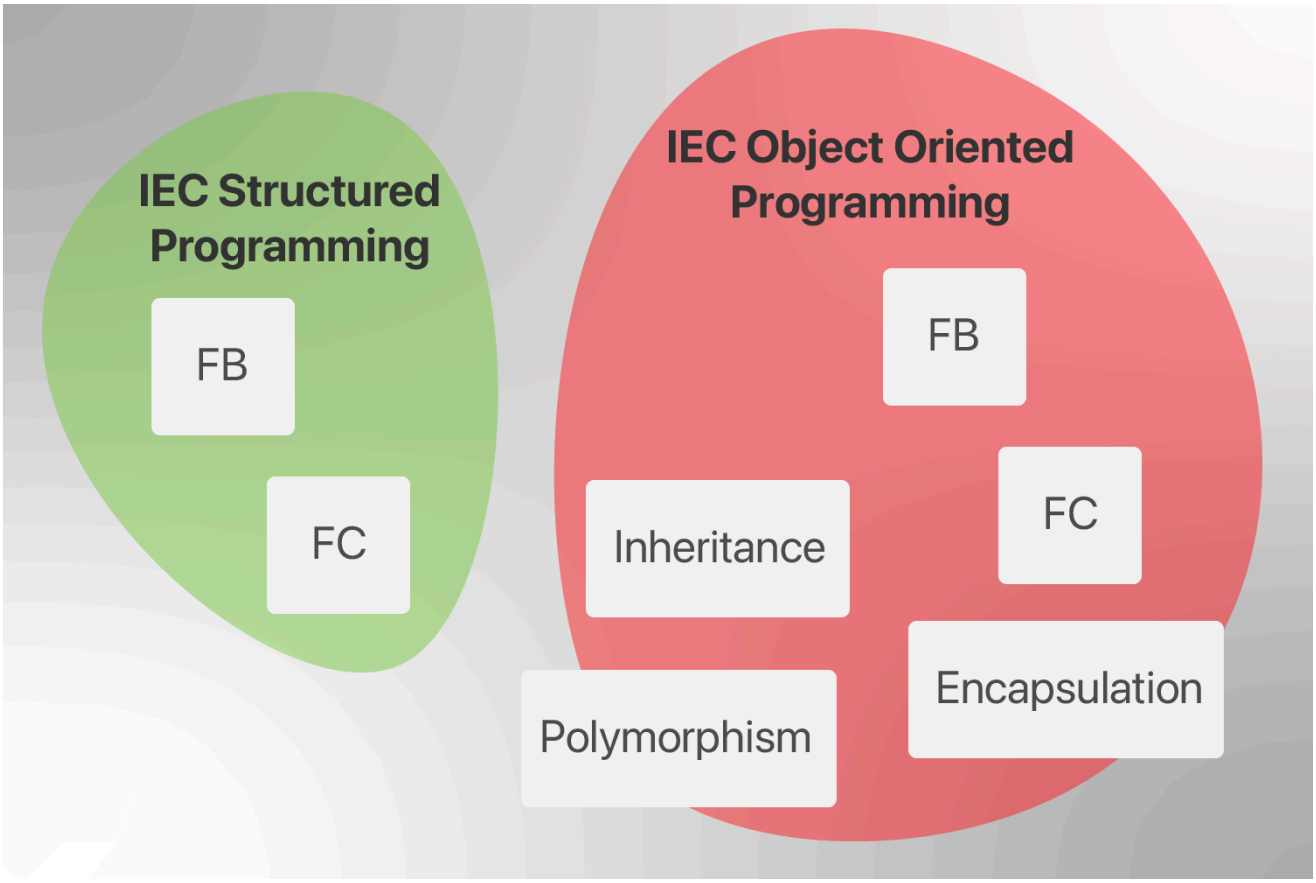


PLC Object Oriented Programming, WHY?

Around 10 years ago object oriented programming feature added to the standard IEC 61131-3 programming languages for industrial automation controllers. But you will find very few PLC producers to support OOP (Object Oriented Programming). Maybe because there are very few automation engineers interested in OOP. Basic IEC programming consists of function and function blocks and can be considered as structured programming in contrast to object oriented programming. Most automation engineers think these features of standard IEC languages are enough to make great programs as they have done it during the last decades without OOP. They believe there is no need to deal with relatively complicated concepts of OOP to achieve their routine goals. But I want to analyze what benefits OOP can provide for me in details and answer this question that is it worth to challenge with PLC object oriented programming complexities or not?

Object Oriented Programming Definition

If you ask software engineers about what is the object oriented definition, they will probably answer “this is a programming language that supports these features:”

1. Encapsulation
2. Inheritance and
3. Polymorphism

Encapsulation

The reason encapsulations is considered as part of OOP is that an OO language provides easy and effective encapsulation of data and functions. That means, in an OO language, you can have a class, we call it FB, that has some variables and functions that the other parts of the program can’t access. Those are called and private variables and functions in terms of object oriented. The public variables and functions can be seen and called by the other parts fo the program in contrast to private ones.

All of these features can also be implemented in basic FBs. In FBs we have input and output variables that can be accessed by the others and also we have local variables that can’t be accessed from outside. We can also define several operations for an FB and execute each of these operations by changing input variables state. It is exactly same as calling functions of a class.

In fact encapsulation does not add new features to basic IEC languages. Even though using OOP, integrating the data and functions in a package – a class – is more structured and easier than basic FBs, encapsulation does not add a big advantage to the basic IEC programming because we simply had it before. So there is no point for encapsulation comparing to normal IEC programming.

Inheritance

Inheritance is simply the redeclaration of variables and functions in inherited classes automatically. If you omit the term ‘automatically’ from this definition, you will have the inheritance in basic FB and FCs. I am sure you will compromise that the main power of

inheritance is in its automatic nature. Sure, you are right. But this feature can be the source of some difficult problems. Fragile base class problem is one of them. You can also read the other ones here.

In fact, in inheritance, making small changes in base classes can cause the extended classes to fail. Thus deep inheritance hierarchies can easily turn to a disaster during development and maintenance. That's the reason why all object oriented experts recommend object composition over inheritance.

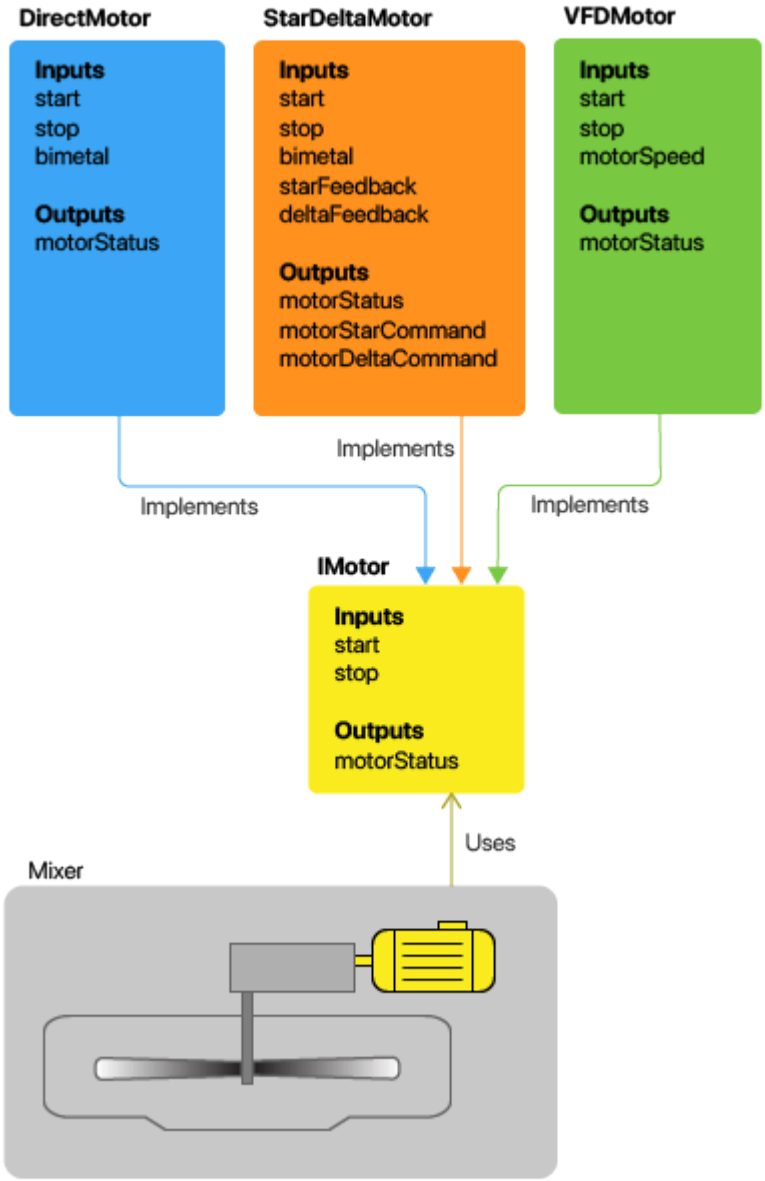
What is object composition? Composing simple objects together to build but more complex features rather than inheriting from the base classes. We also have had this feature using basic FB and FCs. I am not going to say that inheritance is completely useless. But it is not a feature that persuade me to use object oriented programming and accept all OOP overhead.

Polymorphism

Polymorphism is the ability of a programming language to present the same interface for several different underlaying data types. In higher level languages like C# and Swift, polymorphism can be implemented using both inheritance and interfaces. At the time of writing this post and as I know it is not possible to implement polymorphism by inheritance in object oriented IEC 61131-3 languages. But interface ability is available in IEC OOP completely. So let's take a look at it using an example.

An Example for Polymorphism

Mixer



Mixer program architecture using interface

Suppose a mixer program, it needs a motor to run the mixer. Mixer does not care about the type of the motor. A direct started synchronous motor with a lot of startup current consumption, a star/delta started motor, a VFD controller motor or even a DC motor. Mixer only wants a motor to start and stop it based on the mixing time in the recipe. So I define the Motor as an Interface like below which is called IMotor. the 'I' prefix is to denote that it is an interface.

Motor Interface

```
INTERFACE IMotor
PROPERTY run: BOOL
```

```
PROPERTY stop: BOOL
PROPERTY motorStatus: BOOL
```

Various Kinds of Motors

As I mentioned above there may be a wide range of motors that can implement this interface. For example a direct started motor that implements Motor has all above variables and also checks a bimetal signal for safety.

DirectStartedMotor

```
FUNCTION_BLOCK DirctMotor IMPLEMENTS IMotor
VAR_INPUT
    bimetal: BOOL;
END VAR
VAR_OUTPUT
    motorCommand: BOOL;
END VAR
```

A start/delta motor drives two different outputs to run a motor but illustrates the motorsStatus output as on in both star and delta conditions.

```
FUNCTION_BLOCK DeltaStarMotor IMPLEMENTS IMotor
VAR_INPUT
    bimetal: BOOL;
    starFeedback: BOOL:
    deltaFeedback: BOOL;
END VAR
VAR_OUTPUT
    motorStarCmmand: BOOL;
    motorDeltaCommand: BOOL;
END VAR
```

A VFD motor needs a speed value to be specified but it does not ask mixer to define it because mixer does not care about speed. For mixer the faster the better so we set the speed to 50 Hz before passing it to the Mixer.

VFDMotor

```
FUNCTION_BLOCK VFDMotor IMPLEMENTS IMotor
VAR_INPUT
    motorSpeed: REAL;
END VAR
VAR_OUTPUT
    motorCmmand: BOOL;
END VAR
```

Now we declare that the Mixer needs a Motor to work with:

Mixer needs an IMotor

```
FUNCTION_BLOCK Mixer
VAR_INPUT
    motor: IMotor;
END VAR
```

IMotor is an interface and it does not have any implementation by itself. So we need to pass an FB that implement this interface. What kind of motor do you have in your project? Let's say a star/delta motor. So somewhere else in your program I assign a VFDMotor to the mixer. Let's call this part of program that decides about the kind of motor and assign it to the mixer, the Composer.

Mixer Composition

```
FUNCTION_BLOCK Composer
VAR_INPUT
    mixer: Mixer;
    starDeltaMotor: StartDeltaMotor;
END VAR

mixer.motor := starDeltaMotor; //We can do it due to polymorphism
```

It is the polymorphism that lets me to assign starDeltaMotor to the mixer.motor. Actually we can do that because the StarDeltaMotor FB, besides being a StarDeltaMotor can act as an IMotor. This is the concept of polymorphism. In action, that means there

can be many different FBs that can be assigned to the mixer.motor as long as they implement the IMotor interface. In contrast to encapsulation and inheritance, polymorphism is a completely new feature and we are not able to emulate it using basic IEC languages.

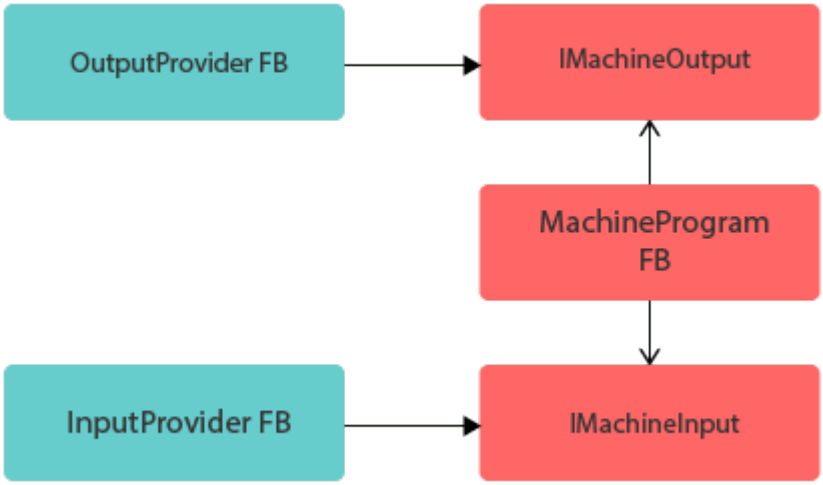
Unleashing the Power of Polymorphism

Your Boss Order

Let’s say one day your boss comes at you and says we are going to show our machine in an exhibition and we want to show people how it works. We have access to pneumatic and hydraulic pressure but we do not want to waste materials as it is dangerous and costly. The machine should work completely but it should not produce anything, nonetheless the HMI must show real products are producing process. You may say this is a dumb request and it costs a lot of time to develop and test such an ability. but I will say it’s not. Even though this feature is very useful for fairs, it is great to test the machine during commissioning phase. This kind of operations are called normally Cold Test. It helps you to check most of your mechanical, electrical and programming features without accessing utilities and wasting materials.

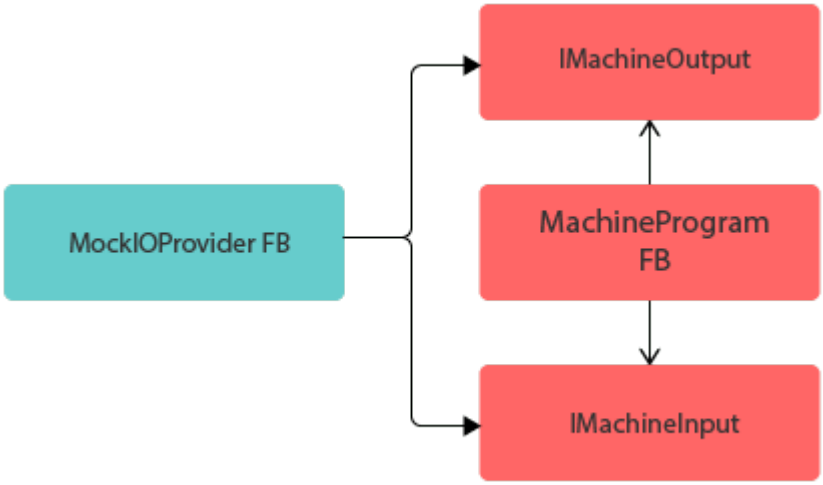
Your Solution

Now suppose you have designed your PLC program in a way that all your machine program is placed in the MachineProgram FB. The MachineProgram does not know anything about the real IOs. Instead it depends on two interfaces to access inputs and outputs, IMachineInput and IMachineOutput. There are also two FBs that implement these interfaces and provide accessing to the required digital, analog and other kind of IO for MachineProgram.



Isolating machine program from IO access

In such an architecture, implementing the cold test operation mode would be very easy because we can easily replace the real IO providers with a mock version that simulation the real IOs for us. For example when the MachineProgram starts a motor, the mock IO provider turn the motor feedback on without really running a motor and receiving real run feedback.



Mock IO Provider Implementation

Changes Never End

Imagine after some days your boss come again asks you to change all your IOs from local to Profibus for a specific customer. After a while you understand that using EtherCAT is by far better than Profibus for your machine. Are you worried about the time and impacts of these sort of changes on your program? Definitely not. For the last

crazy assumption, assume today morning you have received an email from your boss that asks you to implement all your program for a new faster and cheater automation control system. Still you are happy as long as you have separated different parts of your program gracefully and interfaces are a magical tool to do so.

Conclusion

Using interfaces we can isolate different parts of our program into independent modules and by composing those modules together we can build a software which is really soft. Soft to change and reshape. This level of modularization is not available in old IEC programming techniques. Hence, regarding interface abilities, I would pick the object oriented programming instead of the old procedural certainly.

If you are eager to know how to design your PLC programs using these tools to achieve a higher level of modularity, follow me on my next blog post, "PLC Object Oriented Programming, HOW".

If you have any questions or suggestions about above subjects leave your comments below.

Posted January 13, 2021 in Blog
by amirreza

Tags:

IEC, Object Oriented Programming, OOP, PLC, Programming