Обучающие материалы PLCopen

Руководство по применению объектно-ориентированного подхода

Версия 1.0, официальный релиз

Дата публикации: 18.11.2021

Переводчик:	Е.Кислов
Редактор:	А.Осинский
Версия перевода:	1.1
Дата публикации:	27.12.2021

Оглавление

Список авторов и история версий

Данный документ является официальным документом организации **PLCopen**:

Руководство по использованию ООП

Он представляет собой результат работы комитета по продвижению и обучению **PLCopen** и включает вклад каждого из участников:

Участник	**Компания**
Rene Simon	Hochschule Harz
Wolfgang Doll	3S / 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

История версий

Версия	**Дата**	**Описание**
0.1	28.05.2019	Черновик, подготовленный Eelco van der Wal в качестве отправной точки для первой видеоконференции
0.2	19.06.2019	Дополнения по результатам видеоконференции 11 июня
0.3	02.07.2019	Дополнения по результатам видеоконференции 2 июля
0.4	30.07.2019	Дополнения по результатам видеоконференции 30 июля
0.5	30.10.2019	Дополнения по результатам видеоконференции и личного общения
0.6	02.02.2021	Переработка документа в связи с решением использовать пример управления бойлером в качестве основы
0.6c	05.03.2021	Добавлены примеры и пояснения в п. 3
0.7	22.04.2021	Дополнения по результатам видеоконференции и обратной связи
0.8	06.05.2021	Дополнения по результатам видеоконференции и обратной связи
0.9	20.05.2021	Дополнения по результатам обратной связи и решений, принятых на видеоконференции
0.99	27.05.2021	Предварительная публикация для получения комментариев (RFC)
1.0	18.11.2021	Дополнения по результатам видеоконференции 17 ноября и обратной связи

1. Введение

Третья редакция стандарта МЭК 61131-3 сделала возможным использование объектно-ориентированного подхода (ООП) при разработке приложений для программируемых логических контроллеров (ПЛК). Одновременно с этим организация PLCopen разработала концепцию моделей поведения функциональных блоков (ФБ), которые учитывают возможности ООП, алгоритмы для управления движением, промышленной безопасности и обмена данными.

При использовании ООП необходимо уже на ранней стадии работы над проектом принять ряд решений: будут ли все ФБ принадлежать одному классу? На примере ФБ управления движением от PLCopen – нужна ли AxisRef в качестве ссылки на ось или префикс MC_ в имени ФБ? Какие методы будут включать в себя ФБ? Будем ли мы использовать прямой доступ к переменным, помимо использования методов? Будет ли машина состояний управляться методами? Будут ли оси представлены объектами с методами? Будет ли доступ к осям осуществляться только при помощи методов? А что насчет интерфейсов? Что предпочтительнее – композиция или наследование?

Такое многообразие вариантов приводит к различным способам разработки и обслуживания, характерным для разных сред программирования, и требует различных методик обучения. Поэтому PLCopen решила разработать единую согласованную методологию по разработке проектов для ПЛК с использованием ООП.

1.1. Цели рабочей группы

В целом, информации об использовании ООП для программирования ПЛК крайне мало. Поэтому в задачи рабочей группы входило:

- создать руководство по использованию ООП в качестве дополнения к «классическому» способу программирования ПЛК;
- обеспечить единый подход и пользовательский опыт при использовании ООП в различных средах разработки;
- создать общие шаблоны проектирования для разработки ПО промышленных систем управления;
- обеспечить интеграцию «классического» способа программирования ПЛК и средств ООП (например, должна быть возможность дополнить «обычные» ФБ методами, свойствами, и, возможно, входами/выходами).

Различные подходы к программированию в данном документе рассматриваются на примере системы управления бойлером.

2. Базовые понятия ООП-расширений в 3-ей редакции стандарта МЭК 61131-3

2.1. 3-я редакция стандарта МЭК 61131-3

Большинство современных языков программирования поддерживают объектно-ориентированный подход. К таким языкам относятся Python, C++, Objective-C, Smalltalk, Delphi, Java, Swift, C#, Perl, Ruby, PHP и другие. Важным изменением в 3-ей редакции стандарта МЭК 61131-3 стало добавление средств ООП в языки программирования ПЛК. К таким средствам относятся классы (а также методы и интерфейсы), пространства имен и ООП-расширения для функциональных блоков.

3-я редакция МЭК 61131-3 принята в качестве международного стандарта. Она является официальным документом и доступна для приобретения на сайте www.iec.ch. 3-я редакция имеет полную обратную совместимость со 2-й редакцией стандарта, выпущенной в 2003 году.

Включение в стандарт ООП-расширений позволяет применять при программировании ПЛК те же проверенные подходы и практики, которые используются в разработке «обычного» ПО на протяжении десятилетий. В результате созданный с помощью этих средств программный код будет более модульным, читабельным и удобным в сопровождении. Более того, сближение языков программирования ПЛК с упомянутыми в первом абзаце языками высокого уровня позволяет значительно увеличить число специалистов, которые смогут разрабатывать приложения для ПЛК. А для компаний-разработчиков ПО проекты для ПЛК могут стать отличным дополнением для портфолио.

Обратите внимание, что в данном документе есть отсылки к тексту стандарта МЭК 61131-3. Они носят пояснительный характер; для получения цельной картины следует приобрести и изучить текст стандарта.

2.2. Основные понятия

3-я редакция МЭК 61131-3 содержит ряд нововведений и изменений, касающихся основных понятий стандарта. Например, в понятие POU (Program Organization Unit) теперь входят не только функции, функциональные блоки и программы, но и классы. Функциональные блоки и классы могут включать в себя методы. Поэтому, например, требуется определить термины «базовый класс» и «класс-наследник» (по аналогии с пользовательскими типами данных и ФБ), а также «метод». Основные понятия, связанные с ООП-расширениями, приведены в таблице:

Термин	**Описание**
Базовый тип	Тип данных, функциональный блок или класс, от которых осуществляется наследование
Вызов	Конструкция языка, позволяющая организовать выполнение функции, функционального блока или метода

Класс	POU, включающий в себя структуру данных и набор методов, выполняющих операции над этими данными. Класс представляет собой реализацию интерфейса, в которой для методов написан программный код
Класс-наследник	Класс, полученный в результате наследования от другого класса (также может называться «производным классом» или «дочерним классом»). Понятие «класс» достаточно близко к понятиям «пользовательский тип данных» и «пользовательский ФБ» из стандарта МЭК 61131-3
Динамическое связывание	Ситуация, в которой реализация метода определяется в момент его вызова в процессе работы программы (в зависимости от класса, к которому принадлежит объект или интерфейс)
Наследование	Создание нового класса, функционального блока или интерфейса на базе уже существующего класса/функционального блока/интерфейса
Входная переменная	Переменная, которая используется для передачи значения в POU (за исключением класса)
Экземпляр	Конкретная именованная копия структуры данных, связанной с ФБ, классом или программой, которая сохраняет свое состояние в промежутке между вызовами её (программы, ФБ, класса) методов
Интерфейс	Элемент языка, определяющий набор прототипов методов (не имеющих конкретной реализации). Аналогия: интерфейс похож на фланец двигателя – он описывает диаметр отверстий, расстояние между ними, диаметр вала – но при этом не является самим двигателем
Метод	Элемент языка, схожий с функцией, которая может быть определена только в рамках функционального блока или класса и поддерживает неявный доступ к переменным этого экземпляра ФБ или класса. Пример: у бойлера могут быть методы «Наполнить» и «Нагреть», отвечающие за соответствующие действия
Override	Ключевое слово, используемое в методе класса-наследника или функционального блока-наследника, которое позволяет изменить реализацию метода базового класса/функционального блока при сохранении его сигнатуры
Выходная переменная	Переменная, которая используется для получения значения от POU (за исключением класса)
POU	Функция, функциональный блок, программа или класс
Сигнатура	Информация, которая однозначно определяет конкретный метод – его имя, тип возвращаемого значения, названия, типы и порядок аргументов

2.3. Отличия в реализации ООП в разных языка программирования

В таблице ниже приведен обзор возможностей, связанных с ООП, доступных в различных языках программирования.

Возможность	МЭК 61113-3 (2-я редакция)	МЭК 61113-3 (3-я редакция)	C++	Java	C#
Использование разных языков программирования в одном проекте	+	+	-	-	-
Совмещение ООП и процедурного подхода	-	+	+	-	-
Классы	~ (ФБ)	+	+	+	+
Методы	~ (действия)	+	+	+	+
Интерфейсы	-	+	+	+	+
Полиморфизм	-	+	+/-	+	+
Ссылки	-	+ (интерфейсы)	-	+	+
Конструкторы/деструкторы	-/+	-/+	+	+	+
Свойства	-	-[1]	-	-	+
Динамическое выделение памяти	-	-[2]	+	+	+
Ограничение доступа к данным	~ (переменные)	~ (переменные)[3]	+	+	+

[1] В некоторых реализациях стандарта МЭК 61131-3 (например, среде CODESYS V3.5) свойства присутствуют.
[2] В некоторых реализациях стандарта МЭК 61131-3 (например, среде CODESYS V3.5) доступно динамическое выделение памяти.
[3] В некоторых реализациях стандарта МЭК 61131-3 (например, среде CODESYS V3.5) присутствует ограничение доступа к методам с помощью спецификаторов PUBLIC, PRIVATE, PROTECTED.

3. Общий обзор

3.1. Основная информация

Данный документ содержит набор рекомендаций по проектированию и разработке ПО для ПЛК с использованием языков стандарта МЭК 61131-3. Также читателям настоятельно рекомендуется ознакомиться с руководством по созданию библиотек от PLCopen.

В рамках данного документа применительно к задачам разработки ПО для ПЛК используются принципы SOLID:

- **SRP** (single responsibility principle) – принцип единой ответственности;
- **OCP** (open–closed principle) – принцип открытости/закрытости;
- **LSP** (Liskov substitution principle) – принцип подстановки Лисков;
- **ISP** (interface segregation principle) – принцип разделения интерфейса;
- **DIP** (dependency inversion principle) – принцип инверсии зависимостей.

3.2. Модули и команды как объекты

При разработке приложений и библиотек с использование ООП следует различать модули и команды.

- Модули представляют собой обособленные части приложения и могут иметь иерархические зависимости друг от друга. Например, исполнительные механизмы, датчики и другие узлы системы управления могут быть представлены в коде в виде «модулей»;
- Команды соответствуют отдельным действиям модуля. Они могут быть связаны с одним модулем или с группой модулей, реализующих общий интерфейс.

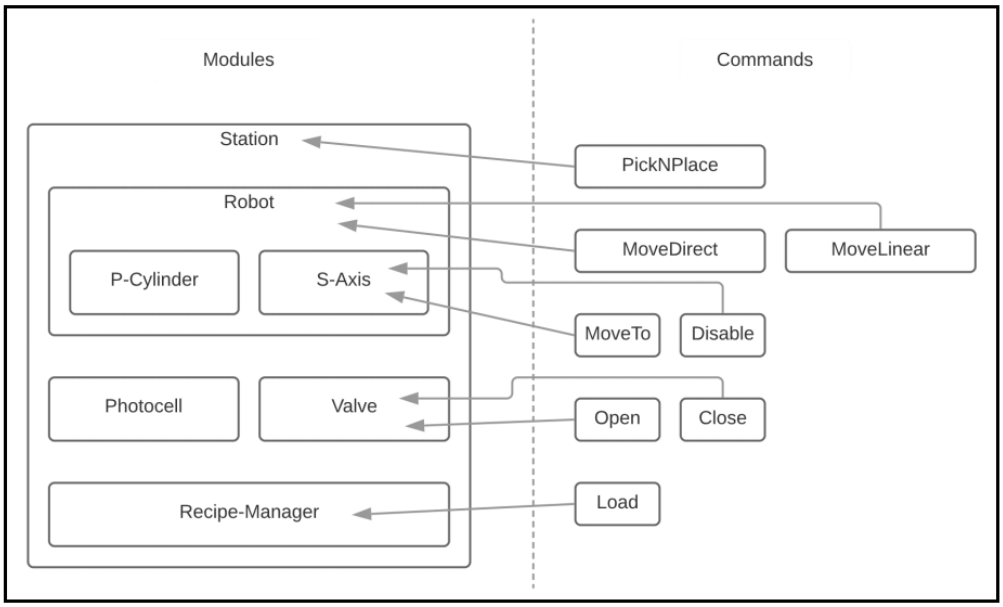


Рис. 1. Визуальное представление модулей и команд на примере интерфейсов **IModule** и **ICommand**

Каждый модуль реализует обобщенный интерфейс **IModule**. Некоторые модули также реализуют другие специализированные интерфейсы – например, модули, связанные с управлением перемещением по осям, реализуют интерфейс **IAxis**.

Каждая команда реализует интерфейс **ICommand** и соответствует рекомендациям по разработке библиотек от PLCopen. Команды, связанные с перемещением по осям, имеют вход типа **IAxis**, через который они получают экземпляр конкретной оси, с которой будут взаимодействовать.

Такой подходит позволяет гармонично использовать «классический» процедурный подход к программированию ПЛК совместно с объектно-ориентированным подходом. Например, блоки управления движением от PLCopen можно использовать в их «традиционном» виде. С другой стороны, этими же блоками можно управлять через интерфейсы. Это открывает новые возможности для приложений автоматизации – например, переконфигурирование приложения в процессе его работы.

4. Пример: управление бойлером

4.1. Обзор этапов подготовки примера

Первая версия примера была представлена на Ганноверской промышленной выставке-ярмарке совместно с консорциумом OPC Foundation.

Исходный код примера был адаптирован под требования руководств PLCopen, что позволило разработать унифицированный интерфейс функциональных блоков.

Следующим шагом является переработка примера с использованием объектно-ориентированного подхода с последующим добавлением обработчика ошибок и менеджера тревог.

Проект написан на языке ST в среде CODESYS V3.5 и доступен на сайте PLCopen.

4.2. Функциональная схема примера

Функциональная схема примера управления бойлером приведена на рисунке:

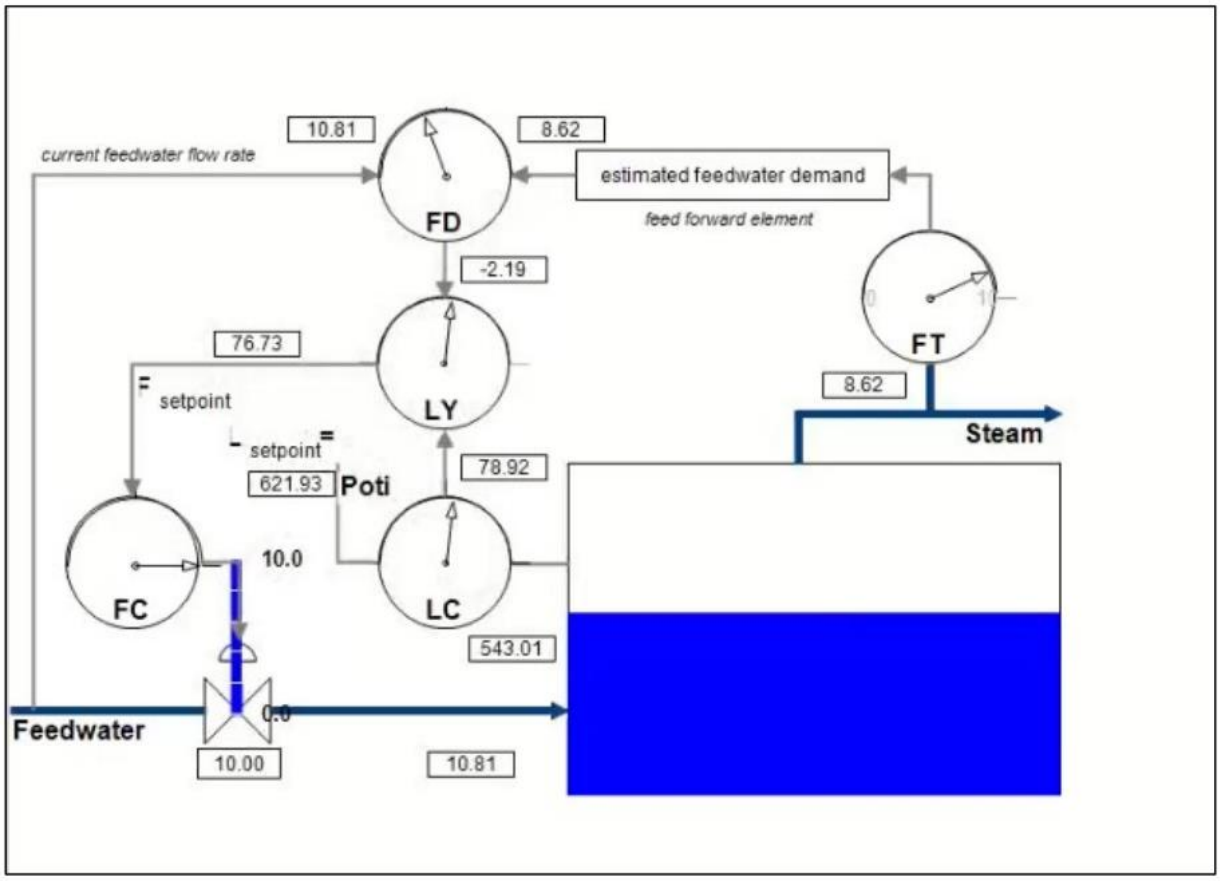


Рис. 2. Функциональная схема примера управления бойлером

4.3. Структура примера с «классическим» подходом к программированию ПЛК

Проект моделирует систему управления бойлером с использованием ПИД-регулятора, входным сигналом для которого является сигнал датчика уровня (LC), а выходным – положение клапана расхода (FC). В состав проекта входит генератор случайных возмущений для симуляции сбоев в подаче питающей воды. Проект включает в себя 2 программы, 4 пользовательские функции и 17 пользовательских функциональных блоков.

Программы
MAIN
P_Initialisation

Функции
F_FeedwaterEstimator
F_AlignValues
F_IntToReal
F_RealToInt

Функциональные блоки
FB_AutomaticMode
FB_InputParameter
FB_DRAND
FB_BasicPIDCtrl
FB_Boiler
FB_OutputPipe
FB_SteamFlow
FB_InputPipe
FB_Valve
FB_FeedwaterRate
FB_Drum
FB_FillLevel
FB_Potentiometer
FB_Motor
FB_LevelController
FB_FlowController
FB_ControlComputation

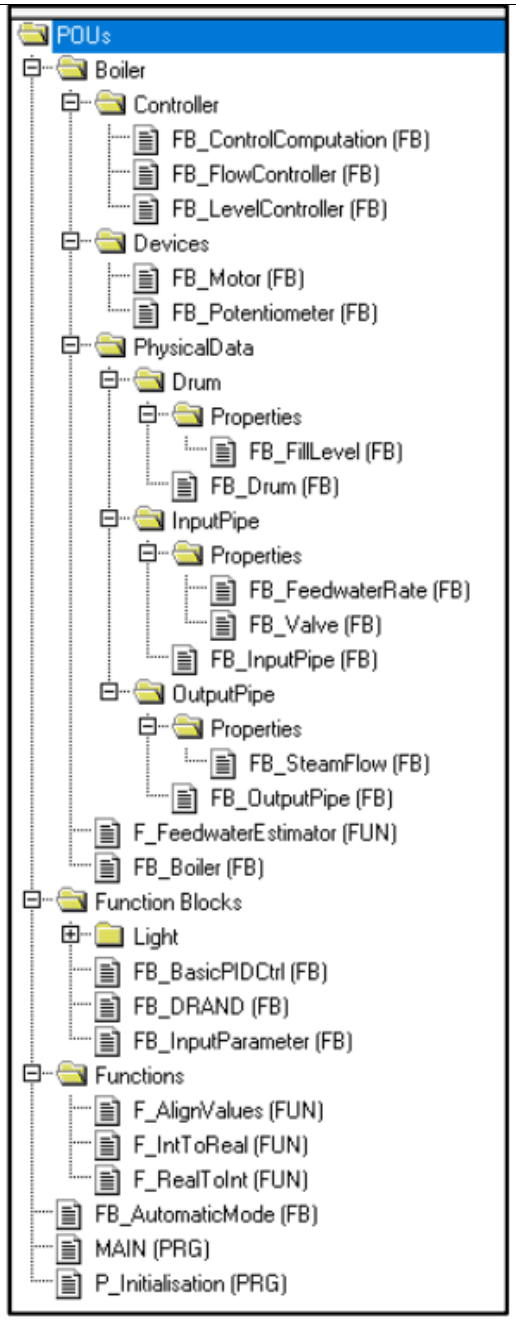


Рис. 3. Структура примера с использованием «классического» подхода к программированию ПЛК

4.4. Адаптация примера под требования руководств PLCopen

Исходный пример хорошо структурирован и соответствует принципу единой ответственности. Для каждого элемента оборудования и каждого действия используется свой функциональный блок.

К сожалению, это приложение изначально проектировалось без использования объектно-ориентированного подхода, поэтому другим принципам SOLID оно не соответствует. В нем не используются интерфейсы и нет абстракций.

Существует два варианта адаптации примера под требования руководств PLCopen:

1. Полная переработка примера в соответствии с принципами объектно-ориентированного подхода. Это позволит создать продуманную архитектуру приложения. К сожалению, этот вариант потребует много времени и ресурсов, из-за чего разработчику может быть сложно получить одобрение руководства.

2. Другим вариантом является частичное переиспользование уже имеющегося кода с некоторыми изменениями в структуре проекта. В данном случае это возможно из-за изначально проработанной структуры приложения. Для давно разрабатываемых и модифицируемых проектов, за годы обросших «костылями» и превратившихся в сильно связанный клубок кода, будет лучше переписать всё с нуля.

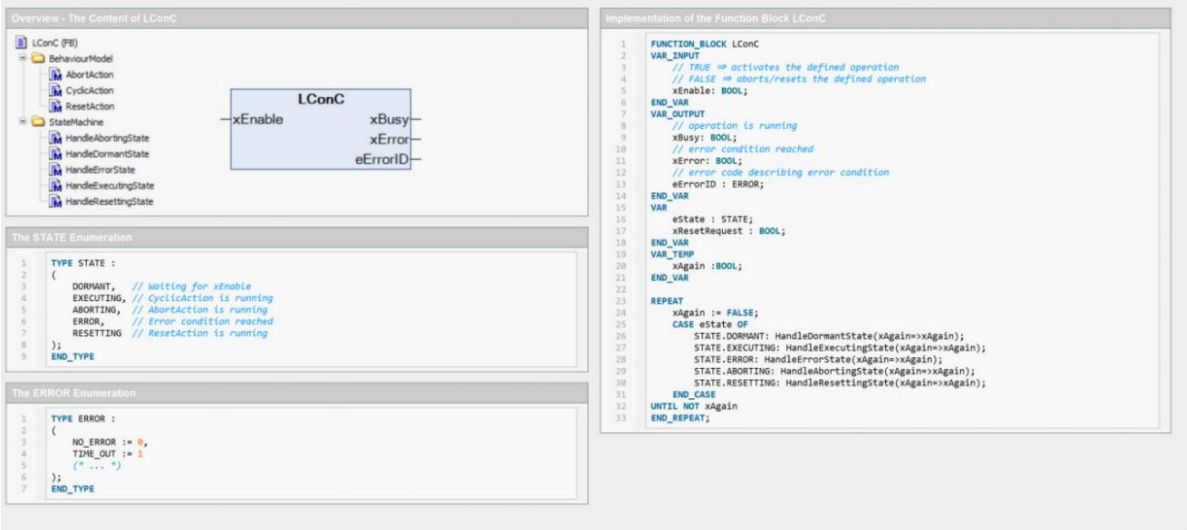
В данном примере рассматривается вариант 2. В первую очередь необходимо провести унификацию POU и разделить их по уровням абстракции. Руководство PLCopen по разработке библиотек рекомендует использовать для этого понятный интерфейс и стандартизированное поведение. Для этого используются модели поведения (behaviour model). Для начала POU примера следует адаптировать под соответствующие модели поведения, создав для них машины состояний и обработку ошибок. Поскольку модели поведения проектировались с учетом объектно-ориентированного подхода, то это можно сделать с помощью всего лишь нескольких изменений в исходных POU.

Также по возможности будет использоваться принцип открытости/закрытости и принцип инверсии зависимостей. Кроме того, будет создан первый интерфейс, который впоследствии можно будет дополнять и расширять.

Поскольку «строительные блоки», из которых состоит программа могут быть адаптированы под использование моделей поведения (в примере используется модель **LConC** – «выполнение по уровню, без ограничения времени выполнения, с постоянным выполнением»), то мы сможем расположить их на верхнем уровне абстракции, чтобы они не зависели от реализации ФБ реализующих модель поведения. **LConC** предоставляет интерфейс, который теперь должен быть реализован старыми ФБ**.** Но на первом этапе можно ограничиться копированием кода старого функционального блока в метод **Cyclic Action** нового.

Благодаря принципу открытости/закрытости – блок сохранит свою функциональность, и существующий программный код не потребует внесения изменений. Это также позволит

разработчикам плавно переходить от «классического» процедурного стиля программирования ПЛК к объектно-ориентированному подходу, поскольку интерфейс блоков остается идентичным, а реализация не претерпевает существенных изменений. Зато теперь модули готовы для дальнейшего расширения. Кроме того, в методе **StartAction** можно произвести валидацию значений входов функционального блока (как, например, это производится в блоках с моделью поведения «выполнение по фронту», описанных в руководстве по разработке библиотек PLCopen), чтобы повысить стабильность работы. Кроме того, теперь модули имеют абстрактный и единообразный интерфейс, который можно будет в дальнейшем расширять в соответствии с [принципом разделения интерфейса](#).



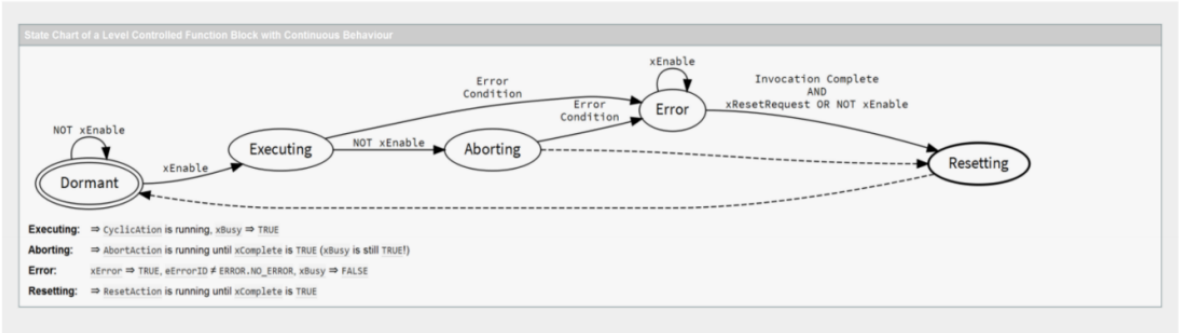


Рис. 4. Реализация и диаграмма состояний ФБ **LConC**

Рис. 5. Все функциональные блоки реализуют единый интерфейс

Согласованный дизайн моделей поведения позволил создать абстракцию, не зависящую от конкретной машины состояний, потому что все функциональные блоки с их машинами состояний являются наследниками одного абстрактного класса и реализуют общий интерфейс.

Это дает возможность вызывающему коду использовать различные ФБ вне зависимости от конкретной реализации (см. принцип разделения интерфейса) – например, реализовать обработчик ошибок.

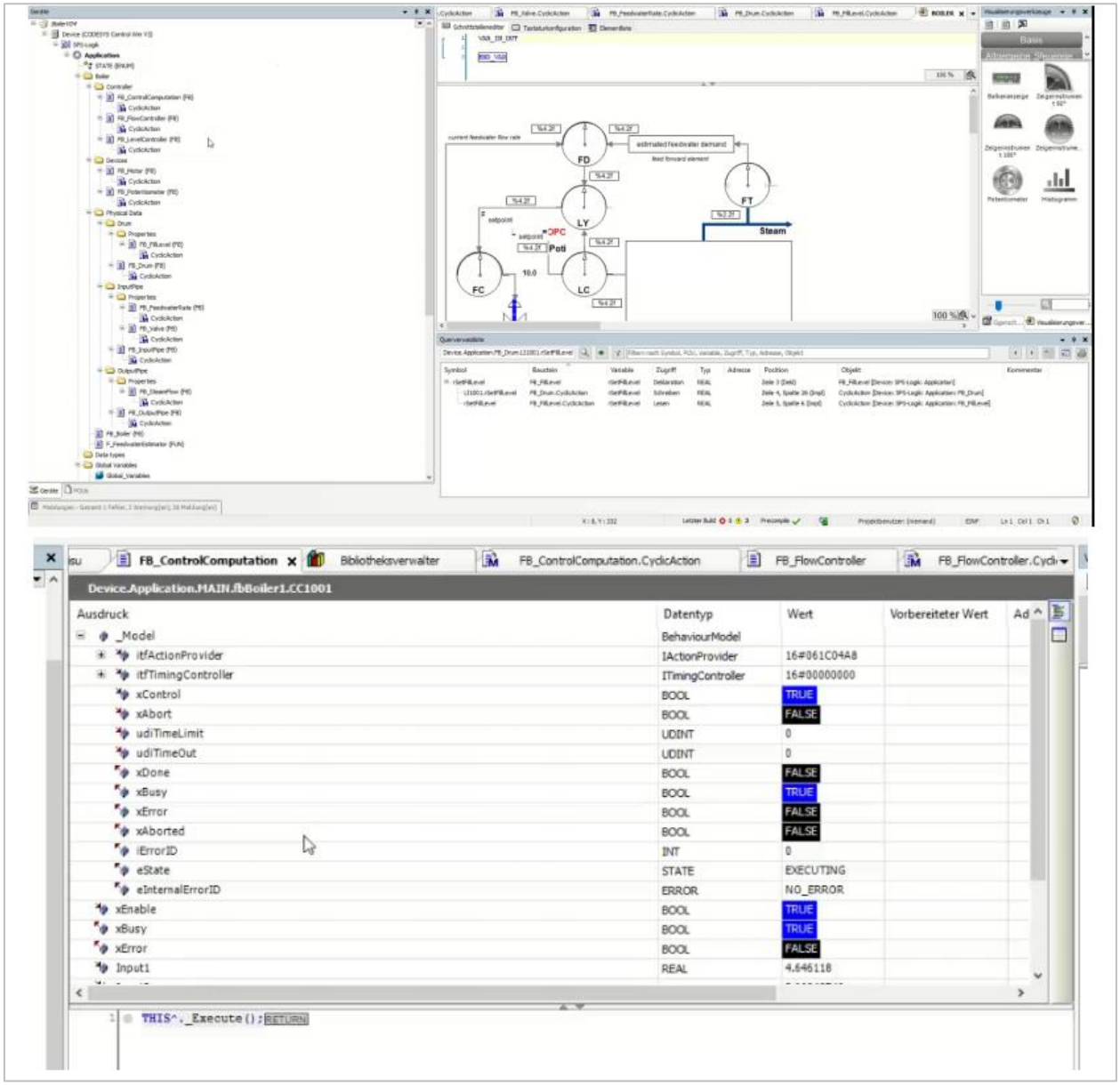


Рис. 6. Обзор состояния блока в симуляции

Важно отметить, что программа **MAIN** из примера не может использоваться на реальном объекте, поскольку в ней отсутствует обработка состояний запуска и останова, а также обработка ошибок. Этот функционал является уникальным для каждого конкретного проекта. Далее мы продемонстрируем преимущества объектно-ориентированного подхода при добавлении этого функционала за счет повторного использования кода.

4.5. Добавление обработки ошибок

В рамках примера будет создан отдельный функциональный блок, который будет контролировать ошибки всех остальных блоков. Продемонстрируем это, используя два разных подхода: наследование и композицию.

Мы не будем напрямую обращаться к функциональным блокам; вместо этого мы воспользуемся абстрактным интерфейсом **IBehaviourModel**, который они реализуют (см. рис. 5). Этот интерфейс предоставляет доступ к выходам **xDone**, **xAborted**, **xError**, **iErrorID**. Такой подход соответствует принципу разделения интерфейса, и этот интерфейс могут реализовывать все функциональные блоки проекта – неважно, какому типу и модели поведения они соответствуют. Все они могут быть подключены к блоку обработки ошибок (**FB_ErrorHandler**).

Идея заключается в том, чтобы собрать все блоки в массив **aritfObserver** и обрабатывать их итеративно. В этом массиве будут храниться не указатели на конкретные функциональные блоки, а экземпляры интерфейса **IBehaviourModel**. С помощью метода **GetModelState** можно получить информацию по всем блокам проекта (см. рис. 8).

Блок-обработчик ошибок позволяет добавлять устройство в список устройств, для которых необходимо обрабатывать ошибки (**AttachDevice**). Если для какого-либо устройства больше нет необходимости обрабатывать ошибки – его можно удалить из списка (метод **DetachDevice**). Данные о состоянии каждого устройства сохраняются в массиве структур **_stInfoState**. Поля структуры приведены на рисунке 10.

Теперь можно получать информацию о состоянии экземпляров блоков с помощью метода **GetModelState** и сохранять ее в массиве структур.

```
METHOD AttachDevice
(*Register a Device by the BehaiourModel Interface*)
VAR_INPUT
    (*Device which will be monitor from Error Handler*)
    IDevice : CBML.IBehaviourModel;
END_VAR

VAR
    uiLoop: UINT; //Help Variable for the For Loog
END_VAR
```

```
//go to the next free input in the arry and save the Interface
FOR uiLoop := 0 TO GVL_Const.MAX_Observer DO
    IF IDevice <> 0 AND THIS^.aritfObserver[uiLoop]= 0 THEN
        THIS^.aritfObserver[uiLoop] := IDevice;
        EXIT;
    END_IF
END_FOR
```

Рис. 7. Метод **AttachDevice** блока обработки ошибок

```
//Error Handler whcih will be monitor all FBs which has an behaiour
FUNCTION_BLOCK FB_ErrorHandler IMPLEMENTS I_ErroHandlerSubject

VAR
    //Help Variables for Attaching and read Information from Devices
    _stInfoState        : ARRAY [1..GVL_Const.MAX_Observer] OF tsCBMLInfo;          //State from all existing FB
    aritfObserver       : ARRAY [0..GVL_Const.MAX_Observer] OF CBML.IBehaviourModel;// Interface list from all FB which will be monitord
    uiLoop              : UINT; //Help Varialbe to get access to the Interface during a For Loop

END_VAR
```

```
//monitor all register Device by the IBehaviour Modell

FOR uiLoop := 0 TO GVL_Const.MAX_Observer DO
    IF aritfObserver[uiLoop] <> 0 THEN
        aritfObserver[uiLoop].GetModelState(    xCommit:= _stInfoState[uiLoop].xCommit,
                                    xDone=> _stInfoState[uiLoop].xDone,
                                    xBusy=> _stInfoState[uiLoop].xBusy,
                                    xError=> _stInfoState[uiLoop].xError,
                                    xAborted=> _stInfoState[uiLoop].xAborted,
                                    iErrorID=> _stInfoState[uiLoop].iErrorID,
                                    eState=> _stInfoState[uiLoop].eState);
    END_IF
END_FOR
```

Рис. 8. Реализация блока обработки ошибок

Методы **AttachDevice** и **DetachDevice** определены в интерфейсе **I_ErrorHandlerSubject**, который реализует блок обработки ошибок. Этот интерфейс может использоваться блоком управления бойлером для подключения своих экземпляров к блоку обработки ошибок.

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

```
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);
        itfErrorHandler.AttachDevice(IDevice := Drum1001);
        itfErrorHandler.AttachDevice(IDevice := FC1001);
        itfErrorHandler.AttachDevice(IDevice := LC1001);
        itfErrorHandler.AttachDevice(IDevice := Pipe1002);
        itfErrorHandler.AttachDevice(IDevice := CC1001);
    END_IF
END_IF
```

Рис. 9. Подключения экземпляров блока **FB_Boiler** к блоку обработки ошибок

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

Рис. 10. Структура диагностической информации блока обработки ошибок

В основной программе происходит объявление экземпляра блока обработки ошибок и передачи его в качестве аргумента блоку управления бойлером при его вызове.

```
PROGRAM MAIN
VAR
    fbBoiler1    : FB_Boiler;        (*~ (OPC : 1 : enabled for OPC) *)
    fbInput1     : FB_InputParameter;
    fbErrorHandler : FB_ErrorHandler;
    bInitDone    : BOOL := FALSE;
    test : FB_DeviceBasic;
END_VAR
```

```
IF NOT bInitDone THEN
    P_Initialisation();
    bInitDone := TRUE;

    //
END_IF

counter := counter +1;

(*Get Input*)
fbInput1();

(*Simulate Boiler*)
fbBoiler1(
    rSetFillLevel := fbInput1.rFillLevel,
    rSteamDemand := fbInput1.rSteamDemand,
    itfErrorHandler := fbErrorHandler
);

(*Monitor all Instance by IBehaviourModell*)
fbErrorHandler();
```

Рис. 11. Код основной программы

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

Рис. 12. Пример отображения диагностической информации функционального блока

Теперь обработка ошибок является стандартным функционалом для всех блоков проекта. При этом обработка ошибок производится централизованно без необходимости вызова отдельных блоков в блоке-обработчике.

Этот поход позволяет получить диагностическую информацию о всех экземплярах всех блоков проекта в одном массиве – достаточно вызвать для них метод **AttachDevice**.

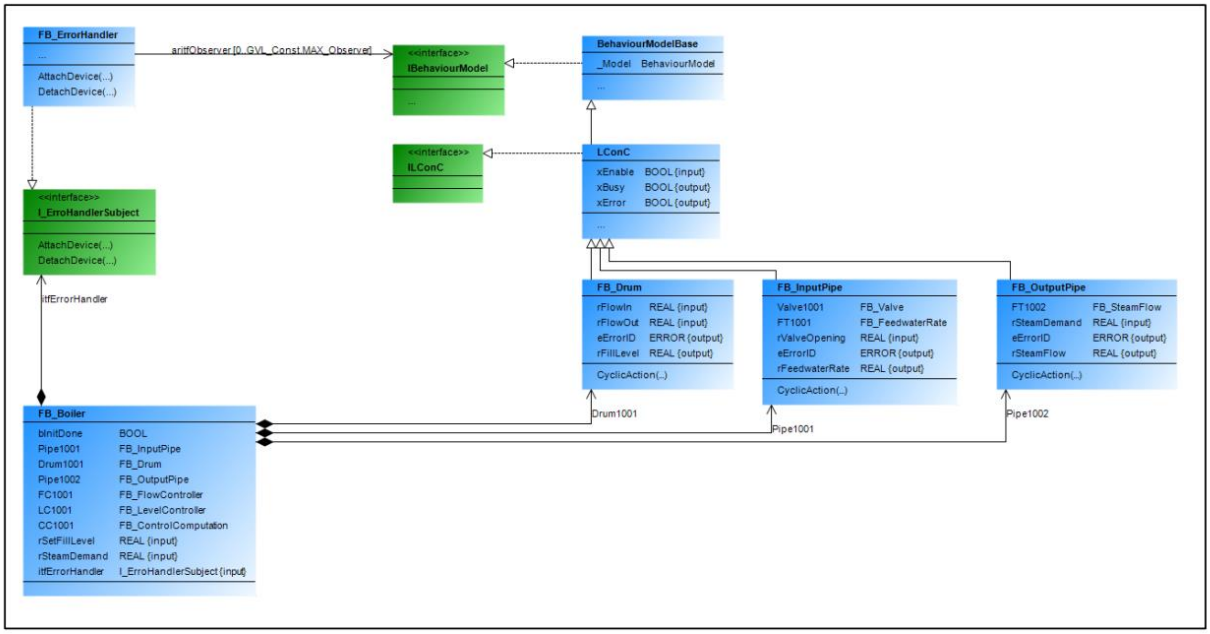


Рис. 13. Архитектура приложения

На рис. 13 показано, как интерфейс **LConC** наследуется интерфейсом **BehaviourModelBase**, а он, в свою очередь, наследуется интерфейсом **IBehaviourModel**. Интерфейс **I_ErrorHandlerSubject** реализуется блоком **FB_ErrorHandler** и всеми блоками проекта, связанными с техпроцессом. Таким образом, обработчик ошибок может использоваться для контроля ошибок всех блоков проекта.

Использование наследования в качестве основного механизма создания блоков предоставляет хорошие средства по разделению уровней абстракции, но в то же время имеет некоторые ограничения. Например, не получится расширить функционал ФБ **BehaviourModelBase** за счет наследования, так как **BehaviourModelBase** уже является наследником **LConC**, а множественное наследование запрещено.

В данном случае нужно проверить, есть ли возможность оптимизировать уже созданную архитектуру, используя для модели поведения композицию вместо наследования (см. п. 5.1). Для этого необходимо определить модель поведения в виде отдельного блока, который будет входить в состав всех пользовательских блоков, и добавить нужные методы для различных состояний, как было сделано в предыдущем варианте (в соответствии с принципом открытости/закрытости).

В среде CODESYS V3.5 это можно сделать с помощью библиотеки **Common Behaviour Model**. Блоки библиотеки поддерживают подключение блока **ActionController**, что позволяет блокам, реализующим интерфейс **IActionController** выполнять свои методы вместо методов, определяемых моделью поведения.

Основная идея заключается в создании нового абстрактного базового класса, одно из полей которого содержит путь к конкретному экземпляру функционального блока. Чтобы поддержать композицию для модели поведения, этот базовый класс реализует интерфейс **IActionController**. Для следования принципу открытости/закрытости рекомендуется также реализовать интерфейс **IActionProvider**. Это позволит наследуемому блоку работать с тем же поведением, что и раньше.

В каждом методе **ActionController** вызывается соответствующий метод из **ActionProvider**.

В примере ниже показана реализация циклически вызываемого метода **ControlCyclicAction**.

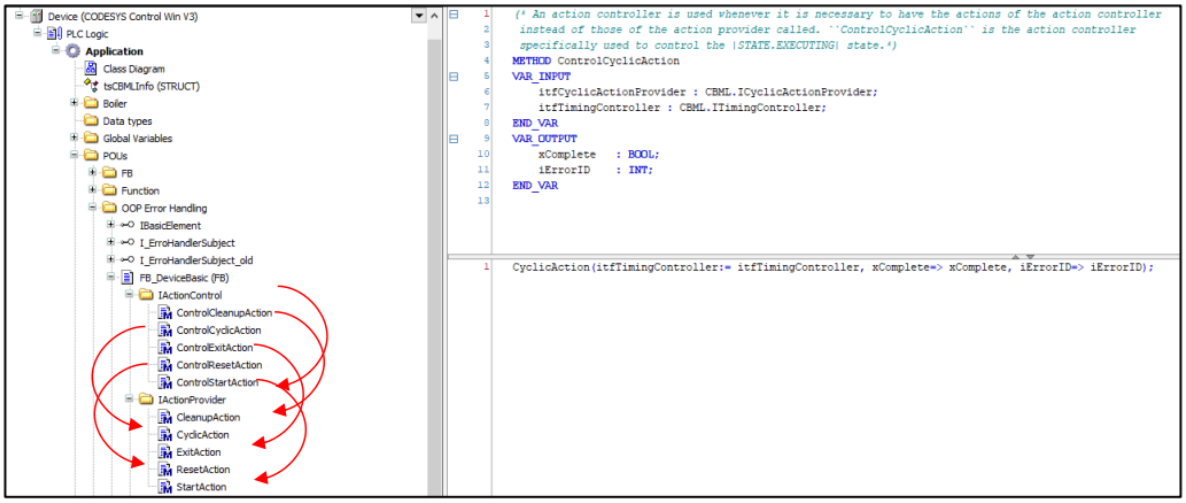


Рис. 14. Пример циклически вызываемого метода, связанного с интерфейсом **IActionControl**

Базовый класс также содержит свойства (**Get/Set**), позволяющие связать блок с экземпляром интерфейса модели поведения.

FB_DeviceBasic.IBehaviourModel.Set

```
VAR
END_VAR
```

```
_iBehaviourModel1 := IBehaviourModel;
IBehaviourModel.ActionController := THIS^;
```

Рис. 15. Реализация записи свойства для интерфейса **IBehaviourModel**

ФБ-наследник теперь может сам задать модель поведения, в соответствии с которой он будет работать. Для этого достаточно:

- Вместо наследования от **LConC** унаследовать ФБ от базового класса **FB_DeviceBasic**;
- Объявить входную переменную **xEnable** или **xExecute** (в зависимости от выбранной модели поведения);
- Объявить экземпляр блока библиотеки **Common Behaviour Model**, соответствующий выбранной модели поведения;
- В коде пользовательского блока передать экземпляр блока, реализующего модель поведения, в свойство **IBehaviourModel**;
- Вызвать ФБ реализующий модель поведения с необходимыми параметрами (см. рис. 16).

FB_OutputPipe

```
FUNCTION_BLOCK FB_OutputPipe EXTENDS FB_DeviceBasic
VAR_INPUT
    rSteamDemand :  REAL;
    xEnable : BOOL;
END_VAR
VAR_OUTPUT
    eErrorID : ERROR;// error code describing error condition
    rSteamFlow :    REAL;
END_VAR
VAR
    FT1002 :        FB_SteamFlow;
    StateBehaviour : CBML.LConC;
END_VAR
```

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

Рис. 16. Изменения в пользовательском блоке

Это свойство передается при вызове метода **AttachDevice** блока обработки ошибок, что позволяет не вносить изменения в интерфейс.

```
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(
```

Рис. 17. Вызов метода **AttachDevice** для отредактированного блока

Новая архитектура приложения приведена на рис. 18. Ее преимуществом является возможность использования абстрактного ФБ **FB_DeviceBasic** для согласованного расширения возможностей уже созданных блоков и сокращение количества дублируемого кода. Кроме того, как было продемонстрировано выше, такой подход позволяет определить модель поведения одного блока с помощью другого (через использование свойства **IBehaviourModel**).

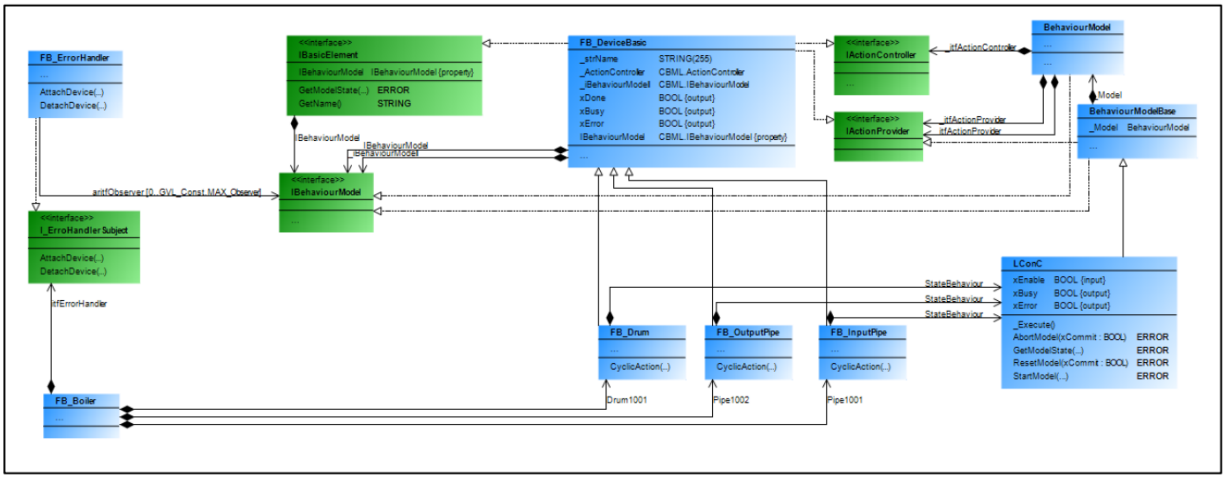


Рис. 18. UML-диаграмма новая архитектуры приложения

Это позволяет с легкостью добавлять дополнительный функционал (например, список тревог, рассматриваемый в следующем пункте).

4.6. Добавление списка тревог

Следует различать тревоги, связанные с техпроцессом, и тревоги системы управления, чтобы иметь возможность отфильтровать их для отображения различным пользователям (например, оператору и системному администратору). В рамках примера рассматриваются только тревоги системы управления, но по аналогии можно реализовать обработку и тревог техпроцесса. Можно даже добавить к функциональным блокам выход **ProcessAlarm** и обрабатывать эти тревоги отдельно.

При формировании списка тревог полезно знать, с каким функциональным блоком (или даже с каким его экземпляром) связана конкретная ошибка.

Для этого в абстрактном базовом классе следует добавить поле, содержащее имя экземпляра блока, и метод для получения этого имени. Это имя автоматически формируется CODESYS, но может быть задано пользователем во время инициализации блока через соответствующее свойство. Использование интерфейса, содержащего нужные методы, соответствует принципу разделения интерфейса.

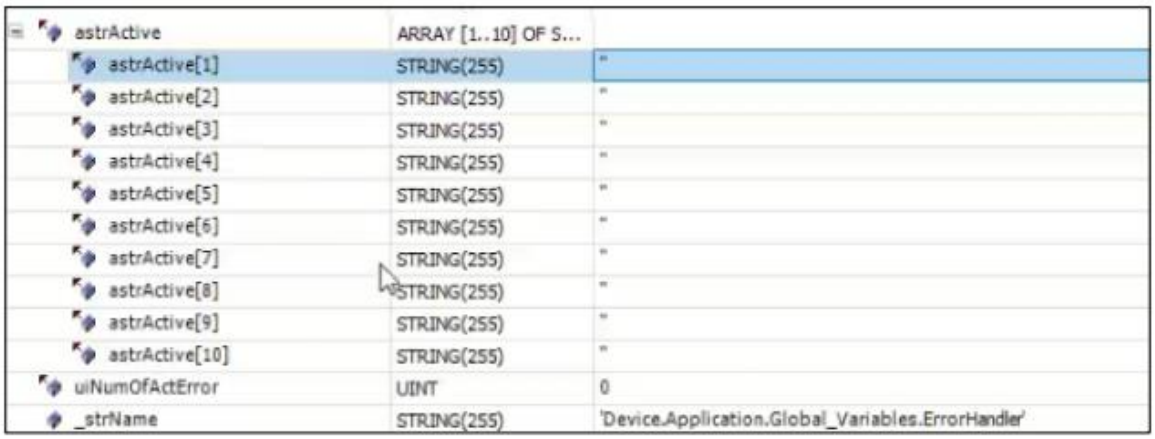
Чтобы получить доступ к этому методу – потребуется создать новый или расширить уже существующий интерфейс обработчика ошибок. Изменение интерфейса – это не самый лучший подход, но в данном случае это позволит оставить в нем только нужные методы (**GetModelState** и **GetName**). Поэтом разумно изменить интерфейс методов **AttachDevice** / **DetachDevice** и массив интерфейсов таким образом, чтобы они использовали в качестве типа не **IBehaviourModel**, а новый **IBasicElement**. Метод **GetModelState** будет делегировать полученный запрос внутренней машине состояний (перечисление **ERROR** определено на рис. 4):

```
METHOD GetModelState : ERROR
VAR_INPUT
    xCommit : BOOL;
END_VAR
VAR_OUTPUT
    xDone : BOOL;
    xBusy : BOOL;
    xError : BOOL;
    xAborted : BOOL;
    iErrorID : INT;
    eState : CBML.STATE;
END_VAR
```

```
IF _iBehaviourModel1 <> 0 THEN
    _iBehaviourModel1.GetModelState(xCommit:= xCommit, xDone=> xDone, xBusy=> xBusy, xError=> xError, xAborted=> xAborted, iErrorID=> iErrorID, eState=> eState);
END_IF
```

Рис. 19. Реализация делегирования в методе **GetModelState**

Сообщения об ошибках предоставляются пользователям в виде массива строк, что упрощает их восприятие.



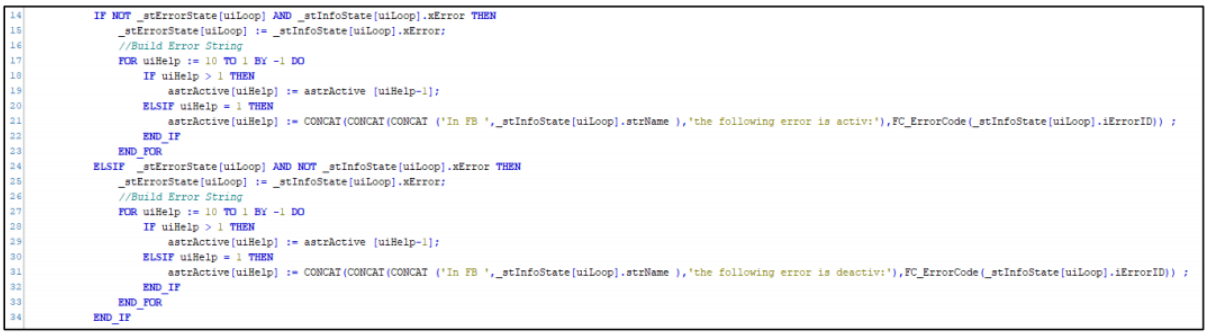
astrActive	ARRAY [1..10] OF S...	
astrActive[1]	STRING(255)	''
astrActive[2]	STRING(255)	''
astrActive[3]	STRING(255)	''
astrActive[4]	STRING(255)	''
astrActive[5]	STRING(255)	''
astrActive[6]	STRING(255)	''
astrActive[7]	STRING(255)	''
astrActive[8]	STRING(255)	''
astrActive[9]	STRING(255)	''
astrActive[10]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

Рис. 20. Массив обработчика ошибок

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)	''

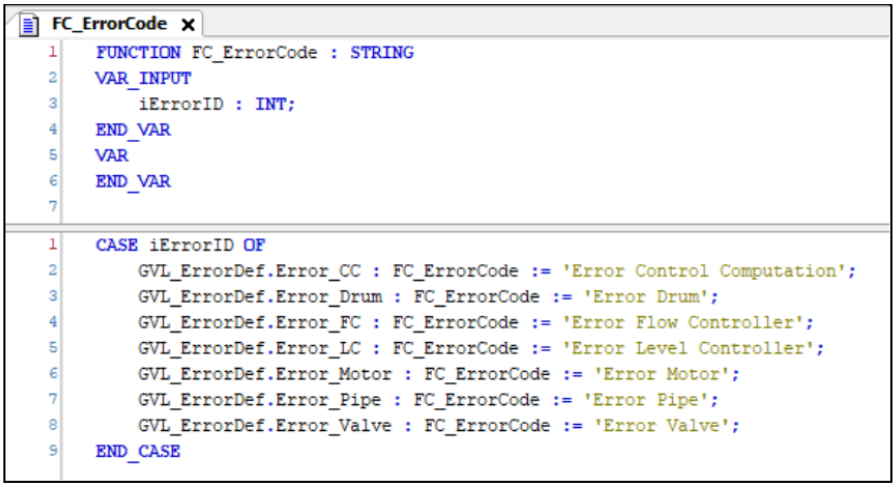
Рис. 21. Пример сообщения об ошибке

На рис. 22 приведен код формирования сообщения об ошибке, который вызывается только по переднему и заднему фронту выходной переменной **xError**. В этом коде используется функция **FC_ErrorCode**, которая преобразует код ошибки в ее название.



```
IF NOT _stErrorState[uiLoop] AND _stInfoState[uiLoop].xError THEN
    _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
    //Build Error String
    FOR uiHelp := 10 TO 1 BY -1 DO
        IF uiHelp > 1 THEN
            astrActive[uiHelp] := astrActive [uiHelp-1];
        ELSIF uiHelp = 1 THEN
            astrActive[uiHelp] := CONCAT(CONCAT(CONCAT ('In FB ',_stInfoState[uiLoop].strName ),'the following error is activ:'),FC_ErrorCode(_stInfoState[uiLoop].iErrorID)) ;
        END_IF
    END_FOR
ELSIF  _stErrorState[uiLoop] AND NOT _stInfoState[uiLoop].xError THEN
    _stErrorState[uiLoop] := _stInfoState[uiLoop].xError;
    //Build Error String
    FOR uiHelp := 10 TO 1 BY -1 DO
        IF uiHelp > 1 THEN
            astrActive[uiHelp] := astrActive [uiHelp-1];
        ELSIF uiHelp = 1 THEN
            astrActive[uiHelp] := CONCAT(CONCAT(CONCAT ('In FB ',_stInfoState[uiLoop].strName ),'the following error is deactiv:'),FC_ErrorCode(_stInfoState[uiLoop].iErrorID)) ;
        END_IF
    END_FOR
END_IF
```

Рис. 22. Код формирования сообщения об ошибке



FC_ErrorCode

```
FUNCTION FC_ErrorCode : STRING
VAR_INPUT
    iErrorID : INT;
END_VAR
VAR
END_VAR

CASE iErrorID OF
    GVL_ErrorDef.Error_CC : FC_ErrorCode := 'Error Control Computation';
    GVL_ErrorDef.Error_Drum : FC_ErrorCode := 'Error Drum';
    GVL_ErrorDef.Error_FC : FC_ErrorCode := 'Error Flow Controller';
    GVL_ErrorDef.Error_LC : FC_ErrorCode := 'Error Level Controller';
    GVL_ErrorDef.Error_Motor : FC_ErrorCode := 'Error Motor';
    GVL_ErrorDef.Error_Pipe : FC_ErrorCode := 'Error Pipe';
    GVL_ErrorDef.Error_Valve : FC_ErrorCode := 'Error Valve';
END_CASE
```

Рис. 23. Код функции **FC_ErrorCode**

Текущую реализацию можно улучшить, избавившись от циклической проверки всех блоков обработчиком ошибок. Для этого можно использовать паттерн Наблюдатель – чтобы **ErrorHandler** был подключен к нужным блокам в качестве «наблюдателя», и они отправляли бы ему сообщения в случае изменения своего состояния.

5. Дополнительные вопросы

5.1. Соображения по основным моментам для руководства по ООП

В этом документе также должны быть рассмотрены следующие важные моменты:

- Хорошая архитектура и плохая производительность. Какие детали реализации следует учитывать, чтобы не слишком сильно ухудшить производительность приложения?
- Наследование и композиция, и как наследование ослабляет инкапсуляцию. Многие уважаемые системные архитекторы[4] считают, что по возможности следует предпочитать композицию наследованию, и использовать наследование только в тех случаях, когда это действительно необходимо. Однако это высказывание слишком туманно. Совет к использованию композиции «по возможности» скрывает реальную проблему. Опыт показывает, что в большинстве случаев использование композиции является более уместным по сравнению с наследованием. Но это не значит, что нужно избегать наследования. Каждый из методов должен использоваться в подходящей для этого ситуации;
- Избегание зависимостей. Вопрос добавления зависимости в процессе выполнения (внедрение зависимости) играет ключевую роль при обсуждении предпочтения композиции наследованию. Важно понимать, что это дискуссионный вопрос. Цель дискуссии – не найти «оптимальный» метод разработки классов, а сформулировать рекомендации по выбору между использованием композиции и наследования.

5.2. Заметки по вопросам производительности

При изменении архитектуры приложения следует учитывать ограниченность ресурсов ПЛК по сравнению с ПК и серверами.

Можно выделить три аспекта, которые способны повлиять на эту ситуацию:

- *Архитектура аппаратного обеспечения.*
  Время выполнения кода может сильно зависеть от аппаратных ресурсов контроллера – например, вызов виртуального метода может занимать гораздо больше времени по сравнению с подпрограммой – или почти столько же. Для циклически выполняемой программы ПЛК даже небольшое увеличение времени цикла может иметь принципиальное значение. Решающим фактором здесь является способ подключения памяти приложения к шинам адресов и данных, а также используемый механизм кэширования. Кэширование может помочь сэкономить ресурсы для циклически выполняемого кода. Сегодня при разработке контроллера вопрос уже не только в том, сколько времени потребуется на выполнение 1024 инструкций языка IL!
- *Архитектура компилятора.*
  Генератор кода должен гарантировать, что код приложения будет размещен в кэшируемой области памяти. Это позволяет обрабатывать код и данные

[4] Например, см. Приёмы объектно-ориентированного проектирования. Паттерны проектирования от «Банды Четырёх»

циклически выполняемой программы без их копирования из основной памяти. Реализация интерфейсов, методов, свойств и ссылок должна соответствовать задачам ПЛК. Выделение памяти в стеке (для локальных переменных) не должно создавать проблем при использовании методов. Инициализация значительного объема данных (например, массивов структур) может стать проблемой при циклическом вызове и должна контролироваться разработчиком приложения;

- *Архитектура ПО.*
  Ключевые слова ABSTRACT, FINAL, PRIVATE и т.д. должны использоваться осознанно. Они позволяют программисту влиять на важные аспекты выполнения своего кода еще на начальном этапе разработки. Программист должен контролировать инициализацию локальных переменных – нужно ли это делать и если да – то когда именно. Среда разработки должна обеспечивать способ отключения инициализации определенного набора переменных, чтобы повысить производительность приложения.

5.3. Заметки для будущих обсуждений

- Синхронный и асинхронный обмен / выполнение программ;
- Следует ли использовать динамическое выделение памяти в системах автоматизации?
- Не следует отдавать предпочтение уже существующим подходам и стилям программирования;
- Возможно, смена парадигмы программирования ПЛК неизбежна.