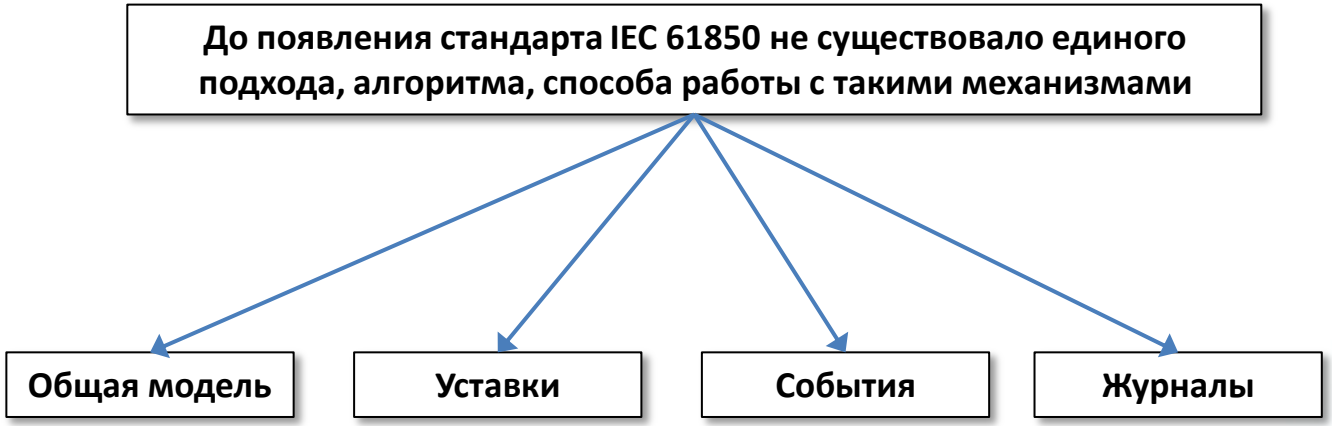
ЧЕБОКСАРСКИЙ ЭЛЕКТРОАППА

SOLID принципы и проектирование ЦПС

Удиков Дмитрий Александрович

До появления стандарта



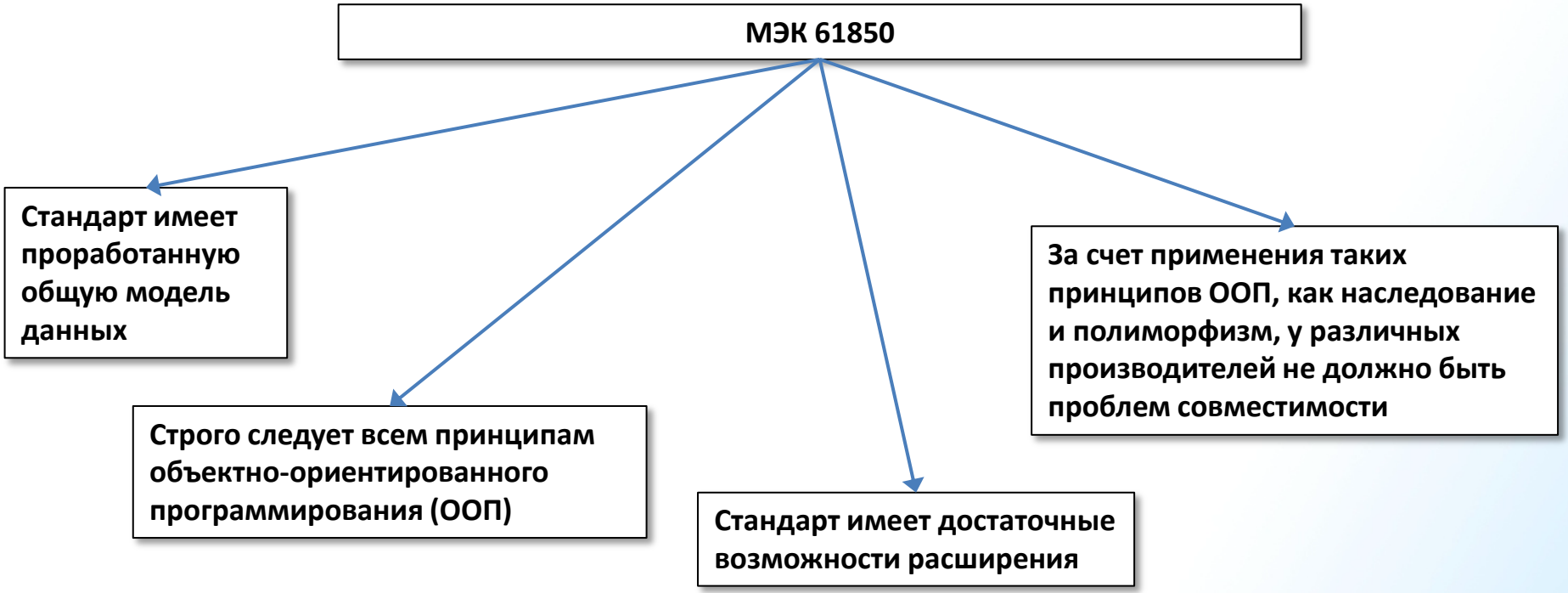
До появления стандарта

Разработка проекта, а тем более введение ЦПС в эксплуатацию, требовали

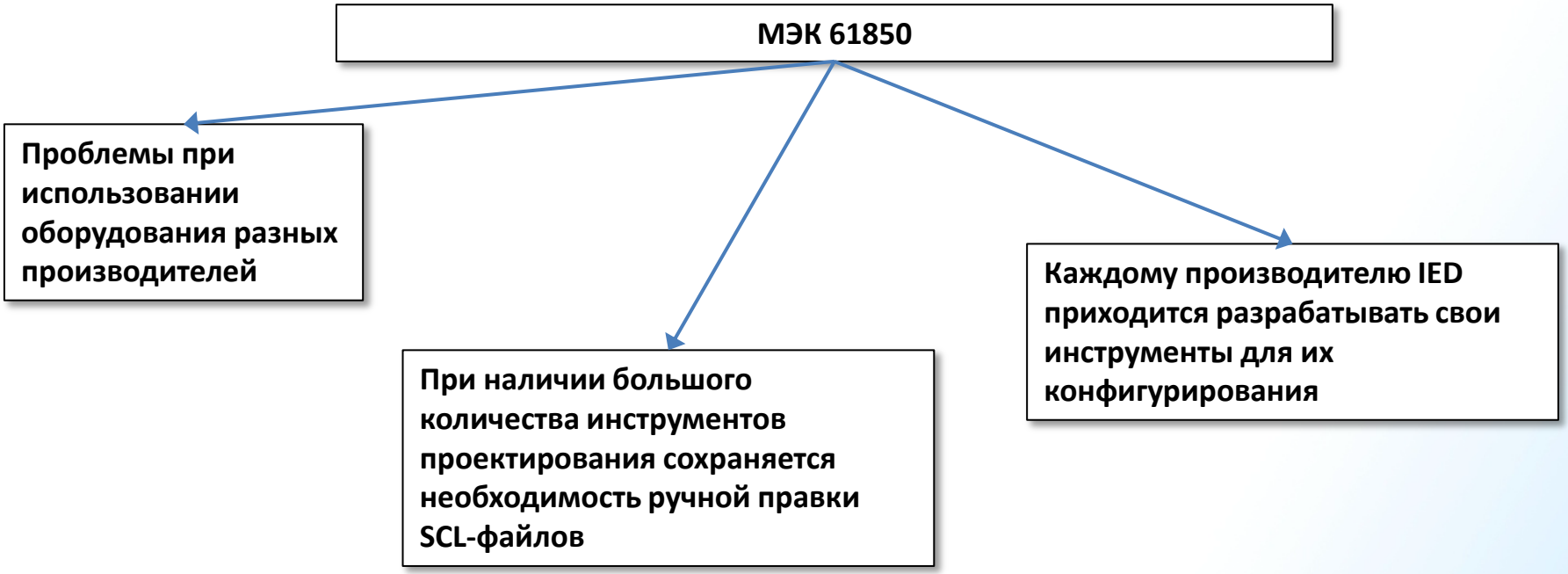
- **Очень высокой компетенции участников процесса**
- **Многократных правок и изменений проекта, кабельных журналов и другой проектной документации**
- **На этапе разработки часто не хватало информации о проприетарных алгоритмах работы устройств РЗА с вышеперечисленными механизмами**

При этом все тяготы приходились как на разработчика АСУ ТП, так и особенно на наладчика (этап пуско-наладочных работ)

Внедрение МЭК 61850



Проблематика МЭК 61850



Возможное решение

Как было упомянуто выше, стандарт использует объектную модель данных и ориентируется на принципы ООП, наиболее важные из которых известны как

S.O.L.I.D.

Рассмотрим данные принципы и то, как нарушения некоторых из них создают проблемы совместимости.

Что такое SOLID?

The Single Responsibility Principle ***(Принцип единственной ответственности)***

The Open Closed Principle ***(Принцип открытости/закрытости)***

The Liskov Substitution Principle ***(Принцип подстановки Барбары Лисков)***

The Interface Segregation Principle ***(Принцип разделения интерфейса)***

The Dependency Inversion Principle ***(Принцип инверсии зависимости)***

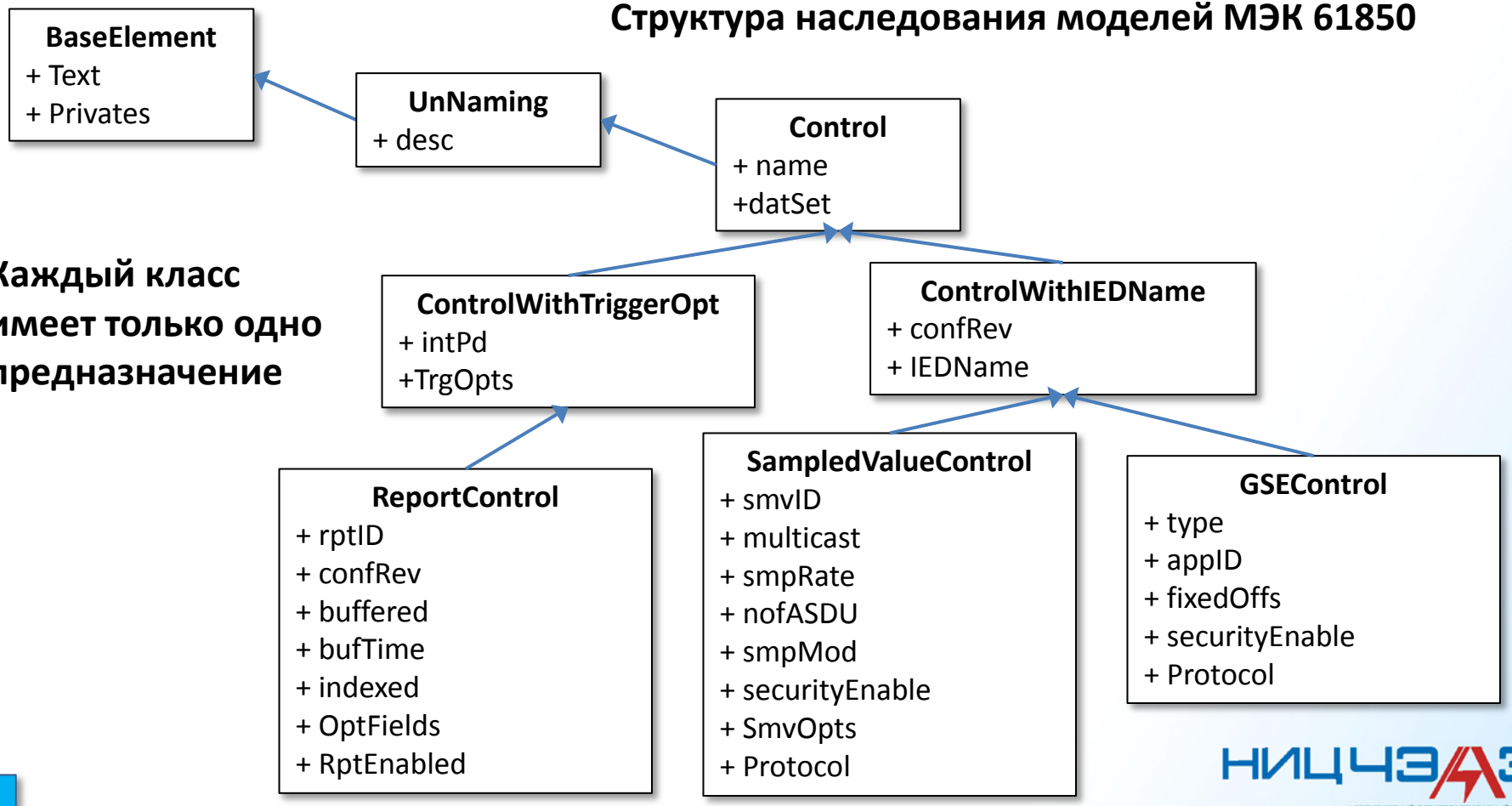
SOLID – Мнемонический акроним, введенный Майклом Фэзерсом для первых пяти принципов, названных Робертом Мартином в начале 2000-х, которые означали пять основных принципов объектно-ориентированного программирования и проектирования.

Принцип единственной ответственности

Класс должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

Принцип единственной ответственности

Структура наследования моделей МЭК 61850



Каждый класс имеет только одно предназначение

НИЦ ЧЭАЗ

Принцип открытости/закрытости

Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.

Принцип открытости/закрытости появился в 1988 году в книге Бертрана Мейера (Object-Oriented Software Construction) и он является одним из ответов на вопрос:

«Как можно разработать проект, устойчивый к изменениям, срок жизни которых превышает срок существования первой версии проекта?».

Принцип открытости/закрытости

Данный принцип прямым текстом прописан на страницах МЭК 61850-6-1 в разделе 8.2 (Версия языка и совместимость)

Допускается:

- Добавление новых необязательных атрибутов. Если атрибуты нуждаются в задании значения, то у них должны быть заданы значения по умолчанию такие, что эта величина, насколько это возможно, должна соответствовать отсутствию какой-либо величины в старых версиях.
- Добавление новых элементов допустимо в конце существующих определений типов.
- Чтобы обеспечить прямую совместимость, любой новый элемент, понимание значения которого необходимо для обеспечения совместимости при передаче данных, должен быть промаркирован атрибутом mustUnderstand.

Принцип открытости/закрытости

Запрещается:

- Удаление старых возможностей. Для обеспечения обратной совместимости они по-прежнему разрешены в старых экземплярах языка, но использование в новых версиях является нежелательным. Указание на нежелательность использования обычно делается путем удаления функции из схемы новой версии. Тем не менее, старые возможности разрешены в экземплярах конфигурации, полученных из старых версий, которые должны быть приняты получателем.
- Изменение существующих возможностей, особенно их семантики. Данное изменение ставит под угрозу совместимость, поэтому запрещено. Вместо этого "старые" возможности объявляются нежелательными, а новые добавляются, что впоследствии считается замещением нежелательных возможностей.
- Изменение существующих значений по умолчанию. Это позволяет приёмнику или процессору использовать значения по умолчанию также для более старых экземпляров конфигураций.

Принцип подстановки Барбары Лисков

*Пусть **q(x)** является свойством верным относительно объектов **x** некоторого типа **T**. Тогда **q(y)** также должно быть верным для объектов **y** типа **S**, где **S** является подтипом типа **T**.*

Барбара Лисков

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Роберт С. Мартин

Более простыми словами можно сказать, что поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

Принцип подстановки Барбары Лисков

На наш взгляд, одной из причин несовместимости между различными, производителями является нарушение данного принципа.

Проблема заключается в том, что стандарт позволяет разработчикам использовать различные расширения и, расширяя свои модели элементами <Private/>, производители добавляют функционал, который может нарушить данный принцип. Например, если вместо использования стандартных механизмов, вынести в Private элементы, информацию о формировании или использовании подписки на GOOSE или SV, работу с настройками и т.п., то другие производители не смогут работать с таким «расширенным» типом.

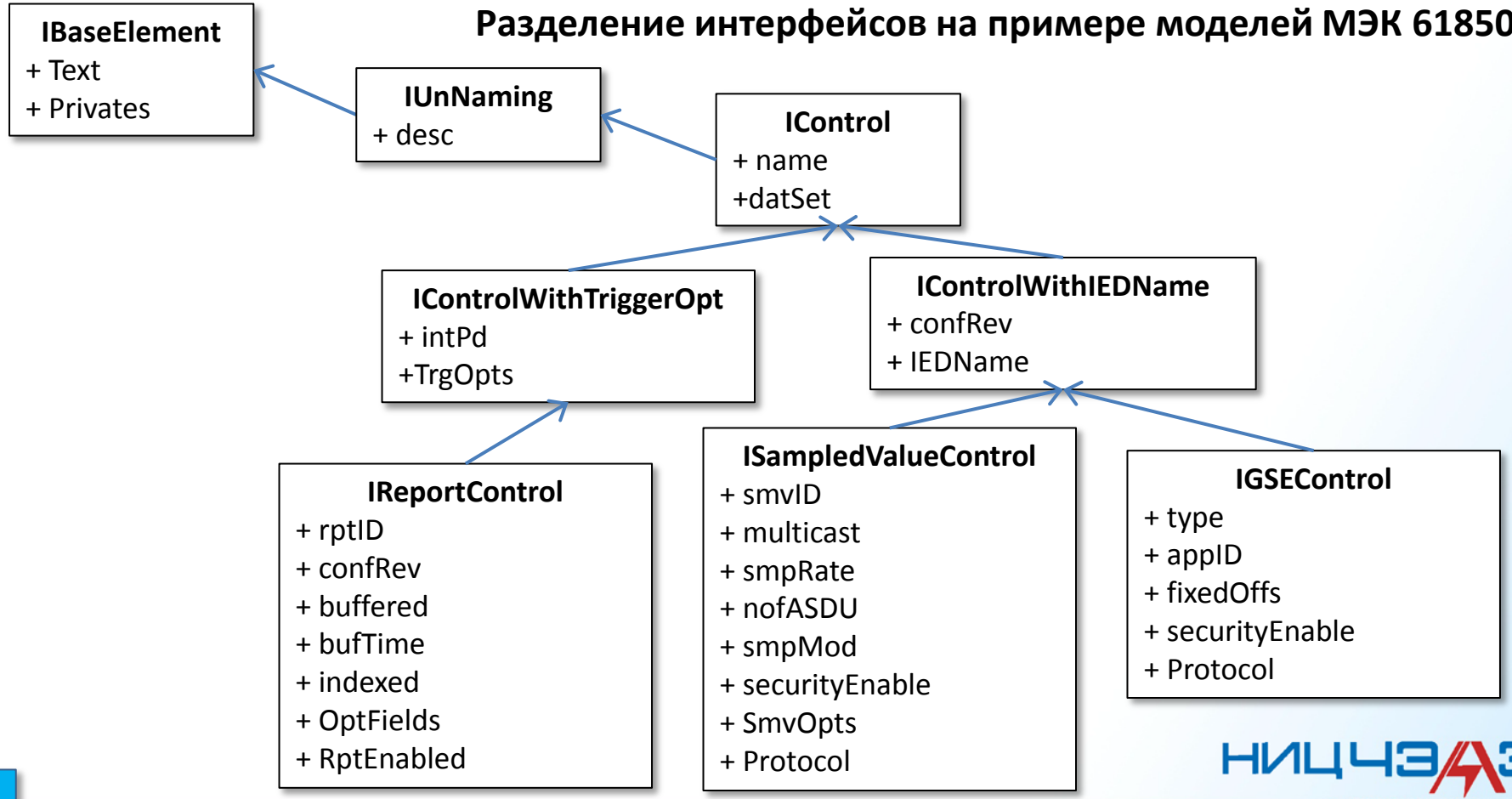
<Private/> - элементы должны расширять функционал, а не изменять его (Принцип открытости/закрытости).

Принцип разделения интерфейса

Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться программные сущности, которые этот метод не используют.

Принцип разделения интерфейса

Разделение интерфейсов на примере моделей МЭК 61850



Принцип инверсии зависимости

1. Модули верхних уровней не должны зависеть от модулей нижних уровней. Модули обоих уровней должны зависеть от абстракций.
2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Принцип инверсии зависимости

```
public class Client
{
    private Server _server;
    public Client()
    {
        _server = new Server();
    }
}
```

Рисунок 1

На Рисунке 1 имеем класс Client, который использует класс Server.

Если нам необходимо изменить реализацию класса Server, то придется изменять и реализацию класса Client. (Рисунок 2)

```
public class Client
{
    private AnotherServer _server;
    public Client()
    {
        _server = new AnotherServer();
    }
}
```

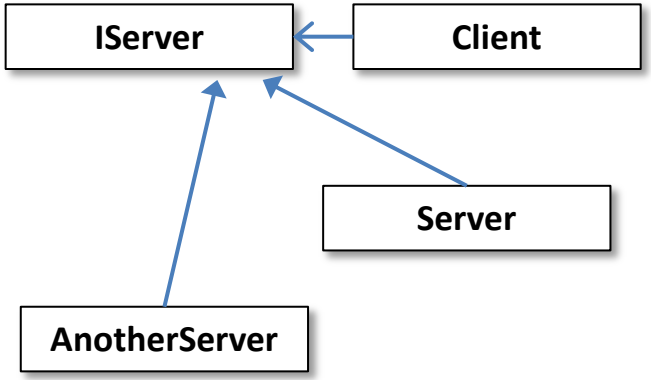
Рисунок 2

Проблема:
Класс Client сильно зависит от класса Server

Принцип инверсии зависимости

Решение:

Клиент не должен знать о конкретной реализации сервера



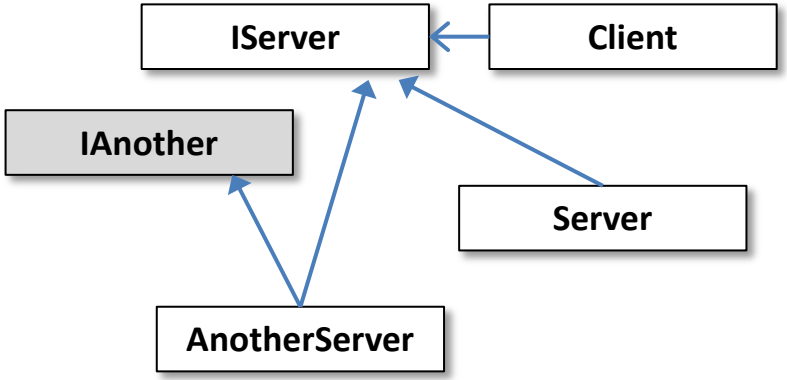

```
public class Client
{
    private readonly IServer _server;
    public Client(IServer server)
    {
        _server = server;
    }
}
```

Этот механизм называется принципом инверсии зависимостей (DIP). Но почему? Класс Client по-прежнему зависит от интерфейса сервера. Если интерфейс изменяется, Client также должен измениться. Да, это правильно. Но Client решает, какие функции ему нужны в этом интерфейсе. Поэтому обычно интерфейс изменяется, когда Client говорит, что его нужно изменить. Интерфейс изменяется, поскольку Client изменяется.

Принцип инверсии зависимости

Решение:

Клиент не должен знать о конкретной реализации сервера

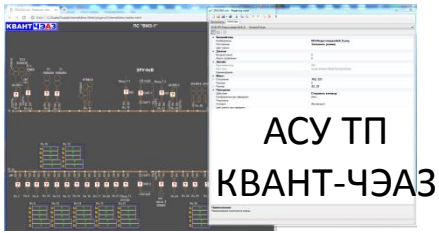



```
public class Client
{
    private readonly IServer _server;
    public Client(IServer server)
    {
        _server = server;
    }
}
```

Этот механизм называется принципом инверсии зависимостей (DIP). Но почему? Класс Client по-прежнему зависит от интерфейса сервера. Если интерфейс изменяется, Client также должен измениться. Да, это правильно. Но Client решает, какие функции ему нужны в этом интерфейсе. Поэтому обычно интерфейс изменяется, когда Client говорит, что его нужно изменить. Интерфейс изменяется, поскольку Client изменяется.

Принцип инверсии зависимости

Клиенты



АСУ ТП
КВАНТ-ЧЭАЗ

Типовые решения

Проектные организации

Interface

.ICD файл

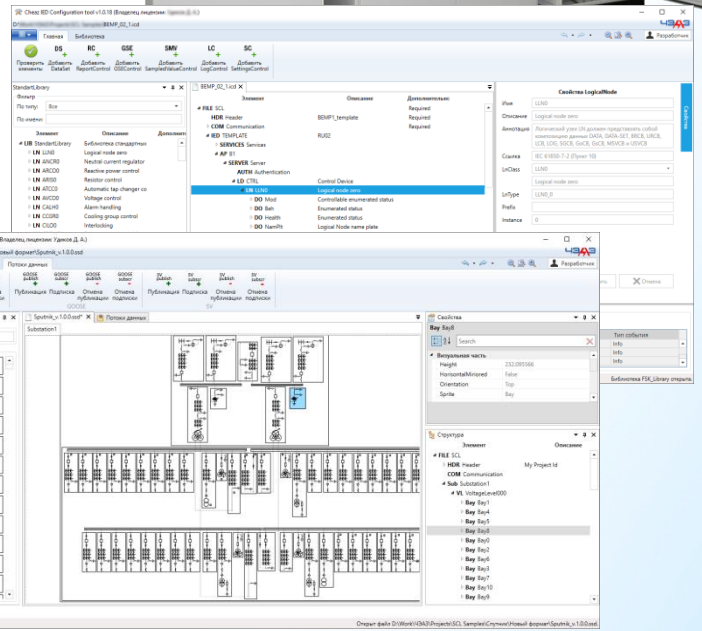
Реализации серверов

Терминал 1 (IED)

Терминал 2 (IED)

Заключение

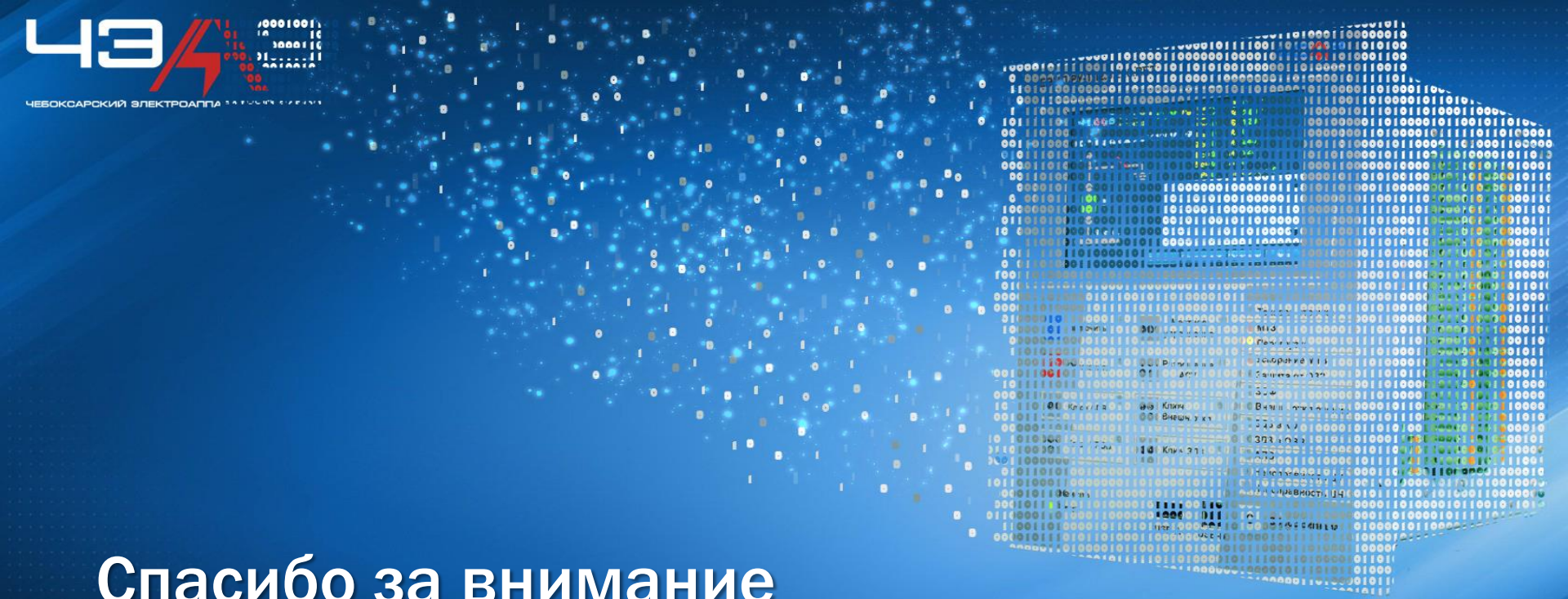
Описанная проблема актуальна для нас, поскольку группа компаний «ЧЭАЗ» является разработчиком не только оборудования для ЦПС (широкого ассортимента шкафов и терминалов серии БЭМП), но также инструментов проектирования, в том числе уровня подстанции («CHEAZ Substation Configuration Tool»)



Если производители будут придерживаться основных принципов SOLID, то проблем совместимости станет значительно меньше, и процесс проектирования станет действительно бесшовным.

НИЦ ЧЭАЗ
ЧЕБОКСАРСКИЙ ЭЛЕКТРОАППАРАТНЫЙ ЗАВОД

ЧЕБОКСАРСКИЙ ЭЛЕКТРОАППА



Спасибо за внимание