OOP in IEC 61131-3 for experts

Elegantly solving the handling of data and call information with OOP

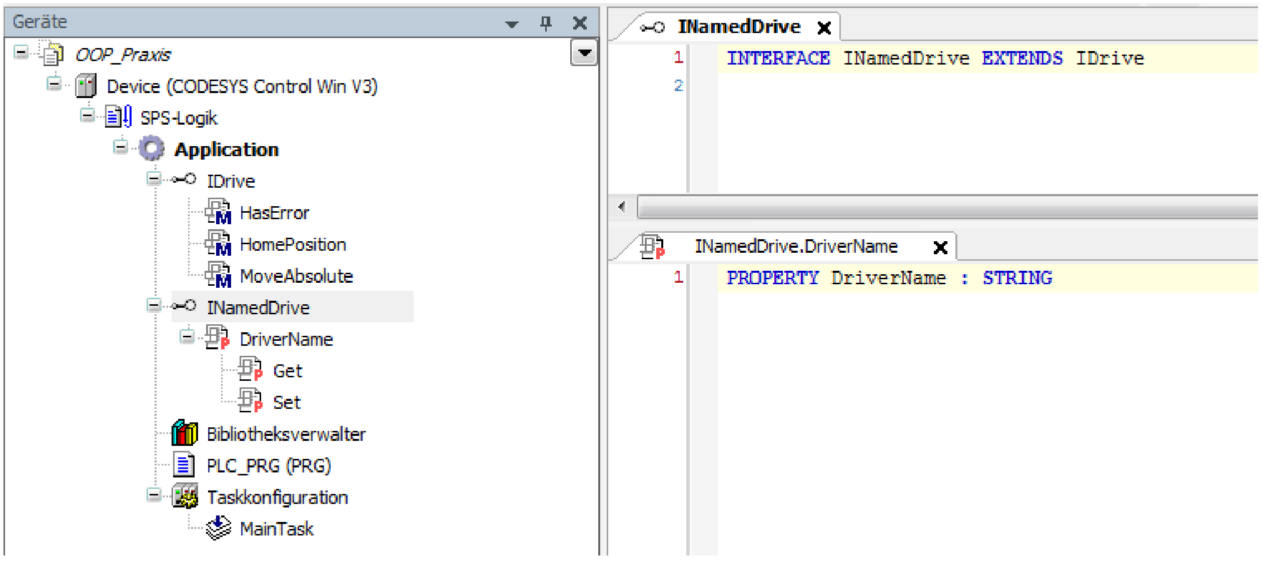
After the introduction to object-oriented programming in automation technology, the benefits of the use of interfaces and the inheritance of function blocks was explained in the second of the series of articles. But how does one deal with data? How does one evaluate information about implemented interfaces? With the answers to these questions, expert features of object-oriented programming such as those available in the IEC 61131-3 programming system CODESYS are described.

Data handling with properties

As already explained, interfaces contain only methods – but function blocks also have data. Thus the question arises: how does one access this data via an interface?

Simple answer: one writes a method that returns the desired data, such as "GetName" or "IsReady" for instance. If applied consistently, the application programmer can quickly create pairs of functions which are always of the same type, such as "GetName" / "SetName" or "IsReady" / "SetReady". Such pairs of methods, which essentially only enable access to a piece of data, can be combined into a *property*.

Let us take as an example the interface *IDrive* from the preceding article in the series and create an extended interface called *INamedDrive*.



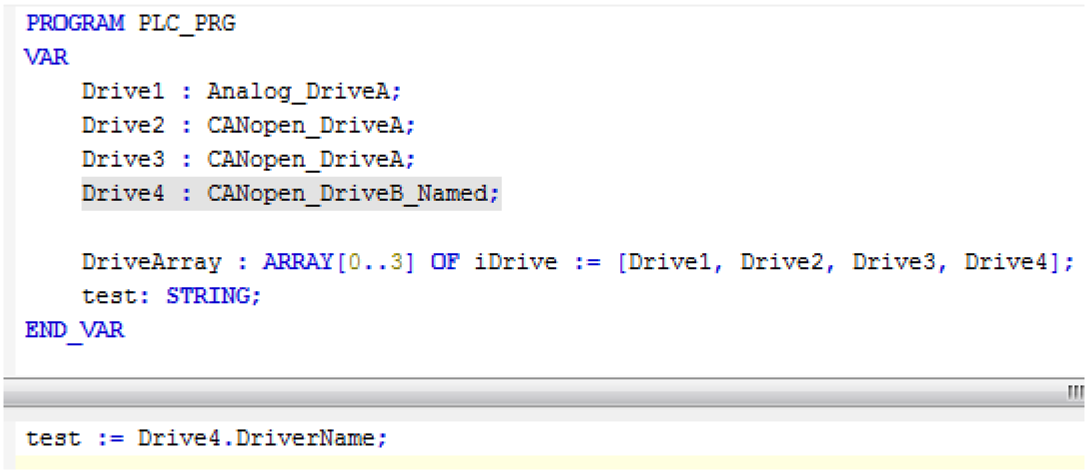
```
INTERFACE INamedDrive EXTENDS IDrive
```

```
PROPERTY DriverName : STRING
```

Fig. 1: Simple property

A property is thereby a combination of two methods that encapsulate the write access or read access to a piece of data. This presents itself as a variable to the user of the properties. The compiler automatically ensures the call of the correct access method or signals an error if this is not implemented. Hence, one would probably equip *INamedDrive* in the above example only with a reading access to the name, thus preventing writing.

In the main block PLC_PRG (from the preceding article) we declare a function block *CANopen_DriveB_Named*, which implements the new interface. In the body of the module in Fig. 2 the property can now be accessed in the same way as a variable.



```
PROGRAM PLC_PRG
VAR
    Drive1 : Analog_DriveA;
    Drive2 : CANopen_DriveA;
    Drive3 : CANopen_DriveA;
    Drive4 : CANopen_DriveB_Named;

    DriveArray : ARRAY[0..3] OF iDrive := [Drive1, Drive2, Drive3, Drive4];
    test: STRING;
END_VAR
```

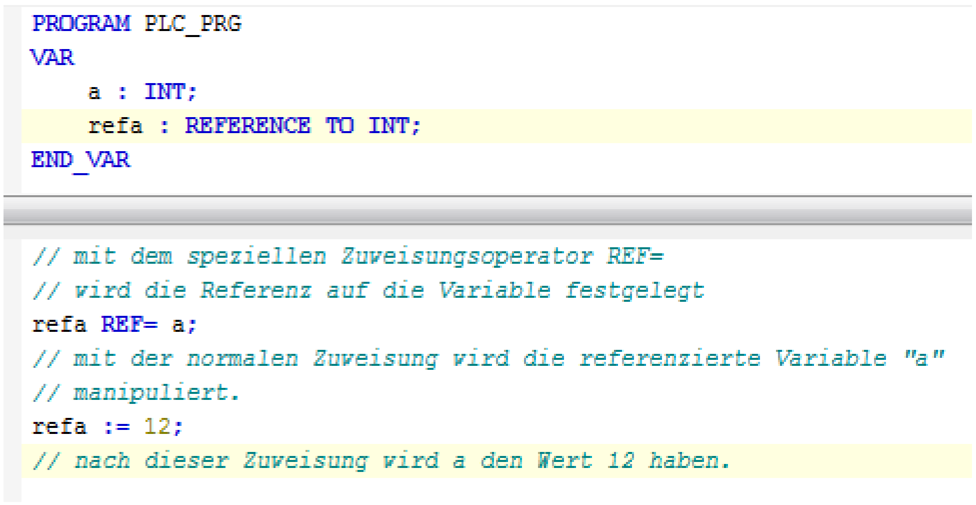
```
test := Drive4.DriverName;
```

Fig. 2: Instancing and use of FBs with extended interfaces

A property can be problematic, however, if one then wishes to allow access to complex data types such as a structure. Usually the application programmer wishes to access only one element of a structure. A method behind that, however, always returns the entire structure. This means that if the data type *Structure* is transferred directly to a property, then too much data will be copied - and that has an effect on the runtime of the program. How can one get around this problem? The data type "REFERENCE TO" supplies the answer.

A *Reference* is a variable that always refers to another variable. If one manipulates the reference, then one actually manipulates the referenced variable. The reference differs from the pointer by the fact that the reference is not explicitly de-referenced, but instead each access takes place directly to the referenced variable.

An example:

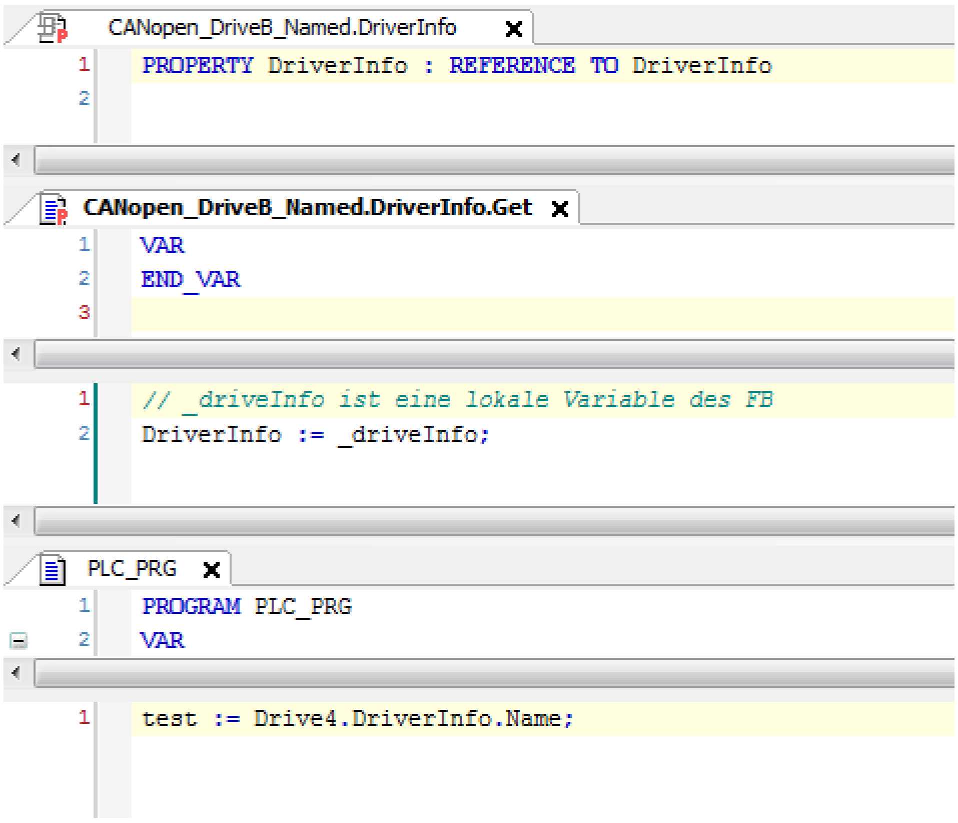

```
PROGRAM PLC_PRG
VAR
    a : INT;
    refa : REFERENCE TO INT;
END_VAR
```

```
// mit dem speziellen Zuweisungsoperator REF=
// wird die Referenz auf die Variable festgelegt
refa REF= a;
// mit der normalen Zuweisung wird die referenzierte Variable "a"
// manipuliert.
refa := 12;
// nach dieser Zuweisung wird a den Wert 12 haben.
```

Fig. 3: A simple reference

Naturally the example in fig. 3 is not practical, it is intended merely to explain the use of references in practical applications. One of these is the return of data in the case of properties.

For our sample project we assume a structure, *DriverInfo*, with the elements *Name* and *Version*. Instead of two properties for the name and the version, the entire information could also be returned as a structure. We give the function block *Canopen_DriveB_Named* an additional method. This does not return a simple data type, but a whole structure.


CANopen_DriveB_Named.DriverInfo

```
PROPERTY DriverInfo : REFERENCE TO DriverInfo
```

CANopen_DriveB_Named.DriverInfo.Get

```
VAR
END_VAR
```

```
// _driveInfo ist eine lokale Variable des FB
DriverInfo := _driveInfo;
```

PLC_PRG

```
PROGRAM PLC_PRG
VAR
```

```
test := Drive4.DriverInfo.Name;
```

Fig. 4: Complex data types and properties

Properties thus offer the possibility to publish a functional access to data. This form of access corresponds to the demands of OOP for data encapsulation and at the same time provides the programmer with the convenience of simple data access. As explained in the example, the application programmer can also define such properties in an interface and as a result indirectly formulate a regulation for the data in a function block.

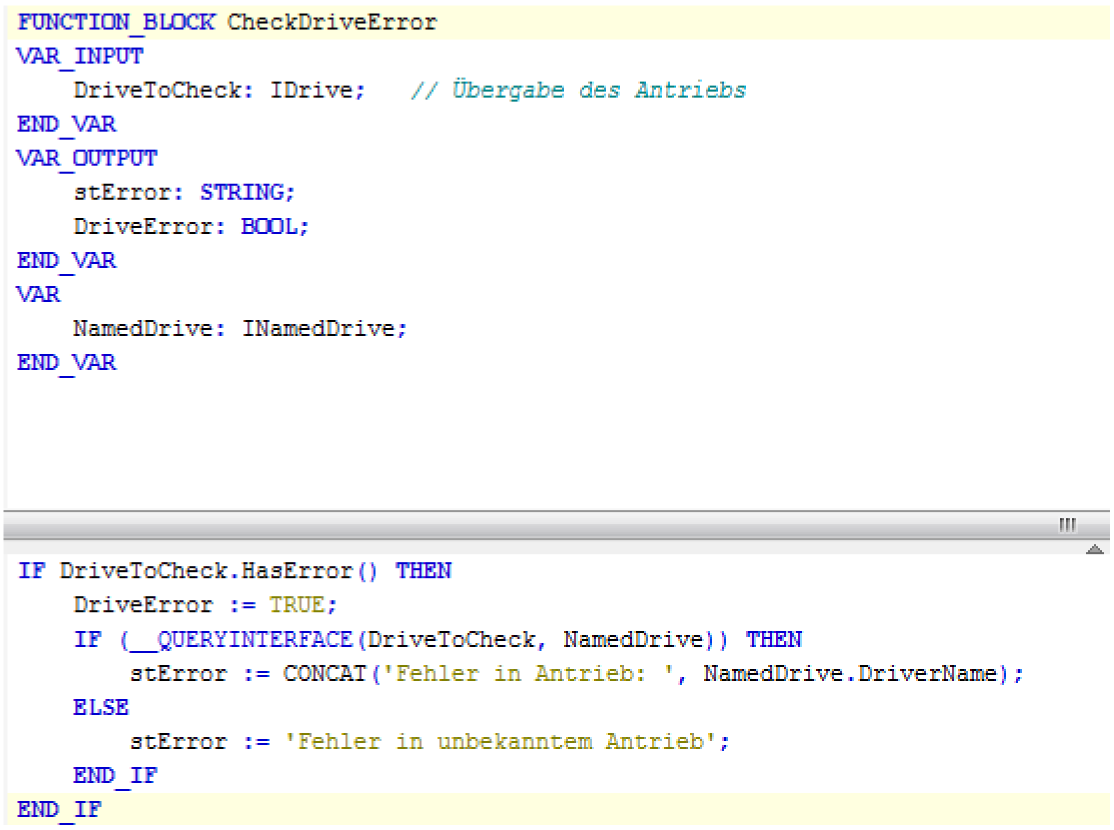
A further possible use of properties is the return of scaled values: in this way a function block could store a value in the unit centimeters. With its own property, however, the value is returned in the unit inches.

Data type queries on interfaces: casts

The example leads us directly to the next topic, type conversion and type queries (casts). We defined an array of elements of the type *IDrive* in fig. 2 in the *PLC_PRG*. Only one element of the array additionally defines the specialized interface *INamedDrive*. It may be of importance at the place of use, however, whether the drive has a name or not, i.e. whether it implements the interface *INamedDrive*. To this end one must be able to query the type information of an object at runtime.

CODESYS contains the operator *__QUERYINTERFACE* for this function. It expects two operands: on the one hand the interface object, from which one wishes to query another interface, and on the other an interface variable with the type which one wishes to check. The operator itself returns the result TRUE if the cast, i.e. the type query was successful.

For the explanation we use the function block CheckDriveError (see 2nd part of the series of articles) and extend it by the output of an error text via the variable stError. The code part in fig. 5 is easy to comprehend: __*QUERYINTERFACE asks the drive transferred by the interface IDrive whether it additionally implements the interface INamedDrive.* If that is the case, then stError returns an error message with the drive name instead of a general text.



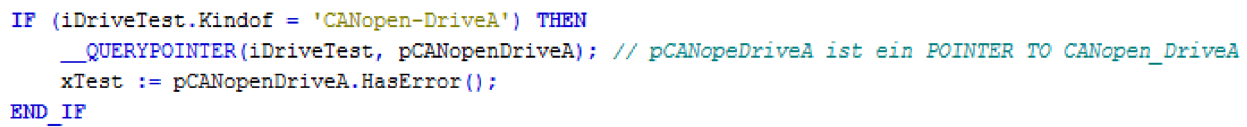
```
FUNCTION_BLOCK CheckDriveError
VAR_INPUT
    DriveToCheck: IDrive;    // Übergabe des Antriebs
END_VAR
VAR_OUTPUT
    stError: STRING;
    DriveError: BOOL;
END_VAR
VAR
    NamedDrive: INamedDrive;
END_VAR
```

```
IF DriveToCheck.HasError() THEN
    DriveError := TRUE;
    IF (__QUERYINTERFACE(DriveToCheck, NamedDrive)) THEN
        stError := CONCAT('Fehler in Antrieb: ', NamedDrive.DriverName);
    ELSE
        stError := 'Fehler in unbekanntem Antrieb';
    END_IF
END_IF
```

Fig. 5: Use of __*QUERYINTERFACE* for the generation of a plain text error message in a heterogeneous environment

A further application of the casts is rarer, but nevertheless possible: an interface reference requires the instance to which it refers. The suitable operator for this in CODESYS is __*QUERYPOINTER*.

__*QUERYPOINTER also expects two operands:* an interface reference and a pointer to a function block. However, the programmer must ensure in this case that the type of the POINTER is also correct after the cast.



```
IF (iDriveTest.Kindof = 'CANopen-DriveA') THEN
    __QUERYPOINTER(iDriveTest, pCANopenDriveA); // pCANopeDriveA ist ein POINTER TO CANopen_DriveA
    xTest := pCANopenDriveA.HasError();
END_IF
```

Fig. 6: Use of __QUERYPOINTER

The piece of code in fig. 6 explains a possible use of the operator: an identifier is queried by an interface reference, which in this case corresponds to the type name. Hence, the programmer knows the type of the instance that points to the reference and he can continue working safely with the pointer.

Conclusion

If one continues consistently down the path to object-oriented application programming within IEC 61131-3, then properties assist in the encapsulation of data. Additional operators based on OOP also retain the overview in complex applications. The benefit: simply re-usable control programs. The language scope of the market-leading IEC 61131-3 programming system CODESYS fulfils the expectations of experienced application programmers in this regard.