheritance in LiveCode

There is no true inheritance in LiveCode. Messages pass through a definite hierarchy (see Figure 2-4), and this hierarchy has some of the

characteristics of OOP inheritance structures, but the analogy is less complete when it comes to inheritance than on any other point.
The hierarchy in Figure 2-4 extends up from the specific object where the action takes place that triggers the event, to the card, group(s), and stack, then to the Home stack, and finally to LiveCode itself.
A mouseUp message that originates with the press of the mouse on a button gets passed up the hierarchy until one of two things happens: A handler named mouseUp is encountered and executed or the top of the hierarchy is reached with no handler having intercepted and acted on the message. This behavior is quite similar to the message-processing approach of OOP systems.
But the opposite is not true. In other words, just because a particular button on a card has the ability to respond to a specific type of message does not mean that all other buttons have the same capability. The same can be said of groups, cards, and stacks. If you create a new card, it may has the same group as the currently visible card depending on a specific property of the group. If the group has a handler for a particular system message in its script, the new card also has that same handler.
But this is not inheritance, because the new card is not a descendant of the original; both are on the same level of the hierarchy.
You can also modify the inheritance path dynamically in your scripts. You can place stacks and the scripts associated with specific objects in the current and other stacks into the message hierarchy between your stack and the Home stack. This makes it possible for you to redefine at least a significant part of the hierarchy and to create libraries of scripts that can be shared among other stacks only when they are needed. This type of reuse, which strongly resembles inheritance in OOP, can be a major design win when you build applications using LiveCode.

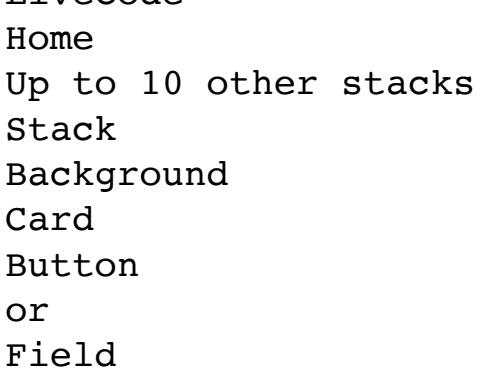

Figure 2-4. LiveCode hierarchy of inheritance

How OOP thinking helps in stack design

You can see why we said at the outset of this discussion that LiveCode is not a true OOP language, but shares enough with that approach to software design to merit consider-ation of the parallels.

We mentioned that code reusability is a major advantage of OOP systems. Because of the absence of true inheritance, that advantage does not accrue to LiveCode. Thanks to cut-and-paste editing power, you can easily copy scripts and handlers from one object to another object of the same type (or even a different type). But this manual process, no matter how facile, hardly qualifies as inheritance of behavior from object to descendant object.

On the other hand, the isolated nature of a handler and its ease of modification mean that maintaining scripts is much easier than modifying and managing traditional procedural programs. If the handler works in response to the message it handles in one card or button, it will work correctly in another place. Similarly, if a script has more than one handler, even if they interact, the functionality is isolated to a sufficient degree that software maintenance is quite straightforward.

Finally, and perhaps most significantly to LiveCode programmers, the language does a remarkably good job of emulating the world of which it is a model: that of a computer application. It makes working with the complex world of objects much simpler, more readable, and more enjoyable.

Partly because of its strong object influence and partly because of the nature of the computer world that lends itself well to such object emulation, LiveCode removes many of the barriers between programmers and elegant, usable, GUI-based applications.

Dan Shafer is the LiveCode guru behind the LiveCode Pros website, and the author of the 3-volume official third-party LiveCode book, LiveCode: Software at the Speed of Thought.