OBJECT-ORIENTED COMMUNICATIONS

L. J. Chapman
Fermi National Accelerator Laboratory *
P.O. Box 500
Batavia, IL 60510
MS 307

Abstract

OOC is a high-level communications protocol based on the object-oriented paradigm. OOC's syntax, semantics, and pragmatics balance simplicity and expressivity for controls environments. While natural languages are too complex, computer protocols are often insufficiently expressive. An object-oriented communications philosophy provides a base for building the necessary high-level communications primitives like "I don't understand" and "the current value of X is K." OOC is sufficiently flexible to express data aquisition, control requests, alarm messages, and error messages in a straightforward generic way. It can be used in networks, for inter-task communication, and even for intra-task communication.

Overview

Fermilab's control system's existing protocol for microprocessor communication uses a variable-oriented communications philosophy (i.e. communication is reading and writing each other's variables). OOC's philosophy is that communication is objects sending each other messages and replying to those messages. This object-oriented paradigm is used in the programming languages Smalltalk, C++, Common LISP, and others. This paper presents an informal introduction to the paradigm and OOC's use of it.

Objects and Classes

Objects are software entities within processors. A typical OOC object might correspond to an accelerator device such as a vacuum pump. Such an object responds to standard messages like READ, SET, and TURNON, as well as to application-specific messages. When an object receives a message, it executes some code called a method. Different kinds ("classes") of objects execute different methods for the same messages. For example, the code that turns on a vacuum pump might be very different from the code that turns on a power supply, but in both cases objects are responding to TURNON messages. The sender of the message cares only about turning on, not about the details of how that is accomplished.

Every object has a class. The methods are associated with the class (i.e. all vacuum pumps do SET commands the same way). Every class has a superclass, from which it "inherits" methods. A new kind of vacuum pump which differs from the standard vacuum pump only in its SET method can easily be implemented as a subclass of the standard vacuum pump class. This new subclass would need a new method for SET messages but would inherit all other methods from its superclass, its superclass's superclass, and so on up the inheritance hierarchy.

Every object contains its own private variables, called instance variables. The structure of these variables is determined by the class (e.g. all vacuum pumps have readings, settings, etc.). Each object has its own values for the variables (e.g. vacuum pump #17 is currently set at 27.4). Just as methods are inherited from superclasses, so are instance variable structures.

Objects are used not just for accelerator devices such as vacuum pumps. They are also used to represent more abstract entities such as PID loops, finite state machines, and classes. For example, the vacuum pump class is itself an object. The instance variables for a class object include, among other things, the methods for the class! This tricky idea is typical of the way this paradigm uses abstract concepts in powerful and generic ways.

All objects are created dynamically. This is done by sending a CREATE command to the appropriate class object. Thus it is easy to create, say, a new PID loop "on the fly." Objects can be created by an operator typing a CREATE message, by other objects sending a CREATE message, by any task, or by OOC itself.

Usually objects are passive in that they only act on receipt of a message. OOC also provides "volitional" objects which contain a task, and therefore can act on their own. For example, a PID loop object might run periodically, sending a READ message to its input object, transforming the resulting value, and then sending the new value to its output object in a SET command.

Addressing Objects

Objects can be addressed in several ways by an "object descriptor." Much flexibility is allowed; OOC routes messages using whatever information is given. The most primitive kind of object descriptor is simply the address of the object in memory. This is normally used only by other objects, typically by the object's creator and owner. Another kind is the object identifier, a 4-byte integer. These object identifiers must be unique within a particular OOC speaker, but OOC itself does not require them to be globally unique. Each OOC speaker maps its object identifiers to object descriptors, typically to the object's address, but perhaps to another OOC speaker. More elaborate object descriptors are used by objects on other nodes of a network. These can include network identifiers and nodes, and task identifiers and task names. Objects can also have ASCII names which are typically used by an operator typing OOC messages.

OOC keeps track of these object identifiers and names using objects of the standard OOC-supplied class DICTIONARY. Several other standard classes are used, including ARCHIVES and HISTOGRAMS for recording significant events such as errors.

Messages

Messages have two parts, the "message type" and "message parameters." The message type (e.g. SET, TURNON) is the portion used to help select the appropriate method. The message parameters are as simple or complex as necessary, their structure being determined by the sender and understood by the receiving object. OOC itself has no expectations regarding message parameters.

Replies to messages also have two parts, the "status" and the "result." The status indicates OOC's opinion of whether the message succeeded and, if not, how it failed; the result was generated by the method itself. OOC places no restrictions on results.

Extensibility

* Operated by the Universities Research Association under contract with the U.S. Department of Energy.

OOC is extensible in that it comes with standard object classes, standard objects, and standard message types. Each application can add its own classes (which can be subclasses of OOC's standard ones), its own objects (of OOC classes or the application's own), and its own message types. Also, libraries of useful object classes will be written and used by more than one application. These should include control objects like finite state machines and PID loops, and also objects representing standard hardware such as terminals, LEDs, etc.

Why Object-Oriented?

The best argument supporting the object-oriented paradigm is that it allows knowledge to be better organized. For example, everything a system knows about valves is stored inside the valve class object. This includes their ability to be read and set, and exactly how to read and set them (i.e. the methods). Everything a system knows about a particular valve is stored inside the object representing that particular valve: its current setting and current position, for example. This "data hiding," the notion that data are accessible only by the object containing them as instance variables, also improves system reliability.

Another advantage of objects is that they allow software entities within a processor to address each other in a standard way (the same way the outside world addresses them). For example, a PID loop gets its input by sending a READ message to its input object. This sort of thing cannot be hard-coded because an operator may wish to change the input to a loop, and not just to another channel, but to an entirely different kind of thing like the average of two channels. Such an average could be established as a new object of the AVERAGER class.

In variable-oriented communications all one can do is read or write variables (or pseudo-variables). Active variables, which trigger some action when set, are then necessary to accomplish simple things like turning a pump on (one must "set" the pump's "on-ness" to "true" and hope that the pump somehow knows to actually turn on!). Such requests can be much more clearly expressed as messages (one sends a TURNON message to the pump).

The inheritance hierarchy of classes and superclasses minimizes the coding necessary to add features to a system because the author need only write code describing how the new feature differs from existing ones. One outstanding example of how much effort this can save is the dynamic creation of objects. This is actually done by the CREATE method of the object at the top of the inheritance hierarchy. This means that this one piece of code allows ANY kind of object to be created dynamically!

Inheritance also allows very generic objects to be created and used in different ways. Several times in the course of developing OOC, existing classes have met new, unanticipated needs. Dictionary objects are heavily used to support operator I/O, for example.

Tagged Data

OOC is built on a base of tagged data called molecules. Each molecule contains a tag which describes its data type. In addition to simple tags such as BOOLEAN and INTEGER, there are more complicated tags like MESSAGE for representing an entire OOC message. Most importantly, molecules with the special tag "#" contain any number of sub-molecules recursively. Thus arbitrarily large trees can be manipulated easily.

Tags allow OOC to provide powerful generic functions for reading, printing, and evaluating any molecule. The generic dictionary objects map molecules to molecules. Archive objects record noteworthy (arbitrarily complex) events as chains of molecules, and are easily interrogated using a single generic match function.

Another important use of molecules is the parameter portion of messages and the result portion of replies. Since molecules can be arbitrarily complex or simple, they are ideally suited for these uses. When a message is sent, OOC passes the parameter molecule to the appropriate method which generates the result molecule and returns it to OOC.

Environment

OOC messages and replies exist in three formats: ASCII, flat, and fluffed. That all OOC messages can be expressed as ASCII strings allows operators and system developers to type any possible message and get the reply as another ASCII string. Flat format is the one normally used for sending OOC messages and replies over ACNET, the network protocol, on token ring. Fluffy format is used internally by objects and involves pointers to substructures.

In addition to terminals, networks, and local objects, there are some special senders and receivers of OOC messages. The most important is a set of tasks which translate between OOC and the existing data aquisition services of Fermilab's control system.

OOC is a function library written in portable C to run on VAXes, Motorola 680x0s, and Intel 80386s. Messages are sent and replies received via calls to these functions. The microcomputer versions use Microtec development tools, the VAX version uses DEC tools. OOC routines can be called from other languages as allowed by the tools. OOC uses the standard Fermilab Controls Department microcomputer operating system MTOS on microcomputers, and a very primitive program called FAKEMTOS provided by the author for VAXes.

Status

OOC presently works on the VAX in single-tasking mode. The 680x0 version working in multi-tasking mode and translating to Fermilab's data aquisition services should work within a month or two. There are no definite plans to implement the network version and the Intel version. Major enhancements which will probably be implemented within six months include multiple inheritance, which would allow classes to inherit methods from more than one superclass.