



Обучающие материалы PLCopen

Руководство по применению объектно-ориентированного подхода

Версия 0.99 – для получения обратной связи (до 14 августа 2021)

Отказ от ответственности

Название 'PLCopen[®]' является зарегистрированным товарным знаком и совместно с логотипом PLCopen является собственностью ассоциации PLCopen.

Данный документ предоставляется "как есть" и в будущем может быть подвергнут изменениям и исправлениям. PLCopen не предоставляет никакие гарантии (явные или подразумеваемые), включая любые гарантии по поводу пригодности использования документа для конкретной цели. Ни при каких обстоятельствах PLCopen не несет ответственности за ущерб или убытки, вызванные ошибками в данном документе или его использованием.

Copyright © 2021 by PLCopen. Все права защищены

Дата публикации: 27.05.2021

Переводчик:	Е.Кислов
Редактор:	А.Осинский
Версия перевода:	1.0
Дата публикации:	12.07.2021

Оглавление

Оглавление.....	2
Список авторов и история версий.....	3
1. Введение	4
1.1. Цели рабочей группы	4
2. Базовые понятия ООП-расширений в 3-ей редакции стандарта МЭК 61131-3	5
2.1. 3-я редакция стандарта МЭК 61131-3	5
2.2. Основные понятия	5
2.3. Отличия в реализации ООП в разных языка программирования	7
3. Общий обзор	8
3.1. Основная информация	8
3.2. Модули и команды.....	8
4. Пример: управление бойлером.....	10
4.1. Обзор этапов подготовки примера	10
4.2. Функциональная схема примера	10
4.3. Структура примера с «классическим» подходом к программированию ПЛК	11
4.4. Адаптация примера под требования руководств PLCopen	12
4.5. Добавление обработки ошибок.....	16
4.6. Добавление списка тревог	23
5. Дополнительные вопросы	26
5.1. Соображения по основным моментам для руководства по ООП	26
5.2. Заметки по вопросам производительности.....	27
5.3. Заметки для будущих обсуждений	27

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	Interval Zero
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
0.6с	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. 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 обусловлено необходимостью более эффективно использовать ограниченные человеческие ресурсы инженеров-программистов. Поскольку многие из них, например, имеют опыт разработки мобильных приложений – то разумно адаптировать стандарт под современные технологии, которые являются общепринятыми в этой области.

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 РОУ (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 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 РОУ (за исключением класса)
РОУ	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 рекомендации 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

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

- Модули представляют собой обособленные части приложения и могут иметь иерархические зависимости друг от друга. Модуль объявляется с помощью ключевого слова **CLASS**;
- Команды соответствуют отдельным действиям модуля. Они могут быть связаны с одним модулем или с группой модулей, реализующих общий интерфейс. Команды объявляются с помощью ключевого слова **FUNCTION_BLOCK**.

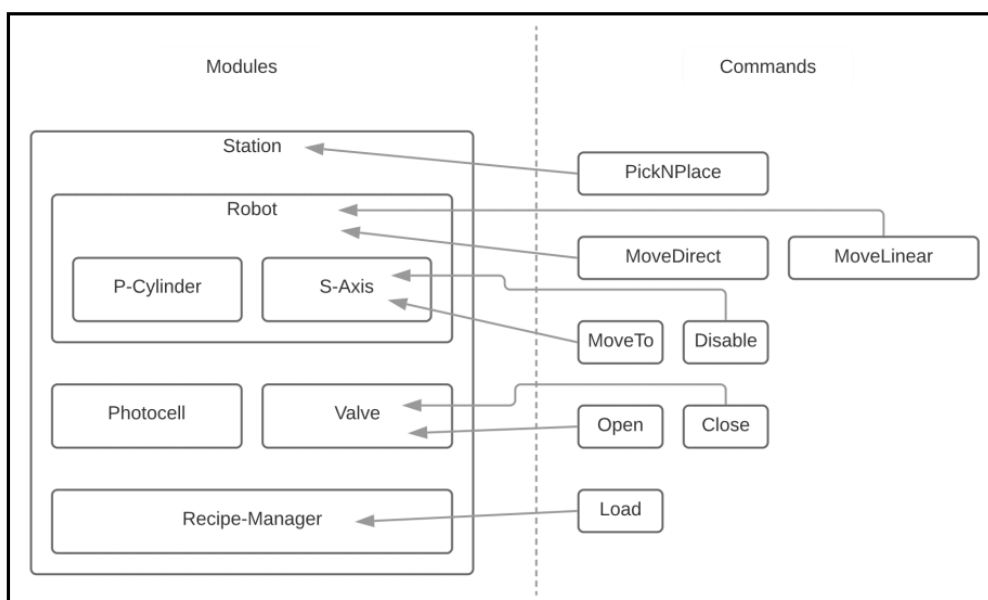


Рис. 1. Визуальное представление модулей и команд на примере интерфейсов **IModule** и **ICommand**

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 подход 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.

Ниже приведен пример ФБ **LConC** из [руководства по созданию библиотек от PLCopen](#). «LConC» означает «выполнение по уровню, без ограничения времени выполнения, с постоянным выполнением⁴».

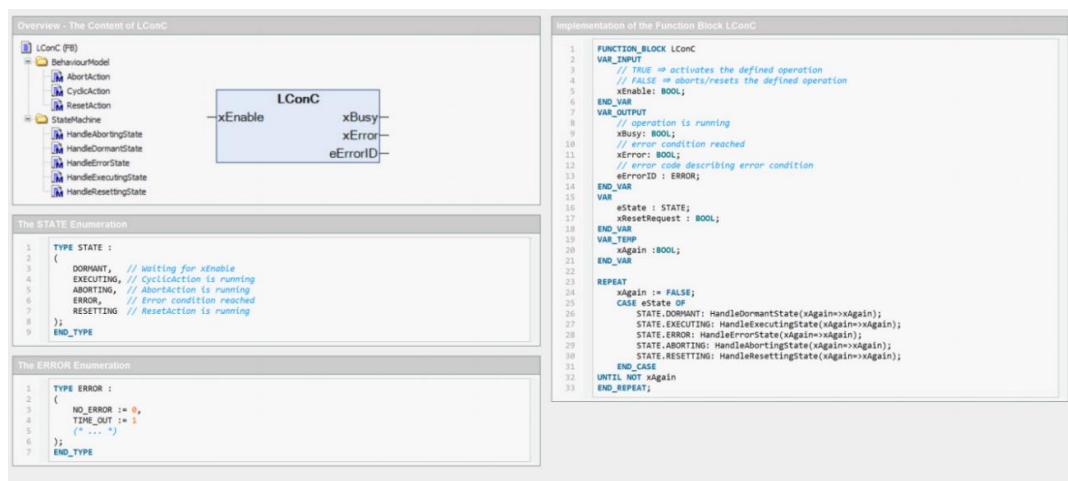


Рис. 2. Реализация ФБ LConC

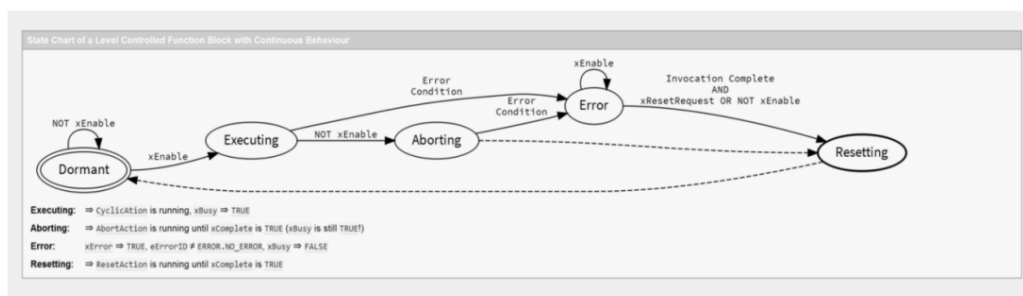


Рис. 3. Диаграмма состояний ФБ LConC

⁴ «Постоянное выполнение» подразумевает, что у блока нет выхода xDone – то есть у выполняемой им операции нет момента завершения

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 руководства 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:

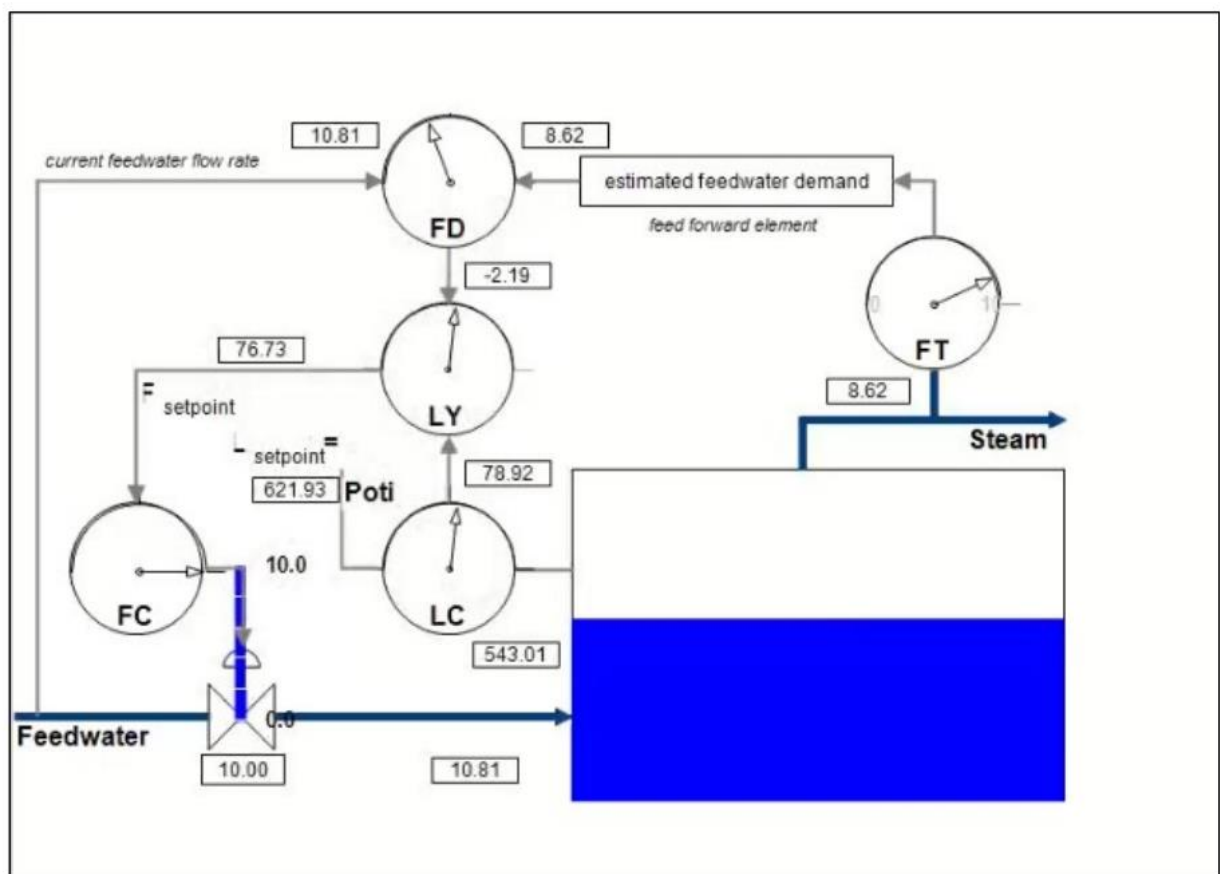


Рис. 4. Функциональная схема примера управления бойлером

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.

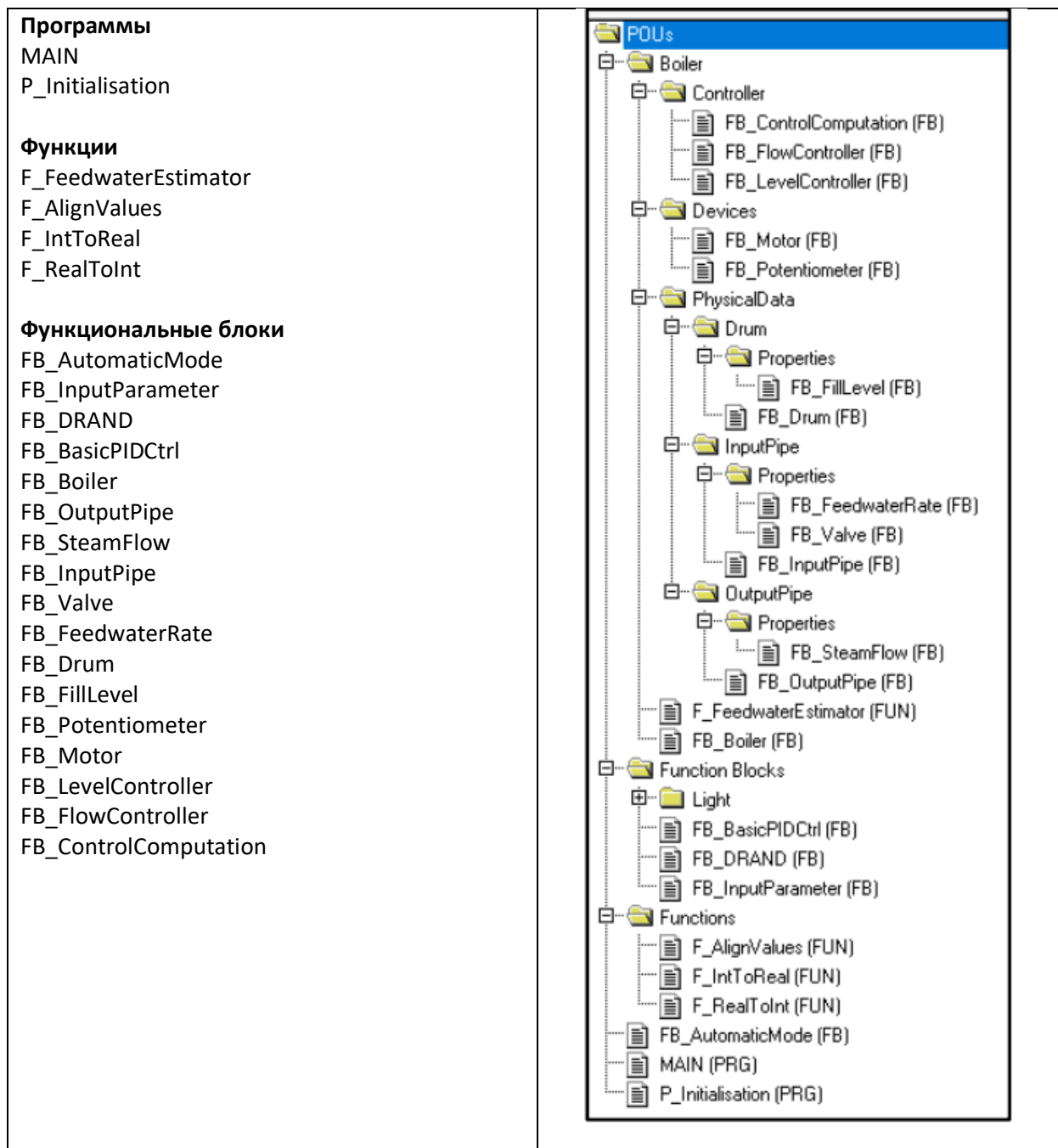


Рис. 5. Структура примера с использованием «классического» подхода к программированию ПЛК

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 ROU 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 ROU 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 ROU.

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](#).

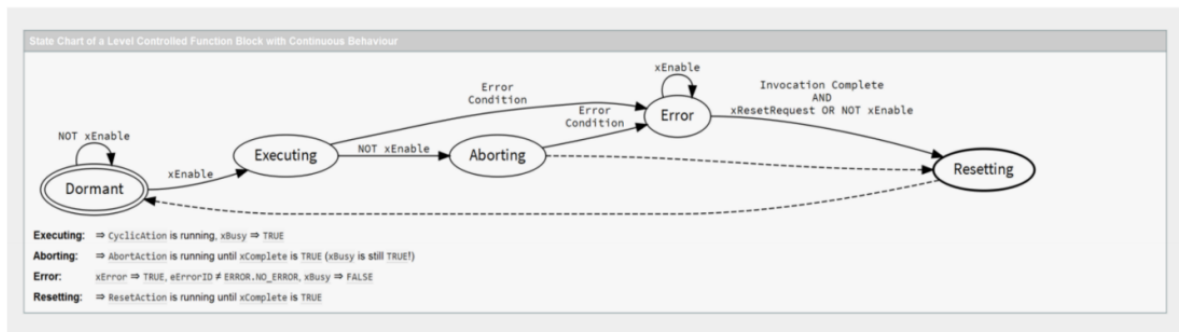
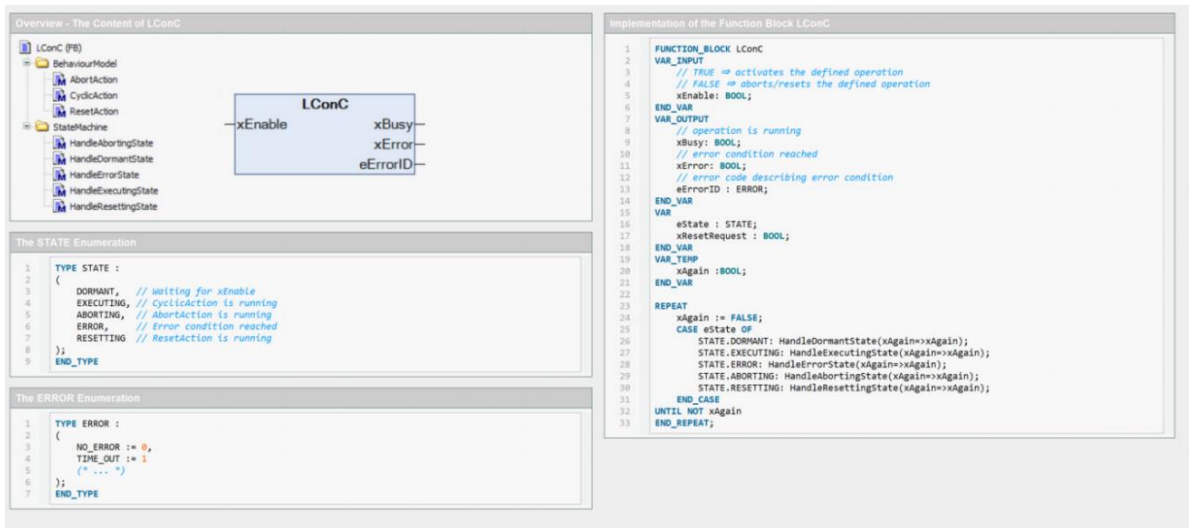


Рис. 6. Реализация и диаграмма состояний ФБ LConC

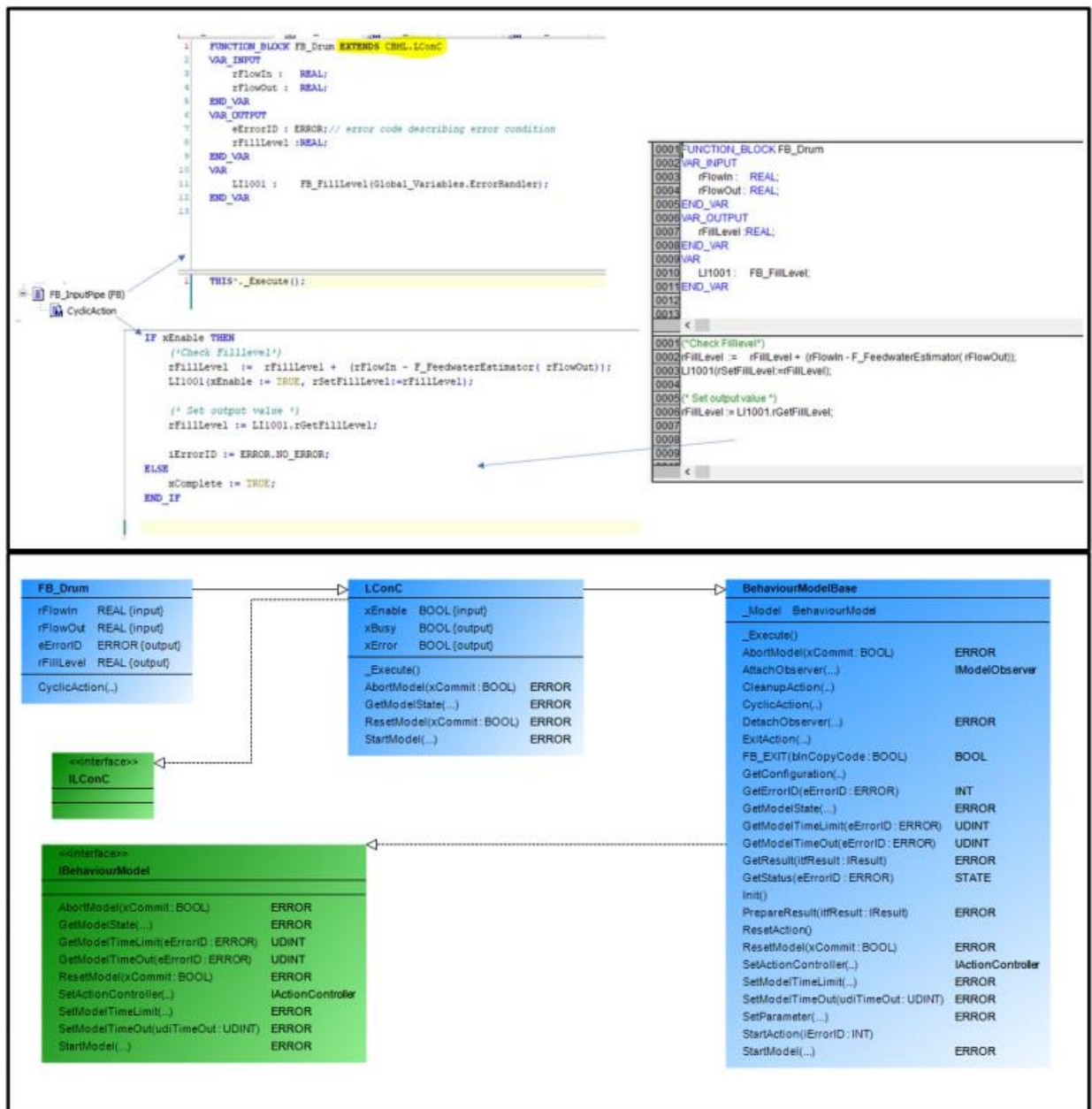


Рис. 7. Все функциональные блоки реализуют единый интерфейс

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.

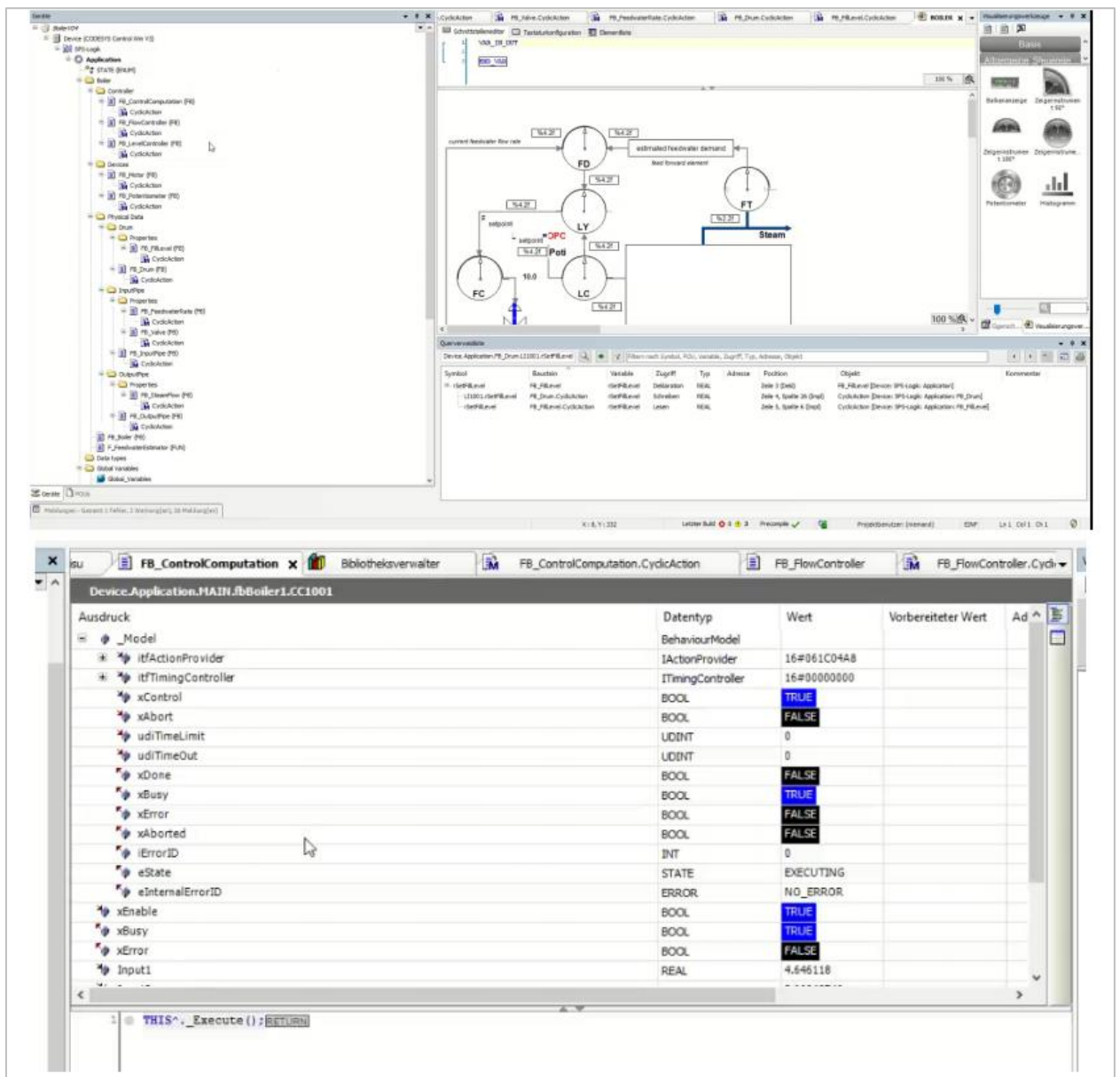


Рис. 8. Обзор состояния блока в симуляции

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. 7). 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 **aritifObserver** 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. 10).

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 12.

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

```

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

```

Рис. 9. Метод **AttachDevice** блока обработки ошибок


```

1 //Error Handler whcih 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

```

Рис. 10. 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;
  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

```

Рис. 11. 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. 12. 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.

```

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 IF NOT bInitDone THEN
12     P_Initialisation();
13     bInitDone := TRUE;
14
15     //
16 END_IF
17
18 counter := counter +1;
19
20 (*Get Input*)
21 fbInput1();
22
23 (*Simulate Boiler*)
24 fbBoiler1(
25     rSetFillLevel := fbInput1.rFillLevel,
26     rSteamDemand := fbInput1.rSteamDemand,
27     itfErrorHandler := fbErrorHandler
28 );
29
30 (*Monitor all Instance by IBehaviourModell*)
31 fbErrorHandler();

```

Рис. 13. 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. 14. 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 подход позволяет получить диагностическую информацию о всех экземплярах всех блоков проекта в одном массиве – достаточно вызвать для них метод **AttachDevice**.

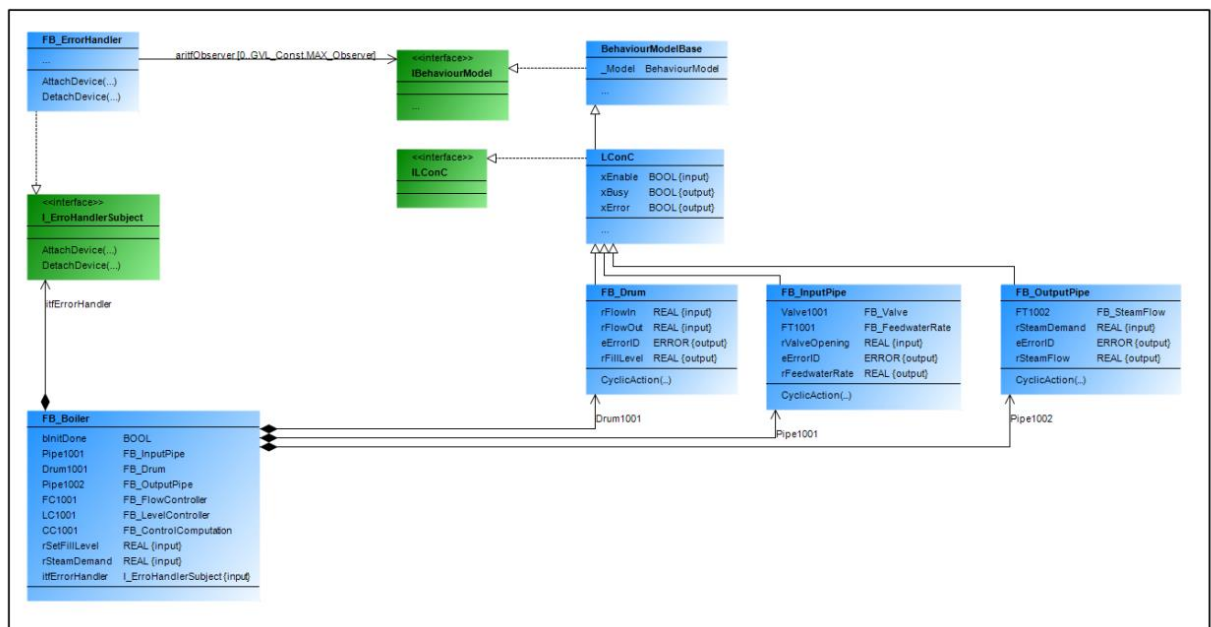


Рис. 15. Архитектура приложения

На рис. 15 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. 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.

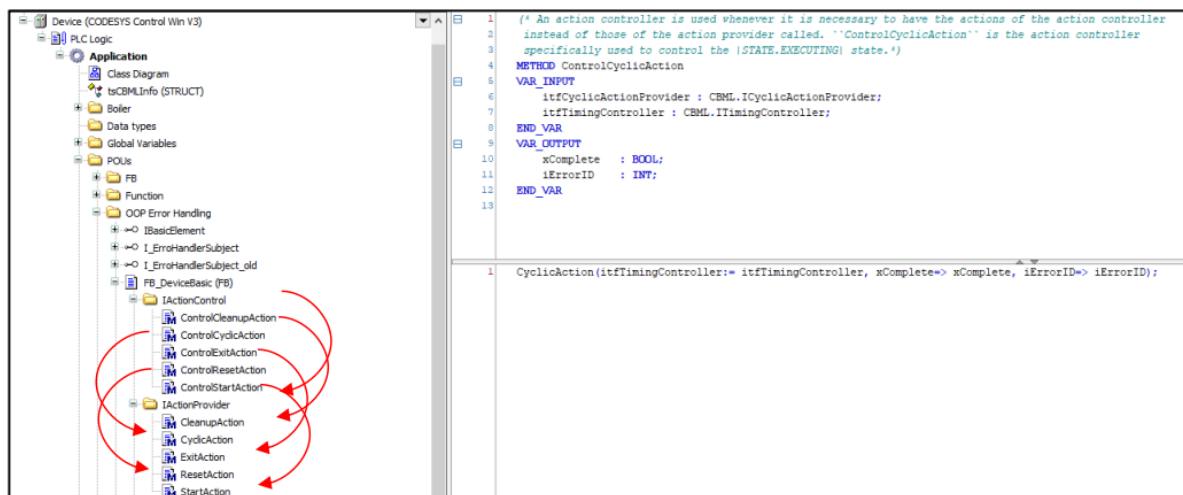


Рис. 16. Пример циклически вызываемого метода, связанного с интерфейсом **IActionControl**

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

```

FB_DeviceBasic.IBehaviourModel.Set
1  VAR
2  END_VAR
3
1  _iBehaviourModel := IBehaviourModel;
2  IBehaviourModel.ActionController := THIS^;

```

Рис. 17. 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. 18).

```

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);

```

Рис. 18. 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. 19. Вызов метода **AttachDevice** для отредактированного блока

Новая архитектура приложения приведена на рис. 20. 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**).

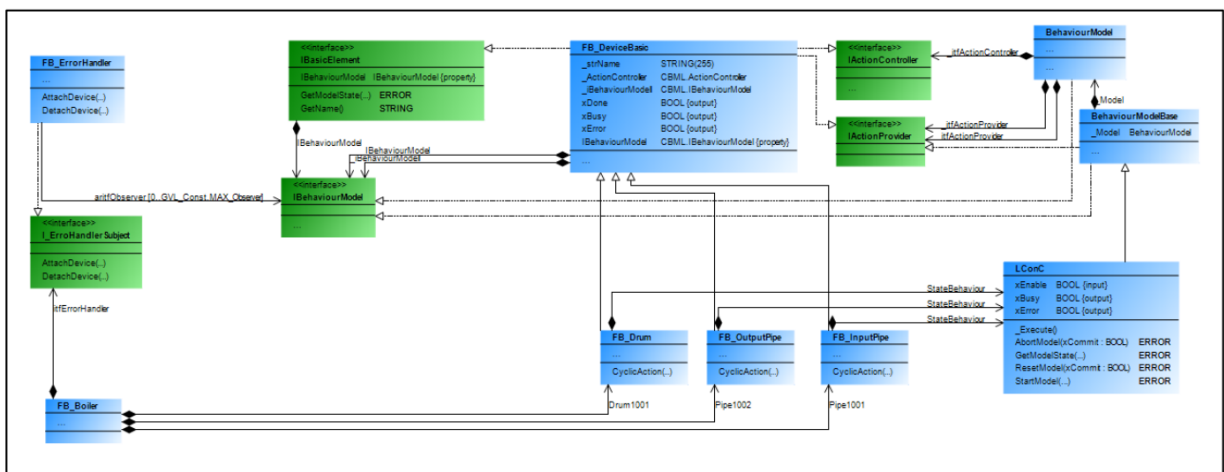


Рис. 20. UML-диаграмма новая архитектуры приложения

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](#).

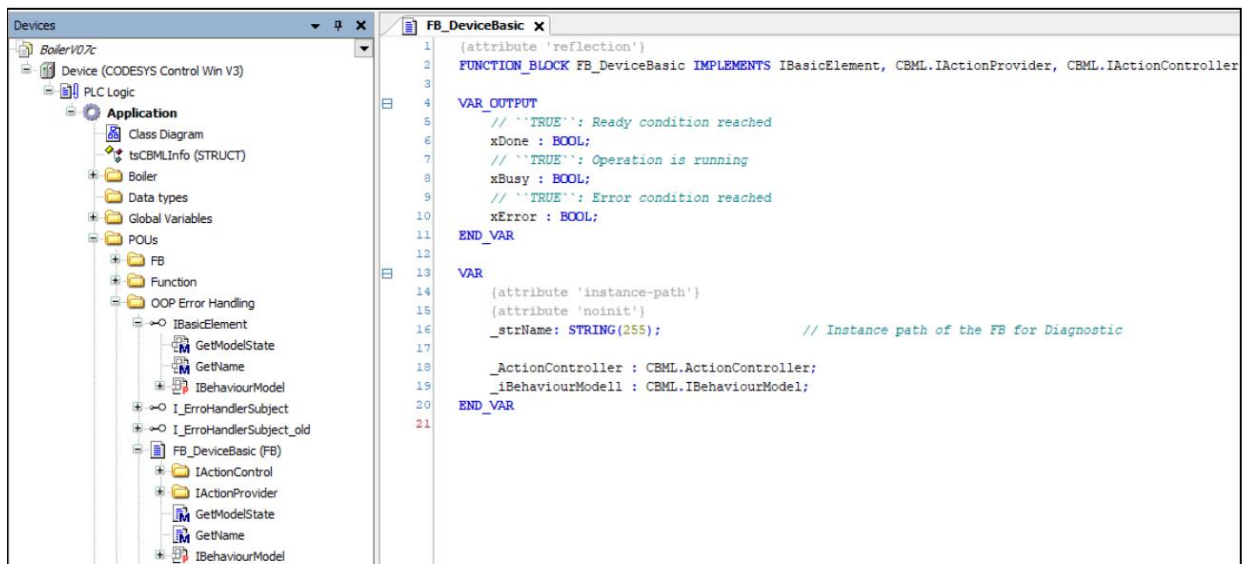


Рис. 21. Добавление метода GetName в базовом классе

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**). Поэтому 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:

```

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 : CML.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

```

Рис. 22. Реализация делегирования в методе **GetModelState**

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

Property	Type	Value
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. 23. Массив обработчика ошибок

Property	Type	Value
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. 24. Пример сообщения об ошибке

На рис. 25 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.

```

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     ELSEIF 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 ELSEIF _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     ELSEIF 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

```

Рис. 25. Код формирования сообщения об ошибке


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

Рис. 26. 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

В этом разделе перечислены вопросы, обсуждавшиеся на заседаниях рабочей группы, но на текущий момент недостаточно проработанные, чтобы быть включенными в основную часть документа.

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 архитекторы⁵ 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;
- Сборник паттернов проектирования. Необходимо создать библиотеку PLCore с интерфейсами, абстрактными классами и т.д., которая поможет при разработке приложений с использованием ООП.

⁵ Например, см. [Приёмы объектно-ориентированного проектирования. Паттерны проектирования](#) от «Банды Четырёх»

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 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.