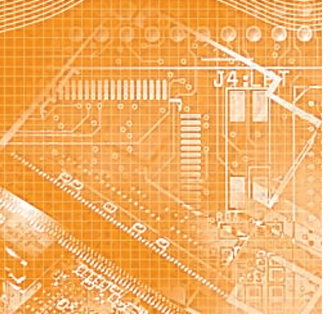


Технология OPC UA: возможности полноценного объектно-ориентированного проектирования цифровых коммуникаций промышленного оборудования

Максим Нейзов

Технология OPC UA поддерживает возможности объектно-ориентированного проектирования (ООП), но наличие этой поддержки ещё не гарантирует высокого качества принятых проектных решений. Для этого требуются как минимум знание основных принципов ООП и умение их применять на практике. В статье на примере проектирования цифровых коммуникаций с котельным оборудованием рассмотрены абстракция, наследование, композиция, инкапсуляция и полиморфизм, а также определён их положительный вклад в разработку.

Введение

Технология OPC уже длительное время является стандартом де-факто в области промышленной автоматизации. OPC UA (Unified Architecture) – последняя спецификация стандарта OPC, кардинально отличающаяся от предыдущих [1]. OPC UA имеет ряд особенностей – кроссплатформенность, сервис-ориентированная архитектура, сетевая безопасность и поддержка резервирования, передача данных в реальном времени, информационное моделирование и т.д. Нас будет интересовать последняя особенность.

Технология OPC UA позволяет вести разработку информационных моделей в объектно-ориентированном стиле. Цель статьи – продемонстрировать возможности технологии OPC UA в плане *объектно-ориентированной* разработки.

Важно, что само по себе наличие объектно-ориентированных возможностей ещё не гарантирует высокого качества спроектированных систем. Объектно-ориентированная парадигма требует «перестройки» мышления разработчиков. Так, при появлении объектно-ориентированных языков программирования многие программисты продолжали писать код по-старому – в структурном стиле [2]. При этом использовались далеко не все возможности объектно-ориентированных языков. Это как сменить отвёртку на шуруповёрт, но при этом вкручивать шурупы, вращая его рукой. Таким образом, вы имеете не шуруповёрт, а всё ту же отвёртку, только в форме шуруповёрта. Потребовалось длительное время, чтобы перейти от структурного программирования к объектно-ориентированному. Скорее всего, также потребуется некоторый период времени для перехода на полноценное *объектно-ориентированное проектирование* (ООП) цифровых коммуникаций в промышленности с использованием технологии OPC UA.

Для определения того, что такое ООП и какими характеристиками оно должно обладать, обратимся к классической работе [2] по объектно-ориентированному анализу и проектированию. ООП – методология проектирования, в основе которой лежит *объектная модель*. Проектируемая система состоит из взаимодействующих объектов, которые являются экземплярами определённых классов, организованных иерархически [2].

Мы рассмотрим следующие принципы ООП: *абстракция, иерархичность (наследование, композиция), инкапсуляция и полиморфизм.*

Задачи статьи:

- проверить наличие поддержки этих принципов технологией OPC UA;
- определить вклад данных принципов в разработку.

Для лучшего понимания сделаем это на конкретном примере: выполним ООП цифровых коммуникаций с котельным оборудованием, проведём анализ полученных результатов.

Постановка задачи проектирования

Имеется газовый водогрейный котёл, оборудование которого подключено к модулям ввода-вывода станции, предоставляющей доступ к данным по протоколу OPC UA. Также имеется контроллер (OPC-клиент), который подклю-

чён к станции (OPC-серверу). Контроллер предназначен для управления технологическим процессом нагрева воды. Задача: разработать соответствующую информационную модель котельного оборудования, которая будет поддерживаться OPC-сервером и использоваться OPC-клиентом. Дополнительным условием является то, что оборудование может меняться в процессе эксплуатации.

Разработка архитектуры решения

Если использовать OPC UA как «классический» OPC, тогда информационная модель просто будет содержать все имеющиеся сигналы ввода-вывода с доступом на чтение и запись, чтобы OPC-клиент мог управлять оборудованием напрямую. Но это будет плохим решением. Если поменяется оборудование, то потребуется вносить изменения и в информационную модель, и в управляющий алгоритм контроллера. Поддерживать такое решение будет крайне сложно.

Объектно-ориентированный подход позволяет эффективно бороться со сложностью информационной модели. Задача, которую преследует разработчик объектно-ориентированной системы, – создать иллюзию простоты у пользователя этой системы [2]. В нашем случае пользователем является управляющий алгоритм контроллера.

В управляющем алгоритме можно выделить две части: одна отвечает за управление оборудованием (драйверная часть), другая – за высокоуровневое управление технологическим процессом (технологическая часть). Драйверная часть зависит только от оборудования, технологическая часть, наоборот, вообще не зависит от оборудования, а зависит только от особенностей технологического процесса. Как непосредственно управлять котлом, решается на уровне его драйвера. Задачи более высокого уровня решаются технологическим алгоритмом. Это, например, может быть задача термостатирования или нагрева по заданному графику. Также на этом уровне определяются закон регулирования, уставки, настройки коэффициентов регулятора и т.д.

Замена оборудования будет приводить только к изменению драйверной части. А доступ к оборудованию обеспечивает OPC-сервер. В таком случае хорошим декомпозиционным решением будет оставить в контроллере только технологическую часть алгоритма (далее – технологический алгоритм), а драйверную часть (далее – алгоритм драйвера) перенести на OPC-сервер.

Такое разделение обеспечит относительную независимость технологического алгоритма в контроллере и алгоритма драйвера на OPC-сервере. OPC-сервер будет предоставлять полноценный сервис для работы с оборудованием. В итоге при смене оборудования не потребуется вносить изменения в технологический алгоритм контроллера. Потребуется внести изменения только на OPC-сервере. Если информационная модель будет иметь хорошую архитектуру, то эти изменения будут незначительными. Такое решение будет легко поддерживать.

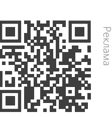
Теперь возникает задача ООП информационной модели котельного оборудования, предоставляющей некоторый сервис. OPC-сервер должен обеспечивать иллюзию простоты работы с оборудованием для OPC-клиента.

- 2 в 1: программируемый источник питания и электронная нагрузка в одном приборе
- Двунаправленная мощность с автодиапазонным выходом
- Полностью цифровой контроль и управление (U, I, P, R)
- КПД до 96%
- Опциональное герметичное водяное охлаждение
- Установленные интерфейсы (аналоговый, LAN, USB)
- Слот Anybus для установки дополнительных интерфейсов
- Моделирование (батареи, PV, FC), встроенный генератор функций
- Мощность 1,5; 3; 5; 10; 15 и 30 кВт, ширина 19", высота от 2U до 4U

ОФИЦИАЛЬНЫЙ ДИСТРИБЬЮТОР

(495) 234-0636
INFO@PROSOFT.RU

WWW.PROSOFT.RU



Реклама

Абстракция

Абстракция – это модель, содержащая только существенные характеристики для решения поставленной задачи. Выбранная абстракция определяет интерфейс, посредством которого с ней осуществляется взаимодействие. Интерфейс связан с набором обязательств, которые необходимо выполнять, чтобы обеспечить поддержание выбранной абстракции.

Потенциальное множество всех объектов с одинаковыми характеристиками образуют *класс*. Отсюда следует, что классов существенно меньше, чем объектов. Поэтому при проектировании лучше описать сразу весь класс, а потом использовать объекты как экземпляры этого класса. У класса есть *атрибуты* (свойства) и *методы*. Атрибуты описывают состояние объекта, а методы – операции, которые может выполнять объект. Совокупность методов и их обязательств определяет поведение объекта. Открытые (доступные снаружи объекта) атрибуты и методы образуют интерфейс. *Класс* и *тип* объекта при определённых условиях можно считать синонимами [2].

В нашем примере OPC-сервер должен предоставить простую абстракцию OPC-клиенту для взаимодействия с котельным оборудованием. Пусть этой абстракцией будет котёл. Для этого определим класс *Boiler* (см. UML-диаграмму на рис. 1). Назначение котла – нагрев воды, поэтому существенной характеристикой является температура воды на выходе котла. Для этого определим атрибут *outWaterTemp* типа *Double*, который будет предоставлять текущее значение температуры в виде вещественного числа. Данный атрибут доступен для OPC-клиента (открытые члены класса обозначаются знаком +), но предназначен только для чтения (в фигурных скобках указано это ограничение).

Также для работы с котлом необходимо знать его состояние (остановлен, работает, запускается, неисправен и т.д.). Для этого определим атрибут *status* типа *Int16*, предназначенный только для чтения (OPC-клиент не может изменять его значение). Статус котла будет кодироваться целым числом, сами коды пока неизвестны.

Простой абстракцией для управления котлом является набор из двух команд: запустить и остановить нагрев. Для этого определим методы *startHeating()* и *stopHeating()* соответственно. Эти методы не имеют параметров и ничего не возвращают. Названия методов выделены курсивом – это означает, что они абстрактные, т.е. не имеют реализации, так как на данном этапе проектирования ещё неизвестно, как именно запускать и останавливать котёл.

Таким образом, класс *Boiler* задаёт простой интерфейс для работы с абстрактным котлом. Название класса *Boiler* также выделено курсивом – это означает, что класс абстрактный, т.е. невозможно создать экземпляры этого класса, так как не спроектированы многие вещи, и использовать такой объект пока нельзя.

Рис. 1. UML-диаграмма классов котлов

Предложенная абстракция фокусирует разработку только на существенных характеристиках, устраняя всё лишнее. В итоге это упрощает и процесс проектирования, и полученный результат.

Иерархия

Иерархия используется для упорядочивания абстракций по уровням [2]. Рассмотрим два механизма, порождающих иерархию: *наследование* и *композиция*.

Наследование – заимствование классом не скрытых атрибутов и методов у другого класса. Класс, который наследует, является *дочерним*. Класс, от которого наследуют, является *родительским*. Наследование порождает иерархию классов с отношением «обобщение-специализация». Дочерний класс является специализацией родительского, и наоборот, родительский класс является обобщением дочернего.

Композиция – объединение частей. Композиция порождает иерархию объектов с отношением «часть-целое». Компонентные объекты являются частями композиционного (целого) объекта.

В нашем примере оборудование может меняться – проработаем его возможные специализации и организуем это в иерархии классов. Допустим, котёл может быть с электрическим или газовым нагревом. Для этого определим два класса: *ElectricBoiler* и *GasBoiler* соответственно. Оба класса наследуют атрибуты и методы родительского класса *Boiler* (родительский класс указывают стрелкой на UML-диаграмме). Кроме того, в классах указана их специализация: электрический котёл имеет нагреватель (атрибут *heater* типа *ElectricHeater*), а газовый котёл – газовую горелку (атрибут *burner* типа *GasBurner*). Класс *GasBoiler* задаёт иерархию объектов: горелка (объект *burner*) является частью газового котла (объекта класса *GasBoiler*).

Класс *GasBurner* определяет абстрактную газовую горелку и является корнем уже другой иерархии классов (см. UML-диаграмму на рис. 2). Абстрактная горелка позволяет подать искру для розжига – для этого атрибут *spark* типа *Boolean* нужно установить в *True*. Также горелка имеет датчик наличия пламени (атрибут *flame* типа *Boolean* только для чтения).

По способу подачи воздуха горелки могут быть атмосферными или наддувными – этому соответствуют два дочерних класса *AtmoGasBurner* и *AirBoostGasBurner* соответственно. Для подачи газа атмосферная горелка имеет газовый

клапан (атрибут *gasValve* типа *Boolean*, доступный для чтения и записи). Подчеркнём, что конструктивно клапан может и не быть в составе горелки, т.е. класс отражает не физическую, а логическую структуру.

Далее определим класс *BurnerModel-M1* – класс атмосферных горелок конкретной модели M1. Данный класс не является абстрактным (имя класса на UML диаграмме не выделено курсивом), т.е. возможно создавать экземпляры данного класса. Это отражает тот факт, что мы можем воспользоваться только конкретной горелкой определённой модели.

Наддувные газовые горелки (класс *AirBoostGasBurner*) могут быть одноступенчатыми (дочерний класс *OneStepGas-Burner*) и модулируемыми (дочерний класс *ModulatedGasBurner*). Наддувные одноступенчатые горелки (класс *One-StepGasBurner*) имеют газовый (атрибут *gasValve*) и воздушный (атрибут *airValve*) клапаны. Наддувные модулируемые горелки (класс *ModulatedGasBurner*) будут иметь регулируемый газовый клапан и регулируемый воздушный вентилятор.

Фактическое положение регулируемого газового клапана будет отображаться в виде вещественного значения (тип *Double*) атрибута *gasValvePosition*. Для задания уставки положения газового клапана будет использоваться метод *setGasValvePosition* (*value: Double*), который принимает на вход значение уставки *value* вещественного типа. При вызове метода происходит передача уставки на сервопривод клапана.

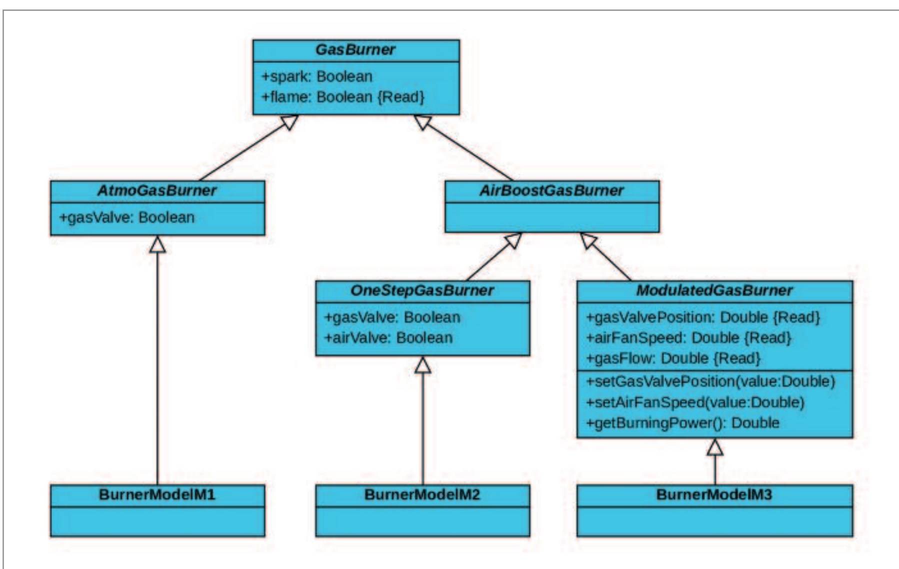


Рис. 2. UML-диаграмма классов газовых горелок

Фактическая скорость регулируемого воздушного вентилятора будет отображаться в виде вещественного значения (тип *Double*) атрибута *airFanSpeed*. Для задания уставки скорости вентилятора будет использоваться аналогичный метод *setAirFanSpeed* (*value: Double*). При вызове метода происходит передача уставки на преобразователь частоты вентилятора.

Решения на DIN-рейку от Delta Electronics

— Источники питания от 7 до 960 Вт с выходными напряжениями 5, 12, 24, 48 В
— ИБП постоянного тока 24 В/24 В с током нагрузки до 40 А
— Модули резервирования N+1, 1+1
— Буферные модули со временем удержания питания от 200 мс до 8 с
— Батарейные модули (для монтажа двух батарей 7-9 Ач)

DELTA

PROSOFT®
Официальный дистрибьютор
+7 (495) 234-06-36
info@prosoft.ru
www.prosoft.ru
Реклама

У горелки будет производиться измерение массового расхода газа (атрибут *gasFlow* типа *Double* только для чтения). Для определения тепловой мощности горения будет вызываться метод *getBurningPower()* без параметров, возвращающий значение типа *Double*. При вызове метода происходит вычисление мощности горения на основании измеренного расхода газа.

Далее определим классы конкретных горелок. *BurnerModelM2* – класс наддувных одноступенчатых горелок модели M2, *BurnerModelM3* – класс наддувных модулируемых горелок модели M3.

Продолжим прорабатывать ветку газовых котлов (см. рис. 1). Газовые котлы (класс *GasBoiler*) могут быть с естественной тягой (дочерний класс *NaturalPullGasBoiler*) и принудительной (дочерний класс *ForcedPullGasBoiler*). Газовые котлы с принудительной тягой (класс *ForcedPullGasBoiler*) имеют вытяжной вентилятор (атрибут *exhaustFan* типа *Boolean*).

Определим класс газовых котлов с естественной тягой конкретной модели – *BoilerModelM1*. Пусть эти котлы имеют в своём составе атмосферную горелку модели M1 (атрибут *burner* абстрактного типа *GasBurner* уточнили его конкретным подтипом *BurnerModelM1*).

Определим класс газовых котлов с принудительной тягой модели M2 (класс *BoilerModelM2*), которые имеют в своём составе наддувную одноступенчатую горелку модели M2 (класс *BurnerModelM2*). Определим ещё один класс газовых котлов с принудительной тягой модели M3 (класс *BoilerModelM3*), которые имеют в своём составе наддувную модулируемую горелку модели M3 (класс *BurnerModelM3*). В итоге, все неабстрактные классы котлов имеют неабстрактные типы горелок. Это отражает тот факт, что реальный котёл конкретной модели всегда имеет реальную горелку, модель которой также известна.

Предложенные иерархии классов и объектов упорядочивают по уровням абстракции котельного оборудования, что позволяет их эффективно хранить, проектировать, дорабатывать и использовать.

Инкапсуляция

Инкапсуляция – скрытие внутреннего устройства (реализации) объекта [2]. Цель инкапсуляции – обеспечить изоляцию между интерфейсом и реализацией объекта, что позволяет менять реализацию, не затрагивая интерфейс. В скрытую реализацию невозможно вмешательство снаружи, что позволяет реализации сохранять заданные инварианты состояния и поведения объекта.

В нашем примере OPC-клиент может управлять горелкой любой модели напрямую, изменяя доступные для записи атрибуты объекта. Например, у экземпляра класса *BurnerModelM1* можно изменять значения атрибутов *spark* и *gasValve*, управляя тем самым подачей искры и газовым клапаном. Но когда разрозненное оборудование (горелка и вытяжной вентилятор) используются в составе котла, их работа должна быть согласована (например, нельзя зажигать горелку без команды запуска котла, нельзя выключать вентилятор во время горения и т.д.).

Выполним инкапсуляцию – скроем детали реализации управления оборудованием котла. Для этого разрешим только чтение атрибутов вложенного объекта *burner* (см. классы *BoilerModelM1*… *Boiler-ModelM3* на рис. 1). Также сделаем доступным только для чтения атрибут *exhaustFan* класса *ForcedPullGasBoiler*. Теперь OPC-клиент не сможет управлять оборудованием напрямую. Управление горелкой (*burner*) и вытяжным вентилятором (*exhaustFan*) будет доступно только из методов, относящихся к котлу (*GasBoiler*).

При управлении этим оборудованием потребуется работа с таймером и генератором импульсов. Таймер может использоваться для реализации выдержки времени, необходимой, например, для зажигания пламени горелки или продувки топки газового котла. Генератор импульсов может использоваться для активации периодической проверки определённых условий, например, возникновения аварийной ситуации. Необходимость в таймере и генераторе возникает на уровне абстракции газового котла, поэтому определим атрибуты *timer* типа *Timer* и *generator* типа *PulseGenerator* в классе *GasBoiler*. Таким образом, объекты *timer* и *generator* будут вложены в любой объект типа *GasBoiler*. Например, объект типа *BoilerModelM1* также является и объектом типа *GasBoiler*, поэтому он унаследует таймер и генератор.

Объект *timer* по истечении заданного времени после запуска будет вызывать метод *timeout()*, а объект *generator* будет вызывать метод *tick()* с заданной периодичностью. Данные объекты и методы являются защищёнными (помечены знаком #). Защищённые члены класса недоступны извне, а доступны только для дочерних классов. Методы определены как абстрактные, так как неизвестна их реализация на данном уровне иерархии.

Как именно должен работать котёл, становится известно только на уровне определения класса конкретной модели котла. Поэтому реализация защищённых методов *timeout()* и *tick()*, а также открытых методов *startHeating()* и *stopHeating()* определена на уровне неабстрактных классов *BoilerModelM1*… *BoilerModelM3*. Здесь данные методы уже не являются абстрактными (имена методов на рис. 1 не выделены курсивом). На этом уровне полностью известна внутренняя реализация класса (класс не содержит ни одного абстракт-

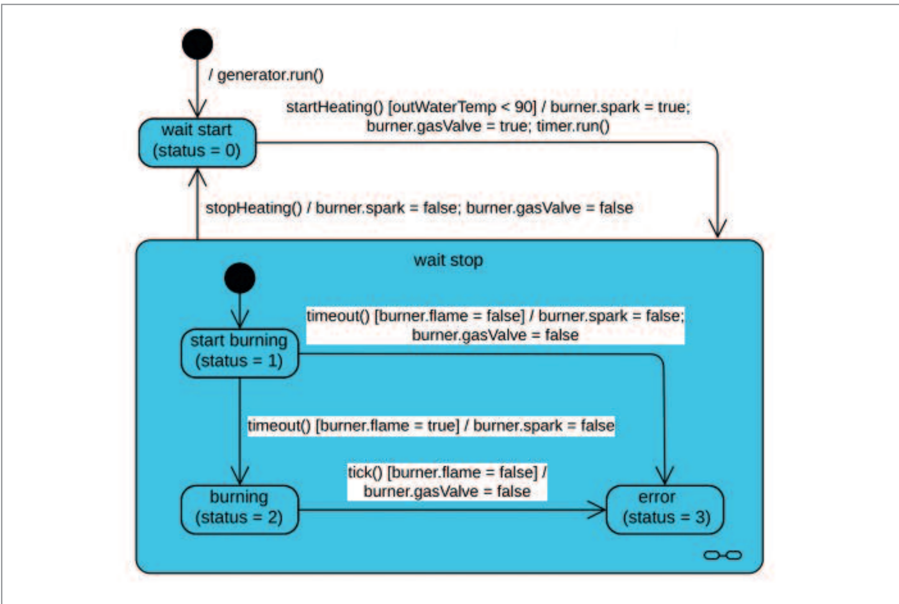


Рис. 3. UML-диаграмма состояний котла модели M1

ного метода), что позволяет создавать и использовать его экземпляры.

Для примера опишем требуемое поведение котла модели М1 (класс *BoilerModelM1*) с помощью *UML-диаграммы состояний* (см. рис. 3). При инициализации начального состояния запускается генератор, т.е. он будет всегда работать и периодически (например, каждые 0,5 с) вызывать метод *tick()*. В начальном состоянии (*wait start*) котёл ожидает запуска. Каждое состояние имеет свой код, который хранится в виде значения атрибута *status*. При событии вызова метода *startHeating()*, если температура воды (*outWaterTemp*) меньше 90 градусов, то подаётся искра (*spark*), открывается газовый клапан (*gasValve*) и запускается таймер контрольного времени розжига горелки. Также происходит переход в составное состояние ожидания останова котла (*wait stop*). Здесь начальным состоянием является состояние запуска горения (*start burning*). В итоге *status* примет значение 1. По истечении контрольного времени (например, 4 с) возникает событие вызова метода *timeout()*. Если у горелки есть пламя (*flame* = *true*), то отключается подача искры (*spark* = *false*) и происходит переход в состояние горения (*burning*). Если у горелки нет пламени, то отключается подача искры, закрывается газовый клапан (*gasValve* = *false*) и происходит переход в состояние ошибки (*error*). Если в состоянии горения (*burning*) пламя погасло, то при очередном событии вызова метода *tick()* будет закрыт газовый клапан и произойдёт переход в состояние ошибки (*error*). В составном состоянии (*wait stop*) котёл ожидает останова. При событии вызова метода *stopHeating()* горелка немедленно прекращает свою работу и происходит переход в начальное состояние *(wait start)*.

Реализация всех четырёх методов класса *BoilerModelM1* задана на диаграмме состояний. Например, метод *startHeating()* проверяет два условия: (*status* = *0*) и (*outWaterTemp* < *90*). Если оба условия выполняются, то *spark* и *gasValve* устанавливаются в *True*, запускается таймер – *timer.run()*, *status* устанавливается в 1. Если хотя бы одно из условий не выполняется, то ничего не происходит. Остальные методы строятся аналогично.

Драйвер для управления газовым котлом задаётся набором из этих четырёх методов: *timeout()*, *tick()*, *startHeating()*, *stopHeating()*. Для каждой модели котла разрабатывается свой драйвер, который должен подгружаться OPC-сервером при создании объекта.

Драйвер может быть более сложным, чем в приведённом примере, но вся эта сложность скрыта от OPC-клиента. Для управления котлом со сложным алгоритмом запуска и останова OPC-клиент всегда использует только два простых метода: *startHeating()* и *stopHeating()* соответственно. Драйвер для управления газовым котлом может быть изменён, но это никак не повлияет на работу OPC-клиента – для него интерфейс с объектом OPC-сервера, представляющим реальный котел, остаётся неизменным. Таким образом, изменения инкапсулированной реализации класса прозрачны для OPC-клиента.

Инкапсуляция делает реализацию класса недоступной, т.е. исключает любое влияние на неё извне. В итоге за поведение оборудования отвечает только реализация. Следовательно, требования корректности управления обору-

дованием уже относятся не ко всей системе в целом, а только к реализации класса. Таким образом, инкапсуляция упрощает задачу обеспечения корректного управления оборудованием.

Полиморфизм

Полиморфизм – возможность использовать методы родительского класса независимо от того, какие они имеют реализации в дочерних классах.

В нашем примере родительский класс *GasBoiler* имеет два защищённых метода (*timeout*, *tick*) и два открытых (*start-Heating*, *stopHeating*), унаследованных от класса *Boiler*. Технологический алгоритм можно спроектировать для работы только с конкретной моделью котла, например, с экземпляром класса *Boiler-ModelM1*. Но тогда при изменении модели котла придётся вносить изменения и в технологический алгоритм. Этого можно избежать, если спроектировать технологический алгоритм для работы со всеми газовыми котлами. Для этого потребуется вместо класса *BoilerModelM1* указать класс *GasBoiler*.

В итоге технологический алгоритм (OPC-клиент) будет вызывать открытые методы класса *GasBoiler*, а OPC-сервер уже будет их отрабатывать в соответствии с конкретным типом созданного объекта (*BoilerModelM1*…*BoilerModelM3*). Таким образом, полиморфизм позволяет менять котлы, имеющие различные алгоритмы работы, не затрагивая технологический алгоритм управления этими котлами в контроллере.

Разработка OPC-сервера

Разработка OPC-сервера может осуществляться на платформе .NET (язык C#) с помощью библиотеки от OPC Foundation [3]. Выполним описание разработанной информационной модели (две UML-диаграммы классов) в формате XML. Далее этот XML-файл преобразуется с помощью компилятора моделей [4] в набор файлов для среды разработки.

Листинг 1 содержит XML-описание типа (класса) *GasBurner* (рис. 2). Тег описывает тип объекта (*opc:ObjectType*), указывая его имя (*SymbolicName*), базовый тип (*BaseType*) и абстрактность (*IsAbstract*). Тип с именем *GasBurner* унаследован от базового типа *Base-ObjectType*, принятого в OPC UA, и является абстрактным. Тег *opc:Children* описывает открытые (доступные для клиента) атрибуты и методы. Здесь определены две переменные (*opc: Variable*) типа *Boolean*. Уровень доступа задан в *AccessLevel*: переменная *spark*

Листинг 1

```
<opc:ObjectType SymbolicName="GasBurner"
                BaseType="ua:BaseObjectType"
                IsAbstract="true">
    <opc:Children>
        <opc:Variable SymbolicName="spark"
                      DataType="ua:Boolean"
                      ValueRank="Scalar"
                      TypeDefinition="ua:BaseDataVariableType"
                      ModellingRule="Mandatory"
                      AccessLevel="ReadWrite"/>
        <opc:Variable SymbolicName="flame"
                      DataType="ua:Boolean"
                      ValueRank="Scalar"
                      TypeDefinition="ua:BaseDataVariableType"
                      ModellingRule="Mandatory"
                      AccessLevel="Read"/>
    </opc:Children>
</opc:ObjectType>
```

Листинг 2

```
<opc:ObjectType SymbolicName="AtmoGasBurner"
                BaseType="GasBurner"
                IsAbstract="true">
    <opc:Children>
        <opc:Variable SymbolicName="gasValve"
                      DataType="ua:Boolean"
                      ValueRank="Scalar"
                      TypeDefinition="ua:BaseDataVariableType"
                      ModellingRule="Mandatory"
                      AccessLevel="ReadWrite"/>
    </opc:Children>
</opc:ObjectType>
```

Листинг 3

```
<opc:Method SymbolicName="ActionMethodType"></opc:Method>
<opc:ObjectType SymbolicName="Boiler"
                BaseType="ua:BaseObjectType"
                IsAbstract="true">
    <opc:Children>
        <opc:Variable SymbolicName="outWaterTemp"
                      DataType="ua:Double"
                      ValueRank="Scalar"
                      TypeDefinition="ua:AnalogItemType"
                      ModellingRule="Mandatory"
                      AccessLevel="Read"/>
        <opc:Variable SymbolicName="status"
                      DataType="ua:Int16"
                      ValueRank="Scalar"
                      TypeDefinition="ua:BaseDataVariableType"
                      ModellingRule="Mandatory"
                      AccessLevel="Read"/>
        <opc:Method SymbolicName="startHeating"
                    TypeDefinition="ActionMethodType"
                    ModellingRule="Mandatory"/>
        <opc:Method SymbolicName="stopHeating"
                    TypeDefinition="ActionMethodType"
                    ModellingRule="Mandatory"/>
    </opc:Children>
</opc:ObjectType>
```

Листинг 4

```
<opc:ObjectType SymbolicName="GasBoiler"
                BaseType="Boiler"
                IsAbstract="true">
    <opc:Children>
        <opc:Object SymbolicName="burner"
                    TypeDefinition="GasBurner"
                    ModellingRule="Mandatory">
        </opc:Object>
    </opc:Children>
</opc:ObjectType>
```

Листинг 5

```
<opc:ObjectType SymbolicName="BoilerModelM1"
                BaseType="NaturalPullGasBoiler"
                IsAbstract="false">
  <opc:Children>
        <opc:Object SymbolicName="burner"
                TypeDefinition="BurnerModelM1"
                ModellingRule="Mandatory">
                <opc:Children>
                    <opc:Variable SymbolicName="spark"
                            AccessLevel="Read"/>
                    <opc:Variable SymbolicName="gasValve"
                            AccessLevel="Read"/>
                </opc:Children>
        </opc:Object>
  </opc:Children>
</opc:ObjectType>
```

Листинг 6

```
<opc:Object SymbolicName="BoilerM1"
          TypeDefinition="BoilerModelM1">
  <opc:References>
      <opc:Reference IsInverse="true">
          <opc:ReferenceType>ua:Organizes</opc:ReferenceType>
          <opc:TargetId>ua:ObjectsFolder</opc:TargetId>
      </opc:Reference>
  </opc:References>
</opc:Object>
<opc:Object SymbolicName="BoilerM2"
          TypeDefinition="BoilerModelM2">
...
<opc:Object SymbolicName="BoilerM3"
          TypeDefinition="BoilerModelM3">
...
```

доступна для чтения и записи, переменная *flame* – только для чтения.

Листинг 2 содержит XML-описание типа (класса) *AtmoGasBurner* (рис. 2). Тип *AtmoGasBurner* унаследован от определённого ранее типа *GasBurner* и является абстрактным. Здесь добавляется одна переменная *gasValve* типа *Boolean*, доступная для чтения и записи.

Листинг 3 содержит XML-описание типа (класса) *Boiler* (рис. 1). Тег *opc:Method* определяет тип (*ActionMethodType*) используемых далее открытых методов. Здесь указано, что методы – это некоторые действия без входных и выходных параметров. Далее тег *opc:ObjectType* описывает сам тип *Boiler*, который унаследован от базового типа *BaseObjectType* и является абстрактным. Здесь определены две переменные (*opc:Variable*): *outWaterTemp* типа *Double* и *status* типа *Int16* – обе только для чтения (*Read*). Далее определены два метода: *startHeating* и *stopHeating* типа *ActionMethodType*.

Листинг 4 содержит XML-описание типа (класса) *GasBoiler* (рис. 1), который унаследован от определённого ранее типа *Boiler* и является абстрактным. Любой объект типа *GasBoiler* содержит вложенный объект (тег *opc:Object*) с именем *burner* типа *GasBurner*. Объекты *timer* и *generator*, а также методы

3onedata

IES6200-PN

коммутаторы с поддержкой протоколов MRP и PROFINET

- Конфигурация портов: 4×1G SFP + 16×10/100Base-T(X) (RJ45)
- Поддержка работы в сетях PROFINET RT CC-B
- Поддержка протокола резервирования MRP (IEC 62439-2)
- Резервированный вход по питанию: 12–48 В (DC)
- Диапазон рабочих температур: -40~75°C

Официальный дистрибьютор

+7 (495) 234-06-36
info@prosoft.ru

www.prosoft.ru

Реклама

timeout и *tick* не объявляются, так как являются защищёнными (недоступными снаружи объекта).

Листинг 5 содержит XML-описание типа (класса) *BoilerModelM1* (рис. 1), который унаследован от типа *NaturalPullGasBoiler* и не является абстрактным. Далее переопределяется уровень доступа к атрибутам объекта *burner*: переменные *spark* и *gasValve* становятся доступными только для чтения.

Реализация всех методов (*startHeating*, *stopHeating*, *timeout*, *tick*) в нашем случае определяется на языке C#. При создании объекта сервер должен назначить всем методам соответствующую реализацию (подгрузить драйвер).

Пусть имеется три газовых котла разных моделей (*BoilerModelM1*...*BoilerModelM3*). Создадим экземпляры этих классов (см. **листинг 6**). Тег *opc:Object* задаёт объект *BoilerM1* типа *BoilerModelM1*, находящийся в корневой папке объектов (*ObjectsFolder*) в дереве узлов. Аналогичным образом задаются объекты *BoilerM2* и *BoilerM3*. При запуске OPC-сервер создаст соответствующие им узлы (рис. 4, 5, 6). OPC-клиент может работать с объектами *BoilerM1*...*BoilerM3* – вызывать их методы, а также читать их атрибуты (запись напрямую недоступна).

Заключение

В статье с помощью технологии OPC UA выполнено ООП цифровых коммуникаций с котельным оборудованием. Проведена декомпозиция задачи проектирования – представлена двухуровневая архитектура ПО: верхний уровень – технологический алгоритм управления тепловыми процессами, нижний уровень – алгоритм непосредственного управления котельным оборудованием (драйвер устройства – котла). Технологический алгоритм управления находится в контроллере, который является OPC-клиентом. OPC-сервер предоставляет сервис для работы с котлом OPC-клиенту. Клиент взаимодействует с объектом, в который встроен драйвер котла. Объект на сервере представлен в виде узла типа *Object*.

Данная декомпозиция упрощает программу контроллера, так как сложный алгоритм управления оборудованием заменяется простой абстракцией, которую предоставляет и поддерживает объект на OPC-сервере.

Становится возможным раздельное проектирование: верхний и нижний уровни могут проектироваться разными людьми, командами и даже организациями.

Также драйвер для оборудования может предоставляться самим производителем. В данном случае особую актуальность приобретает проблема отраслевой стандартизации информационных моделей. Также снижается общая трудоёмкость разработки, так как возможно переиспользование драйверов устройств в другом проекте.

Наличие драйверного слоя повышает надёжность и безопасность работы оборудования, так как технологический алгоритм управляет оборудованием не напрямую, а через посредника – драйвер устройства, который обеспечивает корректное управление оборудованием и не допускает нарушения требований надёжности и безопасности, даже если в технологическом алгоритме допущены ошибки. Становится возможной «горячая» замена оборудования, так как OPC-сервер поддерживает динамическое создание объектов (узлов).

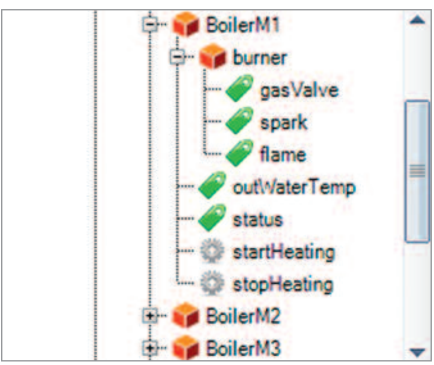


Рис. 4. Узел котла модели M1

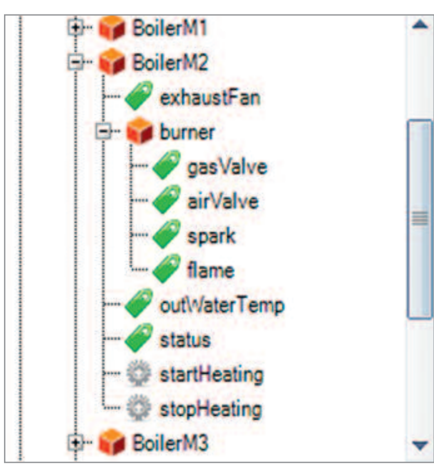


Рис. 5. Узел котла модели M2

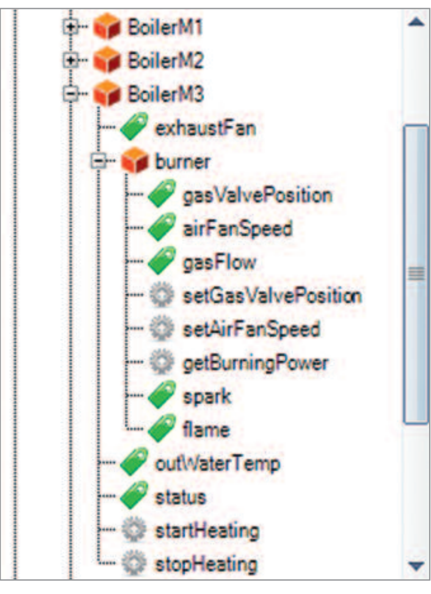


Рис. 6. Узел котла модели M3

Технология OPC UA позволяет придерживаться следующих принципов ООП: абстракция, иерархичность (наследование, композиция), инкапсуляция и полиморфизм.

Абстракция предназначена для упрощения проектных решений, иерархичность – для упорядочивания абстракций по уровням. В итоге это облегчает процесс проектирования. Всё многообразие котельного оборудования было упорядочено иерархически.

Инкапсуляция предназначена для скрытия внутреннего устройства (реализации) класса. Это позволяет легко и безболезненно менять реализацию – внесение изменений в драйвер котла прозрачно для технологического алгоритма.

Интерфейс, предоставляемый абстракцией, и полиморфизм позволяют использовать разнотипные объекты – изменение модели котла (типа объекта) не приводит к изменению технологического алгоритма.

ООП – достаточно мощная парадигма проектирования, основанная на объектной модели. Технология OPC UA поддерживает возможности полноценного объектно-ориентированного проектирования цифровых коммуникаций промышленного оборудования. ●

Литература

1. *Mahnke W., Leitner S.-H., Damm M.* OPC Unified Architecture. Berlin: Springer, 2009.
2. *Буч Г., Максимчук Р.А., Энгл М.У.* и др. Объектно-ориентированный анализ и проектирование с примерами приложений. 3-е изд. / пер. с англ. М.: ООО «И.Д. Вильямс», 2008.
3. URL: http://opcfoundation.github.io/UA-.NETStandard.
4. URL: https://github.com/OPCFoundation/UA-ModelCompiler.

Автор – сотрудник ФГБУН «Институт автоматики и электрометрии» СО РАН
E-mail: neyzov.max@gmail.com

НОВОСТИ реклама НОВОСТИ реклама НОВОСТИ реклама

Новая серия ультракомпактных твердотельных накопителей SATADOM от Innodisk

Компания Innodisk представляет серию SATADOM-SV/SL 3IE7 – твердотельные накопители с интерфейсом подключения SATA III, которые благодаря своему компактному размеру, – 40,4×21,03×10,4 мм в вертикальном исполнении и 36,7×31,2×10.7 мм в горизонтальном исполнении могут использоваться в любой компактной системе.

Подача питания может быть реализована как стандартным способом через кабель, так и через этот же SATA-разъём (седьмой или восьмой контакт в зависимости от модификации), что исключает необходимость использования дополнительных кабелей.

Новинка производится на базе флеш-памяти нового поколения iSLC. Компромиссная память iSLC позволяет использовать физическую память 3D TLC более длительный срок, т.е. программным образом достигается значение до 30 000 циклов перезаписи для одной ячейки, что в 10 раз больше, чем у накопителей на базе памяти 3D TLC. Преемственные серии наделены всеми основными преимуществами своих старших аналогов на базе памяти MLC, а по заявленному количеству циклов перезаписи даже превосходят их, несмотря на более низкую стоимость.

Новинка характеризуется следующими преимуществами:

- увеличение уровня выносливости накопителей с помощью использования новейшей архитектуры L3, включающей технологию LDPC;
- встроенный термодатчик, предотвращающий отказ работы системы;
- полное сохранение данных благодаря отсутствию DRAM буфера;
- защита данных с помощью технологий S.M.A.R.T и ATA Security;
- питание через 7/8-й контакт SATA-разъёма без использования дополнительных кабелей;
- ёмкость 20–80 Гбайт.
- скорость чтения/записи 550/485 Мбайт/с;
- расширенный диапазон рабочих температур, от –40 до +85°C;

Данный накопитель может использоваться как загрузочное устройство, обеспечивая высокую надёжность системы.

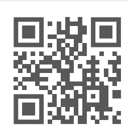
Возможна установка второго накопителя для резервирования операционной системы благодаря поддержке RAID и технологии хранения данных Intel Rapid (Intel RST). Серия SATADOM предоставляет возможность использования прочих накопителей только для хранения данных.

Очередной выпуск слушателей семинара профессора G. Cockrell

17 апреля в демонстрационном зале НИТ ГУАП профессор Gerald Cockrell (США), президент ISA 2008 года, Почётный доктор ГУАП, принял участие в заключительном занятии интернет-семинара «Управление проектами». Профессор Cockrell уже в 18-й раз провел семинар. За эти годы свыше пятисот студентов, аспирантов, преподавателей ГУАП и членов регулярной и студенческой секций ISA приняли в нём участие. Традиционно, на заключительном занятии семинара были вручены сертификаты слушателям семинара, успешно завершившим программу.

Видеонаблюдение без потери данных от Innodisk. Поколение 2.0.

Переход на цифровые технологии диктует постоянно растущий спрос на надёжные средства хранения данных. Учитывая, что в современном мире широкое распространение получили системы видеонаблюдения, идентификации и верификации человека, которые обеспечивают безопасность в местах массового скопления людей и на охраняемых объектах, очень важным аспектом в выборе накопителя служит стабильность записи данных. Традиционным решением всегда служили накопители на жёстких магнитных дисках, но данная технология уже достигла предела производительности, в то время как производство твердотельных накопителей уверенно развивается и активно занимает всё новые и новые рынки, включая видеонаблюдение.



Нестабильность работы накопителя при записи данных приведёт к потере кадров, что может привести к серьёзным последствиям, таким как несанкционированное проникновение, несвоевременное реагирование на нештатную ситуацию, потеря важных данных для поиска злоумышленников. Компания Innodisk представляет обновлённую серию накопителей для систем видеонаблюдения, – теперь на чипах памяти 3D TLC.

InnoREC™ – это запатентованная разработка компании Innodisk, предназначенная специально для систем видеонаблюдения. Благодаря интеллектуальному слиянию программной и аппаратной части скорость записи данных и производительность накопителей отвечают самым высоким требованиям к современным цифровым решениям.

Технология InnoREC реализована в серии 3TV6-P, представленной в формате 2.5"SSD.

Основные характеристики:

- ёмкость 128 Гбайт до 4 Тбайт;
- тип памяти 3D TLC;
- скорость чтения/записи 510/460 Мбайт/с;
- диапазон рабочих температур –40…+85°C;
- технология RECLine для оптимизации работы дисков в системах видеозаписи;
- технология iData guard, контролирующая целостность данных при сбоях питания;
- технология Garbage collection and Trim для оптимизации данных при операциях чтения/записи;
- технология iCell – применение суперконденсаторов для хранения данных на период небольшого количества времени при пропадании питания и для безопасного завершения работы;
- технология InnoRobust – функция уничтожения данных на программном уровне;
- встроенный термодатчик, предотвращающий отказ работы системы;
- поддержка технологий ATA Security /iSMART;
- соответствие стандартам JESD218 и JESD219;
- наличие сертификата E-mark (E13).