Benefiting from OOP in IEC 61131-3

Realizing machine variants elegantly with dynamic binding

The first article in the series on object-oriented programming in automation technology ("Goto OOP", SPS magazine 09-2012) has already shown that structured PLC programming leads step by step to genuine object orientation, as is familiar from high-level languages. What specific engineering advantages can the application programmer achieve as a result?

"Polymorphy" – which sounds at first like a rare disease to those who have no knowledge of high-level language programming – is a useful property of object-oriented programming. It enables the dynamic binding of program code. This property was briefly outlined in the first part of this series of articles. Its benefits will now be explained below.

Different delivery variants of a machine series by interfaces

An application programmer develops a PLC program that is to be used without large changes on different delivery variants of a machine series. Drives are controlled by the PLC in all variants. However, the end customer decides which drives will ultimately be used, for example on the basis of performance data or the level of knowledge of its maintenance staff. Despite the differences, all the drives have functions such as "HomePosition", "HasError" or "MoveAbsolute".

If the application programmer has written his program functionally as before, then he must adapt all function calls that access the drives for each variant of the machine. With the possibilities of OOP, however, he can reduce both work and sources of error. To do this he defines all uniform call functions for the drives employed as methods in an interface.

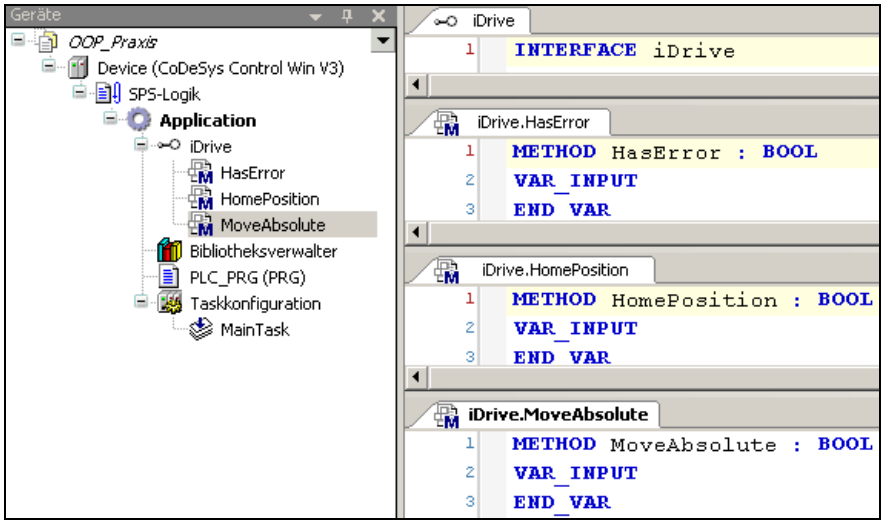

Fig. 1: Uniform interface for the drive functions: definition as an interface

The interface contains for each method only its call interface, i.e. inputs and outputs, but no local variables and no code. The programmer will fill the methods with specific program code when he integrates them in a function block using the new keyword IMPLEMENTS. The function block thus becomes a "class" in the sense of OOP. The

programmer typically carries out the necessary instancing of the function blocks in a higher-level block that manages these instances centrally.

Dynamic binding via an array

```
PROGRAM PLC_PRG
VAR
    Drive1: Analog_DriveA;
    Drive2: CANopen_DriveA;
    Drive3: CANopen_DriveA;

    DriveArray:ARRAY[0..2] OF iDrive:=[Drive1, Drive2, Drive3];
END_VAR
```

Fig. 2: Instancing of the classes in the main block

As shown in fig. 2, several such instances ("objects") can be combined in an array. What is surprising here is that the data type of the array no longer has to be BOOL, INT or a function block, but can now be an interface – in the example the interface *iDrive*. Hence, the array can actually be filled with completely different contents – depending on the function block instanced. In fig. 2 the individual fields of the array are filled immediately during the declaration – with instances of different function blocks for the different drives.

Via the array the programmer can access the methods in the FB instances by loop in an indexed manner. It is initially irrelevant which methods are actually called in the long run – the allocation took place by the filling of the array. Hence, the actual method call is dynamically bound.

```
1  //Alle Antriebe auf HomePosition
2  FOR I:=0 TO 2 DO
3      DriveArray[I].HomePosition();
4  END_FOR
```

Fig. 3: Abstracted call of the methods in a loop

If the next machine variant uses other drives than those defined in fig. 2, then the FB instances used for the new function blocks that are to be inserted need only be declared by the application programmer in the declaration section of the main block. A change of the method calls is no longer required!

Dynamic binding via a function block

Instead of calling the methods dynamically via an array, the programmer can also create a further function block. This is given an interface as an input parameter. Hence, the FB *CheckDriveError* in fig. 4 knows that it will call the method *HasError*.

```
FUNCTION_BLOCK CheckDriveError
VAR_INPUT
    DriveToCheck: iDrive;   //Übergabe des Antriebs
END_VAR
VAR_OUTPUT
    DriveError: BOOL;
END_VAR
VAR
```

```
IF DriveToCheck.HasError() THEN
    DriveError := TRUE;
END_IF
```

Fig. 4: FB with interface as VAR_INPUT parameter

The programmer transfers the desired FB, or the class in which the method is called, as an instance of the drive, e.g. in turn on being called from the main block.

PLC_PRG

```
PROGRAM PLC_PRG
VAR
    CheckDrive:CheckDriveError;
    Drive1: Analog_DriveA;
    Drive2: CANopen_DriveA;
```

```

CheckDrive(DriveToCheck:=Drive1);
```

Fig. 5: FB call with instance transfer by an interface

Using this procedure the programmer can also change the drives employed centrally in one place without having to laboriously wade through all the calls in the project.

Re-use of methods – inheritance

Let us look once again at the function blocks for the specific functions of the drives. Depending on the drive properties the application developer must program-out all required methods individually. However, similar drives from the same manufacturer often have identical basic functions, e.g. for error query or homing. If that is the case, then one would also like to re-use these identical functions. Precisely this request is met in OOP by inheritance.

To this end the programmer creates a new function block that extends an existing block (a "basic class") with the new keyword EXTENDS. This new object thus immediately has all methods of the basic class without them being displayed again in the object tree as a child. For its part the object can implement further interfaces or methods of its own are assigned to it.

Typically, however, not all methods of CANopen_DriveC will be identical to CANopen_DriveA. Hence, the method *MoveAbsolute*, for example, must be executed differently. To do this the application developer can create the method *MoveAbsolute* for CANopen_DriveC, which was already originally defined by the inheritance, and program it out accordingly. The originally inherited method is thus overwritten and is no longer valid for this function block.

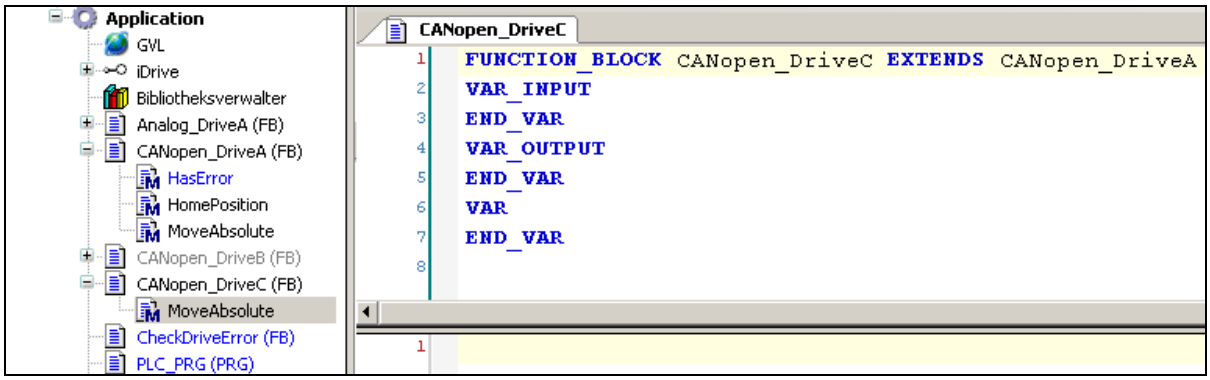

Fig. 6: FB extends basic class and overwrites one of the inherited methods

The programmer can elegantly link inherited and specific program code with one another: as shown in fig. 6, he does this by creating a new method, thus overwriting the inherited one. In the program code of the new method, however, he initially calls the code of the overwritten method with the command

Super^.MoveAbsolute();

Subsequently, he extends the processing by the specific program code.

```
METHOD MoveAbsolute : BOOL
VAR_INPUT
END_VAR

SUPER^.MoveAbsolute();
.zaehler:=.zaehler+2;
```
Fig. 7: Calling an inherited method using the SUPER pointer

Conclusion

These simple examples make it clear that OOP is useful for making application software that is modular and re-usable. That applies equally to programming in high-level languages and to the programming of controllers in automation technology. Automators can already program their PLC applications with CODESYS today, object-oriented in IEC 61131-3 - if they want to.

However, some questions still remain: How can data and functions be encapsulated in such a way that they are not inadvertently changed? How can the code in the body of a function block be used within a method? To be continued…