



## **PLCopen - Promotional Committee 2**

### **Training**

## **Guidelines for usage of Object-Oriented Programming**

**Version 1.0 – Official Release**

#### DISCLAIMER OF WARRANTIES

THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS AND MAY BE SUBJECT TO FUTURE ADDITIONS, MODIFICATIONS, OR CORRECTIONS. PLCOPEN HEREBY DISCLAIMS ALL WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, FOR THIS DOCUMENT. IN NO EVENT WILL PLCOPEN BE RESPONSIBLE FOR ANY LOSS OR DAMAGE ARISING OUT OR RESULTING FROM ANY DEFECT, ERROR OR OMISSION IN THIS DOCUMENT OR FROM ANYONE’S USE OF OR RELIANCE ON THIS DOCUMENT.

Copyright © 2021 by PLCopen. All rights reserved.

Date: November 18, 2021

Total number of pages: 27

## PLCopen Guidelines for usage of Object-Oriented Programming

The following paper is a document created within the PLCopen Promotional Committee 2 – Training.

It summarizes the results of the PLCopen Promotional Committee meetings, containing contributions of its members as well as external sources:

<i>Name</i>	<i>Company</i>
Rene Simon	Hochschule Harz
Wolfgang Doll	Codesys
Yves de la Broise	IntervalZero
Anders Lekve Brandseth	Framo
Daniel Wall	Eaton
Filippo Venturi	SACMI
Georg Rempfler	Wyon
John Dixon	ABB
Dominik Franz	ABB
Ralf Dreesen	Beckhoff
Saele Beltrani	SACMI
Yo Takahashi	Mitsubishi Electric
Juliane Fischer	TUM
Eelco van der Wal	PLCopen

### Change Status List:

<i>Version number</i>	<i>Date</i>	<i>Change comment</i>
<b>V 0.1</b>	May 28, 2019	Document created by EvdW as input for the webmeeting
<b>V 0.2</b>	June 19, 2019	As result of the webmeeting June 11
<b>V 0.3</b>	July 2, 2019	As result of the webmeeting on July 2
<b>V 0.4</b>	July 30,2019	As result of the webmeeting on July 30
<b>V 0.5</b>	Oct. 30, 2019	As a result of the discussions upfront and webmeeting
<b>V 0.6</b>	Feb. 2, 2021	Conversion to boiler demo example
<b>V0.6c</b>	March 5, 2021	Added examples and text in Ch. 3
<b>V0.7</b>	April 22, 2021	After feedback and webmeeting
<b>V0.8</b>	May 6, 2021	After feedback and webmeeting
<b>V0.9</b>	May 20, 2021	After additional feedback and decisions at webmeeting
<b>V0.99</b>	May 27, 2021	Published as Release for Comments to the public
<b>V1.0</b>	Nov. 18, 2021	As a result of the feedback and webmeeting on Nov. 17

**Contents**

<b>1</b>	<b>INTRODUCTION TO THIS DOCUMENT</b> .....	<b>5</b>
1.1.	GOALS OF THIS WORKING GROUP .....	5
<b>2</b>	<b>INTRODUCTION TO OOP FEATURES OF THE 3<sup>RD</sup> EDITION OF THE IEC 61131-3 STANDARD.</b> .....	<b>6</b>
2.1.	THE 3 <sup>RD</sup> EDITION OF IEC 61131-3.....	6
2.2.	DEFINITIONS IN THE IEC 61131-3 STANDARD .....	6
2.3.	DIFFERENCES IN OOP LANGUAGES .....	8
<b>3</b>	<b>GENERIC APPROACH</b> .....	<b>9</b>
3.1.	GENERAL .....	9
3.2.	MODULES AND COMMANDS AS OBJECTS.....	9
<b>4</b>	<b>THE BASIC EXAMPLE: BOILER DEMO</b> .....	<b>11</b>
4.1.	OVERVIEW OF THE DIFFERENT PHASES .....	11
4.2.	APPLICATION DESCRIPTION .....	11
4.3.	THE EXAMPLE IN A CLASSICAL PROGRAM STYLE .....	12
4.4.	CONVERTING THE PROGRAM TO A PLCOPEN COMPLIANT VERSION .....	13
4.5.	ADDING ERROR BEHAVIOUR .....	16
4.6.	ADDING A SYSTEM ALARM LIST .....	24
<b>5</b>	<b>ADDITIONAL REFERENCES</b> .....	<b>26</b>
5.1.	FUNDAMENTALS FOR OOP GUIDELINES – CONSIDERATIONS.....	26
5.2.	NOTES ON PERFORMANCE CONSIDERATIONS .....	26
5.3.	NOTES FOR FURTHER INVESTIGATIONS .....	27

**List of figures**

Figure 1:	Representation of IModules and ICommands.....	9
Figure 2:	The boiler demo. ....	11
Figure 3:	Layout of the POU's .....	12
Figure 4:	Function block interface and state diagram of LConC .....	14
Figure 5:	Same interfaces for different devices.....	15
Figure 6:	Overview of the states in a simulation.....	16
Figure 7:	The AttachDevice Method of the Error handler .....	17
Figure 8:	The Error Handler FB with the list of the devices (FBs).....	18
Figure 9:	Initialization from the Error Handler in FB_Boiler .....	18
Figure 10:	The structure for Error Handler .....	19
Figure 11:	Main program with error handling .....	19
Figure 12:	Example of the error message for one FB .....	20
Figure 13:	Overview of the architecture .....	20
Figure 14:	Example of a cyclic action linked to the IActionControl .....	21
Figure 15:	Implementation of the Set action of the Behaviour Model .....	22
Figure 16:	The needed changes in the Device FB.....	22
Figure 17:	Corresponding changes to the attached devices .....	23
Figure 18:	UML Diagram of the new design.....	23
Figure 19:	Implementation of the delegation of the GetModelState.....	24
Figure 20:	String array for the error handler.....	24
Figure 21:	Example of an error string.....	25
Figure 22:	Example of creating the string in the error message .....	25
Figure 23:	The function FC_ErrorCode.....	25

**List of abbreviations:**

FB	Function Block conform IEC 61131-3
FC	Flow Controller
FD	Feedwater Drum
FT	Flow Transfer
LY	Level Yield
LC	Level Controller
OOP	Object-Oriented Programming
OT	Operation Technology
PID	Proportional, Integral Derivative control algorithm
PLC	Programmable Logic Controller

## **1 Introduction to this document**

The 3rd edition of the IEC 61131-3 standard makes the usage of Object-Oriented Programming, OOP, possible. Parallel to this, PLCopen has defined the concept of function block libraries including OOP and the PLCopen Common Behaviour Model (see on the PLCopen Website “[Creating PLCopen Compliant Libraries V1\\_0](#)” dated May 4, 2017), as well as sets of function blocks for motion control, safety, and communication.

Starting with OOP, several choices need to be made right from the beginning: are all function blocks in one Class? Think about the PLCopen Motion Control function blocks as an example, do we need then the AxisRef as the reference to the axis or even the MC\_ as part of the FB name? How do they contain the methods? And will we use only methods, or also direct access to variables? Is the state machine for the axis controlled by the methods? Are all axes’ objects with methods, and we access them only via these methods? How about interfaces? And do we prefer composition above inheritance?

All these choices give a different look & feel to the users across the different systems, different training guidelines and differences in maintenance. And this is where PLCopen wants to help and give guidance to create a more homogeneous programming methodology.

### ***1.1. Goals of this working group***

Overall, there is little information on how to use OOP for industrial control or the operation technology (OT). The goal of this group is to help here with the following recommendations:

- Guidance to using OOP in addition to the “classical” way.
- Provide the same look & feel in using OOP across the different platforms and implementations.
- Create generic design patterns for industrial control programming.
- The classical programming way should be possible to be used in addition to the OOP way (e.g. this can mean that we have to extend the classical FBs with interfaces, methods, properties, and maybe input and outputs).

As example, a boiler demo will be used to represent the different forms of programming.

## **2 Introduction to OOP features of the 3<sup>rd</sup> edition of the IEC 61131-3 standard.**

### ***2.1. The 3<sup>rd</sup> edition of IEC 61131-3***

Modern programming environments refer in many cases to OOP which is included in Python, C++, Objective-C, Smalltalk, Delphi, Java, Swift, C#, Perl, Ruby and PHP.

Important aspects of the 3<sup>rd</sup> Edition of the IEC 61131-3 standard are the integration of object-oriented features. This includes Classes, including Methods and Interfaces, OOP features for Function blocks, and Namespaces.

The 3<sup>rd</sup> edition of standard IEC 61131-3 – Programming Languages is approved as International Standard. This means that this edition is now official and is available as International Standard at [www.IEC.ch](http://www.IEC.ch). This standard is fully upwards compatible to IEC 61131-3, 2003 (2<sup>nd</sup> edition).

The inclusion of these extensions allows to implement PLC code with the same proven concepts and best practices as the ones used in (object oriented) software development using higher level languages since decades. These concepts yield easier readable, more modular, and finally more maintainable code. Moreover, by becoming more modern and similar to the mentioned highly distributed languages, a whole pool of software engineers may be attracted into the field of PLC development. And therewith, PLC code may become a first-class citizen in the software portfolio of enterprises.

Note that references are made to the IEC 61131-3 standard itself as is available at [www.IEC.ch](http://www.IEC.ch). These used references are for explanation only and for completeness the standard itself should be bought.

### ***2.2. Definitions in the IEC 61131-3 standard***

With these OOP extensions, there are several changes or new definitions in comparison to the 2<sup>nd</sup> edition. For instance, a program organization unit, originally defined as function, function block, or program, is extended to include a category “class”.

Function blocks and classes may contain methods. Moreover, they are inheritable what leads to the concepts of base types, derived function blocks and derived classes, respectively.

The most important new definitions are listed here:

<b>Name</b>	<b>Description</b>
<b>base type</b>	data type, function block type or class from which further types are inherited/derived.
<b>call</b>	language construct causing the execution of a function, function block, or method.
<b>class</b>	program organization unit consisting of: <ul style="list-style-type: none"> <li>• the definition of a data structure,</li> <li>• a set of methods (like subroutines) to be performed upon the data structure</li> </ul> A class is an implementation— a concrete data structure and collection of subroutines— while a type is an interface.
<b>derived class</b>	class created by inheritance from another class. Note 1 to entry: Derived class is also named extended class or child class. Note 2: this is of course very much in line with derived data type and derived function block type as defined in the IEC 61131-3-2003.

<b>dynamic binding</b>	situation in which the instance of a method call is retrieved during runtime according to the actual type of an instance or interface.
<b>inheritance</b>	creation of a new class, function block type or interface based on an existing class, function block type or interface, respectively.
<b>input variable</b>	variable which is used to supply a value to a program organization unit except for <b>class</b> .
<b>instance</b>	individual, named copy of the data structure associated with a function block type, class, or program type, which keeps its values from one call of the associated operations to the next.
<b>interface</b>	language element in the context of OOP containing a set of method prototypes. Example: it is similar to a motor flange: it describes holes diameter, distance, shaft size, but it is not a motor.
<b>method</b>	language element similar to a function that can only be defined in the scope of a function block or class type and with implicit access to instance variables of the function block instance or class instance. Example: a boiler can have a Fill method, a HeatUp method, each one performing a specific task.
<b>override</b>	keyword used with a method in a derived class or function block type for a method with the same signature as a method of the base class or function block type using a new method body
<b>output variable</b>	variable which is used to return a value from the program organization unit except for <b>classes</b> .
<b>program organization unit</b>	function, function block, <b>class</b> , or program.
<b>signature</b>	set of information defining unambiguously the identity of the parameter interface of a METHOD consisting of its name and the names, types, and order of all its parameters (i.e., inputs, outputs, in-out variables, and result type).

### 2.3. Differences in OOP languages

There are other OOP languages, and the table hereunder gives a short overview of the commonalities and differences.

<i>Language Properties</i>	<i>IEC 61131-3 2<sup>nd</sup> edition</i>	<i>IEC 61131-3 3<sup>rd</sup> Edition</i>	<i>C++</i>	<i>Java</i>	<i>C#</i>
<b>Multi languages</b>	+	+	-	-	-
<b>OOP/procedural mixed</b>	-	+	+	-	-
<b>Classes</b>	~ (FB)	+	+	+	+
<b>Methods</b>	~ (Actions)	+	+	+	+
<b>Interfaces</b>	-	+	-	+	+
<b>Polymorphism</b>	-	+	+/-	+	+
<b>Semantic Reference</b>	-	+ (Interfaces)	-	+	+
<b>Constructor/Destructor</b>	-/+	-/+	+	+	+
<b>Properties</b>	-	-	-	-	+
<b>Dyn. Memory („new“)</b>	-	-	+	+	+
<b>Access control</b>	~ (Variables)	~ (Variables)	+	+	+



## 3 Generic Approach

### 3.1. *General*

The PLCopen document "Guidelines for usage of Object Orientation" defines a set of currently generally accepted rules for good OOP design in the context of the IEC61131-3 3<sup>rd</sup> Edition.

It is advised to read the document [PLCopen Software Creation Guidelines: Creating PLCopen Compliant Libraries, Version 1.0 of May 4, 2017](#), which is downloadable from the PLCopen website. As part of this, we will use the SOLID principles in the automation technology environment. SOLID stands for:

- SRP — Single Responsibility Principle
- OCP — Open/Closed Principle
- LSP — Liskov Substitution Principle
- IPS — Interface Segregation Principle
- DIP — Dependency Inversion Principle

### 3.2. *Modules and Commands as Objects*

When designing applications and libraries in an OOP way, we differentiate between modules and commands.

- A *module* represents a part of an automation application including their software function and can also be nested hierarchically as required. For example, actors, sensors, or component assemblies of machines may be represented in code by "modules".
- A *command* represents a concrete action of a module. It is assigned to one module or to a group of modules that implement the same interface.

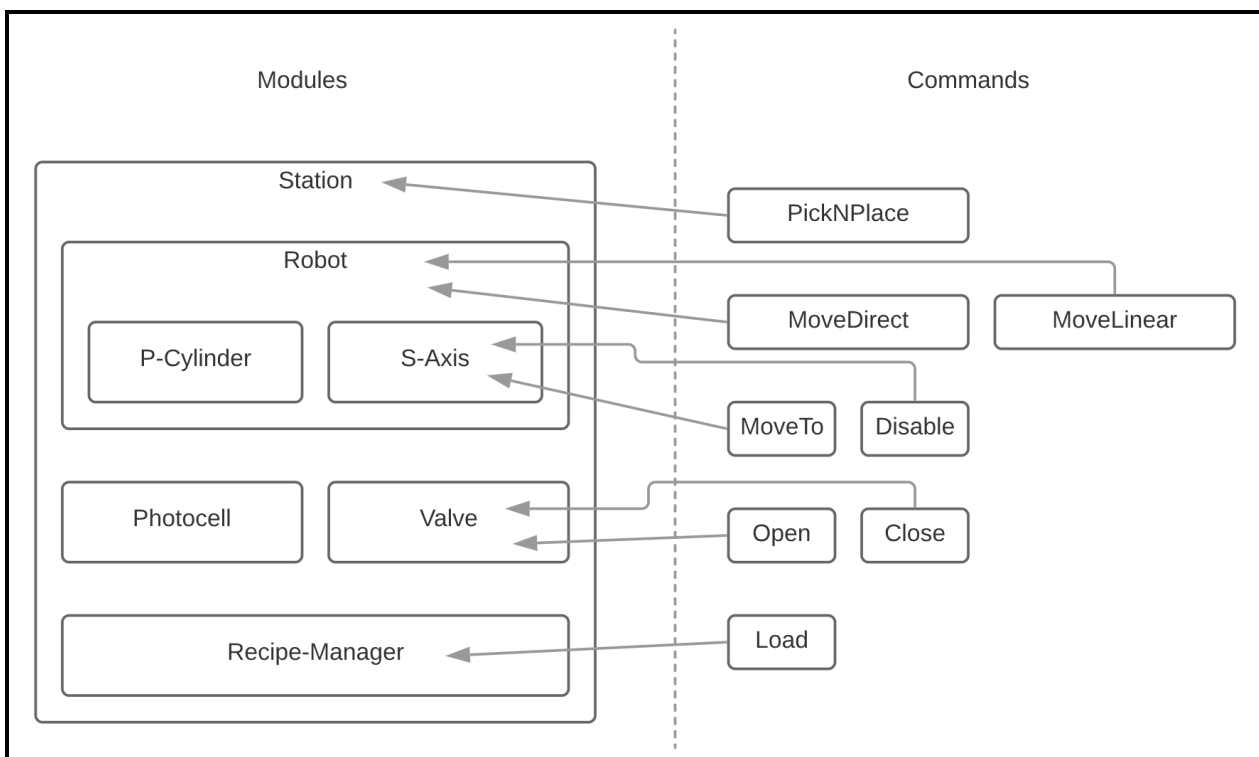


Figure 1: Representation of IModules and ICommands

Every module is implementing a generic *IModule* interface.

Some modules implement additionally a more specialized interface. For example, the modules that provide the functionality of an axis, are implementing an *IAxis* interface.

Every command will implement a generic *ICommand* interface and will follow the design guidelines for PLCopen conforming function blocks.

The axis related commands provide an input of type *IAxis* to get the reference to a related axis instance they will act upon.

With this construction it is possible to bring two worlds reconcilable together.

For example, the classic PLCopen Motion function blocks can be reused in their traditional form. At the same time, these function blocks can also be controlled via their modern interface. This opens new possibilities for innovative solutions such as reconfiguring applications at runtime.

## 4 The basic example: boiler demo

### 4.1. Overview of the different phases

The original example was used for a demonstration at the Hanover Fair in cooperation with the OPC Foundation.

This original code is transferred into a format that is compliant with the PLCopen Guidelines, so based on the harmonized function block interfaces.

The next step is to convert this to Object Orientation, which is followed by the addition of an error handler and alarm management.

The program is written in Structured Text and the code can be found on the [PLCopen website](https://www.plcopen.com/) as a Codesys project.

### 4.2. Application description

The application of the example “Boiler Demo” consists of a simulated boiler as shown in Figure 2: The boiler demo.

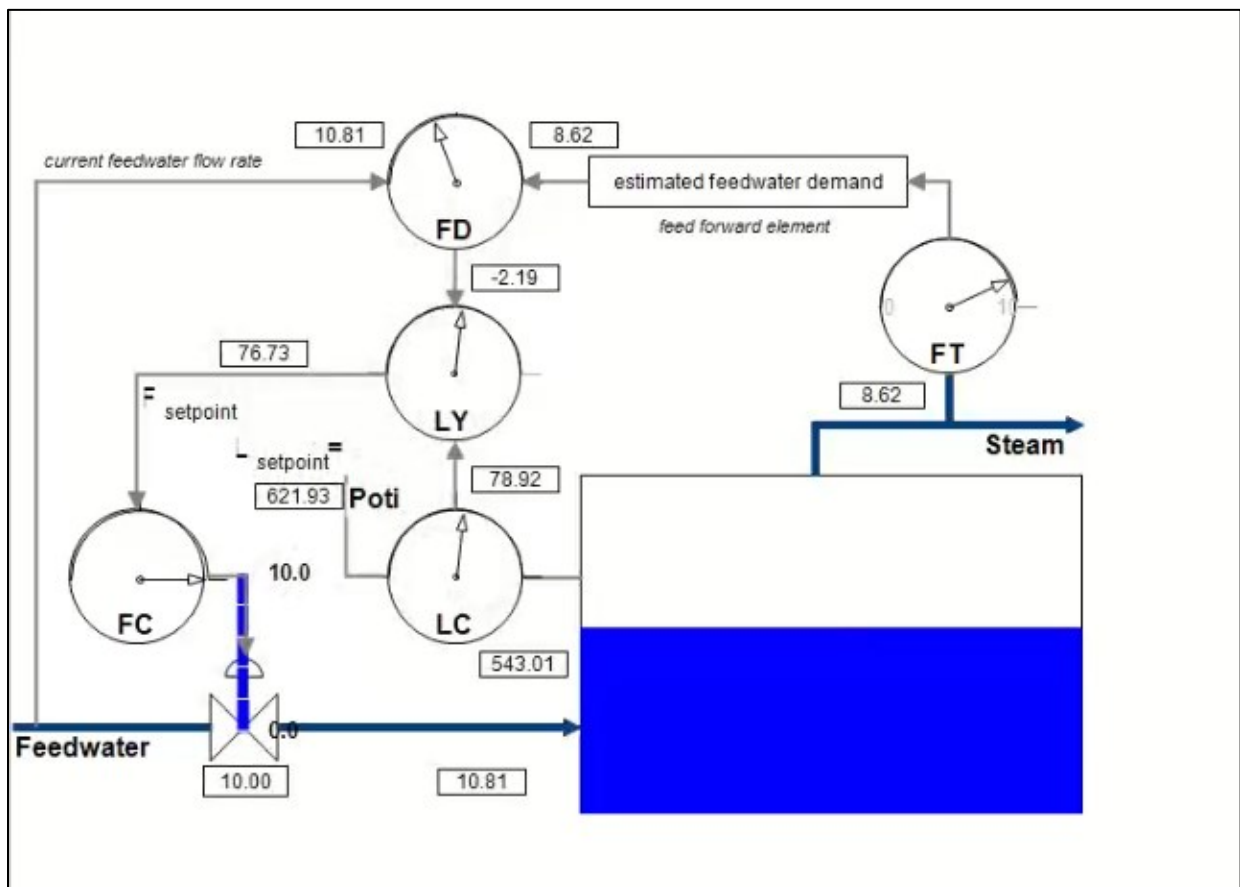


Figure 2: The boiler demo.

### 4.3. The example in a classical program style

The application simulates a boiler process with a basic PID controller supported by flow control (FC) and level control (LC) including a random generator for disturbances in the feeds.

There are two programs, 4 user derived functions and 17 user derived function blocks.

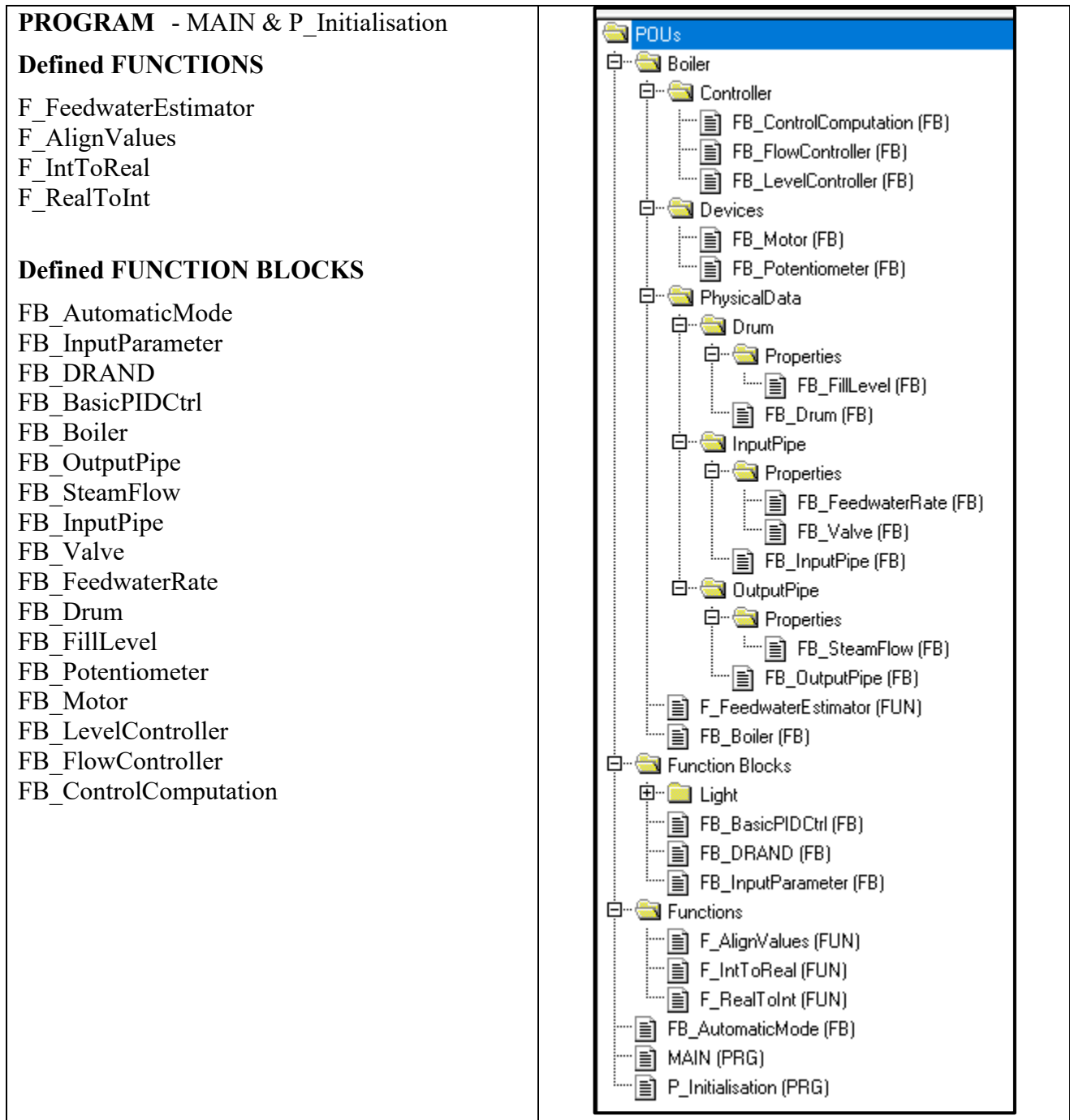


Figure 3: Layout of the POU's

#### **4.4. *Converting the program to a PLCopen compliant version***

The design is already well structured with regards to the first SOLID rule "Single Responsibility Principle". There is a separate FB for every hardware element, as well as for every new task.

Unfortunately, this application has not yet been designed according to an OOP approach, so the other rules are not adhered to.

There is no interface and there is no abstraction. All FBs were programmed and used directly on the detail level.

There are two possible approaches to adapting this application:

1. A complete redesign of the application according to OOP approaches and rules. An exceptionally good architecture can be achieved in this way.

Unfortunately, this solution is very time-consuming and expensive, which makes it difficult for a machine builder to get a completely new software generation approved by the management.

2. Therefore, we try to present a second possibility to partially restore the application. Due to the good starting point, this step can be considered. For (brownfield) applications that have grown over many years and are too nested, it is better to rely on a new generation.

The original building blocks were programmed on the lowest level of detail. The first step is to achieve standardization and abstraction of the building blocks. The PLCopen guidelines for function block libraries offer a clear interface and standardized behaviour for this purpose. The essence of the guidelines is the so-called "behaviour model". So, the first step was to convert the original program including the defined function and function blocks into the applicable behaviour model, including the state machine and the error behaviour. The OOP structure of the PLCopen Common Behaviour Model offers us excellent opportunities to implement this in just a few steps and changes.

Also, we try to use the "Open/Closed Principle" and "Dependency Inversion Principle". In addition, a first interface is defined which can be used for future extensions.

Because the building blocks can be extended by the defined behaviour models, in this case the LConC is used, they get an abstract higher level that does not depend on the details of the lower level. The LConC also defines an interface which must now be implemented by the old FB's. In the first step, however, only the code from the body of the old FB's has to be copied into the Cyclic Action method.

As a result, the "Open/Closed Principle" is adhered to, the block retains its functionality, and the existing software code does not have to be changed. This also gives classic style software developers the opportunity to slowly get used to the changeover. Since the FB's are still 1-to-1 identical, and the implementation has not been changed significantly either. But the modules are now open for expansion. Also, incorrect user entries and parameterization can now be monitored in the StartAction (as used in the edge triggered function blocks as defined in the PLCopen library document), in order to increase the stability. In addition, the modules now have an abstract and uniform interface with which further extensions can be made that follow the "Interface Segregation Principle".

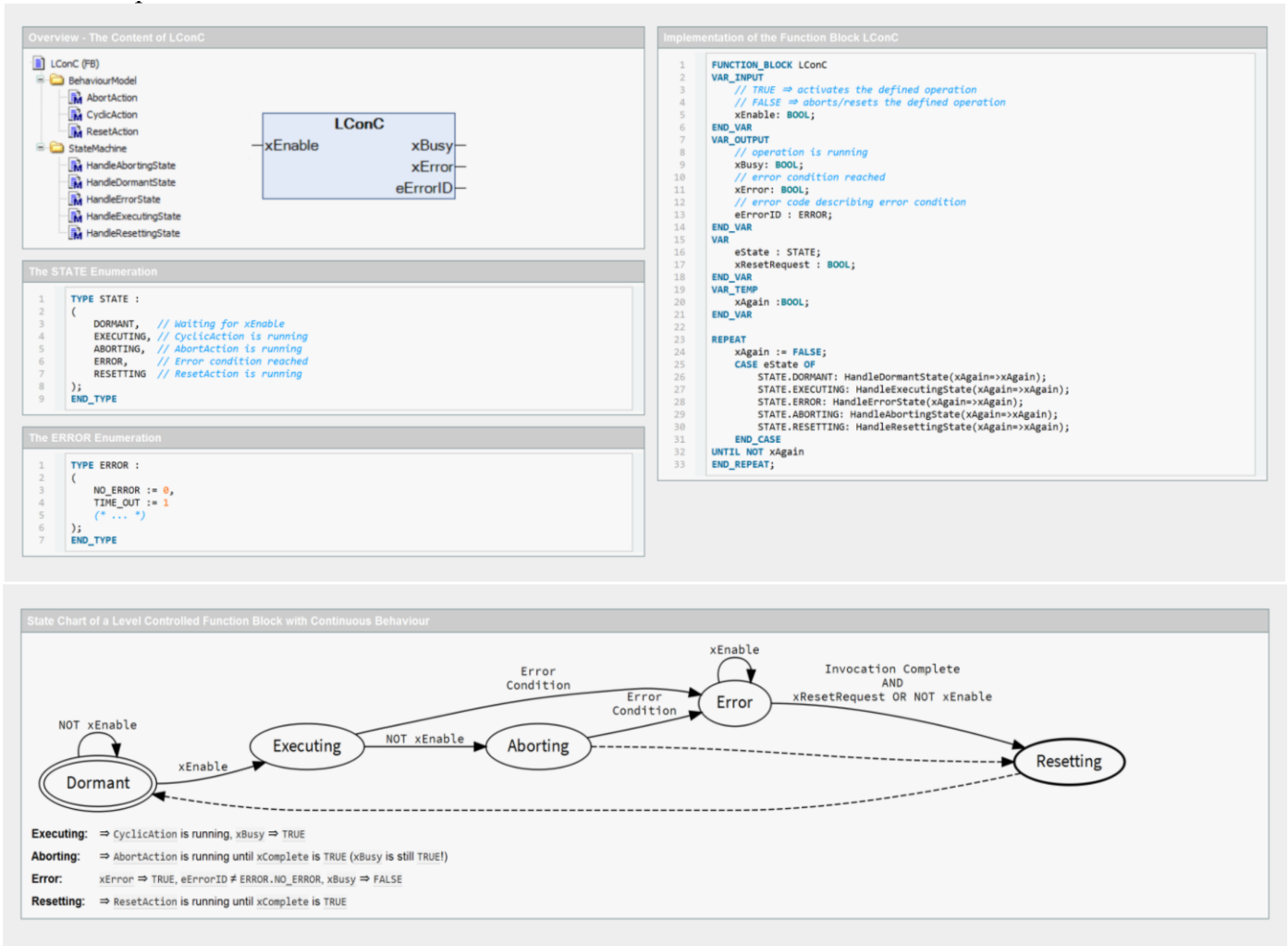


Figure 4: Function block interface and state diagram of LConC



Figure 5: Same interfaces for different devices

The consistent design of the Common Behaviour Model enables us to create an abstraction that is even independent of the state machine chosen. As all state machines, it is derived from an abstract class with a general defined interface.

This now offers the possibility to manage the FBs in a uniform way at a higher level. For example, to implement an error handler that can manage all FBs regardless of their implementation (Interface Segregation Principle).



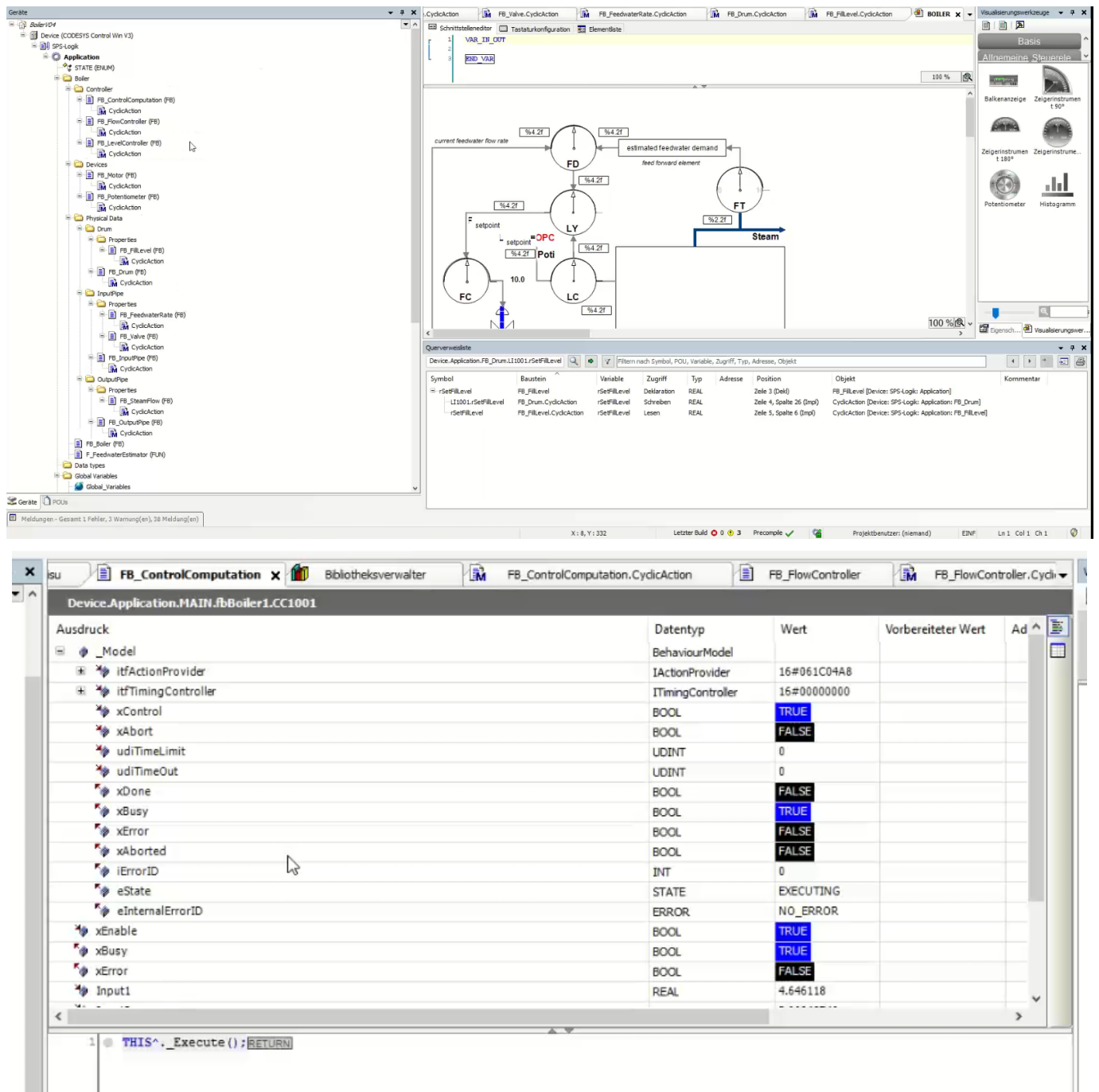


Figure 6: Overview of the states in a simulation

Be aware that the main program here is not usable in real applications as it is missing a start-up sequence, a shutdown sequence, and error handling. This is the part that is customized for every application. These additions will show further benefits of the OOP due to the re-use.

#### 4.5. Adding Error Behaviour

As an example of this, the error behaviour is added as a central error handling function block, which traces all the used function blocks. This is shown in 2 ways: via *inheritance* and via *composition*.



We will not use the different detailed FBs directly, but we will use our abstract Interface IBehaviourModel (see Figure 5: Same interfaces for different devices). This interface gives access to the Outputs from the function blocks which include xDone, xBusy, xError, xAborted, iErrorID. This is the idea from the Interface Segregation Principle, and with this, all the FBs from the Common Model Library can now be connected. This is not corresponding to only one FB-type, nor to one Behaviour Model. All described Behaviours and types can be connected without any adaptation to this ErrorHandler.

The idea is to map all the function blocks to an array which is included in the error handler, to enable an iteration through all the function blocks. In this Array we will not have a Pointer to the different FBs on detail level, but we will have the basic interface IBehaviour Model only. With the Method GetModelState we can monitor the required information from all FB. See Figure 8: The Error Handler FB with the list of the devices (FBs).

The Error Handler has two methods to link it to the devices (FBs): AttachDevice and DetachDevice. The error handler has these two methods for attach and detach, reflecting it in the device list. The error handler references to this list (see Figure 10: The structure for Error Handler).

We can now monitor all FBs with the same Method GetModelState and build a structure with Status Information so that one can see the status from all the function blocks.

```

1  METHOD AttachDevice
2  (*Register a Device by the BehaviourModel Interface*)
3  VAR_INPUT
4      (*Device which will be monitor from Error Handler*)
5      IDevice : CBML.IBehaviourModel;
6  END_VAR
7
8  VAR
9      uiLoop: UINT; //Help Variable for the For Loog
10 END_VAR

1  //go to the next free input in the arry and save the Interface
2  FOR uiLoop := 0 TO GVL_Const.MAX_Observer DO
3      IF IDevice <> 0 AND THIS^.aritifObserver[uiLoop]= 0 THEN
4          THIS^.aritifObserver[uiLoop] := IDevice;
5          EXIT;
6      END_IF
7  END_FOR

```

Figure 7: The AttachDevice Method of the Error handler

```

1 //Error Handler which will be monitor all FBs which has an behaviour
2 FUNCTION_BLOCK FB_ErrorHandler IMPLEMENTS I_ErroHandlerSubject
3
4 VAR
5 //Help Variables for Attaching and read Information from Devices
6 _stInfoState : ARRAY [1..GVL_Const.MAX_Observer] OF tsCBMLInfo; //State from all existing FB
7 aritfObserver : ARRAY [0..GVL_Const.MAX_Observer] OF CBML.IBehaviourModel; // Interface list from all FB which will be monitord
8 uiLoop : UINT; //Help Varialbe to get access to the Interface during a For Loop
9
10 END_VAR
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Figure 8: The Error Handler FB with the list of the devices (FBs)

The definition of the Attach and Detach Methods is done in an Interface which is implemented from the Error Handler. This interface can now be used by the Boiler to attach all the instances to the Error Handler.

```

FUNCTION_BLOCK FB_Boiler
VAR_INPUT
(*Adjustable values*)
rSetFillLevel : REAL;
rSteamDemand : REAL;
itfErrorHandler : I_ErroHandlerSubject ;
END_VAR

IF NOT bInitDone THEN
bInitDone := TRUE;
Pipel001.xEnable := TRUE;
Druml001.xEnable := TRUE;
Pipel002.xEnable := TRUE;
LC1001.xEnable := TRUE;
CC1001.xEnable := TRUE;
FC1001.xEnable := TRUE;

IF itfErrorHandler <> 0 THEN
itfErrorHandler.AttachDevice(IDevice := Pipel001);
itfErrorHandler.AttachDevice(IDevice := Druml001);
itfErrorHandler.AttachDevice(IDevice := FC1001);
itfErrorHandler.AttachDevice(IDevice := LC1001);
itfErrorHandler.AttachDevice(IDevice := Pipel002);
itfErrorHandler.AttachDevice(IDevice := CC1001);
END_IF
END_IF

```

Figure 9: Initialization from the Error Handler in FB\_Boiler

```

TYPE tsCBMLInfo :
STRUCT
    xCommit      : BOOL;
    xDone        : BOOL;
    xBusy        : BOOL;
    xError       : BOOL;
    xAborted     : BOOL;
    iErrorID     : INT;
    eState       : CBML.STATE;
    strName      : STRING;
END_STRUCT
END_TYPE

```

Figure 10: The structure for Error Handler

In the main program we can now build an instance from the Error Handler and connect the Error Handler to the Boiler instance:

```

1  PROGRAM MAIN
2  VAR
3      fbBoiler1   : FB_Boiler;          (*~ (OPC : 1 : enabled for OPC) *)
4      fbInput1   : FB_InputParameter;
5      fbErrorHandler : FB_ErrorHandler;
6      bInitDone  : BOOL := FALSE;
7      test       : FB_DeviceBasic;
8  END_VAR
9
10
11
12
13
14
15
16
17
18
19
20
21

```

```

1  IF NOT bInitDone THEN
2      P_Initialisation();
3      bInitDone := TRUE;
4
5      //
6  END_IF
7
8  counter := counter +1;
9
10 (*Get Input*)
11 fbInput1();
12
13 (*Simulate Boiler*)
14 fbBoiler1(
15     rSetFillLevel := fbInput1.rFillLevel,
16     rSteamDemand  := fbInput1.rSteamDemand,
17     itfErrorHandler := fbErrorHandler
18 );
19
20 (*Monitor all Instance by IBehaviourModell*)
21 fbErrorHandler();

```

Figure 11: Main program with error handling

[-] <b>_stInfoState</b>	ARRAY [1..GVL_Const.MAX_Observer] OF tsCBMLInfo	
[-] <b>_stInfoState[1]</b>	tsCBMLInfo	
<b>xCommit</b>	BOOL	FALSE
<b>xDone</b>	BOOL	FALSE
<b>xBusy</b>	BOOL	TRUE
<b>xError</b>	BOOL	FALSE
<b>xAborted</b>	BOOL	FALSE
<b>iErrorID</b>	INT	0
<b>eState</b>	STATE	EXECUTING
<b>strName</b>	STRING	'Device.Application.MAIN.fbBoiler1.Pipe1001'
[+] <b>_stInfoState[2]</b>	tsCBMLInfo	
[+] <b>_stInfoState[3]</b>	tsCBMLInfo	
[+] <b>_stInfoState[4]</b>	tsCBMLInfo	
[+] <b>_stInfoState[5]</b>	tsCBMLInfo	

Figure 12: Example of the error message for one FB

The error handler is now a standard component of all FBs. It handles all the devices centrally without separately calling all the instances of the used FBs, while being able to centrally read all the relevant information of the FBs.

This concept makes it easier to control all the function blocks with one list only. In an alternative way, one must connect to all the FBs and point to all the different available function blocks. Now you have only one interface with one list from which you can control all listed function blocks.

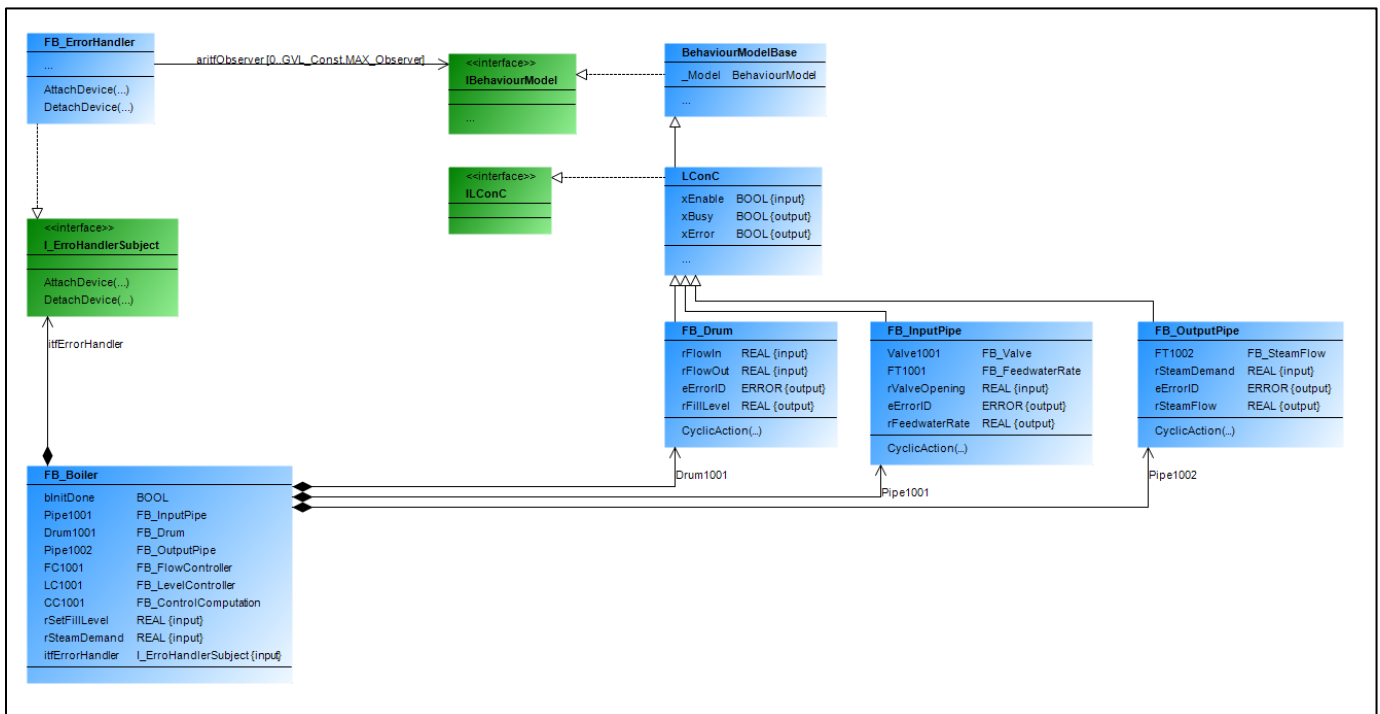


Figure 13: Overview of the architecture

The model in Figure 13: Overview of the architecture shows that LConC (and its interface) inherits from the BehaviourModelBase and so the interface IBehaviourModel.

The I\_ErrorHandler interface is linked to all the FBs which are used in the demo as well as to the FB\_ErrorHandler. In this way, the ErrorHandler can connect to all applicable FBs.

Using inheritance as a base design for FBs gives a nice abstract control option but shows some limitations. We are no longer able to extend the detailed device FB on an abstract Level. The reason for this limitation is that one cannot use inheritance on the abstract Level BehaviourModelBase as it uses inheritance already.

This is the point to check, if we can optimize this design by using **composition** instead of **inheritance** for the Behaviour Model (see 5.1 Fundamentals for OOP Guidelines – considerations). For this, we must show how we can define the Behaviour Model as a member of our FBs but define the Method for the different states in the same way as before. (Open/Closed Principle).

A feature from the Common Behaviour Model can help us to implement this design. The FBs in the Library support the connection of an Action Controller - see red lines in figure 16 - which allows a FB which implements the IActionControl to do a Method call instead of the methods which come from the Behaviour Model.

The idea is now to create a new Abstract Device Base Class, which has as a new feature 'Name' from the FB as a complete Path. To support now the Composition from Behaviour Model, our Base Class implements the IActionControl. To follow the "Open/Closed Principle", it is recommended to implement the IActionProvider Interface also. This allows the detailed Device FB to work with the behaviour implementation on the same way as before.

In the Method from the Action Controller, one calls a Method, which comes from the Action Provider.

The following example shows this situation for the method ControlCyclicAction:

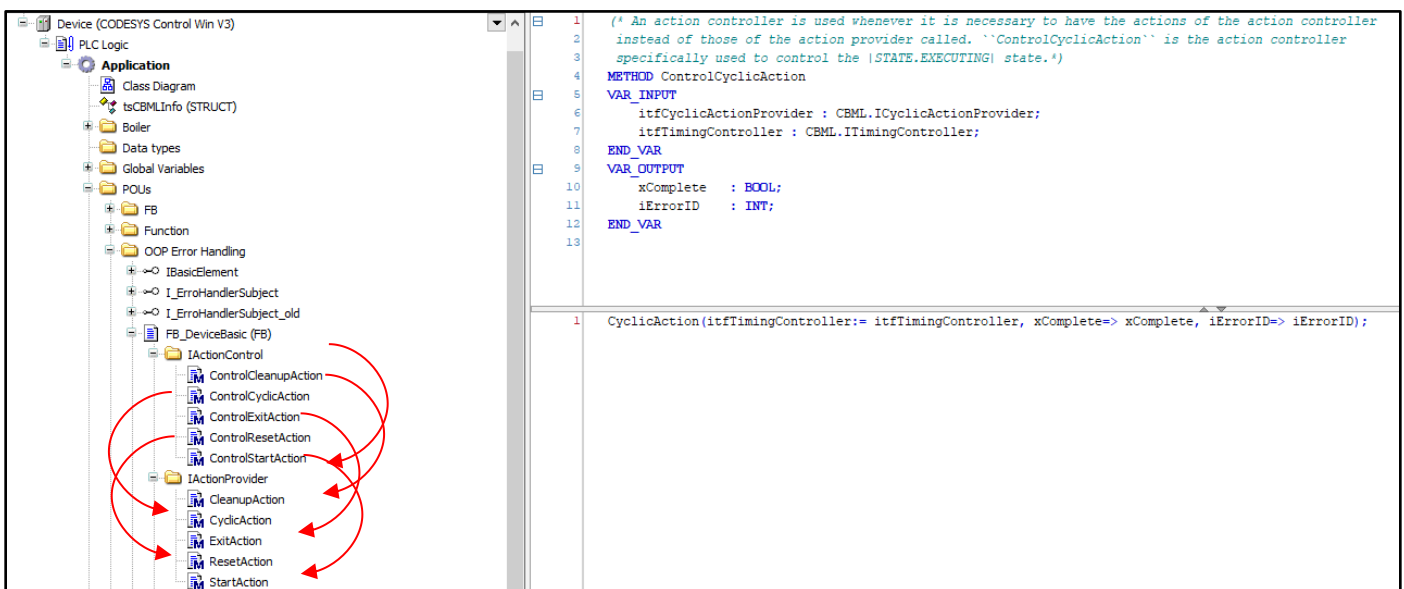


Figure 14: Example of a cyclic action linked to the IActionControl

The Base Class will also get a Property to 'get' and 'set' the IBehaviourModel. In the Setter the IActionControl will directly link to the instance of the created Behaviour Model.

```

FB_DeviceBasic.IBehaviourModel.Set
1  VAR
2  END_VAR
3

```

---

```

1  _iBehaviourModell := IBehaviourModel;
2  IBehaviourModel.ActionController := THIS^;

```

Figure 15: Implementation of the Set action of the Behaviour Model

The detailed FB can now decide, which Behaviour Model it will use. The only applicable changes are the following:

- Change the inheritance from the LConC to our own Base Class.
- Create an Input variable for xEnable or xExecute based on your decided Behaviour.
- Create an instance from the Behaviour you want to use.
- Change the Execution from the Model in the Body and give the member Behaviour instance to the Property.

See the following example:

```

FB_OutputPipe
1  FUNCTION_BLOCK FB_OutputPipe EXTENDS FB_DeviceBasic
2  VAR_INPUT
3      rSteamDemand : REAL;
4      xEnable : BOOL;
5  END_VAR
6  VAR_OUTPUT
7      eErrorID : ERROR; // error code describing error condition
8      rSteamFlow : REAL;
9  END_VAR
10 VAR
11     FT1002 : FB_SteamFlow;
12     StateBehaviour : CBML.LConC;
13 END_VAR
14

```

---

```

1  THIS^.IBehaviourModel := StateBehaviour;
2  StateBehaviour(xEnable:= xEnable, xBusy=> xBusy, xError=> xError);

```

Figure 16: The needed changes in the Device FB

To attach our FBs to the Error Handler we use our property so that the interface to the Error Handler can still be the same.

```

IF NOT bInitDone THEN
  bInitDone := TRUE;
  Pipe1001.xEnable := TRUE;
  Drum1001.xEnable := TRUE;
  Pipe1002.xEnable := TRUE;
  LC1001.xEnable := TRUE;
  CC1001.xEnable := TRUE;
  FC1001.xEnable := TRUE;

  IF itfErrorHandler <> 0 THEN
    itfErrorHandler.AttachDevice (IDevice := Pipe1001.IBehaviourModel);
    itfErrorHandler.AttachDevice (IDevice := Drum1001.IBehaviourModel);
    itfErrorHandler.AttachDevice (IDevice := FC1001.IBehaviourModel);
    itfErrorHandler.AttachDevice (IDevice := LC1001.IBehaviourModel);
    itfErrorHandler.AttachDevice (IDevice := Pipe1002.IBehaviourModel);
    itfErrorHandler.AttachDevice (IDevice := CC1001.IBehaviourModel);
  END_IF
END_IF
(*Simulate waterflow.*)
Pipe1001/

```

Figure 17: Corresponding changes to the attached devices

The architecture is shown in Figure 18:UML Diagram of the new design. It has now the advantage, that we can use the new abstract FB to add new features which are related to all FBs in a consistent way. Code redundancy is also avoided in every FB. Furthermore, it provides the possibility to inject the Behaviour Model from outside with the Property BehaviourModel.

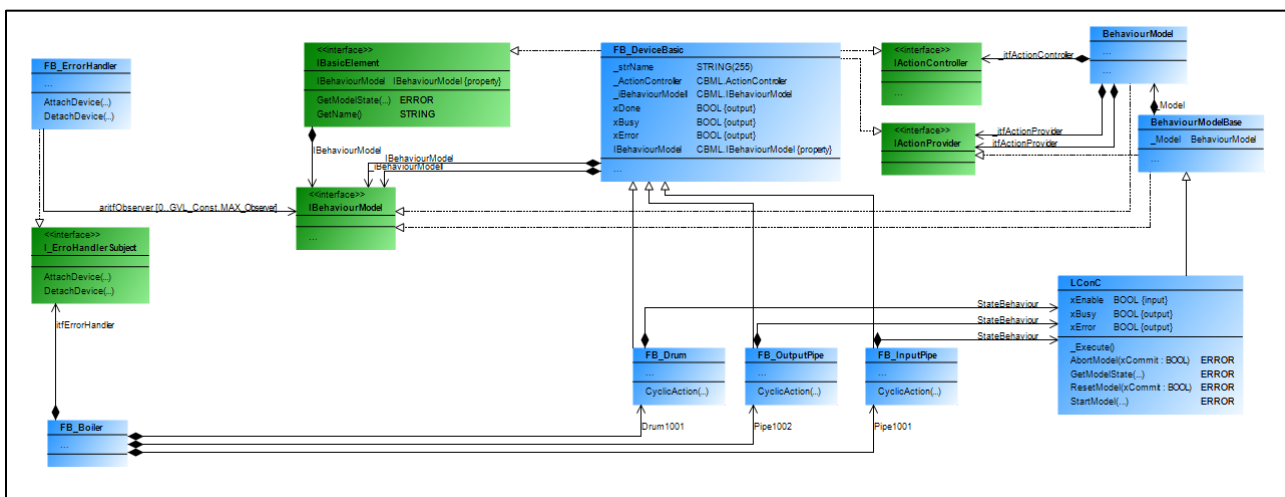


Figure 18: UML Diagram of the new design

Based on this, it is easy to create additional functionalities, like an alarm list, as done in 4.6 Adding a system alarm list.

## 4.6. Adding a system alarm list

One can differentiate between process alarms and system alarms, to filter them for different users (e.g. tank operator and system administrator). This example deals with system alarms, but the process alarms can be created in a similar way. One can even extend the FBs with an output like ProcessAlarm and handle these separately.

To create an error list it may be useful, to know in which FB the Error is happening or even the complete instance path.

For that, we have implemented the name and a Method to have access to the Name in our Abstract Base class. The name is generated here by a feature of the Codesys Development System, but it can also be set via a Property, for example during the Init Routine in the Boiler FB. Using an Interface which describes the Method directly follows the Interface Segregation Principle.

To get access to this method, we need a new or extended Interface for the Error Handler. Changing the interface is normally not the best style but, in this case, we can directly implement the interface in such a way, so that it includes only the necessary information. In this sample these are the GetModelState and the GetName Methods. So, it makes sense to adapt the Interface in the Attach / Detach Method and the Array of Interfaces in a way that they do not use the IBehaviourModel but the new Interface IBasicElement. The Method GetModelState will delegate the request to the internal set State Machine, as shown hereunder (Note: ERROR type is defined in Figure 4: Function block interface and state diagram of LConC):

```

1  METHOD GetModelState : ERROR
2  VAR_INPUT
3    xCommit : BOOL;
4  END_VAR
5  VAR_OUTPUT
6    xDone : BOOL;
7    xBusy : BOOL;
8    xError : BOOL;
9    xAborted : BOOL;
10   iErrorID : INT;
11   eState : CEML.STATE;
12 END_VAR
13
14 IF _iBehaviourModell <> 0 THEN
15   _iBehaviourModell.GetModelState(xCommit:= xCommit, xDone=> xDone, xBusy=> xBusy, xError=> xError, xAborted=> xAborted, iErrorID=> iErrorID, eState=> eState);
16 END_IF

```

Figure 19: Implementation of the delegation of the GetModelState

This is implemented in a way that all errors which happen are presented in a string array, which users can read.

astrActive	ARRAY [1..10] OF S...	
astrActive[1]	STRING(255)	"
astrActive[2]	STRING(255)	"
astrActive[3]	STRING(255)	"
astrActive[4]	STRING(255)	"
astrActive[5]	STRING(255)	"
astrActive[6]	STRING(255)	"
astrActive[7]	STRING(255)	"
astrActive[8]	STRING(255)	"
astrActive[9]	STRING(255)	"
astrActive[10]	STRING(255)	"
uiNumOfActError	UINT	0
_strName	STRING(255)	'Device.Application.Global_Variables.ErrorHandler'

Figure 20: String array for the error handler



astrActive	ARRAY [1..10] OF STRING(255)	
astrActive[1]	STRING(255)	'In FB Device.Application.MAIN.fbBoiler1.Pipe1001the following error is activ:Error Pipe'
astrActive[2]	STRING(255)	"

Figure 21: Example of an error string

Figure 22: Example of creating the string in the error message, shows a small function as part of the error handling which converts the Error into a clear string, only on the rising and falling edge of the xError Output.

Combined with a function FC\_ErrorCode which converts the iErrorID to a string (see Figure 23: The function FC\_ErrorCode).

```

14 IF NOT _stErrorState[uiLoop] AND _stInfoState[uiLoop].xError THEN
15   _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
16   //Build Error String
17   FOR uiHelp := 10 TO 1 BY -1 DO
18     IF uiHelp > 1 THEN
19       astrActive[uiHelp] := astrActive [uiHelp-1];
20     ELSIF uiHelp = 1 THEN
21       astrActive[uiHelp] := CONCAT(CONCAT(CONCAT ('In FB ',_stInfoState[uiLoop].strName ),'the following error is activ:'),FC_ErrorCode(_stInfoState[uiLoop].iErrorID) );
22     END_IF
23   END_FOR
24 ELSIF _stErrorState[uiLoop] AND NOT _stInfoState[uiLoop].xError THEN
25   _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
26   //Build Error String
27   FOR uiHelp := 10 TO 1 BY -1 DO
28     IF uiHelp > 1 THEN
29       astrActive[uiHelp] := astrActive [uiHelp-1];
30     ELSIF uiHelp = 1 THEN
31       astrActive[uiHelp] := CONCAT(CONCAT(CONCAT ('In FB ',_stInfoState[uiLoop].strName ),'the following error is deactiv:'),FC_ErrorCode(_stInfoState[uiLoop].iErrorID) );
32     END_IF
33   END_FOR
34 END_IF

```

Figure 22: Example of creating the string in the error message

```

FC_ErrorCode x
1 FUNCTION FC_ErrorCode : STRING
2 VAR_INPUT
3   iErrorID : INT;
4 END_VAR
5 VAR
6 END_VAR
7
8 CASE iErrorID OF
9   GVL_ErrorDef.Error_CC : FC_ErrorCode := 'Error Control Computation';
10  GVL_ErrorDef.Error_Drum : FC_ErrorCode := 'Error Drum';
11  GVL_ErrorDef.Error_FC : FC_ErrorCode := 'Error Flow Controller';
12  GVL_ErrorDef.Error_LC : FC_ErrorCode := 'Error Level Controller';
13  GVL_ErrorDef.Error_Motor : FC_ErrorCode := 'Error Motor';
14  GVL_ErrorDef.Error_Pipe : FC_ErrorCode := 'Error Pipe';
15  GVL_ErrorDef.Error_Valve : FC_ErrorCode := 'Error Valve';
16 END_CASE

```

Figure 23: The function FC\_ErrorCode

A possible improvement in the design can be to change the way the ErrorHandler checks all the FBs in a cyclic way. One idea can be to use here the Observer Pattern in this way that the ErrorHandler is connected to the Devices as an Observer and the Devices will send out a notification if the Status is changed.

## 5 Additional References

### 5.1. *Fundamentals for OOP Guidelines – considerations*

Within this document, some key aspects shall be taken into consideration:

- Good design but bad performance -- Which details should be considered in order not to influence the performance too much?
- Inheritance and composition -- How inheritance weakens encapsulation  
Many respected OO designers<sup>1</sup> have stated that composition should be given preference over inheritance whenever possible and that inheritance should only be used when necessary. However, this statement is somewhat shortened. The call that composition should be used whenever possible obscures the actual problem. Experience shows that composition is in most cases more appropriate than the use of inheritance. The fact that composition seems more appropriate in most cases does not mean that the use of inheritance is to be avoided. Both methods, inheritance, and composition, have their justification, but in the right context.
- Avoidance of dependencies  
The introduction of dependencies at runtime (Dependency injection) plays a central role in the discussion whether the composition should be given a priority over inheritance. It is very important to note that this is only a discussion. The purpose of this discussion is not necessarily to find the "optimal" way to design a class, but to provide food for thought on how to deal adequately with the decision for or against the use of inheritance or composition.

### 5.2. *Notes on performance considerations*

With the change in the architecture, it is very important not to lose sight of the special situation of a PLC compared to a PC/server system.

Overall, there are three actors who could still do a little more for the application of OOP:

- Hardware design  
Currently we can see a big difference in the measurements of the runtime. Depending on the hardware, a virtual method call takes drastically more time or is of little importance compared to a normal subroutine call. Of course, a small runtime extension in a cyclic executed PLC operation is often of great importance. The decisive factor here is the connection of the data/code memory to address and data buses and the possibilities of proper cache mechanisms. Especially the cache can help us to save time in cyclic operation. When designing a controller today, it is no longer just a question of how much time 1024 IL instructions require!
- Compiler design  
The code generator for a PLC must ensure that the code parts of an application are in relation to the cache structure in the neighborhood. So, code and data which are used cyclically can be processed very fast without a new transfer from the main memory. The implementation of interfaces, methods, and properties but also references must be adapted to the needs of a PLC. So, use cases like OnlineChange and

---

<sup>1</sup> Like: *Design Patterns: Elements of Reusable Object-Oriented Software* by [ErichGamma](#), [RichardHelm](#), [RalphJohnson](#), and [JohnVlissides](#) (the [GangOfFour](#)).

Debugging must be considered early and well. The allocation of memory on the stack (local variables) must not create a problem when using methods. The initialization of extensive data structures can become a problem (e.g., when cyclically called) and should be accordingly controllable by the user.

▪ **Software design**

The keywords ABSTRACT, FINAL, PRIVATE, ... must be used consciously.

This allows the developer to influence essential aspects of the runtime of his software at a very early stage. The developer must keep an eye on whether and when the initialization of his local variables must take place. The PLC development environment should provide a way to prevent the initialization of a set of specific variables so that "much" time can be saved here.

### ***5.3. Notes for further investigations***

- Synchronous versus a-synchronous communication / run-time environments
- Should we introduce dynamic memory allocation to the classical control world?
- Show no preferences for already existing implementations and programming styles.
- It can be that a programming paradigm shift is unavoidable.