Object Oriented Programming and Events

A revolutionary concept that changed the rules in computer program development, object-oriented programming (OOP) is organized around "objects" rather than "actions," data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on your computer desktop (such as buttons and scroll bars).

We are starting to write our NetLinx programs using Object Oriented techniques. There are three parts to the Object Oriented paradigm:

1. Object Oriented Analysis
2. Object Oriented Design
3. Object Oriented Programming

Object Oriented Analysis

Analysis means the process of determining the needs, or requirements, of a system – what the system must do, but NOT how the system does it. There are four major approaches to analysis.

1. Functional Decomposition – Selecting the processing and sub-processing steps required for the system. Functional Decomposition = Functions + Sub-functions + Functional Interfaces
2. Data Flow – Maps the flow of data using event “bubbles”. Data Flow Approach = Data Flow + Data Stores + Terminators + Process Specs + Data Dictionary.
3. Information Modeling – entity-relationship diagrams. Information Modeling = Objects + Attributes + Relationships + Super-type/Sub-types + Associative Objects.
4. Object Oriented – uses all three of the above methods plus new concepts such as:
    a. Encapsulation
    b. Inheritance
    c. Communication with messages

Object Oriented = Classes and Objects + Inheritance + Communication with messages

An object is defined as – A runtime instance of some processing and values, defined by a static description called a “class.”

The OOA model is presented in five layers:

1. Subject layer
2. Class & Object layer
3. Structure layer
4. Attribute layer
5. Service layer

A "Class" is a noun and objects are created that describe that noun.
A "Structure" is a generalization of a specific object. A structure inherits the features of a specific object.
"Attribute is defined as some data (state information) for which each object has its own value. Attributes are adjectives that describe the noun. Another term for "Attribute" is "Property".
"Service" is defined as a specific behavior that an object is responsible for exhibiting. Another term for Service is "Method". They are the actions that an object performs or the verbs describing actions the noun takes.

Objects are things, like a VCR, DVD, CD, switcher, projector, etc. **Objects** have **Properties** and **Methods**. Properties describe the object. Properties are implemented in NetLinx as variables, constants, and structures. Methods are actions that the object performs. Methods are implemented in NetLinx by using the Push, Release, Hold sections of Button Events, in Data Events, etc. using the various NetLinx keywords like Send_String, ON, OFF, Pulse, etc.

Object Oriented Design

Object oriented design is defined as – The practice of taking a specification (done using OOA hopefully) of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details.

What this means is the designer takes the OOA results and figures out how to implement it using a particular hardware/software platform.

There are four main criteria that apply to OOA, OOD, and OOP:

1. Abstraction – The principal of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate on those that are. The simplest interface to an object that provides all of the features and services that the intended user expects (the object acts the way the user expects it to act). It is the unambiguous specific public interface.
2. Encapsulation – A principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision. The interface to each module is defined in such a way as to reveal as little as possible about its inner workings.
3. Inheritance – A mechanism for expressing similarity among Classes, simplifying the definition of Classes similar to one(s) previously defined. It portrays the

generalization – specialization relationship, making common Attributes and Methods explicit within a Class hierarchy or lattice.

4. Polymorphism – is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. There are several different kinds of polymorphism.

These four criteria also determine if a programming language is object oriented.

There are four components/activities in OOD:

1. Designing the Problem Domain
2. Designing the Human Interaction
3. Designing the Task Management
4. Designing the Data Management

Designing the Problem Domain
This activity involves making sure all the objects in the system have been defined and defined properly in the OOA phase. In our A/V example, it also involves selecting the right control devices to control the A/V equipment.

Designing the Human Interaction
This activity in the A/V world is mostly the touch panel or other input panel type, including PC input.

Designing the Task Management
Tasking simplifies the design and code of needed concurrent behavior. In NetLinx, most tasks are implemented in an Event, Function, or Subroutine. In the OOD phase, the Event, Function, or Subroutine is identified, but no code is written to implement it.

Designing the Data Management
This is where you determine your data needs. Do you need an external database? How about storing data in files on the master? Do any variables need to be persistent? Should you use individual variables or a structure for some data? What variables need to have system wide, global, or local scope? How big do your arrays need to be? These are some of the things to be considered in this component.

Now, to relate these concepts to the A/V world here is an example:

Subject = Conference Room A/V Presentation System
Class 1 = Display Devices

1. Display Object 1 = Projector
    a. Attribute 1 = Power Flag
    b. Attribute 2 = Current Input Selected
    c. Attribute 3 = Video Mute Flag
    d. Method (or Service) 1 = Toggle Power
    e. Method 2 = Select Input

        f. Method 3 = Toggle Video Mute
    2. Display Object 2 = Preview Monitor

Class 2 = Source Devices

1. Source Device Object 1 = DVD
2. Source Device Object 2 = CD Player
3. Source Device Object 3 = VCR

Class 3 = Audio Processing Devices

1. Audio Processing Device Object 1 = Mic Mixer
2. Audio Processing Device Object 2 = Equalizer
3. Audio Processing Device Object 3 = Amp

Class 4 = Routing Devices

1. Routing Device Object 1 = Video Router
2. Routing Device Object 2 = Audio Router

Class 5 = Control Device

1. Control Device Object 1 = RS232 Card
2. Control Device Object 2 = IR/S Card
3. Control Device Object 3 = Relay Card
4. Control Device Object 4 = Touch Panel

Let's take a multi-disc CD Changer. The CD Changer is the **Object**. Some **Properties** would be:

- Power Flag – keeps track of whether the power is on or off
- Disc number – keeps track of what disc is selected
- Function – keeps track of what transport function is active (Play, Stop, Pause, etc)
- Track – keeps track of what music track is currently selected

Some **Methods** would be:

- Play
- Stop
- Pause
- Skip Forward
- Skip Reverse
- Fast Forward
- Rewind

The code to implement this might look like this:

```
DEFINE_VARIABLE
INTEGER nCD_PowerFlag
INTEGER nDiscNum
INTEGER nFunction
INTEGER nTrack

DEFINE_EVENT
BUTTON_EVENT[TP,10]
{
```

```
        PUSH:
        {
                PULSE [CD,27]       //Go to Disc #2
                nDiscNum = 2
        }
}

BUTTON_EVENT[TP,1]
{
        PUSH:
        {
                PULSE [CD,1]        //Play
                nFunction = 1
        }
}
```

Buttons on an AMX touch panel can be considered an **Object** also. Some of the **Properties** of a button are:

- Associated device number
- Channel number
- Feedback type (channel, momentary, etc.)
- Variable text channel number
- Level number
- Pop Up page Assignment

The **Methods** of a button are:

- Push
- Release
- Hold

Camera Example

```
DATA_EVENT[dvCam]
{
        ONLINE:                 //RS232 port to camera comes online
        {
                SEND_COMMAND dvCam,'SET BAUD 38400,N,8,1 485 DISABLE'
        }
}



BUTTON_EVENT[TP,50]                     //Pan camera right
{
        PUSH:
        {
                SEND_STRING dvCam,'PAN_RIGHT'
        }

        RELEASE:
        {
                SEND_STRING dvCam,'STOP_PAN_RIGHT'
        }

}

CHANNEL_EVENT[dvCam,250]                //Power channel status
{
        ON:
        {
                PULSE [dvMonitor,26]        //Make sure monitor is turned on
        }

        OFF:
        {
                PULSE [dvMonitor,16]        //Switch input on monitor to RGB
        }
}
```

Mapping OOD to NetLinx

The challenge now becomes taking the objects, with their properties and methods, and implementing the design in NetLinx. This is not as easy as it would be in a truly object oriented language such as Visual C++ or JAVA, since NetLinx is not an object oriented language.

What we need to learn is how to create "Classes", and objects with their properties and methods in Netlinx. We also need to learn how to implement inheritance, encapsulation, and polymorphism in NetLinx.

First, we can create a "Class" as a module in NetLinx. The module will model an object of that "Class". We instantiate, or create an instance of, an object by calling that module in the "Main" source code file.

We can simulate inheritance by calling another module within a "Primary" module. For example:

"Primary" module = VCR
"Inherited" module 1 = RS232 Controlled VCR
"Inherited" module 2 = IR Controlled VCR

The "Inherited" modules will be able to use whatever variables and routines they need from the "Primary" module and will add what they need for their specific type of control.

The module abstracts and encapsulates the data and methods for a particular object. The public interface into the module is the parameter list for the module. Code in another module or in the "Main" source code file can not access, or change, variables in a module unless the variables are passed to the module as parameters. Routines, or methods, in a module can not be executed by another module or the "Main" source code file unless a way has been defined, usually by using a virtual device, and passing the virtual device to all the modules that need it.

Attributes, or properties, of an object are the variables defined in the DEFINE_VARIABLE section of a module or in a structure defined in that module.

Services, or methods, for an object are written as events, such as BUTTON_EVENTS, DATA_EVENTS, etc. in the DEFINE_EVENT section of the module.

One of the major drawbacks to using modules in this manner is that code in modules run in the same thread. What this means is if I am in one module and call a routine in another module, the code in the first module will continue to execute and when it finishes, the code in the 2nd module will execute.. This can lead to timing issues. Say, for example, if you call a routine in another module and expect that module to modify a variable that is used in the calling module in the same routine, the calling module would not see this change when it needs to because the called module has not executed yet.