

# Технологии программирования

Лекция №4  
ИС, весна 2022

# Reflection and codegen

# Поставим перед собой задачу

- Представим, что у нас есть задача:

Я хотел создать множество товаров, которое магазин определяет при создании. Через Assembly можно найти список классов, которые будут представлять каждый товар, они все помечены интерфейсом. У меня также есть Generic Класс, который будет существовать для каждого вида товаров, чтобы можно было удобно менять цену. Как раз таки на моменте с Generic я не могу дать ему знать о существовании товаров, потому что он принимает интерфейс, а не type

# Хейтеры

Hater 1



Нельзя, ты получил тип, а не инстанс сам  
А вообще выглядит как будто ты улетел не туда 20:26

Hater 2



Ты делаешь что-то очень страшное 20:26

Hater 3



я не особо понимаю что ты хочешь, но ты творишь что-то  
очень странное 20:44

Hater 4



Это очень плохо и совсем не объектно ориентированно. Я на  
1000% уверен что прибегать к рефлексии не надо, то что ты  
хочешь можно сделать либо нормальными средствами, либо  
этого делать не стоит. Можешь ещё раз пояснить зачем тебе  
это? 21:13

Hater 3 (again)



если у тебя есть на каждый товар класс, то ты чтото делаешь  
не так 21:16

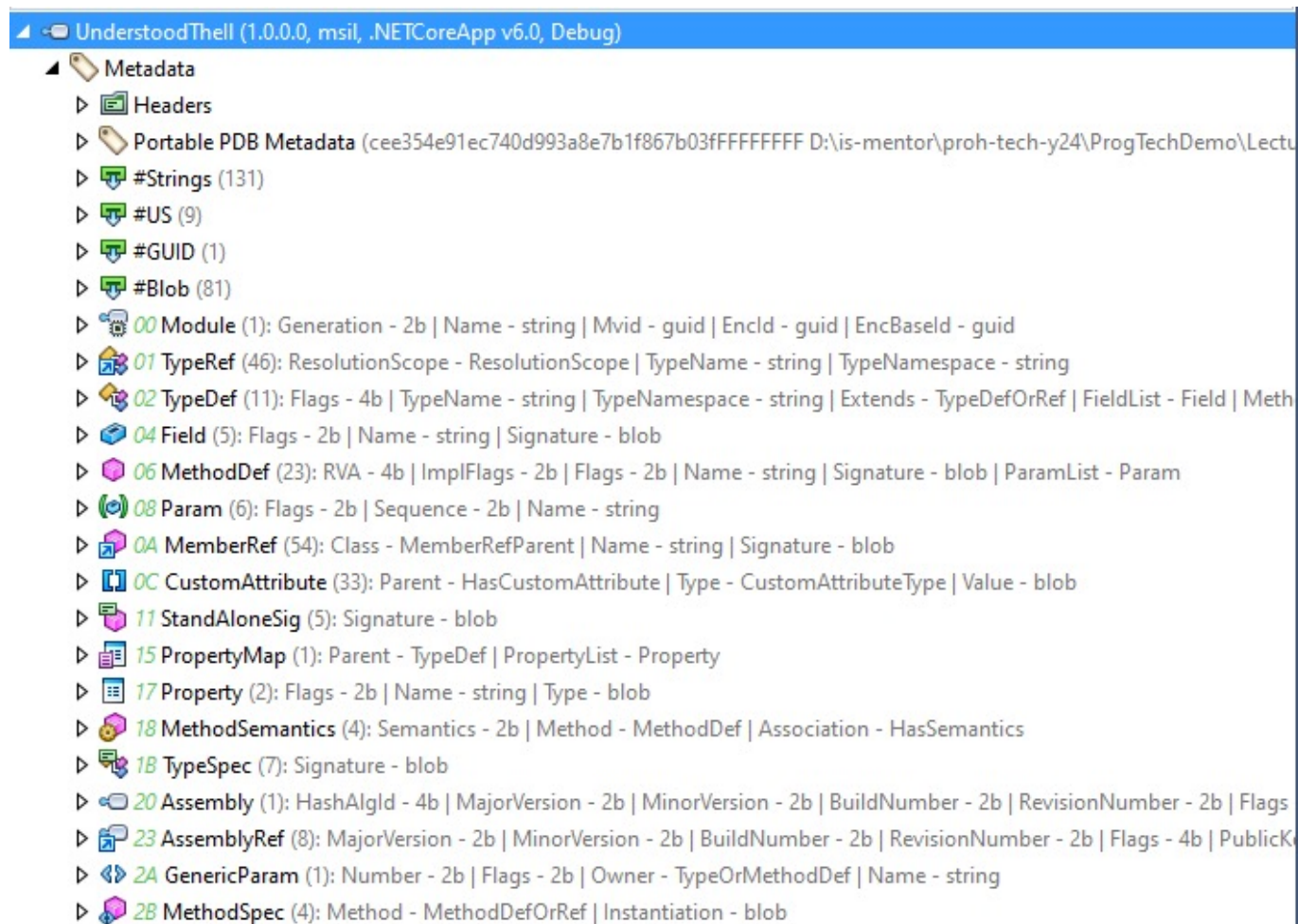
# Проектируем решение

- Есть класс Item, который описывает то, что лежит в базе и возможность с базы получить эти данные.
- Есть интерфейс IShopItem, который должен быть реализован нашими продуктами.
- Ну и наш магазин, который это всё должен сгенерировать.

# Assembly, метадата, манифест

- Assembly – это минимальная единица развёртывания .NET кода. Во время компиляции каждый проект собирается в свою Assembly. Любой nuget-пакет – это тоже Assembly, которую завернули в дополнительную обёртку.
- DLL можно вскрыть через ildasm или dotPeek и увидеть, что в нём описываются зависимости, версия, много другой информации про dll.

# DLL в dotPeek



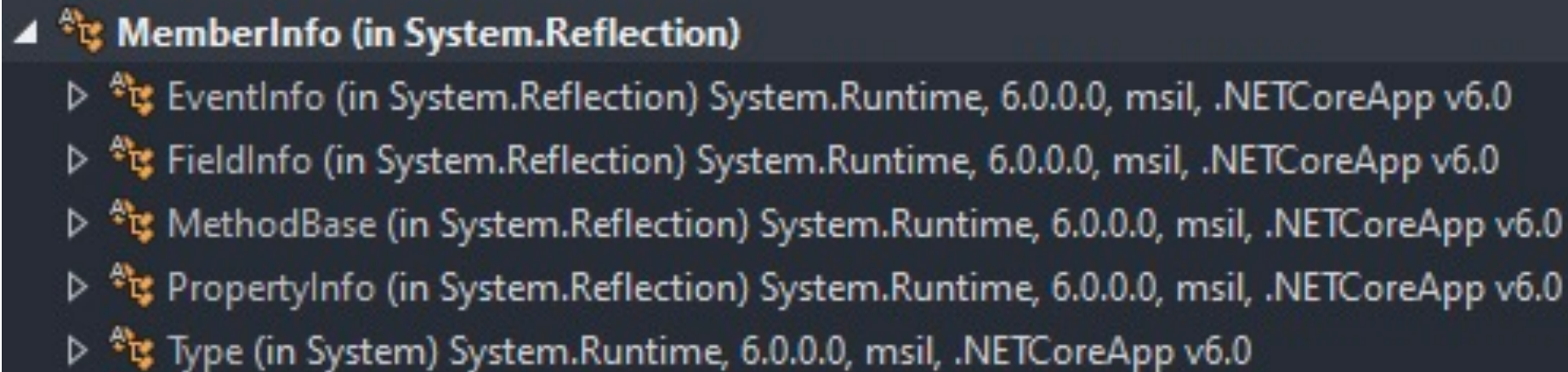
# AssemblyBuilder, module builder

```
1  AssemblyName assemblyName =  
2  |      Assembly.GetExecutingAssembly().GetName();  
3  AssemblyBuilder assemblyBuilder =  
4  |      AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);  
5  ModuleBuilder moduleBuilder =  
6  |      assemblyBuilder.DefineDynamicModule(nameof(UnderTheHoodOfIl));  
7
```



# Object > MemberInfo > Type

- MemberInfo – это базовый класс, от которого наследуются другие типы, которые описывают метайнформацию о различных типах.



▲ **MemberInfo (in System.Reflection)**

- ▷ **EventInfo (in System.Reflection)** System.Runtime, 6.0.0.0, msil, .NETCoreApp v6.0
- ▷ **FieldInfo (in System.Reflection)** System.Runtime, 6.0.0.0, msil, .NETCoreApp v6.0
- ▷ **MethodBase (in System.Reflection)** System.Runtime, 6.0.0.0, msil, .NETCoreApp v6.0
- ▷ **PropertyInfo (in System.Reflection)** System.Runtime, 6.0.0.0, msil, .NETCoreApp v6.0
- ▷ **Type (in System)** System.Runtime, 6.0.0.0, msil, .NETCoreApp v6.0

# Object > MemberInfo > Type

- В Type можно найти много информации про данные:
  - Название, namespace, сборка;
  - Является ли структурой, классом, enum'ом, generic'ом;
  - Получить список полей, конструкторов, методов и пр.

# TypeInfo, GetType, typeof

- TypeInfo в целом позволяет получить около такие же данные с той лишь разницей, что создание TypeInfo - это уже процесс парсинга метаданных и получения всей информации, а Type предоставляет методы получения этой информации.
- Самые простые способы получения Type – это метод GetType() у экземпляров и кейворд typeof() для типов.

# TypeBuilder

- Рассмотрим, как можно создать тип с использованием билдера.
- TypeBuilder:
  - Имя
  - Атрибуты (класс, структура, интерфейс, вложенный, приватный etc)
  - Базовый тип, реализованные интерфейсы
  - Набор методов Define\* (например, DefineConstructor)

# MethodInfo, MethodBuilder

- MethodInfo (внезапно!) описывает метайнформацию о методах.
- Например:
  - Название;
  - Атрибуты: приватный, публичный, виртуальный;
  - Типы аргументов и возвращаемого значения.

# Вызов метода через MethodInfo

```
BindingFlags bindingFlags = BindingFlags.Instance | BindingFlags.NonPublic;

var callOverReflection = new CallOverReflection();
Type currentType = callOverReflection.GetType();
MethodInfo methodInfo = currentType.GetMethod(
    name: "PrivateMethod",
    bindingFlags);
methodInfo.Invoke(callOverReflection, parameters: Array.Empty<object?>());
```

# IL, ILGenerator

Мы уже обсудили, что там внутри где-то всё работает с IL кодом. И вот мы дошли до момента, когда мы тоже можем пописать IL код!

Во время создания нашего типа мы объявили MethodBuilder и дошли до того, что можем получить ILGenerator этого метода и “реализовать” метод.

```
MethodAttributes methodAttributes =  
    MethodAttributes.Public  
    | MethodAttributes.Final  
    | MethodAttributes.Virtual;  
  
MethodBuilder methodBuilder = typeBuilder.DefineMethod(  
    nameof(IShopItem.GetPrice),  
    methodAttributes,  
    typeof(int), Type.EmptyTypes);  
  
ILGenerator ilGen = methodBuilder.GetILGenerator();
```

# OpCodes

```
static int F(int i)
{
    int value = i * 11;
    Console.Write(value);
    return value;
}
```

```
.method assembly hidebysig static
    int32 '<<Main>$>g__F|0_0' (
        int32 i
    ) cil managed
{
    // Method begins at RVA 0x2059
    // Code size 11 (0xb)
    .maxstack 8

    IL_0000: ldarg.0
    IL_0001: ldc.i4.s 11
    IL_0003: mul
    IL_0004: dup
    IL_0005: call void [System.Console]System.Console::Write
    IL_000a: ret
} // end of method 'Program$'::'<<Main>$>g__F|0_0'
```



# Как создать экземпляр

- Представим ситуацию, когда мы знаем какой-то тип и нам нужно создать его экземпляр. Есть три варианта:
  - `where T : new()`
  - `System.Activator.CreateInstance()`
  - `FormatterServices.GetUninitializedObject()`

# GetUninitializedObject

```
class ObjectWithCtor
{
    private readonly int _value;

    public static void Show()
    {
        new ObjectWithCtor().Write();
        var typeless :object = FormatterServices
            .GetUninitializedObject(typeof(ObjectWithCtor));
        var value = (ObjectWithCtor)typeless;
        value.Write();
    }

    public ObjectWithCtor() ⇒ _value = 10;
    public bool Ok() ⇒ _value == 10;
    public void Write() ⇒ Console.WriteLine(_value);
}
```

Демо нашего магазина

# Объявляем сигнатуры

```
public interface IShopItem
{
    public int GetPrice();
}

public class Item
{
    public string Name { get; set; }
    public int Price { get; set; }
}
```

# Создаём билдеры

```
1  AssemblyName assemblyName =  
2  |      Assembly.GetExecutingAssembly().GetName();  
3  AssemblyBuilder assemblyBuilder =  
4  |      AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);  
5  ModuleBuilder moduleBuilder =  
6  |      assemblyBuilder.DefineDynamicModule(nameof(UnderTheHoodOfIL));  
7
```

# Создаём типы

```
foreach (Item item in items)
{
    TypeBuilder typeBuilder = moduleBuilder.DefineType(
        item.Name,
        attr: TypeAttributes.Public,
        parent: null,
        interfaces: new[] {typeof(IShopItem)});

    typeBuilder.DefineDefaultConstructor(
        MethodAttributes.Private);
}
```

# Добавляем метод

```
MethodAttributes methodAttributes =  
    MethodAttributes.Public  
    | MethodAttributes.Final  
    | MethodAttributes.Virtual;  
  
MethodBuilder methodBuilder = typeBuilder.DefineMethod(  
    nameof(IShopItem.GetPrice),  
    methodAttributes,  
    typeof(int), Type.EmptyTypes);  
  
ILGenerator ilGen = methodBuilder.GetILGenerator();
```



# Добавляем имплементацию

```
ILGenerator ilGen = methodBuilder.GetILGenerator();  
  
ilGen.Emit(opcode: OpCodes.Ldc_I4, item.Price);  
ilGen.Emit(opcode: OpCodes.Ret);
```



# Создаём тип и экземпляр

```
Type? newType = typeBuilder.CreateType();  
object? createTypeValue = FormatterServices  
    .GetUninitializedObject(newType);  
shopItems.Add((IShopItem)createTypeValue);
```

# Получаем результат

```
List<Item> items = Database.GetItems();  
  
var stillShopBut = new StillShopBut();  
List<IShopItem> shopItems = stillShopBut.CreateShopItems(items);  
foreach (IShopItem shopItem in shopItems)  
{  
    WriteLine($"{shopItem.GetType()} : {shopItem.GetPrice()}");  
}
```

# Codegen

# Виды генерации

- Статик
  - Конвертация
  - T4
  - Анализаторы и кодфиксеры
- Компайл
  - Препоцессинг
  - Во время компиляции
- Рантайм
  - Работа с метамданными
  - Работа с байтами

# Конвертация между языками

- Плюсы: уже всё написано, просто нужно вставить и запустить
- Минусы: очень сложно, скорее всего нормальной тулы нет или она не поддерживает почти всё, что есть в языке. Кейсы перевода языка странные, а выхлоп очень плохой.

# T4

- Плюсы: очень наглядно, ещё до запуска неплохо так видно что будет генерироваться. Минимальный порог вхождения т.к. почти стринг билдер.
- Минусы: нужно самому отвечать за валидность сгенерированных данных, это просто стрингбилдер на максималках. Чтобы узнать о том, что забыл поставить скобочку в вызове метода, нужно запустить.

# T4

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ output extension=".cs" #>

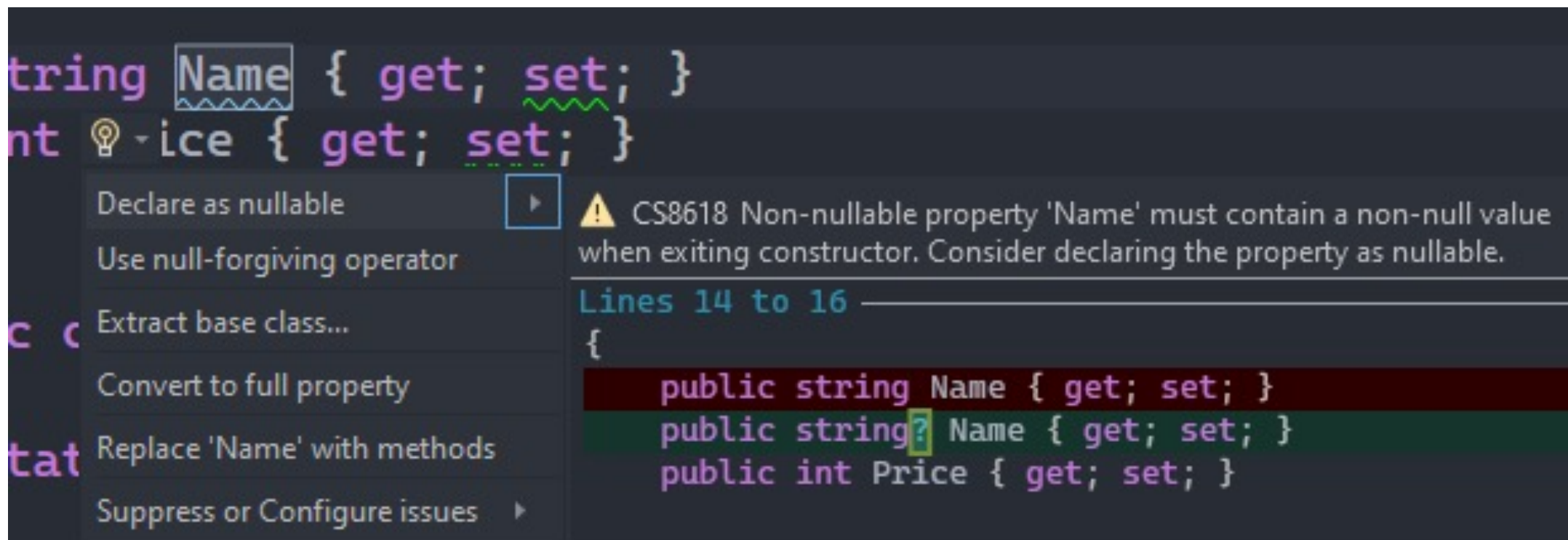
<# var properties = new string [] {"P1", "P2", "P3"}; #>
// This is generated code:
class MyGeneratedClass {
<# // This code runs in the text template:
    foreach (string propertyName in properties) { #>
        // Generated code:
        private int <#= propertyName #> = 0;
    } #>
}
```

T4

```
// This is generated code:  
class MyGeneratedClass {  
    // Generated code:  
    private int p1 = 0;  
    // Generated code:  
    private int p2 = 0;  
    // Generated code:  
    private int p3 = 0;  
}
```



# Анализаторы и кодфиксеры



# Conditional Compilation

```
public static string DatabaseFilePath
{
    get
    {
        var filename = "ToDoDatabase.db3";
        #if SILVERLIGHT
            // Windows Phone 8
            var path = filename;
        #else

        #if __ANDROID__
            string libraryPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
        #else
        #if __IOS__
            // we need to put in /Library/ on iOS5.1 to meet Apple's iCloud terms
            // (they don't want non-user-generated data in Documents)
            string documentsPath = Environment.GetFolderPath (Environment.SpecialFolder.Personal); // Documents
            string libraryPath = Path.Combine (documentsPath, "..", "Library");
        #else
            // UWP
            string libraryPath = Windows.Storage.ApplicationData.Current.LocalFolder.Path;
        #endif
        #endif
        var path = Path.Combine(libraryPath, filename);
    #endif
    return path;
    }
}
```

# Атрибуты

Атрибуты - это функционал языка, который позволяет расширять метайнформацию сборки, типа, метода

- Указание методов, которые являются тестами
- Указание методов для тестирования
- Сериализация

# Работа с атрибутами. Постановка задачи

Есть енам, который описывает режимы работы. Есть реализации интерфейсов, которые соответствуют каждому значению енама. Хочется получить такое решение, где добавление нового алгоритма не будет задевать изменения более чем одного места.

# Объявление типов

```
public interface IAlgorithm {}

public class AnAlgorithm : IAlgorithm {}
public class BnAlgorithm : IAlgorithm {}

public enum Implementations
{
    A,
    B,
}
```

# Создаём атрибут

```
public class ImplementationAttribute : Attribute
{
    public Type Type { get; }

    public ImplementationAttribute(Type type)
    {
        Type = type;
    }
}

public enum Implementations
{
    [Implementation(typeof(AnAlgorithm))]
    A,

    [Implementation(typeof(BnAlgorithm))]
    B,
}
```

# Получение результата

```
public static class ImplementationExtensions
{
    public static IAlgorithm GetImplementation(this Implementations value)
    {
        Attribute customAttribute = value // Implementations
            .GetType() // Type
            .GetCustomAttribute(typeof(ImplementationAttribute));

        var attribute = (ImplementationAttribute)customAttribute;
        IAlgorithm algorithm = (IAlgorithm)Activator.CreateInstance(attribute.Type);
        return algorithm;
    }
}
```

# Генерация в помощью Fody

- Более реальное применение такого подхода - Fody
- Система плагинов для расширения языка по средствам генерации IL
- Пример использования - атрибут ToString, который генерирует