Дитер Хесс

Объектно-ориентированные расширения МЭК 61131-3

Цели

В то время как в сфере компьютерных приложений объектно-ориентированное программирование (ООП) давно стало составной частью всех ведущих языков, в сфере контроллерных приложений оно применяется крайне редко. Говорят, что это происходит в силу некоторой консервативности, свойственной программистам контроллеров (ПЛК). Отчасти это действительно так. Но всё же в более значительной степени здесь сказываются ограниченные возможности инструментов программирования. Конечно, почти все современные контроллерные платформы дают возможность так или иначе использовать C++ (за дополнительные деньги). Однако компилятор обеспечивает только лишь аспекты чистого программирования. Функции отладки и ввода в эксплуатацию этих систем практически непригодны для контроллерных приложений. Даже для элементарного мониторинга значений входов-выходов приходится писать вызовы библиотечных функций. О таких приёмах, как «горячая» замена кода прикладной программы без остановки контроллера, вообще нужно забыть. Помимо этого автоматы и задачи с битовыми операциями реализуются в C++ достаточно сложно.

Требования

В результате компанией 3S-Smart Software Solutions было принято решение расширить нормы стандарта МЭК 61131-3, введя поддержку ООП в новое поколение системы программирования CoDeSys. Расширения стандарта должны подчиняться следующим требованиям:

- ООП-расширения должны быть не обязательными, а опциональными;
- ООП- и не ООП-программирование можно совмещать;
- существующие приложения должны полностью поддерживаться с возможностью их плавной трансформации в ООП с учетом целесообразности;
- ООП должно быть применимо во всех языках МЭК 61131-3;
- программист не должен сталкиваться со сложными определениями.

Расширения

Основное расширение МЭК 61131-3 касается превращения функционального блока (FUNCTION_BLOCK) в класс. Подобным образом структуры выросли в классы в языке C++. Это достигается введением методов. Фактически метод — это функция, встроенная в функциональный блок. В реализации функции доступны не только значения её параметров и локальных переменных, но и данные экземпляра функционального блока. В итоге вызов метода всегда включает имена экземпляра и метода.

Следующий пример показывает определение и вызов простого метода.

```
TYPE Direction: (Forward, Backward);
END_TYPE

FUNCTION_BLOCK Pump
VAR
  Enabled: BOOL;
  Direction: Direction;
END_VAR

METHOD GetState : BOOL
  GetState := Enabled;
END_METHOD

METHOD Start: BOOL  (* Метод Start *)
VAR_INPUT
  WantedDirection: Direction;
END_VAR
  Enabled := TRUE;
  Direction := WantedDirection;
END_METHOD
END_FUNCTION_BLOCK

PROGRAM Main
VAR
  Pump1: Pump;
  Pump2: Pump;
END_VAR
  Pump1.Start(Forward);   (*Вызов метода Start*)
  Pump2.Start(Backward);
END_PROGRAM
```

Естественно, вызов метода можно выполнить и в графических языках (рис. 1).

Даже если функциональный блок имеет методы, ничто не мешает использовать его обычным образом, как определено в стандарте МЭК 61131-3.

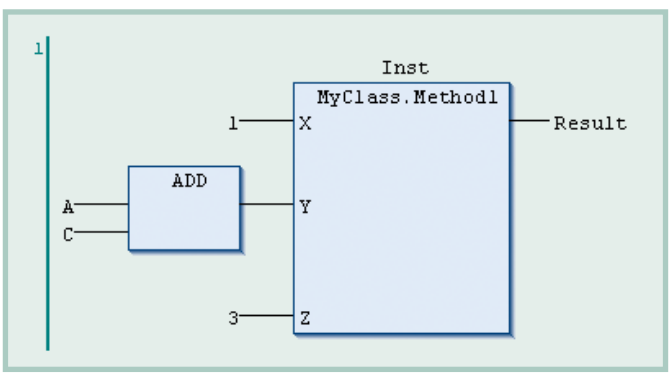

Рис. 1. Пример вызова метода в FBD

Помимо пользовательских методов и стандартной реализации, функциональный блок включает два предопределённых метода: **Init** и **Exit**. **Init** вызывается неявно для всех экземпляров всех функциональных блоков после загрузки кода приложения или «холодного» рестарта контроллера. **Exit** вызывается перед «горячим» обновлением кода экземпляра, перед сбросом или управляемым отключением питания ПЛК. Например, его можно применить для корректного завершения работы.

Для упрощения правила видимости заданы жёстко:

Тип элемента	Внешний доступ на чтение	Внешний доступ на запись	Внешний вызов
VAR	—	—	—
VAR_INPUT	—	√	—
VAR_OUTPUT	√	—	—
METHOD	—	—	√

Уже существующий класс может быть дополнен с помощью ключевого слова **EXTENTS**.

```
FUNCTION_BLOCK MonitoredPump EXTENTS Pump
VAR
  MonitoredState: (OK, Error);
END_VAR

METHOD HasError : BOOL;
  HasError := MonitoredState <> OK;
END_METHOD
END_FUNCTION_BLOCK

PROGRAM Main
VAR
  Pump1: Pump;
  Pump2: MonitoredPump;
END_VAR
  Pump1.Start(Forward);
  IF NOT Pump2.HasError THEN
    (* Все методы базового класса доступны *)
    Pump1.Start(Backward);
END_IF
END_PROGRAM
```

Однако реальную мощь ООП даёт возможность создания интерфейсов. Под интерфейсом понимается набор методов, работающих с одинаковыми параметрами, но разными реализациями для разных функциональных блоков. Интерфейс можно передать в качестве параметра, и программный компонент (POU) не будет в действительности заботиться о том, какой функциональный блок им применяется.

Следующий пример иллюстрирует данную технику:

```
INTERFACE Drive
METHOD HasError : BOOL;
END_METHOD

METHOD Home : BOOL;
END_METHOD

METHOD MoveAbsolute : BOOL;
VAR_INPUT
  Pos: DINT;
END_VAR
END_METHOD
END_INTERFACE
```

Теперь мы можем написать несколько функциональных блоков, реализующих интерфейс Drive (привод) с помощью ключевого слова **IMPLEMENTS**.

```
FUNCTION_BLOCK CANDrive IMPLEMETS Drive
VAR
  CANId: DINT;
  State: (OK, ParamError, DriveError, CommError);
  InHoming: BOOL;
END_VAR

METHOD HasError : BOOL;
  HasError: State <> OK;
END_METHOD

METHOD Home : BOOL;
  IF NOT InHoming THEN
  WriteSDO(CANId, 16#4711, 16#02, 1);(*Команда на исх.*)
  InHoming := TRUE;
  ELSE
    Home := ReadSDO(CANId, 16#4711, 16#03);
    InHoming := NOT Home;
  END_IF
END_METHOD

METHOD MoveAbsolute : BOOL;
VAR_INPUT
  Pos: DINT;
END_VAR
  ... (* Реализация абсолютного перемещения *)
END_METHOD

METHOD SetCanId : BOOL;
VAR_INPUT
  Id: DINT;
END_VAR
  CANId := Id;
END_METHOD
END_FUNCTION_BLOCK
```

Как можно видеть, все методы интерфейса Drive наполнены специальными реализациями, построенными на CAN-сообщениях. Сверх того здесь присутствуют некоторые специфические переменные и методы. В данном случае это метод, устанавливающий CAN Id. Далее мы могли бы описать еще один вид привода, например аналоговый (AnalogDrive). В нём можно реализовывать методы совершенно иначе, чем для цифрового привода (CANDrive).

Теперь можно написать функциональный блок, получающий интерфейс в качестве параметра:

```
FUNCTION_BLOCK InitMove
VAR_INPUT
  D: Drive;
  Pos: INT;
END_VAR

VAR_OUTPUT
  Done: BOOL;
END_VAR
  IF Drive.Home() THEN
    IF Drive.MoveAbsolute(Pos) THEN
      Done := TRUE;
    END_IF
  END_IF
END_FUNCTION_BLOCK
```

Данный POU сможет работать с разными типами приводов, причём обратите внимание, что никакой их дифференциации в нём нет.

```
VAR
  IM1, IM2: InitMove;
  DriveCAN1: CANDrive;
  DriveAna2: AnalogDrive;
END_VAR
  DriveCAN1.SetCANId(12);
  IM1(D := DriveCAN1, Pos := 100);
  IM2(D := DriveAna2, Pos := 200);
```

Можно легко применять интерфейсы так же, как обычные типы данных, например, создавать массивы. Это позволяет использовать следующий приём:

```
VAR
  AD: ARRAY[1..2] OF Drive;
  DriveCAN1: CANDrive;
  DriveAna2: AnalogDrive;
  I: INT;
END_VAR
  DriveCAN1.SetCANId(12);
  AD[1] := DriveCAN1;
  AD[2] := DriveAna2;
  FOR I := 1 TO 2 DO
    AD[I].Home();
  END_FOR
```

Заключение

Может возникнуть вопрос: насколько целесообразны или даже допустимы описанные расширения действующего стандарта МЭК 61131-3?

Дело в том, что главное требование стандарта состоит в выполнении однозначно описанных в нём конструкций, без каких-либо отклонений. Это полностью применимо к функциональным блокам, которые сохраняют все свойства «нормальных» функциональных блоков, несмотря на все нововведения.

Вы могли заметить, что описанные расширения языков программирования МЭК 61131-3 уже есть в других современных языках, таких как Java или C#. Однако инструментов, созданных на их основе специально для решения задач автоматизации, нет. Кроме того, переход на эти языки не соответствует практическим требованиям прикладных программистов.

Наконец, мы сталкиваемся с вопросом: действительно ли программистам ПЛК нужна технология ООП? Наши исследования тысяч приложений, созданных в CoDeSys, показали, что уже сейчас многие программисты пытаются реализовать конструкции ООП в своих проектах. Имея дело с абстрактными приводами, сетями или агрегатами машин, они создают функциональные блоки с поведением, управляемым специальными флагами. Это указывает на растущую необходимость появления объектного подхода в мире автоматизации. Достаточно многие пользователи 3S пытаются самостоятельно компенсировать отсутствие ООП, прилагая значительные усилия, чтобы иметь возможность автоматически генерировать код для однотипных приложений. Некоторые же открыто призывают нас к добавлению объектно-ориентированных функций.

Мир ПК прошёл тот же позитивный путь развития. Так, популярность языка Basic, предназначенного для самого широкого круга программистов, значительно возросла после добавления в него ООП-расширений.

Популярность CoDeSys в области промышленной автоматизации такова, что можно гарантировать непрерывную доработку и развитие новых функций. Кроме того, 3S будет продвигать включение объектно-ориентированных расширений в стандарт МЭК 61131-3. ●

Dieter Hess — директор
3S-Smart Software Solutions GmbH
Телефон: (+49-831) 540-310
Факс: (+49-831) 540-3150

Перевод с немецкого Стефании Клёкнер, Игоря Петрова
Телефон: (4812) 65-8171