Object-Oriented Extensions for IEC 61131-3

Methods, Inheritance, and Interfaces As Additional Elements in Automation Programming

BERNHARD WERNER

Industrial automation is faced with several challenges today. The complexity of applications is ever increasing as is the number of variants available. Development cycle times are decreasing, and more and more tasks are assigned to the controller software, which plays a more than crucial role.

Object-oriented programming (OOP) in languages like C++ or Java is commonplace in desktop application development and an integral part of university education today. It has proved to be absolutely unbeatable when it comes to elegantly handling complex software-development tasks and producing flexible, reusable software components. OOP has clearly reduced the development time of new software and simplified the solution of complex software tasks.

Industrial programmable logic controllers (PLCs), however, are still mostly programmed in the languages of the International Electrotechnical Commission (IEC) 61131-3 standard. To meet the challenges of modern industrial automation, it is therefore only logical to add OOP to the upcoming third edition of the standard.

At the moment, an IEC 61131-3 maintenance group is discussing a working draft of this next edition. This draft contains a proposal for object-oriented extensions

Digital Object Identifier 10.1109/MIE.2009.934795

1932-4529/09/$26.00©2009IEEE

to be added to the IEC 61131-3. This article refers to this proposal. It is, however, a fact that the standardization in regards to OOP is still at an early stage, and it is quite possible that this article may differ from the final edition of the IEC standard.

New Language Elements

The IEC 61131-3 already contains a simple class concept, the function block. A function block has an internal state, a routine manipulating this state, and may be instantiated several times. All IEC 61131-3 programmers are familiar with these concepts. Thus, the extension of the existing function block by object-oriented features is a natural way of introducing object orientation to the IEC 61131-3.

The features proposed in the current working draft are

1) methods
2) inheritance
3) interface abstraction.

With these new language elements, the working draft covers all common elements of OOP languages such as classes, objects, methods, inheritance, and polymorphism.

Methods

A traditional function block contains only one routine for manipulating the function block's internal state, although there may be very different tasks to perform on this state, e.g., initialization, error handling, or cyclic execution. The only way to control this routine is by assigning specific values to the inputs of the function block when calling this routine.

An object-oriented approach would separate the code for the different tasks to separate methods of the function block.

The following example shows a direct comparison of the two concepts:

1) *Start the pump*
   Pump1(start := TRUE, Direction := Forward, Reset := FALSE); (*classical approach*)
   Pump1.Start(Direction := Forward); (*object-oriented approach*)
2) *Reset the pump*
   Pump1(start := FALSE, Reset := TRUE); (*classical approach*)
   Pump1.Reset(); (*object-oriented approach*).

Object-oriented programming in languages like C++ or Java is commonplace in desktop application development and an integral part of university education today.

There are several advantages of object-oriented approach:

- *Structuring*: The classical implementation must contain all functionality in one body, whereas the object-oriented approach allows for the separation of code for different tasks.
- *Readability*: The name of the method clearly shows what the pump is meant to do, whereas in the classic approach, the assignment of inputs determines the usage.

Language Feature: Methods

Methods, as defined in the current working draft, may be seen as a function declared inside a function block with result type, parameters, and local variables. As is the case within functions, these local variables will not keep their state from one call of the method to the next. Methods have an implicit access to the variables of the function block. In addition to traditional object-oriented methods, IEC 61131-3 methods may contribute to the internal state instances. This allows the implementation of EDGE-detection inputs or state machines [sequential function chart (SFC)] within a method.

Thus, all languages of the standard can be used for the implementation of methods [although in the following code samples, the implementation language is always structured text (ST)].

Language Feature: Access Specifiers

For fine-grained access on methods, the proposal defines the following access specifiers:

- *Public*: Access without any restriction
- *Private*: Access is restricted to the function block
- *Protected*: Access is restricted to the function block and its derivations
- *Internal*: Special access in combination with namespaces (also a proposal for the third edition, which cannot be discussed in this article).

Inheritance

In classical IEC 61131-3 programming, variants of a function block are constructed by copying the function block and changing its implementation. There is no way of reusing common parts of the code. Obviously, this approach has a lot of disadvantages for code maintenance.

In OOP, a new function block may be constructed by inheritance. This means that the new function block inherits all variables and methods of the old function block. It may define additional variables and methods, and it may override the methods of its parent.

Language Feature: Extends

EXTENDS in a function block makes it the subclass of another function block.

Interfaces

Language Feature: Interface

For the declaration of abstract superclasses, the interface construct is introduced. An interface is similar to a function block, with subordinate method prototypes. The difference is that the interface does not have variables nor an implementation part, and its method prototypes in turn do not have local variables nor an

An object-oriented approach would separate the code for the different tasks to separate methods of the function block.

implementation. If a function block is derived from an interface, it must contain concrete methods (with implementation) for all method prototypes defined by the interface.

The introduction of interfaces avoids the problems that arise with multiple inheritance. A function block may be derived from one other function block (and not more) but from an arbitrary number of interfaces.

```
INTERFACE ROOM
METHOD DAYTIME : VOID
END_METHOD
METHOD NIGHTTIME : VOID
END_METHOD
END_INTERFACE
```

FIGURE 1 – Abstract interface for rooms.

Language Feature: Implements

The keyword IMPLEMENTS in a function block makes it the subclass of one or more interfaces. An IMPLEMENTS declaration requires the function block to have at least all the methods that the named interfaces have (with the same parameter and result types). The new thing is that in the function block the methods must also have an implementation part!

Language Feature: Reference Semantics for Interface Types

Like a function block name, the name of an interface can be used as a type name in the declaration of variables. These declarations have reference semantics. This means, this declaration does not provide a new object but only a new reference to a function block instance declared elsewhere (initially NULL). This instance must be of a function block type implementing the interface. The assignment of a function block instance to an interface variable makes that variable refer to the function block instance. Thus, the same interface variable can refer to different instances of different function block types (polymorphism).

Note that a function block instance may be assigned to any interface variable of interface types it is derived from.

```
FUNCTION_BLOCK ROOM_CTRL
VAR_INPUT
    RM : ROOM;
END_VAR
VAR_EXTERNAL
    DAYTIME : TOD; // global time definition
END_VAR
IF (RM = NULL) THEN // always test on valid reference!
    RETURN;
END_IF
IF DAYTIME >= TOD#20:15 OR DAYTIME <= TOD#6:00 THEN
    RM.NIGHTTIME();
ELSE
    RM.DAYTIME();
END_IF
END_FUNCTION_BLOCK
```

FIGURE 2 – Implementation of room control.

Example

The following example illustrates one possible usage of the new language elements. It contains a mixture of the IEC 61131-3 languages ST (similar to PASCAL) and function block diagram (a graphical language).

The application is a simplified building application that could, for example, be used to control the lighting system of rooms in an office building. The application mainly consists of rooms with so-called daytime and nighttime modes and a room control that controls these modes. The room control is to handle any function block with a daytime/nighttime mode; therefore, we define an abstract interface for these kind of function blocks (Figure 1).

The room-control function block (Figure 2) can now handle a variable of type ROOM.

This function block is a common function block with one body and no methods. Note that the room-control function block can be called with any function block implementing the ROOM interface.

Now, we need function blocks that implement the interface (Figure 3). The following code fragment contains two such function blocks, the second being inherited from the first one.

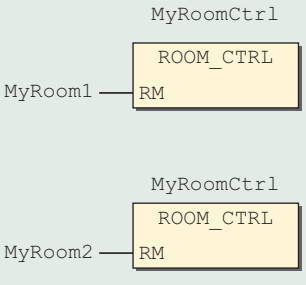
Now we have all parts of our tiny building application, and we can bring them together in a small test program (Figure 4).

```
FUNCTION_BLOCK LIGHTROOM IMPLEMENTS ROOM
VAR
    LIGHT : BOOL;
END_VAR
PUBLIC METHOD DAYTIME : VOID
    LIGHT := FALSE;
END_METHOD
PUBLIC METHOD NIGHTTIME : VOID
    LIGHT := TRUE;
END_METHOD
END_FUNCTION_BLOCK

FUNCTION_BLOCK LIGHT2ROOM EXTENDS LIGHTROOM
VAR
    LIGHT2 : BOOL;
END_VAR
PUBLIC METHOD DAYTIME : VOID
    SUPER.DAYTIME ();        // call of parent implementation
    LIGHT2 := FALSE;
END_METHOD
PUBLIC METHOD NIGHTTIME : VOID
    SUPER.NIGHTTIME ();      // call of parent implementation
    LIGHT2 := TRUE;
END_METHOD
END_FUNCTION_BLOCK
```

FIGURE 3 – Example of interface implementation.

```
PROGRAM TEST
VAR
    MyRoom1 : LIGHTROOM;
    MyRoom2 : LIGHTROOM;
    MyRoomCtrl : ROOM_CTRL;
END_VAR
 room control with two different types of rooms!
```




```
END_PROGRAM
```

FIGURE 4 – Simple building application.

The crucial point in this example is the usage of polymorphism. One variable of an interface type can be assigned variables of different types at runtime. This offers enormous new opportunities for the design of automation applications. Had this program been written in a conventional way without using OOP, it would have been necessary either to define a control function block for every room or to include the control code into every different type of room. In both cases, we would have to duplicate the code.

Conclusions

Even the small example shown earlier clearly demonstrates the strong advantages of OOPs:

- better-structured program code with separation of concerns and information hiding
- flexible extensibility by new types of objects (e.g., software representations of new types of drives)
- reuse of code for defining specialized subclasses (inheritance)
- reuse of code operating on different implementations of an interface (polymorphism).

Biography

Bernhard Werner received his master's degree in computer science (Diplominformatiker) at the Technical University of Munich in 1996. He joined 3S-Smart Software Solution as a software developer and specialized in the field of compiler technologies. He was directly involved in the development of CoDeSys V3. CoDeSys V3 is the first PLC programming system with object-oriented language features. Furthermore, he is a member of the German working group of the Deutsche Kommission Elektrotechnik Elektroni Informationstechnikim DIN und VDE (DKE) and of the maintenance team of the IEC (IEC 65B/Working Group 7/Maintenance Team 3). Both groups are currently working on the new draft of the IEC 61131-3.