CodeStyle FullVersion

SmartPTT team

V 1.0.0

Оглавление

Глава 1

О документе

Этот документ описывает как должен выглядеть хороший код с точки зрения команды SmartPTT.

1.1 Условные обозначения

Good practice
Так отмечены рекомендованные варианты использования.

Bad practice
Так отмечена инфорация о том как делать нельзя, ни в коем случае.

Полезная информация
Информация, которая будет полезна при чтении статьи.

Требует обсуждения
Этот блок содержит информацию которая должна быть утверждена коллегиально.

```
// пример исходного кода

class foo
{
    public void bar()
    {
        return;
    }
}
```

1.2 Назначение документа

В нашем отделе принято решение выработать единый стиль оформления для исходного кода всех разработчиков. Это позволить облегчить сопровождение всех частей программы.

Истиной в последней инстанции при определении того как должен быть оформлен тот или иной код является коллективное решение разработчиков отдела.

Для вопросов по которым решение ещё не выработано, этот документ содержит предварительно заданные значения, чтобы было от чего оттолкнуться.

Для вопросов по которым решение уже выработано, данный документ содержит кэш этого решения. Чтобы по уже обсуждённым вопросам заново не собирать всех

разработчиков.

Также данный документ будет выступать основой для создания настроек инструментов контроля стилистики. В число этих инструментов входят: Visual Studio, ReSharper, StyleCop. После создания этих настроек будет разрешено коммитить только файлы без стилистических ошибок. Также планируется внедрение предкоммитного контроля. То есть при коммите выполняется проверка оформления исходного кода, и если файлы содержат стилистические ошибки, то коммит отвергается.

Порядок внесения изменений в документ

Не значащие изменения вносятся в документ ответственным за CodeStyle. К незначащим изменениям относятся например: добавление примеров, переупорядочивание статей, переименование названий статей, и тому подобное.

Под значимыми понимаются положения которые могут потребовать внесения изменений в исходный код.

Внесение значимых изменений должно проходить в следующем порядке:

1. ответственный за CodeStyle готовит предложение, то есть что внести какие плюсы, какие минусы
2. ответственный за CodeStyle организовывает собрание отдела
3. на собрании высказываются все заинтересованные стороны
4. при необходимости вносятся изменения в предложение
5. предложение утверждается голосованием, простым большинством, или не утверждается

После голосования дальнейшее обсуждение предложения не допускается. Готовьте ваши аргументы, и убеждайте коллег до голосования. Уважайте мнение команды.

1.3 Применение для разных языков

Наша команда использует много языков, как программирования, так и разметки. В их число входят:

- C#
- java
- Objective C
- C++
- скриптовые языки
- XAML
- SQL

Данный документ создан для нашего основного языка разработки - C#. Те положения из этого документа которые можно применить в других языках, следует применять. Специфичные вещи для конкретных языков следует смотреть в главе 11.

Глава 2

Общие принципы

- Простота чтения превыше простоты создания. Код читается намного чаще чем пишется, поэтому не экономьте на понятности и чистоте кода ради скорости набора.
- Чем хуже код читается, тем страшнее костыли, используемые при его сопровождении.
- Есть разница между тем что написано, и тем, что хотел сделать автор. Чем лучше написан код тем больше усилий можно посвятить анализу того что хотел сказать автор, а не продираться сквозь громоздкие языковые конструкции и нелогичное именование.
- Нельзя полагаться только на то как код выглядит и читается в студии. Нельзя полагаться на облегченную навигацию по F12 и intelliSence. Так как во-первых инструменты меняются, а во-вторых для code-review скорее всего будут использоваться сторонние инструменты в которых код может выглядеть по другому.
- В качестве проверки того насколько код хорош представьте что вы его распечатали на бумаге. Сохранится ли удобство чтения? Если да, то код хорош.

2.1 Идеология

2.1.1 Принцип целенаправленности

Элемент(компонент, объект, метод) должен делать только одну вещь, но должен делать её хорошо.

Если вам нужен класс который умеет отправлять сообщения по сети, создайте класс который умеет отправлять сообщения по сети и только это. Этот класс не должен уметь сохранять сообщения в файл, или выводить их на экран. Но он должен уметь отправлять сообщения хорошо: обрабатывать все особые ситуации, быть предсказуемым и логичным.

Если вам всё-таки нужно сохранять сообщения в файл, или выводить их на экран то создайте новые классы. По одному для каждой задачи.

2.1.2 KISS

Keep It Simple Stupid.

Второй вариант расшифровки Keep It Short and Simple.

Если у вас есть два способа сделать что-то выберите более простой.

Если вам нужно найти всех абонентов с идентификаторами больше 200, то сделайте простой метод. Не нужно создавать язык запросов позволяющий искать по любым условиям на любом наборе свойств.

Лучше вызов метода по ссылке, чем разворачивание паттерна команда. Лучше поиск простым перебором, чем наворачивание чёрно-красных деревьев.

Но простота это не абсолютный критерий. Если критично время работы то нужно работать с чёрно-красными деревьями. Если нужно разорвать логику действия и место вызова, то нужно использовать паттерн команда.

2.1.3 YAGNI

You Ain't Gonne Need It

Вам это не нужно.

Не нужно делать вещи которыми никто не пользуется. Не нужно создавать аргументы методов "на будущее". Не нужно сохранять свойства классов которыми никто не пользуется. Не нужно сохранять программные прослойки которые потеряли свою роль. Не нужно делать классы, которые когда-нибудь понадобятся.

Если у вас есть наработки которые могут быть полезны, но у вас нет сейчас времени довести их до ума, их следует развивать в отдельных ветках, а не в транке.

2.1.4 DRY

Don't Repeat Yourself

Не повторяйтесь. Избегайте дублирования.

Дублирование бывает трёх видов

- функциональное - разные компоненты программы делают одну и ту же работу.
- временное - один и тот же код выполняется несколько раз без особой необходимости.
- кода - в разных частях программы исходный код полностью дублирован.

2.1.5 Принцип наименьшего удивления

Поведение интерфейса не должно вызывать удивление у пользователя.

Если вы нажимаете кнопку закрыть, программа должна закрыться, а не закрывать открытые сокеты.

Если вы вызываете метод `Clone()` то ожидается получение копии объекта, а не просто новой ссылки.

Если вы устанавливаете свойство `ParentControl` то ожидается что контрол сам себя перерисует как надо. Если же после установки этого свойства обязательно надо вызвать метод `Refresh()`, то пользователь интерфейса будет в недоумении.

2.1.6 Порядок работы

Make it run. Make it right. Make it fast.

Поговорка устанавливающая порядок работы.

1. Make it run. - сделайте чтобы это работало. То есть работало правильно и проходило тесты.

2. Make it right. - сделайте это правильно. Приведите в порядок именование. Выполните рефакторинг если требуется.

3. Make it fast. - Оптимизируете. Если потребуется.

Кроме прочего этот принцип содержит в себе утверждение о том что преждевременная оптимизация корень всех зол.

2.2 Антипаттерны

Это раздел о том как не надо делать ни в коем случае.

2.2.1 Magic consts

Не используйте литеральные константы (магические числа, зашитые в код размеры буферов, сообщения исключений). Лучше определите константу (если вы никогда не будете ее менять) или переменную только для чтения (если она может измениться в будущем).

Не используйте именованные константы, строковые литералы, или иные фиксированные значения вместо перечислений.

- Использование типов предметной области облегчает понимание кода.
- Значения перечисления легко можно привести к числовым значениям.
- Контроль за соответствием типов перекладывается на компилятор, а не висит на программисте.
- Облегчается процесс добавления/удаления допустимых значений.

Good practice

```
class Scheduler
{
    private const int DayPerWeek = 7;

    //Нельзя создавать задачи дольше 50 дней;
    private static readonly int MaxDayCount = 50;
    private static readonly string TooLongPeriodMessage =
        "Период выполнения задачи слишком длинный";

    private TaskState _lastTaskState;
    ...
    private void CalculatePeriod()
    {
        int dayCount = weekCount * DayPerWeek;
        if (dayCount > MaxDayCount)
        {
            throw new LongPeriodException(TooLongPeriodMessage);
        }
    }

    private enum TaskState
    {
        WaitForActivate,
        Active,
        Finished
    }
}
```

Bad practice

```
class Scheduler
{
    // 1 - в ожидании
    // 2 - задача в процессе выполнения
    // 3 - задача завершена
    private int _lastTaskState;

    private void CalculatePeriod()
    {
        int dayCount = weekCount * 7;
        if (dayCount > 50)
        {
            throw new LongPeriodException(
                "Период выполнения задачи слишком длинный");
        }
    }
}
```

Глава 3

Именование

3.1 Общие правила

- Для именования используйте литературный английский язык. Проверяйте имена идентификаторов на грамматические ошибки.

Bad practice

```
bool calLaod;
int fuckenCounter;
```

- Имя должно ясно и четко описывать смысл и/или предназначение сущности.

Good practice

```
class TextMessageSender ...

int activeSender;

class LongMessageException : Exception ...
```

Bad practice

```
class AnythingSender ...

int mySender;

class BadMessageException : Exception ...
```

- Одна сущность одно имя. Во всём проекте. Если в разных классах собеседник будет называться `Callee`, `Companion`, `Interlocutor`, `RemotePeer`, то скорее всего мы запутаемся.

- Разные сущности разные имена. Во всём проекте. Если компьютер за которым сидит пользователь и самого пользователя мы будем называть одинаково, например `RemotePeer`, то мы снова запутаемся.

- Старайтесь делать имена идентификаторов как можно короче (но не в ущерб читабельности). Помните, что современные языки позволяют формировать имя из пространств имен и типов. Главное, чтобы смысл идентификатора был понятен в используемом контексте. Например, количество элементов коллекции лучше назвать `Count`, а не `CountOfElementsInMyCollection`.

- Не используйте малопонятные префиксы или суффиксы (например, венгерскую нотацию), современные языки и средства разработки позволяют контролировать типы данных на этапе разработки и сборки.

Bad practice

```
class AbonentId ...
class AbonentId1 ...
class AbonentId2 ...
class AbonentId2_ ...
```

- При именовании элементов пользовательского интерфейса, если возникает необходимость включить тип в имя поля, сделайте тип суффиксом. Не нужно использовать венгерскую нотацию.

Good practice

```
Label addUserLabel;
TextBox addUserTextBox;
Button addUserButton;
```

Bad practice

```
Label lblAddUser;
TextBox tbAddUser;
Button btnAddUser;
```

- Имена различающиеся регистром:

  Не рекомендуется, но допустимо:

  – Создавать свойства имена которых совпадают с именем их типа

  – Создавать параметры методов, локальные переменные отличающиеся от имени типа или именем свойства регистром первой буквы.

```
class ServerInviteTransaction
{
    public ServerInviteTransaction(Request request, Gateway gateway)
    {
        FirstRequest = request;
        Gateway = gateway;
    }

    public Gateway Gateway
    {
        get;
        private set;
    }
}
```

  Запрещается:

  – Создавать сущности отличающиеся регистром не первого символа.

Bad practice

```
class CallId...

class Callid...


private void SearchAbonent(AbonentId abonentId)
{
    AbonentId abonentid = new AbonentId();
}
```

- Старайтесь использовать имена с простым написанием. Их легче читать и набирать. Избегайте (в разумных пределах) использования слов с двойными буквами, сложным чередованием согласных. Прежде, чем остановиться в выборе имени, убедитесь, что оно легко пишется и однозначно воспринимается на слух. Если оно с трудом читается, и вы ошибаетесь при его наборе, возможно, стоит выбрать другое.

Bad practice

```
int entrepreneurialRequest;

int ambiguousCondition;

bool b001 = (lo == l0) ? (I1 == 11) : (lol != 101);
```

- Избегайте частей имени которые не добавляют новой информации о сущности. Например `MessageSender` - хорошо; `MessageSenderClass` `MessageSenderSystem`, `MessageSenderComponent` - плохо.

3.2 Используемые написания имён

В проектах команды допустимо использование следующих написаний имён:

- **Паскаль** - идентификатор состоит из нескольких сканкатенированных строк, первые буквы слов - заглавные. `BackgroundColor`, `PascalNameStyle`, `DateTime`, `CodeConventions`.

  Пример использования - имена классов.

- **кэмел** - то же самое что и написание Паскаль, но первая буква идентификатора - строчная. `numberCount`, `camelNameStyle`, `currentWord`, `lastDate`.

  Пример использования - имена локальных переменных.

- **_подчёркиваниеКэмел** - то же самое что и написание кэмел, но используется префикс подчёркивание. `_elementStorage`, `_firstRequest`, `_myIp`.

  Пример использования - имена приватных полей.

- **UPPERCASE_UNDERSCORE** - идентификатор состоит из нескольких слов, слова разделены символом подчеркивания, все буквы слов пишутся с заглавной буквы. `DEBUG`, `RUSSIAN_LANGUAGE`, `UPPERCASE_UNDERSCORE_NAME_STYLE`, `TRACE`.

  Пример использования - директивы `define`.

3.2.1 Написание элементов языка

Language element	Name casing	Example
Класс, структура	Паскаль	ClientInviteTransaction
Интерфейс	IПаскаль	IBusinessService
Перечисления	Паскаль	ErrorLevel
Значения перечислений	Паскаль	FatalError
Свойства	Паскаль	Length
События	Паскаль	Disposed
Приватные поля	_подчёркиваниеКэмел	_upTime
Поля UI контролы	кэмел	addUserButton
protected поля	_подчёркиваниеКэмел	_supportedCodecs
private static поля	_подчёркиваниеКэмел	_instanceCounter
public static свойства	Паскаль	UniqueInstance
Методы	Паскаль	Restart
Аргументы методов	кэмел	socketAddress
Локальные переменные	кэмел	currentWord
Константы	UPPERCASE_UNDERSCORE	DayPerWeek
public readonly поля	Паскаль	MaxValue
private readonly поля	_подчёркиваниеКэмел	_defaultBufferLength
Пространства имён	Паскаль	ElcomPlus.Client
Файлы проекта	Паскаль	Configurator
generic параметры	TПаскаль	TEventArgs
define директивы	UPPERCASE_UNDERSCORE	RUSSIAN

3.3 Именование булевых элементов

Имеются в виду булевы переменные, методы, свойства. Чтобы подобные имена читались легко, они должны представлять собой вопрос ответом на который является либо да, либо нет.

Good practice

```
bool IsValid() ...
bool CanLoadImage ...
```

Bad practice

```
bool GlobalRange ...
bool Overflow ...
```

Хорошей практикой при создании имени булева элемента является использование префиксов `is`, `has`, `can`.

3.4 Сокращения

- Не используйте аббревиатуры или неполные слова в идентификаторах, если только они не являются общепринятыми. Например, пишите `GetWindow`, а не `GetWin`.
- Если есть сокращения которые действительно часто встречаются в используемой предметной области и использование этих сокращений улучшает читаемость,

то использование сокращений допускается, но их следует включить в список используемых сокращений.

На настоящий момент неофициальный список используемых исключений находится тут: список используемых сокращений

Требует обсуждения
Процесс заполнения этого списка, форма списка и прочие вопросы будут обсуждаться после выпуска релизной версии CodeStyle.

- Широко распространенные акронимы используйте для замены длинных фраз. Например, `Ui` вместо `UserInterface` или `Olap` вместо `OnLineAnalyticalProcessing`. Без внесения в список используемых сокращений.

- Сокращения в именах идентификаторов, подчиняются общим правилам. Первая буква заглавная, остальные строчные. В противном случае возможно появление большого числа заглавных букв:

Good practice

```
interface IUiInterconnection ...
class IpIoStream ...
class XmlParser ...
```

Bad practice

```
interface IUIInterconnection ...
class IPIOStream ...
class XMLParser ...
```

3.5 Список слов запрещённых к использованию в качестве имён

Запрещено использовать имена отличающиеся от ключевых слов языка только регистром. Старайтесь не использовать имена похожие на имена стандартных типов, нэймспейсов.

AddHandler	CBool	Date	Event
AddressOf	CByte	Decimal	Exit
Alias	CChar	Declare	Extends
And	CDate	Default	ExternalSource
Ansi	CDec	Delegate	False
As	CDbl	Dim	Finalize
Assembly	Char	Do	Finally
Auto	CInt	Double	Float
Base	Class	Each	For
Boolean	CLng	Else	Friend
ByRef	CObj	ElseIf	Function
Byte	Const	End	Get
ByVal	CShort	Enum	GetType
Call	CSng	Erase	Goto
Case	CStr	Error	Handles
Catch	CType	Eval	If

Implements
Imports
In
Inherits
Instanceof
Integer
Interface
Is
Let
Lib
Like
Long
Loop
Me
Mod
Module
MustInherit
MustOverride
MyBase
MyClass
Namespace
New
Next
Not
Nothing
NotInheritable
NotOverridable
Object
On
Option
Optional
Or
Overloads
Overridable
Overrides
Package
ParamArray
Preserve
Private
Property
Protected
Public
RaiseEvent
ReadOnly
ReDim
Region
REM
RemoveHandler
Resume
Return
Select
Set
Shadows
Shared
Short
Single
Static
Step
Stop
String
Structure
Sub
SyncLock
Then
Throw
To
True
Try
TypeOf
Unicode
Until
Var Volatile
When
While
With
WithEvents
WriteOnly
Xor

Глава 4

Форматирование

4.1 Общие правила

- Одна инструкция - одна строка. Несколько инструкций на одной строке усложняет отладку. Также при чтении визуально сложно определять границы инструкций. В одну строку допускается оформлять свойства без определения акцессоров и инициализацию массивов.

Good practice

```
if (a == null)
{
    throw new NullReferenceException();
}

int firstNumber;
int secondNumber;
int thirdNumber;

int Count
{
    get
    {
        return elementCount;
    }
}
```

Bad practice

```
if (a == null) throw new NullReferenceException();

int firstNumber, secondNumber, thirdNumber;

int Count{ get{ return elementCount; }}
```

- Используйте для формирования отступов символ пробела. Размер отступа должен быть равен 4 пробелам.

- Максимальная длина строки ограничена 150 символами. Для визуального отображения этой границы можно пользоваться расширением Editor Guidelines.

- Не используйте табличное выравнивание. Его тяжело сопровождать. Всё равно все забьют, лучше уж сразу отказаться.

Good practice

```
        internal readonly int _id;
        internal readonly Gateway _gateway;

        private CallState _state;
        private CallParameters _paremeters;
        protected int _duration;
```

Bad practice

```
        internal readonly int     _id;
        internal readonly Gateway _gateway;

        private   CallState      _state;
        private   CallParameters _paremeters;
        protected int            _duration;
```

4.2 вертикальные отступы

- Можно оставлять только одну пустую строку для вертикального отступа.
- Добавляйте пустые строки вокруг:
  - вложенных типов
  - прокомментированных полей
  - свойств
  - событий
  - методов
  - вокруг и внутри регионов
  - между группами using
  - после списка using
  - после заголовочного комментария файла

4.3 горизонтальные отступы

Пробелы нужно ставить в следующих случаях:

- после ключевыз слов:
  - if
  - while
  - catch
  - switch
  - for
  - foreach
  - using
  - lock

- вокруг операторов языка, кроме операторов разыменования в небезопасном режиме
- вокруг символов тернарного оператора
- после запятых
- после точки с запятой в операторе for
- после двоеточия
- перед двоеточием в списке наследования
- перед двоеточием в generic constraint
- вокруг стрелки лямбда выражений
- в начале комментария

4.4 Правила переноса длинных строк

Если строка кода длиннее ограничения 4.1 часть этой строки нужно перенести в соответствии с нижеизложенными правилами.

- если нужно перенести какой-либо список, то начиная со первого элемента каждый элемент переносится на новую строку и выравнивается по первому элементу.

```
ClientInviteTransaction transaction = gateway.CreateTransaction("INVITE",
                                                                fromId,
                                                                toId);
```

  Исключения из этого правила: список наследования для класса или интерфейса, инициализация массива, арифметические выражения, логические выражения. В этих случаях на каждой строке следует располагать столько элементов сколько помещается.

```
private readonly byte[] AvailableSymbolCodes = new byte[]
                                               {
                                                   016, 156, 189, 248, 100,
                                                   003, 192, 079, 154, 010,
                                                   226, 194, 085, 156, 025,
                                                   232, 057, 092, 021, 243,
                                                   147, 026, 185, 017, 133,
                                                   091, 000, 163
                                               };
```

- при переносе длинных логических или арифметических выражений вставляйте разрыв строки после операторов оставляя оператор на предыдущей строке
- при разрыве цепочки вызовов оставляйте точку на предыдущей строке

4.5 Положение открывающих фигурных скобок

Предложение:

В качестве правила постановки открывающей фигурной скобки использовать нотацию Allman.

Возможные варианты:

- K&R

```
if ( PageWidth > PaperWidth ) {
    SetPageWidth( PaperWidth );
    TellUser( MSG_PAGE_CHANGED );
}
```

- Allman

```
if ( PageWidth > PaperWidth )
{
    SetPageWidth( PaperWidth );
    TellUser( MSG_PAGE_CHANGED );
}
```

Аргументы за:

- Чётче разграничивает блоки исходного кода.

```
// использование базового конструктора

// K&R
public A()
    :base(null){
    Initialize()
}

// Allman
public A()
    :base(null)
{
    Initialize()
}
```

```
// generic constraints

// K&R
public int Compare<T>(T par1, T par2)
    where T : System.IComparable<T>{
    if (par1 == null){
        throw new ArgumentNullException("par1");
    }
    ...
}

// Allman
public int Compare<T>(T par1, T par2)
    where T : System.IComparable<T>
{
    if (par1 == null)
    {
        throw new ArgumentNullException("par1");
    }
    ...
}
```

```
// длинное логическое выражение:

// K&R
...
List<int> blocks;
List<string> blockNames;
...
    if(blocks != null && blocks.Count >= minCount &&
        blockNames != null && blockNames.Count >= minCount &&
        blocks.Count == blockNames.Count){
        blocks.RemoveAt(0);
    }
...

// Allman
...
List<int> blocks;
List<string> blockNames;
...
    if(blocks != null && blocks.Count >= minCount &&
        blockNames != null && blockNames.Count >= minCount &&
        blocks.Count == blockNames.Count)
    {
        blocks.RemoveAt(0);
    }
...
```

- Нотация по умолчанию в среде Visual Studio.
- Привычнее большинству разработчиков.

Аргументы против:

- В код добавляется много незначащих строк.

```
public void foo(string source)
{
    if (string.IsNullOrEmpty(source)){
        throw new ArgumentNullException("source");
    }

    try{
        int wordLendth = ReadFromSource(source);
    }
    catch (ArgumentNullException ex){
        Log(ex.Message);
    }
    catch (EncoderFallbackException ex){
        Log(ex.Message);
    }
    catch (InvalidExpressionException ex){
        System.FailFast("всё плохо");
    }
    catch (ConstraintException ex){
        Log(ex.Message);
        throw;
    }
}
```

1: K&R

```
public void foo(string source)
{
    if (string.IsNullOrEmpty(source))
    {
        throw new ArgumentNullException("source");
    }

    try
    {
        int wordLendth = ReadFromSource(source);
    }
    catch (ArgumentNullException ex)
    {
        Log(ex.Message);
    }
    catch (EncoderFallbackException ex)
    {
        Log(ex.Message);
    }
    catch (InvalidExpressionException ex)
    {
        System.FailFast("всё плохо");
    }
    catch (ConstraintException ex)
    {
        Log(ex.Message);
        throw;
    }
}
```

2: Allman

Итоговое решение:

Для положения открывающей фигурной скобки используем нотацию Allman.

4.6 Фигурные скобки вокруг блоков кода

Предложение:

Если требуется оформить обособленный блок кода, например после операторов `if`, `for`, то обязательно использование обрамляющих фигурных скобок.

В простые блоки кода часто добавляются новые строки, а в условиях спешки можно забыть добавить фигурные скобки.

Аргументы за:

- Упрощается модифицирование кода.
- Уменьшается вероятность сложноопределяемых ошибок.

```
for(int i = 0 ; i < 20 ; ++i)
    if(i % 2 == 0)
        evenNumbers.Add(i);

// дэдлайн близится, поэтому в спешке доработали так:

for(int i = 0 ; i < 20 ; ++i)
    if(i % 2 == 0)
        if(i % 4 == 0)
            quarterNumbers.Add(i);
    else
        oddNumbers.Add(i);
    processedNumbers.Add(i);

// Программа работает очень странно, при этом за счёт выравниваний читается
// вполне комфортно.
```

Аргументы против:

- Лишние скобки не добавляют читаемости.
- В код добавляется много строк не несущих смысловой нагрузки.
- Код становится объёмнее и как следствие менее читаемым.

```
public void foo(string firstSource, string secondSource)
{
    if (string.IsNullOrEmpty(firstSource))
    {
        throw new ArgumentNullException("a");
    }

    if (string.IsNullOrEmpty(secondSource))
    {
        throw new ArgumentNullException("b");
    }
    ...

//убираем лишние скобки, код становится компактнее и читаемее.

public void foo(string firstSource, string secondSource)
{
    if (string.IsNullOrEmpty(firstSource))
        throw new ArgumentNullException("a");

    if (string.IsNullOrEmpty(secondSource))
        throw new ArgumentNullException("b");
    ...
```

Итоговое решение:

Если требуется обособленный блок кода, то он всегда окружается фигурными скобками.

Good practice

```
if (string.IsNullOrEmpty(secondSource))
{
    throw new ArgumentNullException("b");
}

for (int i = 0 ; i < buffer.Length ; ++i)
{
    Console.WriteLine(buffer[i]);
}

if(code == ErrorCodes.Success)
{
    return new Transaction();
}
else
{
    throw new SocketException();
}
```

Bad practice

```
if (string.IsNullOrEmpty(firstSource))
    throw new ArgumentNullException("a");

for (int i = 0 ; i < buffer.Length ; ++i)
    Console.WriteLine(buffer[i]);

if(code == ErrorCodes.Success)
{
    return new Transaction();
}
else
    throw new SocketException();
```

4.6.1 Пустые блоки

Предложение:

В случае если оказывается необходимым пустой блок кода, то обязательно наличие комментария объясняющего почему этот блок кода может быть пустым. Причём учитывая правило 4.6 все пустые блоки кода долны быть оформлены так:

```
{
    // Этот блок кода может быть пустым просто потому, что это пример.
}
```

Аргументы за:

- Пустой блок кода само по себе достаточно противоестественное явление и нужно пояснять почему пустой блок имеет право на существование.
- Не придётся гадать почему блок кода может быть пустым.

Good practice

```
for (spaceCount = 0 ; message[spaceCount] == ' ' ; ++spaceCount)
{
    // Подсчёт уже происходит в операторе цикла.
}
```

- Чётко читается наличие блока кода, пусть и пустого.

Bad practice

```
// на первый взгляд кажется что if - инструкция цикла просто выравнивание
// неверное.
int spaceCount;
for (spaceCount = 0 ; message[spaceCount] == ' ' ; ++spaceCount);
if(spaceCount == 0)
...

// использование пустых скобок  не сильно спасает ситуацию
int spaceCount;
for (spaceCount = 0 ; message[spaceCount] == ' ' ; ++spaceCount){}
if(spaceCount == 0)
...
```

Аргументы против:

- Увеличивается число функционально бесполезных строк. Что может даже усложнять читаемость.

Итоговое решение:

Пустой блок кода должен сопровождаться комментарием.

Исключение конструкторы. Пустой конструктор может не содержать поясняющего комментария. Однако если не очевидно почему конструктор нельзя удалить, то лучше описать это в комментарии.

Good practice

```
for (spaceCount = 0 ; message[spaceCount] == ' ' ; ++spaceCount)
{
    // Подсчёт уже происходит в операторе цикла.
}

try
{
    socket.SendMessage("Hello world!");
}
catch (ArgumentNullException)
{
    // Исключение невозможно. Так как аргумент задан явно.
}
catch (TimeoutException)
{
    // Сокет занят более приоритетными сообщениями. Так как отправка сообщения
    // не критична, то можем спокойно игнорировать это исключение.
}
```

Good practice

```
class DigitOnlyTextBox
{
    public DigitOnlyTextBox()
    {
        // Явный пустой конструктор нужен для корректного использования в XAML.
    }
}

class ClientTransaction : Transaction
{
    // И так понятно зачем нужен этот конструктор и что он делает.
    // Комментарий не обязателен.
    public ClientTransaction(Gateway gateway)
        : base(gateway)
    {
    }
}
```

Bad practice

```
// Код нарушает принцип наименьшего удивления. Непонятно почему мы можем
// так сделать.
for (spaceCount = 0 ; message[spaceCount] == ' ' ; ++spaceCount)
{
}

// Неясно почему мы можем игнорировать эти исключения.
// Если мне потребуется доработка этого кода, например отправлять сообщение
// переданное в аргументе метода, то мне будет не очевидно как мне поступать
// с обработчиками исключений. Оставлять ли их пустыми, а если обрабатывать,
// то как?
try
{
    socket.SendMessage("Hello world!");
}
catch (ArgumentNullException)
{
}
catch (TimeoutException)
{
}
```

Глава 5

C#features

5.1 warning

В релизном коде проекта warning'ов быть не должно. Ни пользовательских ни от компилятора.

Если они всё таки есть, то они обязательно должны быть отмечены как костыли 9.6.

5.2 anonymous event handlers

Запрещены. От них нельзя отписаться. Они неявно используют переменные замыкания. Их тяжело искать. Их сложно отлаживать.

5.3 default arguments

По возможности следует избегать.

Так как аргумент по умолчанию прошивается в вызывающем коде, а не в вызываемом. И это чревато проблемами в следующих вещах:

- Использование reflections.
- Изменение default параметра требует перекомпиляции всех проектов использующих этот метод.
- Разработчики языка рекомендуют использовать перегрузку вместо параметров по умолчанию.
- Есть сложноопределяемые ошибки при совместном использовании именованных параметров и параметров по умолчанию.
- Есть сложности при наследовании. Например:

```
interface IFoo
{
    int Get(int index);
}

class Bar : IFoo
{
    public int Get(int index = 0)
    {
        ...
    }
}
```

или вот ещё пример:

```
interface IFoo
{
    int Get(int index = 4);
}

class Bar : IFoo
{
    public int Get(int index = 20)
    {
        ...
    }
}
```

- Нужно тщательно следить за определёнными функциями. Иначе можно получить неожиданное поведение.

```
class Foo
{
    public Foo()
    {
        Console.WriteLine("4");
    }

    public Foo(string arg = "menu")
    {
        Console.WriteLine(arg);
    }
}
...
new Foo();//Что выведется на экран?
```

5.4 extension methods

Использование этой фичи может нарушать принцип наименьшего удивления. Соответственно её использование должно быть как-то регламентировано.

Для наших классов использование этой фичи запрещено.

В качестве альтернативы можно рассматривать класс хелпер, в методах которого первым параметром передаётся объект "расширяемого" типа. Или использовать наследование.

Расширения системных классов должны находиться в проекте Common, в нэймспейсе `SystemExtensions` в статических классах имя которых получается следующим образом: К имени расширяемого класса добавляется суффикс `Extensions`.

Расширения библиотечных классов должны находится в нэймспейсе `ИмяБиблиотекиExtensions` в корне проекта, в котором используются эти расширения.

Если расширяются `enum` то класс `Extensions` должен находиться в том же неймспейсе и в том же каталоге, что и расширяемый `enum`.

5.5 regions

Регионы используются для сокрытия частей кода. Соответственно это может как принести пользу, так и вред. Общим правилом при сокрытии является следующее: нужно скрывать несущественные детали. Навряд ли при открытии класса, методы являются несущественными деталями.

Предложение:

Использовать регионы только для следующих целей:

- Инициализация больших констант.
- Категория членов класса 6.3.1.

Аргументы за:

- Сокрытие редкоизменяемых больших констант не загромождает код.
- Сокрытие редкоизменяемых реализаций интерфейсов не отвлекает от основного предназначения класса.
- Если у вас есть большой метод в котором подэтапы выделены регионами лучше вместо регионов использовать методы. Так как регионы неявно зависят от локальных переменных объявленных в вышележащих регионах и разработчику при внесении правок в этот метод придётся держать в голове не только аргументы, поля, свойства, но и вышележащие регионы, что усложняется ещё и тем, что объявления локальных переменных разбросаны, по всему коду, а не сосредоточены в одном месте.

Аргументы против:

- Если в коде появляется необходимость скрывать часть функций - такой код нуждается в рефакторинге.

Итоговое решение:

- Регионы можно использовать только для группировки категорий членов класса и сокрытия инициализации больших констант. В других ситуациях применение регионов запрещено.
- И открывающая и закрывающая директива должны содержать имя региона. Для закрывающей директивы это имя должно быть установлено в виде комментария.

Good practice

```
class CryptoStream : IStream, IEncryptable
{
    #region Methods

    public void Write(byte[] data)
    {
        ...
    }

    #endregion // Methods

    #region IEncryptable

    public void StartEncrypt();
    public void FinishEncrypt();

    public byte[] Key
    {
        get;
        set;
    }

    #endregion // IEncryptable
}
```

Good practice

```
private readonly byte[] substitutionFunction ={
#region algorithm constant definition
// Табличное задание функции byte -> byte
// массив из 256 значений.
215, 016, 156, 189, 248, 100, 003, 192, 079, 154, 010, 000, 163, 226, 194, 085,
...//ещё 14 аналогичных строк
156, 025, 149, 232, 057, 092, 021, 243, 175, 147, 026, 185, 017, 133, 255, 091
#endregion // algorithm constant definition
};
```

Bad practice

```
class CryptoStream : IStream, IEncryptable
{
    #region IEncryptable

    #region StartEncrypt

    public void StartEncrypt();

    #endregion // StartEncrypt

    #region FinishEncrypt

    public void FinishEncrypt();

    #endregion // FinishEncrypt

    #region Key

    public byte[] Key
    {
        get;
        set;
    }

    #endregion // Key

    #endregion // IEncryptable

    ...
```

Bad practice

```
public void ShowTracks()
{
    #region получение данных

    ...

    #endregion // получение данных

    #region оптимизация

    ...

    #endregion // оптимизация

    #region отображение

    ...

    #endregion // отображение
}
```

5.6 verbatim identifier

Идентификаторы подобного рода: `int @float;` Использование запрещено.

5.7 var

В ходе дискусии об этой фиче ни одной стороне не удалось привести бесспорных аргументов в свою пользу. Использование этой фичи очень субъективная вещь.

Ключевое слово var используется для того чтобы было проще писать код. То есть фактически идеология использования var входит в конфликт с потенциальной читаемостью кода, так как от программиста скрывается тип переменной. И программисту придётся придпринимать усилия чтобы держать в голове конкретный тип переменной.

Однако в ряде ситуаций использование этого слова позволяет уменьшить дублирование.

Итоговое решение:

- Разрешается использование var при создании переменной если явно вызывается конструктор.

Good practice

```
    var a = new List<int>();
    var b = new object();
    var c = new List<int>() as object;
```

- Разрешается использование var в цикле foreach.

Good practice

```
    var a = new List<int>();
    a.Add(5);
    a.Add(4);
    foreach(var item in a)
    {
        ...
    }
```

- Использование var в LINQ выражениях.

Good practice

```
string[] aBunchOfWords = {"One","Two", "Hello", "World", "Four", "Five"};

var result =
from s in aBunchOfWords
where s.Length == 5
select s;
```

Во всех остальных случаях использование var запрещено.

Bad practice

```
    var a = 3;

    var c = IPGlobalProperties.GetIPGlobalProperties().GetActiveUdpListeners();
```

5.8 const vs readonly

При компиляции все `const` инлайнятся в код. Следовательно возникают дополнительные накладные расходы если нужно изменить значение константы.

В ходе обсуждения решено что для нас эти накладные расходы несущественны.

Итоговое решение:

- Если есть выбор используйте `const` а не `readonly`

```
class SipTimeouts
{
    // Sip rfc допускает возможность вендорам менять константы таймаутов.
    // И при изменении этой константы будет необходимо пересобрать все
    // библиотеки в которых используется эта константа.
    // Но так как мы всё равно пересобираем весь проект, и чтобы не вносить
    // путаницу используем const.

    // Хороший подход.
    public const int T1 = 500;

    // Плохой подход.
    public static readonly int T1 = 500;
}
```

- Ключевое слово `readonly` следует использовать для полей которые могут меняться в конструкторе.

Good practice

```
class Call
{
    public Call(int id)
    {
        _callId = id;
    }

    private readonly int _callId;
}
```

- Ключевое слово `readonly` следует использовать для маркирования полей которые не могут быть сделаны `const`, но являются неизменяемыми в ходе работы программы.

Good practice

```
    private static readonly TimeSpan = new TimeSpan(0, 5, 0);
```

Глава 6

Сущности определяемые пользователем

6.1 Общие правила

- Предпочитайте использовать типы предметной области, а не примитивы языка.
- По возможности используйте наиболее скрытую область видимости.

6.2 Пространства имён

Именование

- Для пространства имен используйте имя компании(для нашего отдела это очевидным образом `ElcomPlus`), затем название продукта и, возможно, название подсистемы или существенной части проекта. Например: `ElcomPlus.TiproAPI`, `ElcomPlus.Helpers`, `ElcomPlus.Ais.Protocols`.

  Полезная информация

  Внедрение этого правила отложено до начала разработки и внедрения правил для решарпера.

- Множественное число следует использовать в случае, если пространство имен объединяет некоторое количество разных, но семантически похожих сущностей. И наоборот, когда пространство имен содержит некую подсистему, стоит использовать единственное число. Сравните: `ElcomPlus.Helpers`, но не `ElcomPlus.Helper`; `ElcomPlus.Client.Controls`, но не `ElcomPlus.Client.Control`.

Использование

- Содержимое одного файла должно принадлежать одному неймспейсу.
- Не используйте одно и то же имя для класса и пространства имен. Например, не используйте класс `Debug` и пространство имен `Debug`.

Форматирование

- При объявлении пространства имен используйте единственную директиву `namespace` с полным именем пространства имен. Не используйте вложенные объявления

пространств имен.

- Выравнивайте определение пространства имён по левому краю.
- Содержимое пространства имён должно быть смещено на один отступ вправо.

Good practice

```
namespace ElcomPlus.Ais.Protocols.Sip
{
    class SipStack
    {
        ...
    }
}
```

Bad practice

```
    namespace Common
    {
        namespace Ais.Transmission.VoIpProto
{
class SipStack
{
    ...
}
}
}
```

6.3 Классы

Именование

- Используйте существительное (одно или несколько прилагательных и существительное) для имени класса.
- Не используйте специальных префиксов, поясняющих, что это класс. Например, `FileStream`, а не `CFileStream`.
- В подходящих случаях используйте составные слова для производных классов, где вторая часть слова поясняет базовый класс. К примеру, `ApplicationException` – вполне подходящее название для класса, унаследованного от `Exception`, поскольку `ApplicationException` является наследником класса `Exception`. Не стоит, однако злоупотреблять этим методом, пользуйтесь им разумно. К примеру, `Button` – вполне подходящее название для класса, производного от `Control`. Общее правило может быть, например, таким: «Если производный класс незначительно меняет свойства, поведение или внешний вид базового, используйте составные слова. Если класс значительно расширяет или меняет поведение базового, используйте новое существительное, отражающее суть производного класса». `LinkLabel` незначительно меняет внешний вид и поведение `Label` и, соответственно, использует составное имя.
- Используйте составное имя, когда класс принадлежит некоторой специфичной категории, например `FileStream`, `StringCollection`, inlineCodeExampleIntegrityException. Это относится к классам, которые являются потоками `Stream`, коллекциями `Collection`, `Queue`, `Stack`, ассоциативными контейнерами `Dictionary`, исключениями `Exception`.

- Абстрактные базовые классы должны иметь суффикс `Base`. Только абстрактные базовые классы могут иметь этот суффикс.
- Коллекциям (реализующим интерфейс `ICollection`/`IList`) нужно давать имя в виде `<ИмяЭлемента><ИмяКоллекции>`. Переменным же этих типов лучше давать имена, являющиеся множественным числом от элемента. Например, коллекция кнопок должна иметь имя `ButtonCollection`, а переменная `buttons`.
- Для классов коллекций избегайте использовать в именах членов имени хранимого типа

```
class AbonentManager
{
    public Abonent GetAbonent(string name);

    public Abonent CurrentAbonent
    {
        get;
        set;
    }
}
```

Использование

- Класс должен отвечать только одной цели.
- Запечатанные (`sealed`) классы не должны объявлять ни защищенных, ни виртуальных методов. Такие методы используются в производных классах, а `sealed`-класс не может быть базовым.
- Классы, определяющие только статические методы и свойства, не должны иметь открытых или защищенных конструкторов и должны быть помечены ключевым словом `static`, поскольку никогда не нужно создавать их экземпляры.
- По возможности избегайте статических классов. Так как фактически это шаг в сторону от объектно ориентированного программирования и попытка привнести структурное программирование.
- Наследники не должны менять поведение базового класса. Критерий: если вместо базового класса мы подставили какой-то класс наследник и что-то сломалось, принцип нарушен. Как следствие избегайте сокрытия базовых методов ключевым словом `new`.
- Избегайте в конструкторе вызывать виртуальные функции.

Форматирование

- Базовые классы и интерфейсы должны указываться на той же строке, что и объявление класса. Если список слишком длинный, снесите его, начиная с двоеточия, и сделайте отступ на одну табуляцию вправо.
- Содержимое класса должно быть сдвинуто на один отступ относительно объявления.
- Область видимости члена класса всегда должна быть явно указана.

- Область видимости самого класса всегда должна быть явно указана.
- Порядок определения модификаторов членов класса:
  1. public
  2. protected
  3. internal
  4. private
  5. new
  6. abstract
  7. virtual
  8. override
  9. sealed
  10. static
  11. readonly
  12. extern
  13. unsafe
  14. volatile
  15. async

Good practice

```
class MySample : MyClass, IMyInterface
{
    private int myint;

    public MySample(int myint)
        : base("hello, MyClass")
    {
        this.myint = myint;
    }

    [MyCustomAttribute]
    private void Inc()
    {
        ++myint;
    }

    protected virtual void EmptyVirtualMethod()
    {
    }
}
```

6.3.1 Порядок объявления членов класса

Для того чтобы с классом было удобно работать нужно чтобы он представлял собой некоторую абстракцию. Одна из задач при чтении чужого кода и при проведении code-review попытаться восстановить абстракцию которую имел в виду автор класса. Для решения этой задачи можно пользоваться несколькими инструментами: архитектурные артефакты, коментарии, именование, публичное API. Архитектура и

комментарии в нашим проекте явно в недостаточном количестве. Именование также оставляет желать лучшего. Таким образом единственным средством восстановления исходной абстракции остаётся публичное API. Благо его никуда не денешь.

Также мы исходим из предположения что программист чаще открывает класс для чтения, чтобы посмотреть что и как класс может делать, а не для редактирования.

Предложение:

Располагать члены класса в следующем порядке: `public`, `internal`, `protected`, `private`. Логика расположения: от общего к частному. Если мне нужно просто узнать что класс умеет делать, мне не нужно лезть в конец файла.

В каждой области видимости располагать члены класса в следующем порядке: внутренние типы, константы(в том числе `readonly`), статические члены, конструкторы, события, поля, свойства, методы. Логика расположения: по мере погружения в детали класса. Если мне нужны только константы я не буду читать дальше. Если я могу обойтись статическими полями мне не нужны ни конструкторы ни методы. Если мне нужно работать с объектом, то мне необходимо сначала создать объект и соответственно необходим конструктор. Если мне нужно просто получать какие-то уведомления от класса, мне не нужно знать ничего о его полях.

Допустимо группировать члены класса в публичной области видимости в соответствии с логикой использования. Например свойства и методы реализующий некоторый интерфейс.

Если класс настолько большой, что в нём несколько групп логически связанных членов класса, не являющихся реализацией интерфейсов, то скорее всего класс должен быть разделён на несколько более мелких классов.

Аргументы за:

- Члены класса расположены так, чтобы наиболее часто читаемые находились ближе к началу.
- Порядок просто формализуется.
- Лучше уж такой порядок чем никакого.

Аргументы против:

- Публичный интерфейс класса удобнее смотреть в студии. Например в Object Browser. Можно использовать сторонние плагины.
- API удобнее смотреть в системах построения документации по коду. sphinx, doxygen, etc. Даже если члены класса не комментированы, такие системы позволяют удобно просматривать список членов, группировать по областям видимости. И нам надо по крайней мере частично внедрять эти системы так как у нас есть публичное API.
- Есть другие распространённые способы группировки. Например группировать члены класса по их категориям.

Итоговое решение:

Используем группировку членов класса по категориям. Она привычна большинству членов команды.

Порядок:

1. `const`

2. `enums`
3. `fields`
4. `properties`
5. `events`
6. `constructors`
7. `methods`
8. `interfaces`
9. `internal classes`

Каждая категория должна быть заключена в регион. Если вам не нравятся регионы используйте расширение I Hate #Regions

В каждой категории члены должны располагаться в следующем порядке:

1. `public`
2. `internal`
3. `protected`
4. `protected internal`
5. `private`

Если требуется просмотреть API класса используйте Object Browser.

6.4 Интерфейсы

Именование

- Используйте описывающее существительное, прилагательное или одно или несколько прилагательных и существительное для идентификатора интерфейса. Например, `IComponent` – это описывающее существительное, `ICustomAttributeProvider` – это конкретизированное прилагательными существительное, а `IPersistable` – это характеризующее прилагательное.
- Используйте префикс `I` (заглавная `i`) для интерфейсов, чтобы уточнить, что тип является интерфейсом. Старайтесь избегать интерфейсов с двумя `I` в начале, например `IIdentifiable`. Попробуйте подобрать синоним, например `IRecognizable`.
- Для пары класс-интерфейс, в которой класс является некоторой стандартной или эталонной реализацией интерфейса, используйте одинаковые имена, отличающиеся только префиксом `I` для интерфейса. Например, `IConfigurationManager` и `ConfigurationManager`.

Использование

- Интерфейс должен быть узконаправленным, и решать только одну задачу.
- Используйте интерфейс для уменьшения связности компонентов.

Форматирование

При оформлении интерфеса следует пользоваться правилами оформления классов 6.3.

6.5 События

Именование

- Для типов событий используйте стандартные делегаты `EventHandler` и `EventHandler<T>` это позволит использовать уже созданные методы для безопасного эмитирования события.
- Для данных события используйте класс производный от `EventArgs`, и используйте суффикс `EventArgs`. Другие классы, не описывающие информацию о событии, не должны использовать этот суффикс.
- Для имен событий старайтесь использовать глаголы, которые описывают производимое над объектом действие. Например, `Click`, `DragEnter` или `FontChanged`.
- Не используйте суффиксы наподобие `Before`, `After` для идентификатора события. Используйте соответствующую форму глагола, например `Closing` перед закрытием и `Closed` после закрытия.
- При осздании события постарайтесь использовать уже объявленные делегаты. Создавайте свой делегат только если существующие не отвечают вашим целям. Особенно это касается пользовательского интерфейса. Например, если ваш элемент управления должен реагировать на нажатие кнопки мыши, следует использовать стандартный делегат. `MouseButtonEventHandler`. Обычно, такие события уже объявлены в базовом классе `Control`.
- Для методов эмитирующих событие используйте префикс `Raise`. Другие методы не должны использовать префикс`Raise`.

  Good practice

  ```
  protected virtual void RaiseStateChanged(GatewayState newState)
  {
      StateChanged(this, newState);
  }
  ```

- Имя метода обработчика события должно удовлетворять следующему шаблону: `On<ИмяОбъектаГенерирующегоСобытие><ИмяСобытия>` Другие методы не должны использовать префикс `On`.

  Good practice

  ```
  OkButton.Click += OnOkButtonClick;

  RecieveStream.PacketRecieved += OnRecieveStreamPacketRecieved;
  ```

  Bad practice

  ```
  AcceptCallButton.Click += new System.EventHandler(AcceptCallButton_Click);

  RecieveStream.PacketRecieved += RecieveStreamOnPacketRecieved;
  ```

Использование

- Не рекомендуется чтобы обработчики бросали исключения. Если у события несколько обработчиков, то последующие не будут вызваны. Также, на событие может подписаться кто угодно, на то оно и событие. И эмитирующий событие класс просто не может знать как обрабатывать получаемые исключения.
- Тип делегата события должен иметь стандартную для Microsoft форму. Первый аргумент `sender` типа `object` - ссылка на объект эмитирующий событие. Второй аргумент `e` наследник `EventArgs` - параметры события.
- Используйте наш собственный способ для эмитирования события. Чтобы избежать проблем например при эмитировании события на которое никто не подписан.
- Рекомендуется использовать `protected virtual` метод для генерации события. Чтобы наследники класса имели возможность эмитировать событие.
- Не рекомендуется передавать в качестве `sender` значение `null`.

6.6 Свойства

Именование

- Используйте существительное или одно или несколько прилагательных и существительное для имени свойства.

Использование

- Старайтесь реализовать в виде свойств только то, что отражает состояние класса или объекта. Например, если вы делаете свою коллекцию, то количество элементов `Count` должно быть свойством, а операцию преобразования ее в массив `ToArray` лучше сделать методом.
- Не рекомендуется использовать свойства «только для записи». Потребность в таком свойстве может быть признаком плохого проектирования.
- Мы должны иметь возможность менять свойства в любом порядке. Это не должно вызывать проблемы.
- Если в вашем классе есть взаимоисключающие свойства, то есть если одно свойство имеет значение то второе должно быть `null`, то ваш класс скорее всего объединяет в себе две разные сущности. Подумайте о его разделении.
- Если ваше свойство:
  - Долго работает.
  - Его значение меняется от вызова к вызову, при неизменном состоянии объекта. Например `NewId`.
  - Имеет побочные эффекты.

  используйте метод.

Форматирование

Допускается оформление в одну строку свойств с не определёнными аксессорами.

Good practice

```
public int Id { get; private set; }
```

6.7 Методы

Именование

- Используйте глаголы или комбинацию глагола и существительных и прилагательных для имен методов.

Использование

- Рекомендуется проведение рефакторинга если метод длиннее 200 строк.
- Рекомендуется проведение рефакторинга если метод имеет более 10 параметров.
- Избегайте использования именованых параметров.

6.8 Аргументы методов

Именование

- Из имени и типа параметра должны быть понятны его назначение и смысл.
- Старайтесь избегать указания типа в имени параметра.
- Не усложняйте прототип метода «зарезервированными» параметрами, которые, возможно, будут использоваться в будущих версиях реализации. Если в будущем понадобится новый параметр, используйте перегрузку методов.

6.9 Поля

Именование

- Публичных полей вообще не должно быть. Никогда.

Форматирование

- Одна декларация должна содержать не более одного поля и должна располагаться на одной строке.

6.9.1 Визуальное выделение приватных полей

Есть пожелание особым образом выделять имена приватных полей. Нужно принять решение будем ли мы это делать и если будем то как.

Предложение:

Использовать визуальное маркирование приватных полей. Например особый префикс.

Возможны несколько вариантов подобного маркирования.

- `_privateField` - приватные поля начинаются с символа подчёркивания.
- `m_privateField` - приватные поля начинаются с прификса `m_`. m - сокращение от member.
- `mPrivateField` - приватные поля начинаются с прификса `m`. m - сокращение от member.

Аргументы за:

Этот признак позволяет отличить приватное поле от локальной переменной.

- Локальная переменная должна быть инициализирована внутри метода. Приватное поле как правило уже инициализировано до начала работы метода.
- Можно неосмотрительно использовать приватное поле в качестве рабочей переменной.
- Для code-review скорее всего будет использовать не Visual Studuio. И соответственно использовать функционал GoTo Definitions будет нельзя.
- При проведении code-review часто будет нужно просматривать не весь класс, а только изменения. Соответственно повышается риск перепутать приватное поле и локальную переменную.
- В нашем проекте часто бывает непросто разобраться, так что ещё одно правило структурирования, не помешает.

Аргументы против:

- Если есть риск перепутать локальную переменную и приватное поле, это признак того, что метод слишком большой. Возможно лучше отрефакторить метод, а не придумывать ненужные префиксы.
- Если разработчик настолько слабо знаком с классом, что может перепутать приватное поле и локальную переменную, то он просто не в состоянии провести грамотное code-review.

Итоговое решение:

Для приватных полей использовать кэмэл стиль. В качестве префикса обязательно использование символа подчёркивания "_".

Исключение из правила: UI контролы, для них следует использовать стиль именования кэмэл.

После выполнения рефакторинга это правило следует пересмотреть.

Good practice

```
class MeteoStation
{
    ...
    float _currentTemperature;
    int _pressure;
    CardinalDirection _windDirection;
    ...
}
```

Bad practice

```
class Packet
{
    ...
    int Length;
    int offset;
    IPEndPoint m_sender;
    ...
}
```

6.9.2 Inline инициализация полей

Предложение:

Есть предложение запретить inline инициализацию полей и инициаллизировать поля только в конструкторах.

При необходимости инициализации некоторых полей класса значениями по-умолчанию, инициализировать данные поля исключительно в конструкторах класса, при необходимости возможно создание конструктора по-умолчанию, для инициализации полей значениями по-умолчанию.

Аргументы за:

- Упрощается отладка. Нет необходимости проходить отладчиком по inline инициализации полей. Из конструктора по-умолчанию можно легко выйти по Shift+F11 пропустив инициализацию переменных или просто не заходить в данный конструктор.
- Упрощается чтение кода. Поскольку в большинстве случаев часть полей всё-равно необходимо инициализировать в конструкторе по причине того, что:
  - Часть полей должны заполняться на основе аргументов конструктора
  - Часть полей заполняется нетривиально
  - Авто-свойства пока не поддерживают inline

  С учётом того что между объявлением полей и конструкторов должны находится объявления свойств и событий получаем 2 отдельных участка кода с инициализацией полей.
- При отказе от inline инициализации полей мы избегаем неявной автогенерации конструктора по-умолчанию.

Аргументы против:

- Во многих случаях избавляет от необходимости создания дополнительного конструктора. Это позволяет сократить объём кода для инициализации и избежать случаев, когда программист забывает проинициализировать некоторые поля.

- Inline инициализация позволяет явно увидеть какие поля зависят от параметров консруктора, а какие нет.

```
class MeteoStation
{
    ...
    float _currentTemperature;
    int _pressure;
    CardinalDirection _windDirection;
    ...

    public MeteoStation(...)
    {
      ...
      _currentTemperature = 25.0f;
        _pressure = 756;
        _windDirection = new CardinalDirection(Direction.SouthWest);
        ...
    }
}
```

```
class MeteoStation
{
    ...
    float _currentTemperature = 25.0f;
    int _pressure = 756;
    ...
    CardinalDirection _windDirection;
    ...
}
```

Итоговое решение:
Использование inline инициализации полей не регламентировано.

6.10 Перечисления

Именование

- Не используйте суффикс `Enum` в названии типа, вместо этого используйте более конкретный суффикс, например `Style`, `Type`, `Mode`, `State`. Чтобы код легче читался, используйте следующее правило: «Название перечисления + is + значение должно образовывать простое предложение». Например: `BorderStyle.Single` -> "Border style is single", `ThreadState.Aborted` -> "Thread state is “aborted”".

- Если перечисление обладает атрибутом `[Flags]`, используйте множественное число или суффикс `Flags`.

- Имена членов перечисления не должны содержать имени перечисления и другой не относящейся к конкретному значению информации.

- Если значение одного из членов перечисления зависит от других, используйте булевы операции (`&`, `|`, `^`), а не литеральные константы.

6.11 Локальные переменные

Именование

- Старайтесь не использовать короткие незначащие имена даже для счётчиков циклов. При развитии проекта код как правило дописывается, и соответственно увеличиваются области видимости переменных. И счётчик первоначально использующийся в одной строке, может быть видимым на трёх уровнях вложенности циклов.

  Но на текущий момент мы считаем допустимым использовать короткие незначащие имена для следующих целей:

  – счётчики циклов, рекомендуемые имена: "`i`, `j`, `k`";

  – объекты исключения, рекомендуемое имя `ex`.

Использование

- Объявляйте переменные непосредственно перед их использованием.
- Не объявляйте более одной переменной в одной инструкции.

6.12 Классы атрибутов

Именование

Класс, являющийся атрибутом, должен иметь суффикс `Attribute`. Ни один класс, атрибутом не являющийся, не должен иметь такого суффикса. Если семантика класса требует в названии слова что-то вроде `Attribute`, используйте синонимы, например `Descriptor`, `Sign`, `Qualifier`, `Specifier`, `Declarator`.

Форматирование

Каждый атрибут должен располагаться на отдельной строке

Good practice

```
[Obsolete]
[NotNull]
public Color LinkColor { get; private set; }
```

Bad practice

```
[Obsolete][NotNull]
public Color LinkColor { get; private set; }
```

Глава 7

Языковые инструкции

7.1 Логические выражения

- Избегайте двойного отрицания в логических выражениях.

Bad practice

```
bool canHandlePacket = !(!(_packetQueue.Count < _packetQueue.Capacity &&
                           _registeredHandler.ContainHandler(packet) &&
                           IsPacketValid(packet) &&
                           State == ServerInviteTransactionState.Active));

bool canNotHandlePacket = !(_packetQueue.Count < _packetQueue.Capacity &&
                            _registeredHandler.ContainHandler(packet) &&
                            IsPacketValid(packet) &&
                            State == ServerInviteTransactionState.Active));
```

- Рекомендуется скрывать сложные логические условия в методах или локальных переменных.

Good practice

```
bool canHandlePacket = _packetQueue.Count < _packetQueue.Capacity &&
                       _registeredHandler.ContainHandler(packet) &&
                       IsPacketValid(packet) &&
                       State == ServerInviteTransactionState.Active;

if(canHandlePacket == false)
{
    Log(packet);
    return;
}
```

Bad practice

```
if((_packetQueue.Count < _packetQueue.Capacity &&
    _registeredHandler.ContainHandler(packet) &&
    IsPacketValid(packet) &&
    State == ServerInviteTransactionState.Active) == false)
{
    Log(packet);
    return;
}
```

- Избегайте оператора присвоения в логических выражениях.

- Не используйте Yoda style логические выражения.

Bad practice

```
if(false == canHandlePacket)
{
    ...
}
```

7.2 Типы

- Используйте примитивы языка, а не стандартные .NET типы. `int` но не `System.Int32`.
- Не рекомендуется использование `unsigned` типов. Большинство системных классов используют знаковые типы, а не беззнаковые, даже в тех случаях когда отрицательные значения бессмысленны, например для длины массива. Конвертирование знаковых в беззнаковые типы провоцирует возникновение ошибок. Использование беззнаковых типов также провоцирует возникновение сложнодиагностируемых ошибок.

```
uint lastProcessedPosition = buffer.length;

while(lastProcessedPosition >= 0)
{
    // Обработка элемента буфера.

    --lastProcessedPosition;
}
```

Если вы стоите перед выбором какой тип использовать, знаковый или без знаковый, выбирайте знаковый тип.

7.3 if

Примеры форматирования

if

Good practice

```
if (condition)
{
    DoSomething();
}

if (condition)
{
    DoSomething();
    ...
    SomethingElse();
}
```

Bad practice

```
if(condition) DoSomething();

if (condition)
    DoSomething();
```

if с последующим else

Good practice

```
if (condition)
{
    DoSomething();
}
else
{
    DoSomethingOther();
}

if (condition)
{
    DoSomething();
    ...
    SomethingElse();
}
else
{
    DoSomethingOther();
    ...
    AnotherAction();
}
```

Bad practice

```
if (condition)
    DoSomething();
else
    DoSomethingOther();

if (condition) DoSomething(); else DoSomethingOther();
```

if elseif else

В подобной конструкции всегда должен быть завершающий блок `else`.

Good practice

```
if (condition)
{
    DoSomething();
    ...
}
else if (condition)
{
    DoSomethingOther();
    ...
}
else
{
    DoSomethingOtherAgain();
    ...
}
```

7.4 for

Форматирование

Good practice

```
for (int i = 0; i < 5; ++i)
{
    ...
}

for (int veryVeryVeryVeryLongIndexName = 0;
     veryVeryVeryVeryLongIndexName < 5;
     ++veryVeryVeryVeryLongIndexName)
{
    ...
}
```

for с пустым телом

Good practice

```
for (initialization; condition; update)
{
    // обязательный комментарий. Почему тело может быть пустым.
}
```

7.5 foreach

Форматирование

Good practice

```
foreach (var i in IntList)
{
    ...
}
```

7.6 while

Форматирование

Good practice

```
while (condition)
{
    ...
}
```

while с пустым телом

Good practice

```
while (condition)
{
    // обязательный комментарий
}
```

7.7 do-while

Форматирование

Good practice

```
do
{
    ...
}
while (condition);
```

7.8 switch

Использование

- Всегда добавляйте `default`. Если при нормальной работе мы не должны туда попасть бросайте исключение.

Форматирование

switch с длинными последовательностями в case

Good practice

```
int xCoordinate;

switch (condition)
{
    case 1:
    case 2:
        xCoordinate = XAxis.Projection(Shapes.First()).StartPoint;
        break;
    case 3:
        xCoordinate = XAxis.Projection(Shapes.Last()).StartPoint;
        break;
    default:
        RuntimeChecks.HandleCriticalState();
}
```

switch с короткими последовательностями в case

Good practice

```
int xCoordinate;

switch (condition)
{
    case 1:  xCoordinate = 1; break;
    case 2:  xCoordinate = 2; break;
    case 3:  xCoordinate = 3; break;
    default: xCoordinate = 100; break;
}
```

7.9 try-catch

Форматирование

Good practice

```
try
{
    ...
}
catch (SomeException ex)
{
    ...
}
finally
{
    ...
}
```

7.10 using

Форматирование

- Использовать директиву `using` внутри `namespace`.
- Перечислять импортируемые пространства имен необходимо в следующей последовательности: пространства имен .NET Framework, пространства имен сторонних производителей, собственные пространства имен из других проектов, пространства имен из текущего проекта. Каждый такой раздел должен отделяться одной пустой строкой, а имена внутри раздела должны быть отсортированы по алфавиту.
- Директивы `using`, содержащие alias, должны идти в конце соответствующих разделов, и также быть упорядоченными по алфавиту.

Глава 8

Политика работы с ошибками

- Возникновение ошибки в нашем проекте очень критично. Наше приложение используется в самых разных сферах, например в системе связи скорой помощи, где отсутствие связи может повлечь за собой вред здоровью, или в системах связи больших карьеров, где простой из-за отсутствия связи может привести к многомиллионным убыткам. Соответственно наш проект должен быть максимально надёжным. Мы должны внимательно относиться ко всем ситуациям, которые могут привести к появлению ошибки.

- Если метод обнаруживает, что входные параметры некорректны, следует проинформировать вызывающий код броском исключения. Не нужно пытаться скорректировать входные параметры чтобы была возможность продолжить работу. Например, если в метод `public void InsertElement(object element, int position)` в качестве значения параметра `position` передано `-1` следует бросить `IndexOutOfRangeException`, а не добавлять элемент в начало коллекции.

  Требует обсуждения
  - Как информируем о возникновении исключительной ситуации? Допустимо ли использование кодов ошибок? Допустимо ли возвращать null как результат некорректной работы функции? Или все функции возвращающие null должны иметь суффикс OrDefault?
    Правило хорошее но непонятно как его применять когда есть много старого кода не удовлетворяющего этому правилу.

- В процессе разработки встречаются ситуации в которых выполнение некоторого условия однозначно свидетельствует о ошибке программиста, а также о том что исходный код должен быть доработан. Например:

```
enum Shape
{
    Round,
    Triangle
}

class MarkerFactory
{
    public Marker CreateMarker(Shape shape)
    {
        switch(shape)
        {
            case Shape.Round: return new RoundMarker();
            case Shape.Triangle: return new TriangleMarker();
            default:
                // Если мы сюда попадаем, значит кто-то расширил набор
                // допустимых фигур и мы должны принять решение что делать
                // с новой фигурой.
                RuntimeChecks.HandleCriticalState("Unknown Shape");
        }
    }
}
```

при этом учитывая тот факт, что в нашем проекте есть места в которых исключения проглатываются, хочется иметь механизм оповещения о ошибочной ситуации, который нельзя проигнорировать.

В этих ситуациях следует использовать обёртку над методом `System.Environment.FailFast()`. Этот метод гарантированно крашит приложение, и создаёт memory dump на момент возникновения ошибки.

В обёртке этого метода следует проверять определённый флаг настроек, и вызывать `System.Environment.FailFast()` только если этот флаг установлен. Это нужно чтобы не убивать приложение у наших клиентов(это противоречит сформулированному правилу поведения при обнаружении некорректной ситуации), при этом непонятной ситуации на объекте мы будем иметь возможность попросить на объекте установить этот флаг и будет шанс получить больше информации о возникающих проблемах. Также краш в результате вызова этого метода будет явно свидетельствовать о том, что требуется переписать исходный код.

Этот метод должен находиться в следующем месте: `ElcomPlus.Common.RuntimeChecks.HandleCr`

> **Требует обсуждения**
> После устранения `catch` блоков проглатывающих ошибки, можно будет рассмотреть вопрос замены `FailFast` на генерацию исключения наследника `System.SystemException`.

- Не рекомендуется перехватывать исключения унаследованные от класса `System.SystemExcep` как правило их возникновение свидетельствует об ошибках в исходном коде программы.

  Пример наследников этого класса:

  `System.NotImplementedException`
  `System.NotSupportedException`
  `System.NullReferenceException`
  `System.OperationCanceledException`
  `System.OutOfMemoryException`

- Чем раньше обнаружена ошибка тем дешевле её устранение и тем меньше вреда она наносит в ходе работы приложения. Следовательно предпочтительнее тщательно проверять входные параметры методов, но в то же время это потребует выполнения большого объёма работы, которая может быть излишней.

  В качестве решения принят следующий подход. Все публичные методы должны проверять свои входные параметры и бросать соответствующие исключения. Приватные методы могут этого не делать так как они вызываются публичными методами, которые мы мы создаём сами, и которые уже проверили входные параметры.

- При создании проверок следует проверить является ли эта проверка актуальной в релизной версии. Если нет, то её нужно исключить из релизной версии.

- Если необходимо сгенерировать исключение, для типа исключения используйте следующие варианты(в порядке предпочтения):

  - системные исключения
  - уже определённые исключения
  - новый тип исключения

- Так как у нас есть требование, что все исключения должны иметь семантическое значение, то с высокой вероятностью они будут общими для логически связанных классов, логические связанные классы должны располагаться в одном неймспейсе, соответственно классы исключения нужно располагать в неймспейсе класса который генерирует исключение. Если есть необходимость генерировать одно исключение в нескольких классах, то это исключение должно располагаться в общем нэймспейсе этих классов. Верхним неймспейсом наших классов является неймспейс `ElcomPlus`

- Все бросаемые исключения должны иметь семантическое значение в рамках класса. Иначе это нарушает принцип инкапсуляции. Например Кэш не должен бросать исключение `FileIOException`. Так как если мы захотим изменить механизм хранения данных, например на базу данных нам придётся во всём коде использующем наш кэш заменять `FileIOException` на `DbIOException`. В данной ситуации следует бросать специфичное для кэша исключение типа: `CacheStorageException`, и в нём в `InnerException` передавать конкретное исключение.

- Перехватывать общее исключение допустимо в следующих случаях:

  - Это оправдано семантикой. Например исключения из обработчиков событий. Мы в принципе не знаем и не должны знать всех кто захочет подписаться на наши события, какие исключения они могут сгенерировать и как их нужно обрабатывать.
  - Предотвратить выход непредвиденного исключения за пределы нашего кода. Соответственно блок `catch (Exception ex)` нужно рассматривать как страховку от генерации непредвиденного исключения, попадание в этот блок не может считаться нормальным поведением программы и если мы в него попадаем, то нужно тщательно разбираться почему так произошло и исправляться эту ситуацию. В этом блоке рекомендуется использовать обёртку над методом FailFast как описано тут: 8. Подобный блок может располагаться например в следующих местах: верхний уровень пользовательского интерфейса, методы из clientApi, обработчики событий.

В остальных случаях нужно перехватывать максимально узкое исключение. И перехватывать нужно все исключения которые мы переупаковываем или по которым мы можем принять решение об обработке ошибки. Например:

Good practice

```
try {
    socket.SendTo(data, 0, count, SocketFlags.None, m_companionAddress);
}
catch (SocketException ex) {
    StopOnError(ex);
}
catch (ObjectDisposedException ex) {
    StopOnError(ex);
}
catch (SecurityException) {
    //TODO: залогировать.
    RuntimeChecks.HandleCriticalState();
}
```

Good practice

```
class AbonentEditor
{
    ...
    private void OnSaveButtonClick(object sender, AbonentEventArgs e)
    {
        try
        {
            // Логика сохранения, вместе с обработкой ошибок.
            ...
        }
        catch (Exception ex)
        {
            RuntimeChecks.HandleCriticalState(ex.Message);
        }
    }
    ...
}
```

Глава 9

Комментирование

9.1 Общие правила

- Для комментариев используйте литературный русский язык. Избегайте ошибок, обсценной лексики, расплывчатых и неоднозначных формулировок.

  > **Требует обсуждения**
  > Тут нужно оговорить список исключений. Как минимум это наше public API.

- Используйте систему XML Doc компании Microsoft для разметки комментариев.

- Запрещено использование многострочных комментариев (`/*...*/`). Если вам нужно создать многострочный комментарий то для классов и членов классов используйте специальный тип комментариев (`///`) и разметку Microsoft, для иных случаев используйте подряд идущие однострочные комментарии (`//`). Чтобы закомментировать/раскомментировать большой блок кода пользуйтесь пунктами меню EDIT -> Advanced -> Comment Selection. Либо сочетаниями клавиш, чтобы закомментировать: Ctrl+K, Ctrl+C; чтобы раскомментировать: Ctrl+K, Ctrl+U.

- Отделяйте текст комментария одним пробелом « `// Текст комментария.`».

- Комментируя код, старайтесь объяснять, что он делает, а не какая операция производится. Так, инструкции `if` соответствует выражение «если... то...», причем часть, идущая за «то», является кодом, который будет выполняться, если выражение в `if` будет верным. Таким образом, для конструкции «`if (somePath && File.Exists(somePath))`», нужно написать комментарий « `Если выбранный файл существует, то...`», а не « `Производим проверку на наличие файла и, если он имеется, удаляем его`». Часть предложения, идущую за «то», вписывайте непосредственно перед выполнением конкретных действий. Для инструкций, осуществляющих действия, пишите « `Производим...`» или « `Делаем...`», где вместо троеточия вписывайте описания действий. Описывая действия, старайтесь описывать суть происходящего, а не то, что делают те или иные операторы. Так, совершенно бессмысленны комментарии вроде «`Присваиваем переменной a значение b`» или «`вызываем метод f`».

- Помните, что экономить на комментариях нельзя. Однако не стоит также формально подходить к процессу создания комментариев. Задача комментария – упростить понимание кода. Есть немало случаев, когда сам код отличным образом себя документирует.

- Не коммитьте закомментированный код. Либо пишите почему этот код важно сохранить в таком виде.

9.2 Форматирование XML Doc комментариев

- Максимальная длина строки такая же как и для исходного кода 4.1.
- Открывающий и закрывающий XML тэги должны располагаться на отдельных строках, и отделяться от символов начала комментария пробелом.
- Содержимое комментария должно быть смещено на один отступ относительно открывающего тэга.
- Для коротких записей, без вложенных элементов допустимо оформлять запись в одну строку, вместе с открывающим и закрывающим XML тэгами.
- При определении атрибутов символ ""должен быть окружён пробелами.
- Комментарий не должен содержать пустых строк.
- Если запись содержит много атрибутов то при переносе строки, атрибуты на новой строке должны быть выравнены по первому атрибуту первой строки.

9.3 Комментарии API компонентов

Предложение:

Публичное API компонетов должно быть откомментировано исчерпывающим образом. Исчерпывающий комментарий это комментарий удовлетворяющий следующему критерию: если прочитав только комментарий и сигнатуру класса/члена класса мы можем его использовать абсолютно корректно, комментарий называется исчерпывающим.

Под компонентом понимается минимальный элемент используемый в архитектурных документах. Компонент это некоторый набор логически связанных классов предоставляющих какую-то функциональность. В качестве примеров компонентов можно рассмотреть такие мифические сущности:

- хранилище настроек
- радиосеть(NAI, AIS, MotoTRBO)
- костяк отображающий различные панели настроек

За состоянием и актуальностью этих комментариев должен следить codeOwner этого компонента.

Полезная информация

Да, на текущий момент в нашем проекте компоненты не выделены.

В качестве примера открываем в msdn любую статью о любом методе или свойстве. Либо можно посмотреть документацию по QT: UDP сокет, QMap. Причём по документации QT явно видно что статья к методу вовсе не обязательно должна иметь большой объём. Примеры комментариев можно найти в конце статьи.

Аргументы за:

- Нам всем не нравится что наш проект представляет собой мешанину сильно связанных классов. Мы все хотим разделить его на какие-то независимые компоненты. Но после этого разделения нам будет нужно эти компоненты как-то использовать желательно не влезая в их кишки. Исчерпывающие комментарии дадут нам такую возможность. Они будут документировать предоставляемое API.

- Если разработчик хочет только использовать компонент, то ему не придётся лезть внутрь компонента, чтобы понять как это сделать.

- Если разработчику нужно доработать компонент, то исчерпывающие комментарии дадут ему представление о том как компонент работает на текущий момент.

- Упростится разбор полётов. Если есть чёткое описание как должен работать метод, то проще определить где ошиблись в использовании этого метода, либо понять что метод работает не так как описано. И соответственно найти кто именно допустил ошибку.

- Повысится надёжность проекта. Правки влекущие за собой изменение API компонента будут более строго отслеживаться. Сейчас на необходимость контроля таких правок, фактически на контроль за архитектурными изменениями, просто забивается. Что влечёт за собой непредсказуемость поведения.

- Дополнительный контроль за логичностью поведения API. Если разработчик не может на нормальном русском языке объяснить как работает его классы или методы, то почти наверное он плохо реализовал этот компонент. Написание комментариев часто выявляет нарушение принципа наименьшего удивления.

- Подобные комментарии могут служить мостиком связывающим архитектурные документы и код. Они лежат рядом с исходным кодом но в то же время являются документацией. Соответственно выше если изменилась документация, то нужно проконтролировать, не привело ли это к необходимости изменить архитектуру.

- Позволяют выявить нарушение архитектуры проекта. Если изменения в исходном коде повлекли за собой необходимость изменений в таких комментариях то это один из признаков, что нужно проверить изменения на соответствие архитектуре.

- Позволят сохранить информацию о компоненте

- Это правило относительно просто внедрять. Выделили новый компонент, назначили codeOwner, откомментировали.

Аргументы против:

- Разработчики не любят писать комментарии. Им это не нравится.

- Относительно высокие трудозатраты на создание и поддержку в актуальном состоянии.

- Неэффективны в часто меняющихся проектах, либо часто меняющихся частях проекта.

- Требует наличия архитектурных документов.

Итоговое решение:

Публичное API компонетов должно быть откомментировано исчерпывающим образом. Исчерпывающий комментарий это комментарий удовлетворяющий следующему критерию: если прочитав только комментарий и сигнатуру класса/члена класса мы можем его использовать абсолютно корректно, комментарий называется исчерпывающим.

Good practice

```
class Gateway
{
    /// <summary>
    /// Создаёт новый исходящий звонок.
    /// </summary>
    /// <remarks>
    /// Новый звонок находится в состоянии Initial, и должен быть запущен
    /// методом Start().
    /// </remarks>
    /// <example>
    /// <code>
    /// ICall call = _gateway.CreateCall(from, to);
    /// call.StateChanged += OnCallStateChanged;
    /// call.Start();
    /// </code>
    /// </example>
    /// <param name="sourceAbonentId">
    /// Идентификатор абонента инициирующего вызов.
    /// </param>
    /// <param name="destinationAbonentId">
    /// Идентификатор абонента, которого мы хотим вызвать.
    /// </param>
    /// <exception cref="System.ArgumentException">
    /// Некорректные идентификаторы. Например меньше нуля. Конкретную
    /// информацию смотреть в сообщении ошибки.
    /// </exception>
    /// <exception cref="ElcomPlus.Ais.RfcArt.InconsistantStateException">
    /// Создавать звонок можно только если гейтвей запущен методом Start()
    /// и находится в активном состоянии.
    /// </exception>
    /// <returns>
    /// Созданный объект звонок. В случае ошибки бросается исключение. Данный
    /// метод не возвращает null.
    /// </returns>
    public ICall CreateCall(int sourceAbonentId, int destinationAbonentId)
    {
        ...
    }
}
```

Good practice

```
class Call
{
    /// <summary>
    /// Отправляет порцию голосовых данных.
    /// </summary>
    /// <param name="data">Фрагмент голоса.</param>
    /// <exception cref="System.ArgumentNullException">
    /// Входной аргумент равен null.
    /// </exception>
    /// <exception cref="System.ArgumentException">
    /// Голосовой фрагмент имеет неверный формат.
    /// </exception>
    /// <exception cref="ElcomPlus.Ais.RfcArt.InconsistantStateException">
    /// Отправлять голосовые данные можно только если звонок находится
    /// в состоянии Active.
    /// </exception>
    /// <exception cref="ElcomPlus.Ais.RfcArt.TransportException">
    /// Ошибка передачи данных на транспортном уровне. Конкретную ошибку
    /// смотрите в InnerException. Звонок переведён в состояние Finished.
    /// </exception>
    public void SendVoiceFragment(VoiceFragment data)
    {
        ...
    }
}
```

9.4 Обязательные комментарии членов класса.

Предложение:

Запретить распространённое требование писать обязательные комментарии для публичных членов класса.

Аргументы за:

- Комментарии нужны чтобы упростить понимание исходного кода. Если мы просто будем требовать обязательно комментировать части кода, это правило никак не контролирует качество создаваемых комментариев. И если мы всё равно будем вынуждены контролировать качество комментариев в ходе code-review, то и наличие комментариев в требуемых местах можно проверять с его же помощью.

- Львиная доля комментариев созданная вследствие наличия этого правила имеет абсолютно формальный характер. Как правило это просто перевод имени метода, свойства.

Bad practice

```
/// <summary>
/// Строка поиска
/// </summary>
private string searchString = string.Empty;
```

Bad practice

```
/// <summary>
/// Показывает, является ли выбранный элемент абонентом
/// </summary>
private bool isSubscriberSelected;
```

Bad practice

```
/// <summary>
/// Инициализирует новый экземпляр класса <see cref="RadioSystemViewModel"/>.
/// </summary>
/// <param name="subscribersManager">
/// The subscribers Manager.
/// </param>
/// <param name="callingController">
/// The calling Controller.
/// </param>
/// <param name="categoriesRepository">
/// The categories Repository.
/// </param>
/// <param name="callWindowsService">
/// The call Windows Service.
/// </param>
public RadioSystemViewModel(
    ISubscribersManager subscribersManager,
    CallingController callingController,
    CategoriesRepository categoriesRepository,
    ICallWindowsService callWindowsService)
{
    ...
}
```

Bad practice

```
/// <summary>
/// Получает значение, показывающее, что разрешено добавление-удаление
/// абонентов
/// </summary>
public bool AllowSubscribersModification
{
    get
    {
        ...
    }

    private set
    {
        ...
    }
}
```

- Лучше уж вообще не иметь комментариев, чем иметь формальные.
  - Если комментариев совсем не будет, то не будет и иллюзии что они что-то поясняют.
  - Если есть комментарий который действительно поясняет что-то важное, то он незаметен и легко может быть пропущен, затерявшись в большом количестве формальных комментариев.
  - Если большая доля комментариев формальна, разработчики начинают пренебрежительно относиться к комментариям. И скорее всего не будет вообще их читать так как они бесполезны.
- Формальные комментарии не дают вообще ничего нового при чтении кода.

Аргументы против:

- Наличие секции summary позволяет заполнить всплывающую подсказку.

Итоговое решение:

В нашем проекте не требуется наличие обязательных комментариев, кроме API компонентов.

9.5 TODO комментарии

Предложение:

В итоговом коде, который идёт в релизную ветку, TODO комментариев быть не должно. Допустимо использование TODO комментариев в промежуточном коде и в побочных ветках.

Аргументы за:

- Сам смысл TODO комментария - пометка для разработчика чтобы не забыть сделать что-либо. Если работа над задачей закончена, то соответственно разработчик должен закончить всё что он отложил на потом.
- TODO комментарии всё равно никто не делает, и они просто засоряют "Task List" Visual Studio и соответственно делают этот инструмент сложноиспользуемым.
- Если в ходе выполнения работы над какой-то задачей обнаружена необходимость выполнения значительных доработок, то эта доработка должна быть занесена в рэдмайн, в категорию "developer" или иную подходящую категорию. Чтобы эта задача не потерялась и в подходящее время была назначена на выполнение и результат был проконтролирован.
- Если разработчик дорабатывает или читает чужой код, то наличие TODO комментариев сбивает с толку. Насколько вообще работоспособен код содержащий такой комментарий?
- При возникновении сомнений относительно нужности выполнения задачи, помеченной как TODO, комментарий лучше удалить. Если в ходе code-review замечания на эту тему возникнут, то их всё равно будет нужно устранить. Если замечаний не возникнет, значит и так всё в порядке.

Аргументы против:

- Дополнительная работа перед завершением задачи.

Итоговое решение:

В результирующем коде, TODO комментариев быть не должно.

9.6 Предупреждения о костылях

В тексте программы нужно отмечать места где воткнуты костыли. Например как-то так:

```
// КОСТЫЛЬ имя_автора #2158 Поиск активной конференции нужно переделать.
// Активные конференции нужно хранить в менеджере, а не определять по открытым
// окнам.
```

Где:

- **КОСТЫЛЬ** - специальная метка комментариев, описывающих костыли. В прочих случаях её использовать нельзя.
- **имя_автора** - Имя человека который создал этот костыль, и который может проконсультировать по этому поводу. Фактически может быть определено в рэдмайне по номеру задачи, но для оптимизации лучше указывать явно.
- **#2158** - Номер задачи в рэдмайне в ходе выполнения которой, сделан этот костыль.
- **Поиск активной конференции нужно переделать.** - Описание того что сделано плохо, и как нужно сделать правильно.

Требует обсуждения

Также хорошо бы добавить статус задачи что-то типа "закостылено".

Глава 10

Проект

10.1 Общие правила

- Все не вложенные типы (размещаемые в пространствах имен или непосредственно в глобальном пространстве), должны находиться в отдельных файлах.
- Старайтесь не делать излишней вложенности типов. Помните, что вложенными должны быть только тесно связанные типы.
- Для не generic типов имя файла должно совпадать с именем содержащегося типа.
- Для generic типов имя должно сопадать с именем(за исключением пространств имён) возвращаемым методом System.Type.GetType.

  То есть удовлетворять шаблону: `ИмяТипа`ЧислоGenericАргументов`

  Символ "`" это символ находящийся на клавише с буквой ё в английской раскладке. Microsoft называет этот символ backtick. Его Alt код 96.

  Пример: EventHandlerExtensions`1
- Для хранения делегатов создаётся один файл на неймспейс и все публичные делегаты неймспейса должны лежать в этом файле.
- В релизном проекте не должно быть warning компилятора.
- Наличие и содержание заголовочного комментария файла не регламентируется. Но большинство разработчиков считает эти комментарии бессмысленными.

  Требует обсуждения
  Хотим ли мы использовать дополнительные средства анализа кода.
  Хотим, но развитие должно быть поступательным, поэтому внедрение этого правила отложено до лучших времён.

10.2 Различия DEBUG и RELEASE версий

В нашем проекте допустимо различие между DEBUG и RELEASE версиями.

В DEBUG версии могут быть такие отличия:

- особое лицензирование, чтобы упростить жизнь разработчика;

- дополнительные проверки важные в процессе разработки, но неактуальные в релизной версии;
- отладочная информация для разработчика.

10.3 Большое число однотипных файлов в каталоге

Требование оформления каждого класса в отдельном файле может привести к появлению большого количества файлов. При этом возможно что часть этих файлов будет иметь одну природу. Например файлы параметров событий, или файлы исключений.

Предложение:

Вынести однородные файлы в каталог.

- Для этого каталога должен быть отключён "Namespace Provider".
- Имя каталога должно быть заключено в квадратные скобки.
- Этот каталог не может содержать подкаталогов.

Аргументы за:

- Структурирует дерево решения.
- Дерево решения не загромождается редкомодифицируемыми файлами.
- Не нарушает логику построения `namespace`.

Аргументы против:

- Без дополнительной информации непонятно что и зачем сделано.

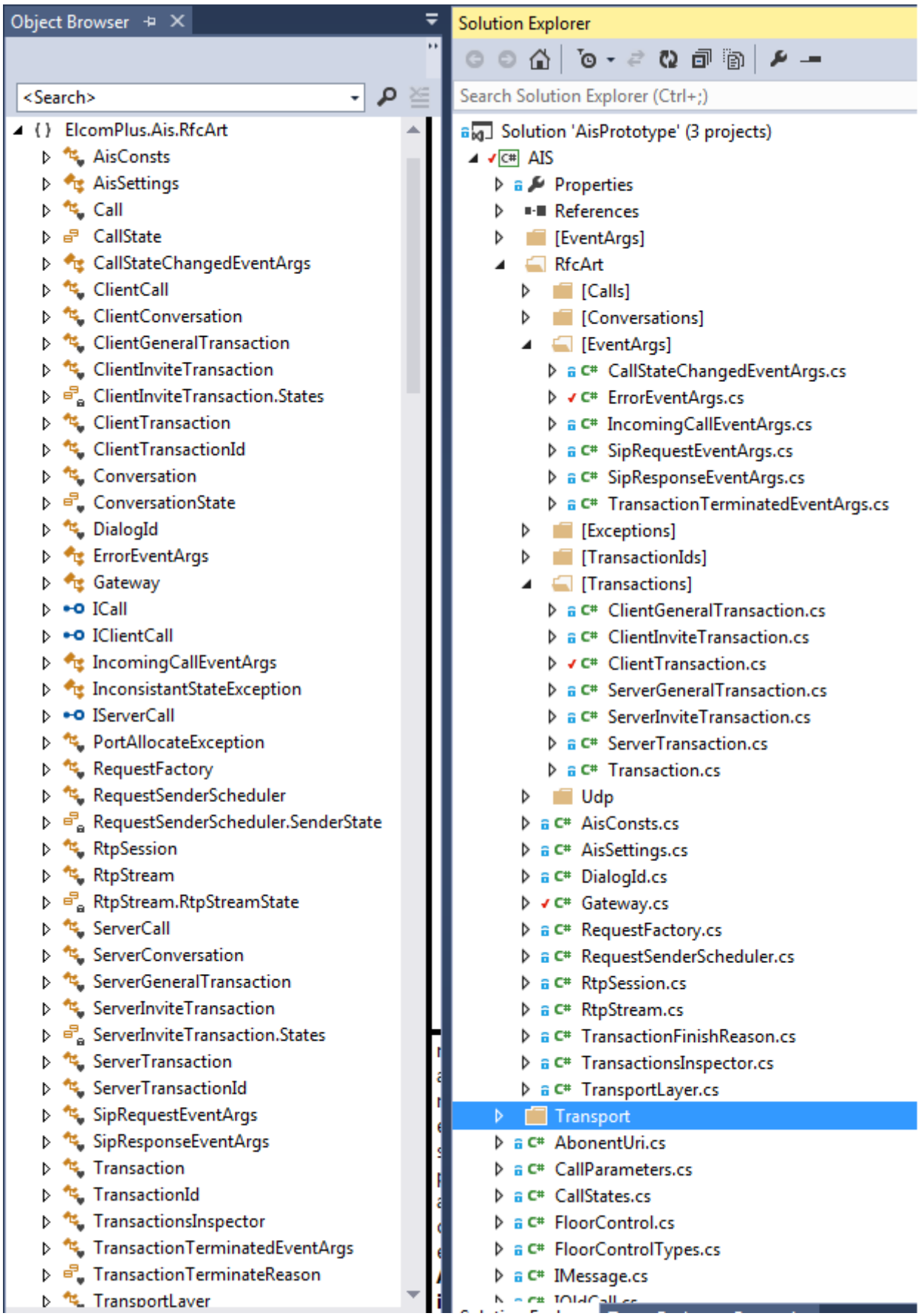
Итоговое решение:

Допустимо помещать однородные файлы в каталоги для которых отключен Namespace Provider, имена этих каталогов должны быть заключены в фигурные скобки.

Например `[EventArgs]`, `[Exceptions]`.

Ниже пример из "боевого" решения где используется эта возможность.

Один и тот же namespace слева и справа. Но справа его файлы сгруппированы с использованием этой возможности. И это сильно облегчает навигацию по этому namespace.


Object Browser
<Search>
ElcomPlus.Ais.RfcArt
AisConsts
AisSettings
Call
CallState
CallStateChangedEventArgs
ClientCall
ClientConversation
ClientGeneralTransaction
ClientInviteTransaction
ClientInviteTransaction.States
ClientTransaction
ClientTransactionId
Conversation
ConversationState
DialogId
ErrorEventArgs
Gateway
ICall
IClientCall
IncomingCallEventArgs
InconsistantStateException
IServerCall
PortAllocateException
RequestFactory
RequestSenderScheduler
RequestSenderScheduler.SenderState
RtpSession
RtpStream
RtpStream.RtpStreamState
ServerCall
ServerConversation
ServerGeneralTransaction
ServerInviteTransaction
ServerInviteTransaction.States
ServerTransaction
ServerTransactionId
SipRequestEventArgs
SipResponseEventArgs
Transaction
TransactionId
TransactionsInspector
TransactionTerminatedEventArgs
TransactionTerminateReason
TransportLayer
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AisPrototype' (3 projects)
AIS
Properties
References
[EventArgs]
RfcArt
[Calls]
[Conversations]
[EventArgs]
CallStateChangedEventArgs.cs
ErrorEventArgs.cs
IncomingCallEventArgs.cs
SipRequestEventArgs.cs
SipResponseEventArgs.cs
TransactionTerminatedEventArgs.cs
[Exceptions]
[TransactionIds]
[Transactions]
ClientGeneralTransaction.cs
ClientInviteTransaction.cs
ClientTransaction.cs
ServerGeneralTransaction.cs
ServerInviteTransaction.cs
ServerTransaction.cs
Transaction.cs
Udp
AisConsts.cs
AisSettings.cs
DialogId.cs
Gateway.cs
RequestFactory.cs
RequestSenderScheduler.cs
RtpSession.cs
RtpStream.cs
TransactionFinishReason.cs
TransactionsInspector.cs
TransportLayer.cs
Transport
AbonentUri.cs
CallParameters.cs
CallStates.cs
FloorControl.cs
FloorControlTypes.cs
IMessage.cs

Глава 11

Прочие языки

Требует обсуждения
Эта глава будет заполнена после выпуска релизной версии документа.

java

Objective C

скриптовые языки

C++

XAML

SQL