

CONTROL



Issue Date: June 2004, Posted On: 6/10/2004

The Straight Scoop on OOP

By Matt Scott

Object-Oriented Programming (OOP) is a programming technique that lends itself to structured and modular software code. Modular coding can be implemented with virtually any programming language from ladder to function block, but OOP was designed from the ground up with modules in mind.

This means that OOP tools and programming languages naturally lead to discrete modules or blocks of code linked together with rules that are part of OOP programming languages.

The flip side to modular programming is simple, monolithic or "spaghetti" style programming. These types of programs don't use modules but rather have all code in one continuous string. Monolithic programming skips advanced programming techniques and goes straight for the quick fix. This often means programmers are writing code without first thinking about the overall structure of the program. This may appear an expeditious way to write a program, but problems often result.

A program written without a pre-defined structure can become very complex, and even incomprehensible to all except the original programmer. Even the original programmer can become confused as the program grows more complex or when the program needs to be revisited after a period of time.

By contrast, structured and modular programming requires thought and planning prior to coding. If structure and modules are desired, then OOP is an ideal tool.

Modular coding in general and OOP in particular offer many advantages over monolithic code (see Table 1) including ease of test, reuse of code and portability to different hardware target platforms.

Table 1

REASONS TO USE OOP
1. Complex code can easily re-used by other programmers.
2. Code is easy to reconfigure, change and extend.
3. Develops problem conceptualizing skills.
4. OOP uses less memory and CPU than ladder logic style programming.

OOP was developed for good reasons, and under the right circumstances it can produce excellent results and do things that are virtually impossible *sans* OOP. With OOP, a skilled programmer can program an application in an organized manner.

It only takes one skilled programmer to properly develop an OOP framework. That framework can then be used by other programmers. The beauty of OOP lies in its ability to manage complex tasks, its flexibility, and its ability to make efficient use of hardware.

Inside OOP

OOP is a high-level extension of the modular code concept. By building cleverly thought out, reusable modules of code, repetitive programming is avoided. Code also becomes more organized and readable

(see Figure 1).

Individual modules of code, or classes, can "inherit" each other's contents. A child class inherits the contents of a parent class. Inherited contents can be modified by adding new code or improving old code. For example, a parent class can be written to allow a PC to exchange data with a particular brand and model PLC.

A child class can also be created for a different PLC, and with the child inheriting the parent's common functionality while still providing a programmer with a way to add hardware specific details.

An object is an instance of a class, and it contains data and code that operates on that data. Multiple objects or modules are linked together to build programs.

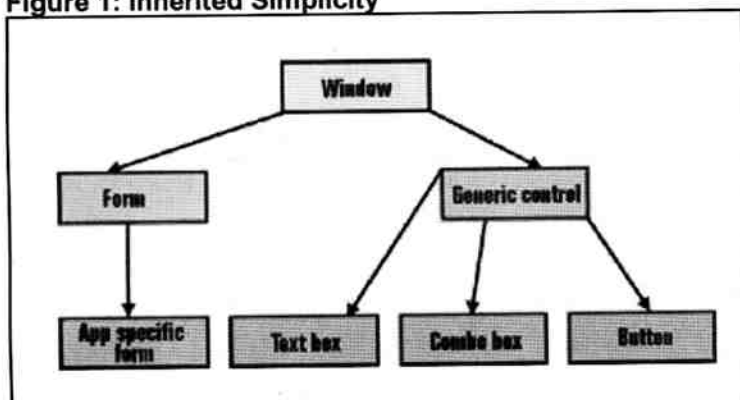
Objects can be linked together in a tree structure by having them contain each other's memory addresses. Those addresses can then be followed from object to object. This functionality is called traversal. As the program traverses, each object is accessed and the program gathers information about that object or provides services such as settings management to that object.

For example, the traversal function can be used to build a test and validation document for the entire program by traversing a tree of objects that contain the file name of a test and validation document that describes the object and its requirements. The traversal function collects all test and validation information together in an organized fashion.

By traversing a similar tree of objects, a programmer can gather all of the pertinent information describing those objects. This information could then be used to build a new program with a similar structure on another machine, a technique called serialization.

An entire Graphical User Interface (GUI) page can be transported from one hardware platform to another by serializing an object that contains a description of the GUI page.

Figure 1: Inherited Simplicity



This diagram is a sketch of the inheritance lineage which is an example of a simplified GUI class hierarchy.

When to OOP

OOP does take more work up front than monolithic code, so OOP should only be applied when dictated by circumstances.

OOP systems are particularly useful when all the eventual uses and applications of a program can not be known in advance. Some OOP languages are also well-suited for situations where a software program is needed to control a variety of hardware. For example, a corporate-level software engineer is responsible for implementing an advanced control algorithm across several plants. These facilities contain a variety of software operating system and hardware platforms. OOP's inherent flexibility would allow the software engineer to create a base program that could then be easily modified for each platform.

Another situation where OOP excels is when equipment or processes need to be connected together in unforeseen ways. Each module can be written to control a particular process, and these modules can then be linked together as needed.

Building with modular connectable blocks lets one swap only the required modules out if and when a process changes. Existing blocks from old projects can also be used to program new projects quickly.

If a module is structured correctly the first time, a permanent solution is created that can be used again and again. If mistakes are found in a module or if a module is improved, it is a simple task to update the module in all of the programs in which that module exists.

S88 recipe management techniques use many of the same terms and embody many of the same principles as OOP, so OOP is particularly well-suited to S88 implementations.

Although S88 was originally conceived for batch processes, Procter & Gamble and other large processors are now using S88 concepts for continuous process control.

According to David Chappell, a technology leader with Procter & Gamble in West Chester, Ohio, his company has successfully created several hybrid (batch & continuous) process control applications. "These hybrid systems benefited greatly from the modular approach described in the S88.00.01 standard," says Chappell.

OOP for GUIs

At Deseret Labs, GUIs are developed using Microsoft's MFC. MFC is a set of classes (or a library) designed to implement GUIs. It is provided when one purchases Microsoft's Visual C++. Text editors, buttons, and other GUI elements are part of MFC. Such libraries are often contained in programming language software packages.

GUI elements contained in MFC are referred to as controls and ActiveX controls by Microsoft. These elements are user interactive object entities that a user can click on to simulate operation of a knob, button, or dial.

When creating GUIs, it is important to manage the position, size, and other attributes of all its elements using the same routines. Otherwise, separate routines have to be written to manage each GUI element. The OOP inheritance scheme is perfect for this task.

With Visual C++, GUI elements are derived from (children of) CWindow. CWindow (part of the MFC library) is a core window functionality class that contains the generic position, sizing and other functionalities inherited by nearly all of the MFC classes. See Figure 1 for a diagram of the inheritance lineage of a simplified GUI class hierarchy.

GUIs generally operate with what is called event-oriented programming. That means that there is a never-ending loop which scans the system for events like a mouse click or a key stroke. The loop then hands off information about the events (as well as CPU control) to event recipients.

With Visual Basic (VB), if one object receives the CPU control from the event loop and doesn't give it back, the entire application freezes. With VB, it is also possible for a particular window to grab the event loop for itself and stop the rest of the application. VB.NET and other OOP languages such as Visual C++ also can solve this event loop problem.

With Visual C++, the event loop/GUI architecture is thread-safe. A thread is a collection of code, like an event loop, that executes simultaneously with other threads. More precisely, a thread is a collection of data managed by the operating system.

The operating system examines data as it schedules tasks for CPU time. The operating system will set up the CPU, begin executing code, stop and come back later to do more.

Because separate threads access the same data space, they can trip over each other by trying to change the same data simultaneously. Avoiding this mishap requires

synchronization, which means that threads block each other from accessing resources. Synchronized code becomes thread-safe and can tolerate multiple simultaneous threads without crashing.

Visual C++ programs can therefore operate multiple GUI windows without having them block each other while effortlessly managing I/O access at the same time. The cost for these advantages is elevated programming effort required by OOP in general and C++ in particular.

Control Algorithms

Control algorithms can be easily programmed as objects with OOP. Keeping algorithms generic is a key concept, because generic control algorithms are reusable where specific algorithms are not.

Many OOP libraries support advanced control concepts like fuzzy logic and neural networks. Fuzzy logic algorithms can be very good for managing extreme complexity. Neural networks are prized for their ability to learn what to do with just a few training examples. Fuzzy-neural systems combine the two.

Finite state machines are a common example of OOP control algorithms. If a process machine has several different behaviors rather than just one, a finite state machine keeps track of which behavior is currently running with a state variable. Example states might be drying, stirring, and heating. Figure 2 diagrams part of the code for a finite state machine used to control a simple dryer/tumbler.

Events such as button pushes or reaching an end-point can trigger a change of state. When the state changes, different code executes and the machinery behaves differently. Finite state machines are very debuggable, easy to validate, inherently well organized and stable.

Matt Scott is the engineering team leader for Desert Laboratories in St. George, Utah. He holds a composite masters degree in technical computer science/physics from Humboldt University of Berlin. Scott can be reached at matts@deseretlabs.com.