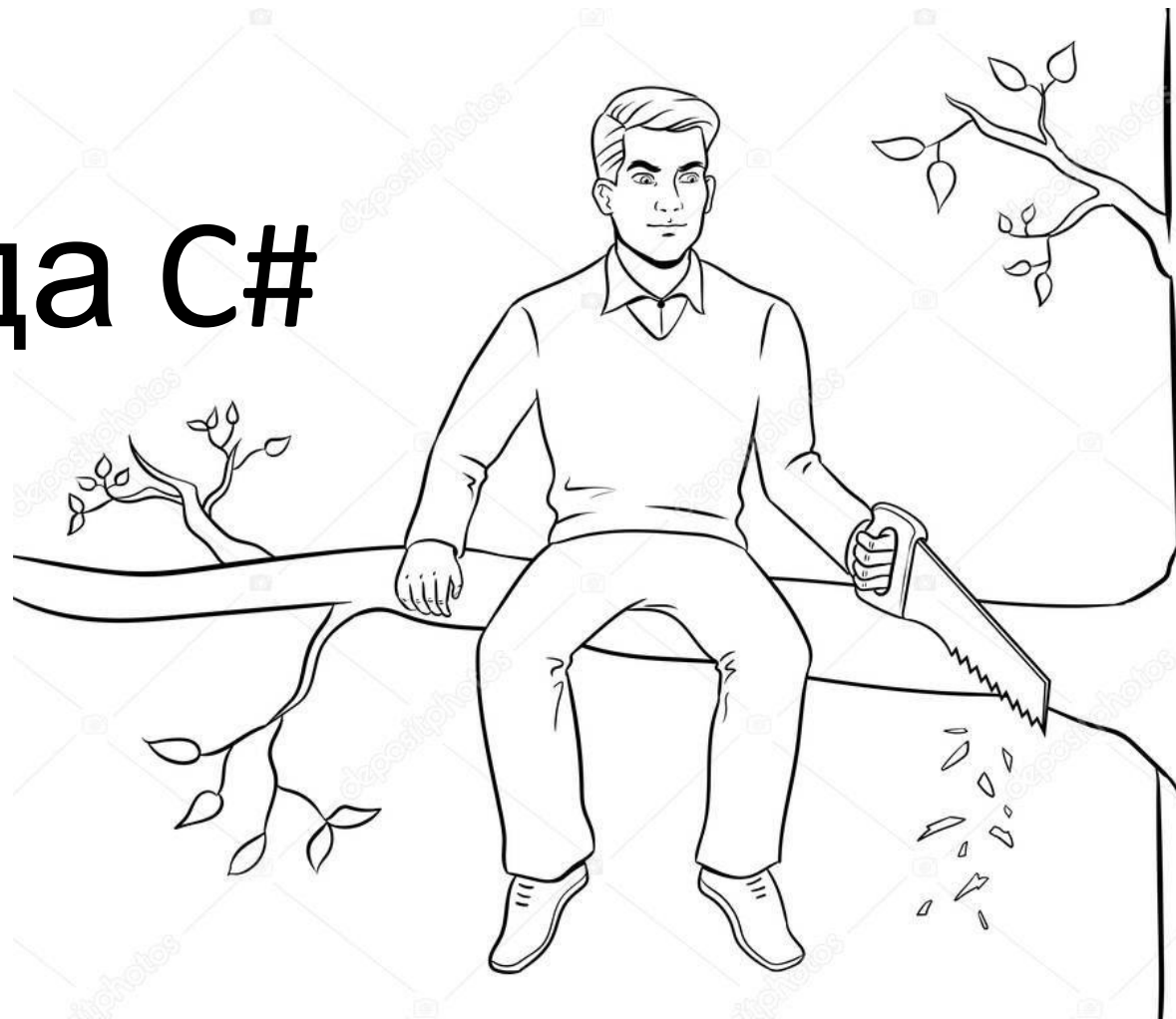


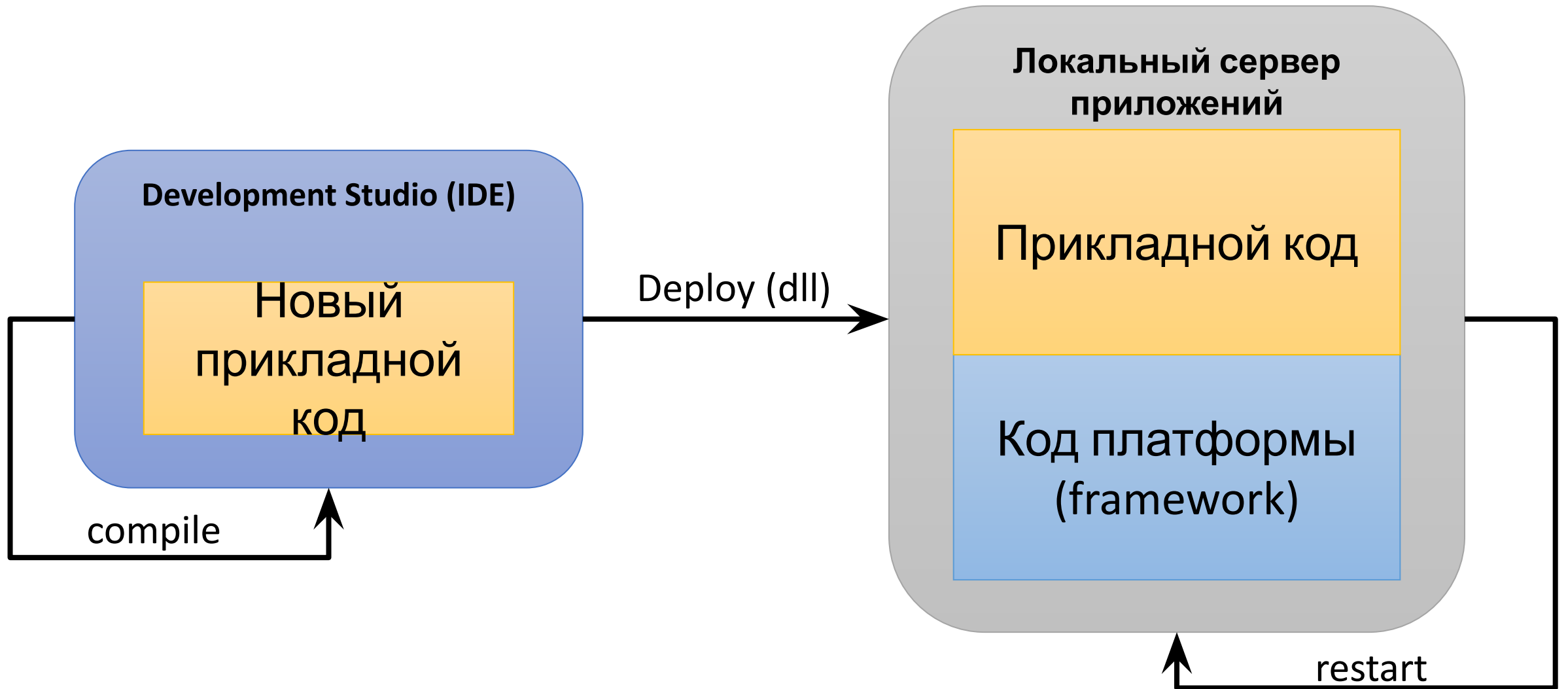


# HotReload кода C#

Другим кодом C#

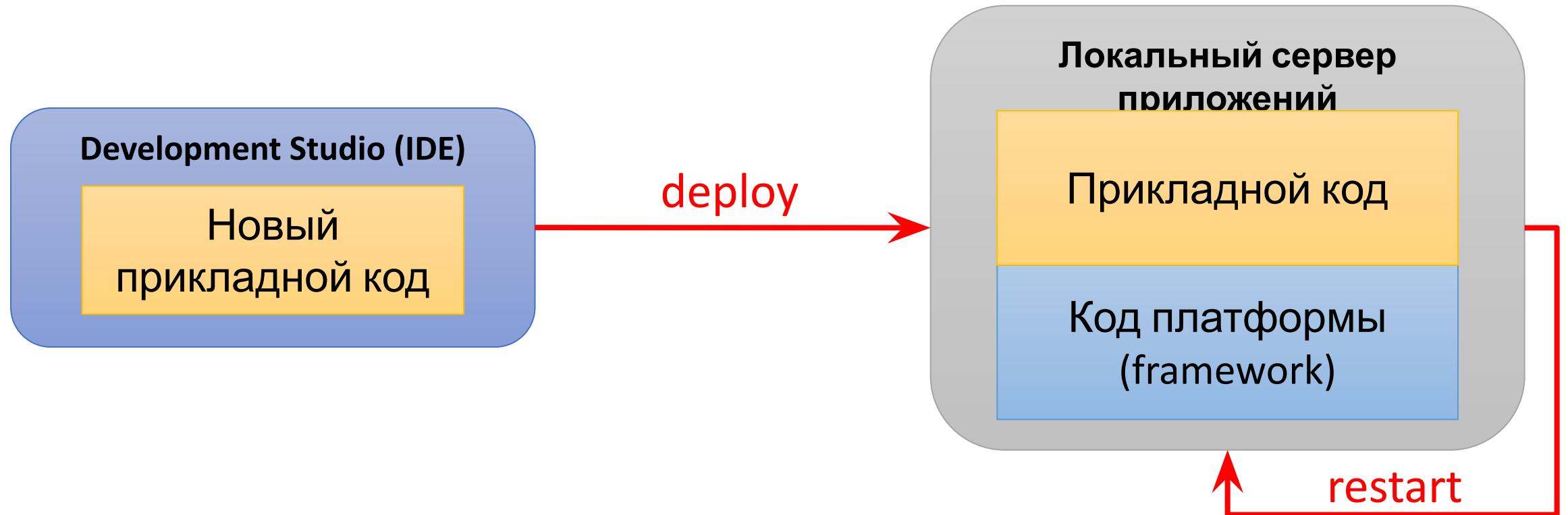


# Прикладная разработка DirectumRX



# Зачем нам HotReload

- Быстрая локальная отладка прикладного кода.
- Обычный deploy хорош и надёжен для прода, но медленный из-за перезапуска сервера.

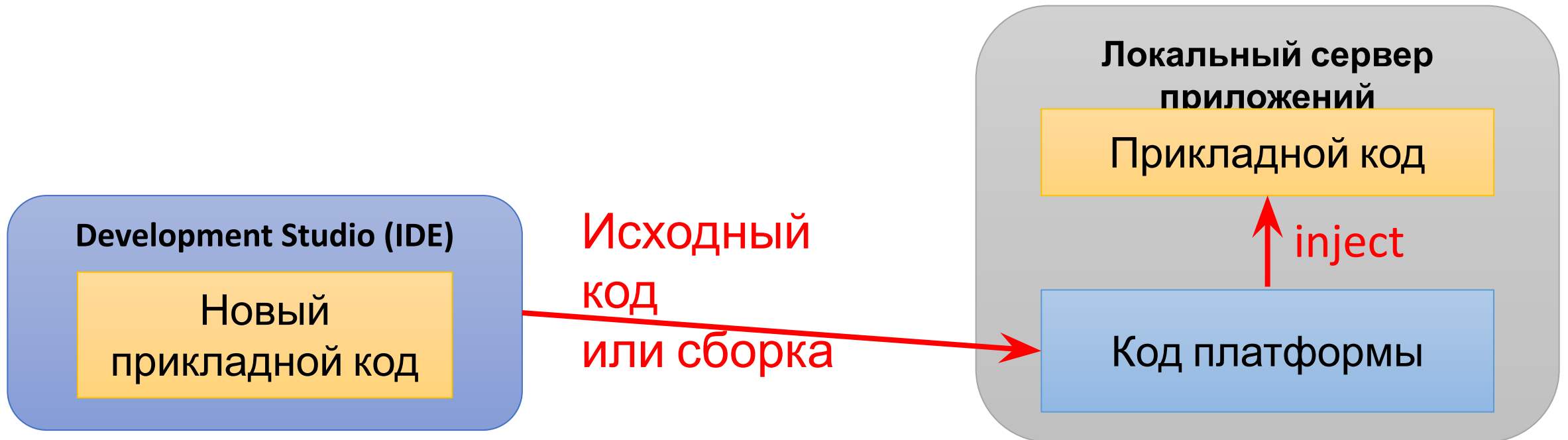


# Дополнительные ограничения и требования

- Не спасёт, если поменялась структура БД.
- Надо, чтобы при Hot Reload могла работать отладка в Dev Studio.
- Хорошо бы, чтобы можно использовать не только для серверного кода, но и для клиента (который уже запущен).

# Основная идея

- Избавиться от перезапуска сервера.
- Делаем на стороне сервера точку (контроллер WebApi, например), которая встроит поданный код в работающее приложение.



# Решения в лоб

- Возня с reflection – LoadAssembly (в т. ч. Shadow Copy Assemblies).
- Managed Extensibility Framework (MEF); в том числе VS-MEF.
- Mono.Cecil.



# Решения в лоб. Проблемы

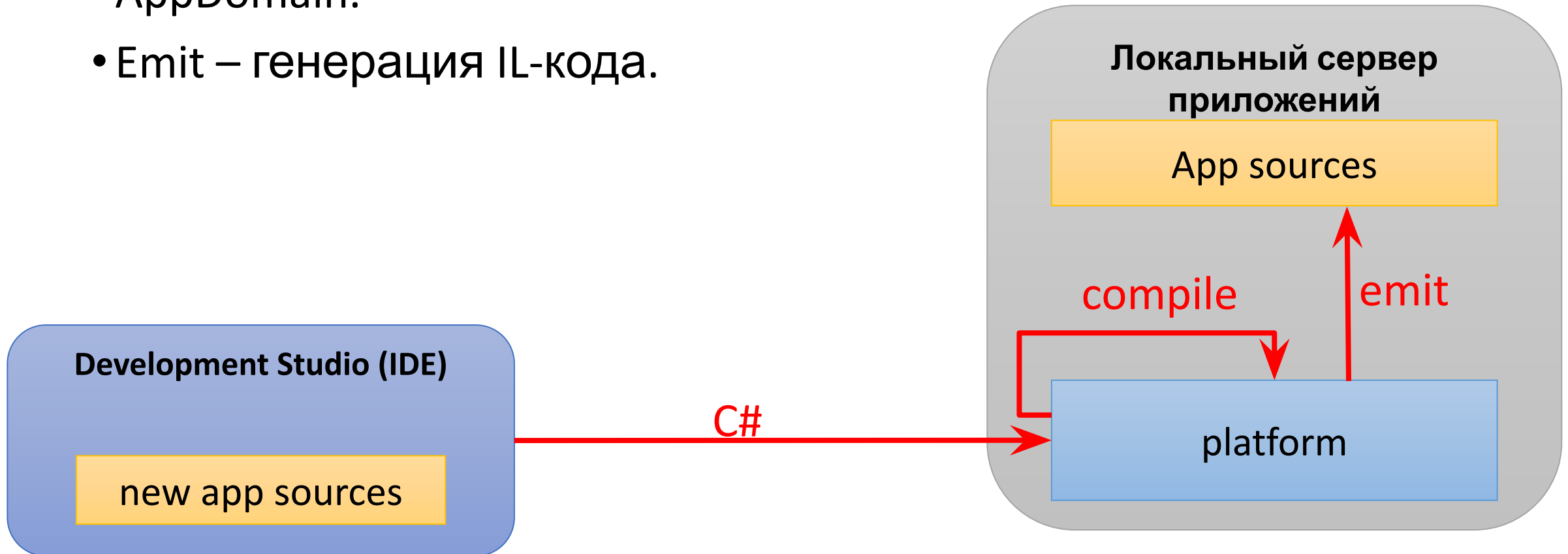
- Никак не повлияем на уже существующие в памяти объекты,
- Новые объекты, создаваемые через *new* в прикладном коде будут ссылаться на старые типы.

## Чтобы это работало нужно:

- писать свой загрузчик типов. Если где-то в прикладном коде какая-то прикладная сущность создаётся через *new*, то придётся модифицировать прикладной код (это плохо);
- в случае с MEF по коду надо либо раскидывать специальные атрибуты, либо наследоваться от нужных интерфейсов.

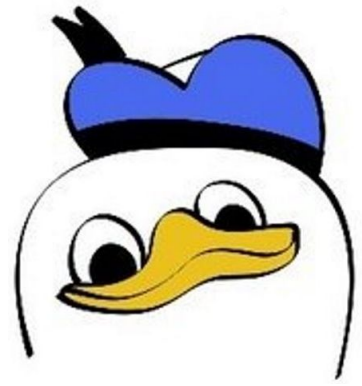
# Компилять на сервере и Emit (v1)

- CodeDom – компиляция c# в Assembly сразу в память в AppDomain.
- Emit – генерация IL-кода.





# Компилять на сервере и Emit



Проблемы:

- Emit работает только с `DynamicAssembly` (соответственно – `DynamicMethod`). А в результате работы `CodeDom` и компиляции мы получаем, по сути, обычную `Assembly`.
- Чтобы динамический класс отправить в работу, нужно вызвать у него `CreateType()`. Это блокирует дальнейшие его модификации

# Edit and Continue

- Встроенный в Visual Studio хитрый механизм, генерирующий некоторые дельты.
- Общедоступного API нет.
- Даже в самой VS механизм не работает в ряде случаев.

# Method inject v1

## Замена указателя на метод:

```
MethodInfo methodToReplace = ... ;
```

```
MethodInfo methodToInject = ... ;
```

```
unsafe
```

```
{
```

```
    long* inject = (long*)methodToInject.MethodHandle.Value.ToPointer();
```

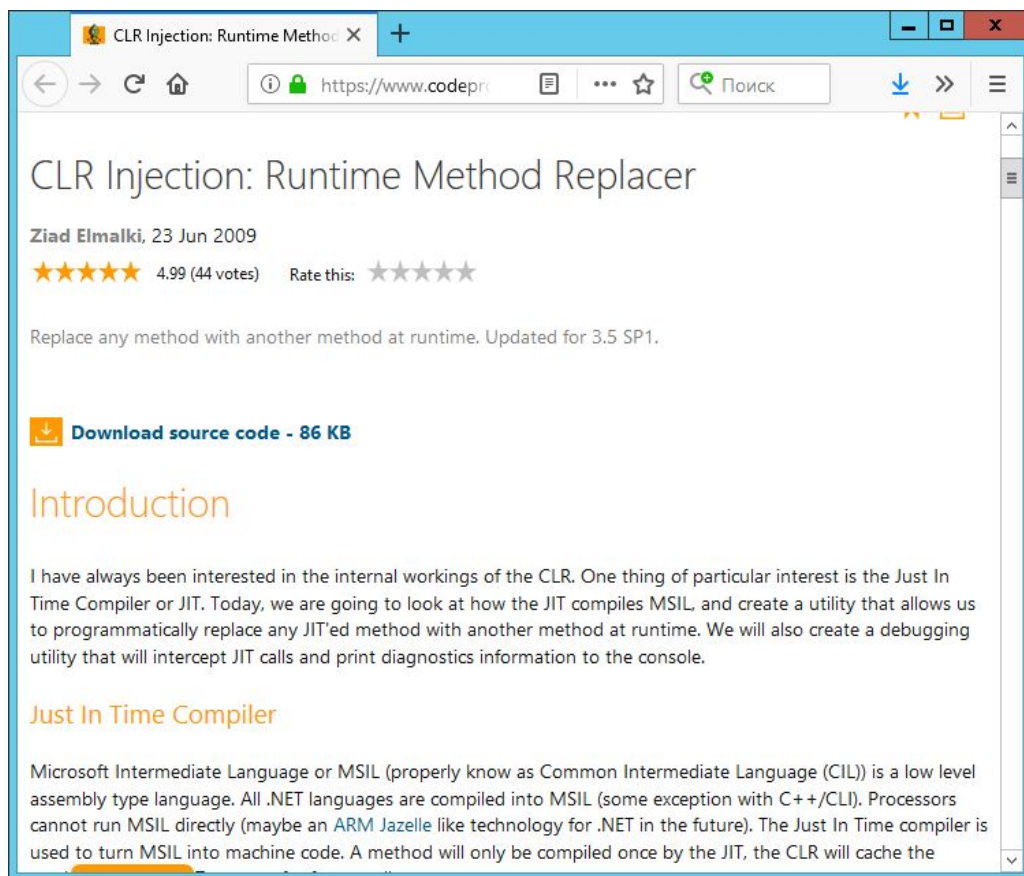
```
    long* target =
```

```
(long*)methodToReplace.MethodHandle.Value.ToPointer();
```

```
    *tar = *inj;
```

```
}
```

В реальности чуток сложнее, потому что надо учесть x86/x64, Debug/Release.



CLR Injection: Runtime Method Replacer

Ziad Elmalki, 23 Jun 2009

★★★★★ 4.99 (44 votes) Rate this: ★★★★★

Replace any method with another method at runtime. Updated for 3.5 SP1.

[Download source code - 86 KB](#)

### Introduction

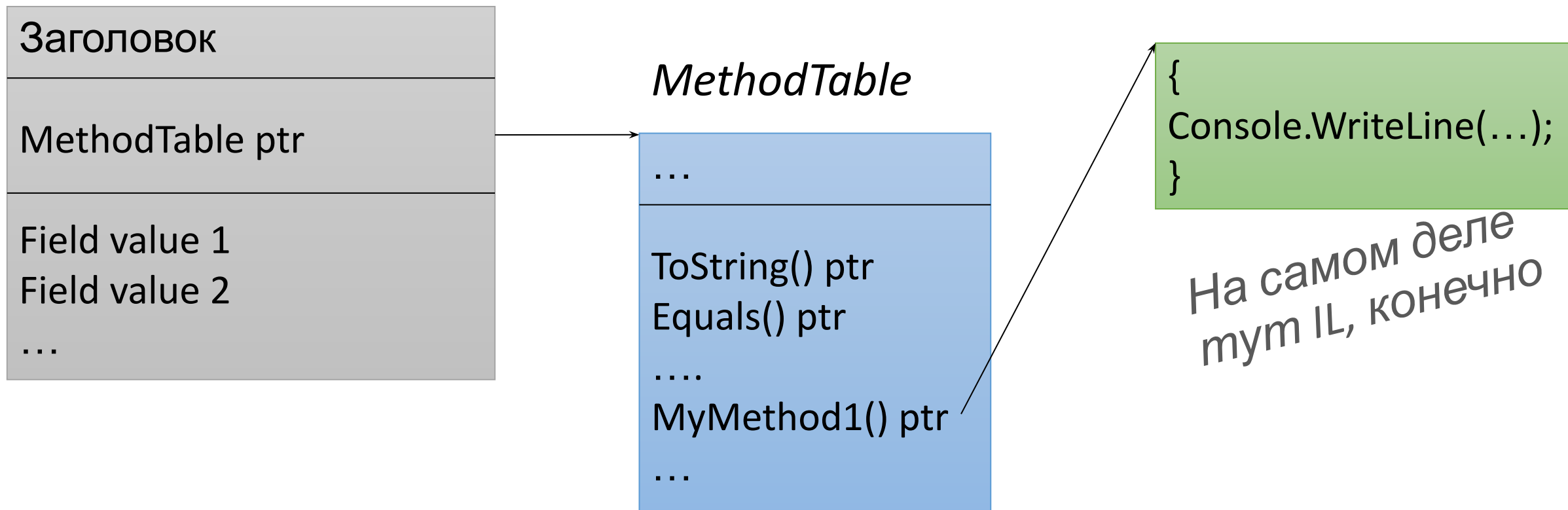
I have always been interested in the internal workings of the CLR. One thing of particular interest is the Just In Time Compiler or JIT. Today, we are going to look at how the JIT compiles MSIL, and create a utility that allows us to programmatically replace any JIT'ed method with another method at runtime. We will also create a debugging utility that will intercept JIT calls and print diagnostics information to the console.

### Just In Time Compiler

Microsoft Intermediate Language or MSIL (properly know as Common Intermediate Language (CIL)) is a low level assembly type language. All .NET languages are compiled into MSIL (some exception with C++/CLI). Processors cannot run MSIL directly (maybe an ARM Jazelle like technology for .NET in the future). The Just In Time compiler is used to turn MSIL into machine code. A method will only be compiled once by the JIT, the CLR will cache the

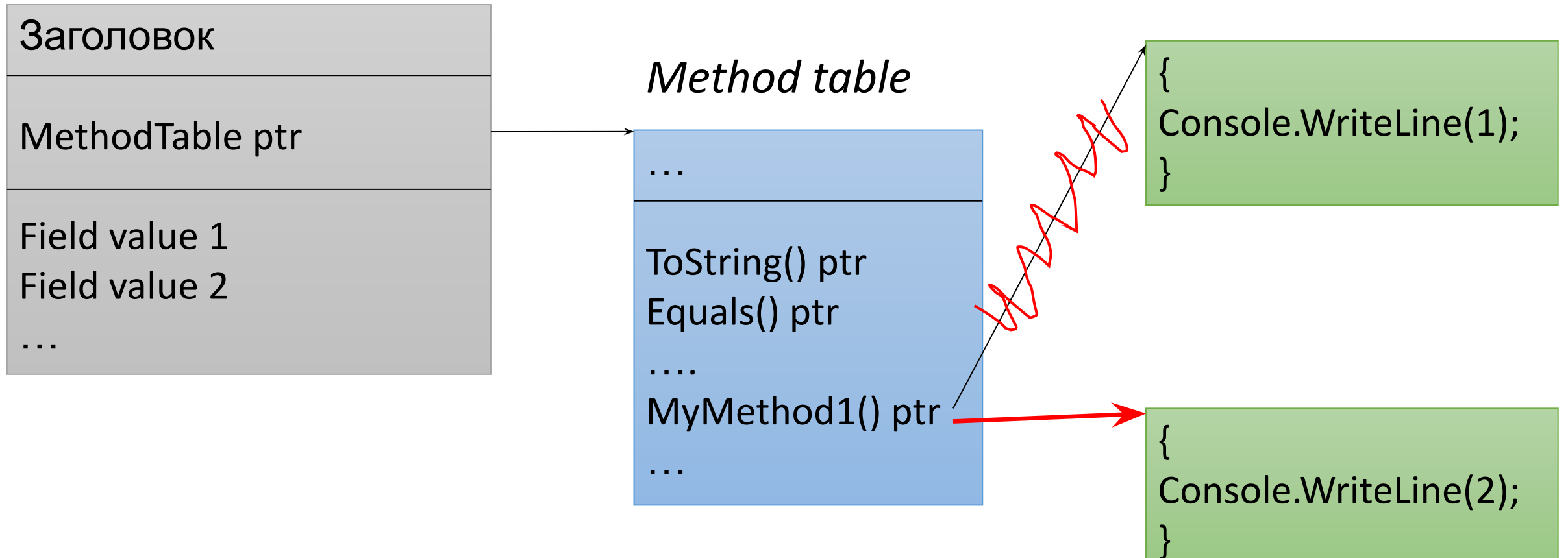
# Как хранятся описания классов в .NET

*ObjectInstance*

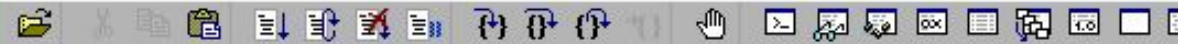


# Суть Method Inject v1

*Object instance*







Command

```
0:000> !Name2EE TestClasses.dll TestClasses.Target
Module:      00007ffc1e745bf0
Assembly:    TestClasses.dll
Token:       0000000020000003
MethodTable: 00007ffc1e7462f0
EEClass:     00007ffc1e8d46f0
Name:        TestClasses.Target
0:000> !DumpMT -md 00007ffc1e7462f0
EEClass:     00007ffc1e8d46f0
Module:      00007ffc1e745bf0
Name:        TestClasses.Target
mdToken:     0000000020000003
File:        D:\work\protos\Hot reload\ConsoleApplication2\bin\Release\T
BaseSize:    0x18
ComponentSize: 0x0
Slots in VTable: 8
Number of IFaces in IFaceMap: 0
```

-----

MethodDesc Table			
Entry	MethodDesc	JIT Name	
00007ffc7c5dc000	00007ffc7c137fb8	PreJIT System.Object.ToString()	
00007ffc7c6440f0	00007ffc7c137fc0	PreJIT System.Object.Equals(System.Object)	
00007ffc7c6b4490	00007ffc7c137fe8	PreJIT System.Object.GetHashCode()	
00007ffc7c60f390	00007ffc7c138000	PreJIT System.Object.Finalize()	
00007ffc1e850880	00007ffc1e746298	JIT TestClasses.Target.Test()	
00007ffc1e8508d0	00007ffc1e7462a8	JIT TestClasses.Target.TargetMethod()	
00007ffc1e8500b0	00007ffc1e7462b8	NONE TestClasses.Target.get_Value()	
00007ffc1e850820	00007ffc1e7462c8	JIT TestClasses.Target..ctor(System.Str	



Command

```
00007ffc1e850880 00007ffc1e746298 JIT TestClasses.Target.Test()
00007ffc1e8508d0 00007ffc1e7462a8 JIT TestClasses.Target.TargetMethod()
00007ffc1e8500b0 00007ffc1e7462b8 NONE TestClasses.Target.get_Value()
00007ffc1e850820 00007ffc1e7462c8 JIT TestClasses.Target..ctor(System.String)
0:000> g
Breakpoint 0 hit
00007ffc`1e85093e 90 nop
0:000> !DumpMT -md 00007ffc1e7462f0
EEClass:     00007ffc1e8d46f0
Module:      00007ffc1e745bf0
Name:        TestClasses.Target
mdToken:     0000000020000003
File:        D:\work\protos\Hot reload\ConsoleApplication2\bin\Release\TestClass
BaseSize:    0x18
ComponentSize: 0x0
Slots in VTable: 8
Number of IFaces in IFaceMap: 0
```

-----

MethodDesc Table			
Entry	MethodDesc	JIT Name	
00007ffc7c5dc000	00007ffc7c137fb8	PreJIT System.Object.ToString()	
00007ffc7c6440f0	00007ffc7c137fc0	PreJIT System.Object.Equals(System.Object)	
00007ffc7c6b4490	00007ffc7c137fe8	PreJIT System.Object.GetHashCode()	
00007ffc7c60f390	00007ffc7c138000	PreJIT System.Object.Finalize()	
00007ffc1e850880	00007ffc1e746298	JIT TestClasses.Target.Test()	
00007ffc1e850b50	00007ffc1e746500	JIT TestClasses.Injection.InjectionMethod()	
00007ffc1e8500b0	00007ffc1e7462b8	NONE TestClasses.Target.get_Value()	
00007ffc1e850820	00007ffc1e7462c8	JIT TestClasses.Target..ctor(System.String)	

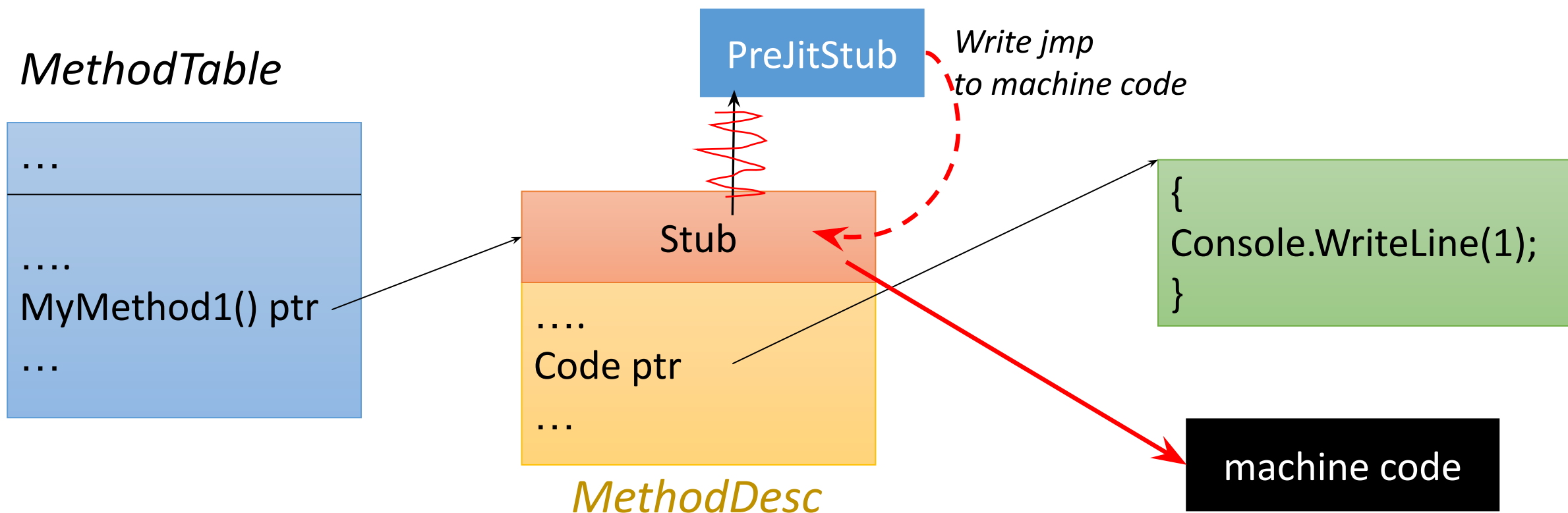
# Method Inject v1. Что нам надо сделать ещё

- Сравнить прикладные исходники и найти изменившиеся методы.
- Передать исходные тексты методов (+ доп. инфа) на сервер
- Создать новый класс, засунуть в него методы и скомпилировать в память
- *(Перед компиляцией надо заменить все обращения к this на обращения через Reflection)*
- Найти старый метод и сделать MethodInject на новый



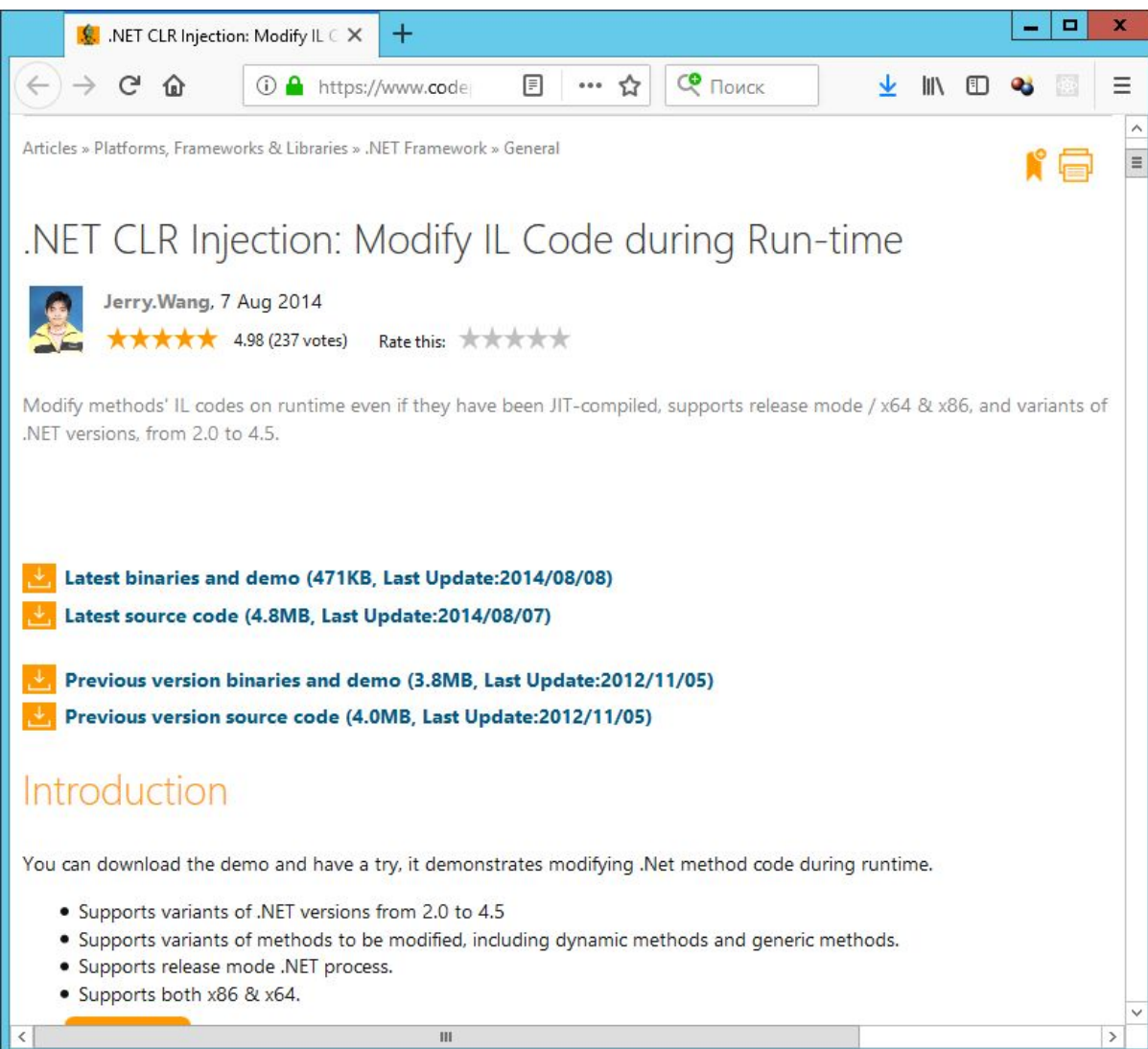
# Method Inject v1. ПРОБЛЕМА!

Прекрасно работает, пока мы не вызываем метод.  
После первого вызова срабатывает JIT и вся магия пропадает.





# Method Inject v2



.NET CLR Injection: Modify IL Code during Run-time

Jerry.Wang, 7 Aug 2014  
★★★★★ 4.98 (237 votes) Rate this: ★★★★★

Modify methods' IL codes on runtime even if they have been JIT-compiled, supports release mode / x64 & x86, and variants of .NET versions, from 2.0 to 4.5.

- Latest binaries and demo (471KB, Last Update:2014/08/08)
- Latest source code (4.8MB, Last Update:2014/08/07)
- Previous version binaries and demo (3.8MB, Last Update:2012/11/05)
- Previous version source code (4.0MB, Last Update:2012/11/05)

## Introduction

You can download the demo and have a try, it demonstrates modifying .Net method code during runtime.

- Supports variants of .NET versions from 2.0 to 4.5
- Supports variants of methods to be modified, including dynamic methods and generic methods.
- Supports release mode .NET process.
- Supports both x86 & x64.

## Использование EasyHook

### Основная идея:

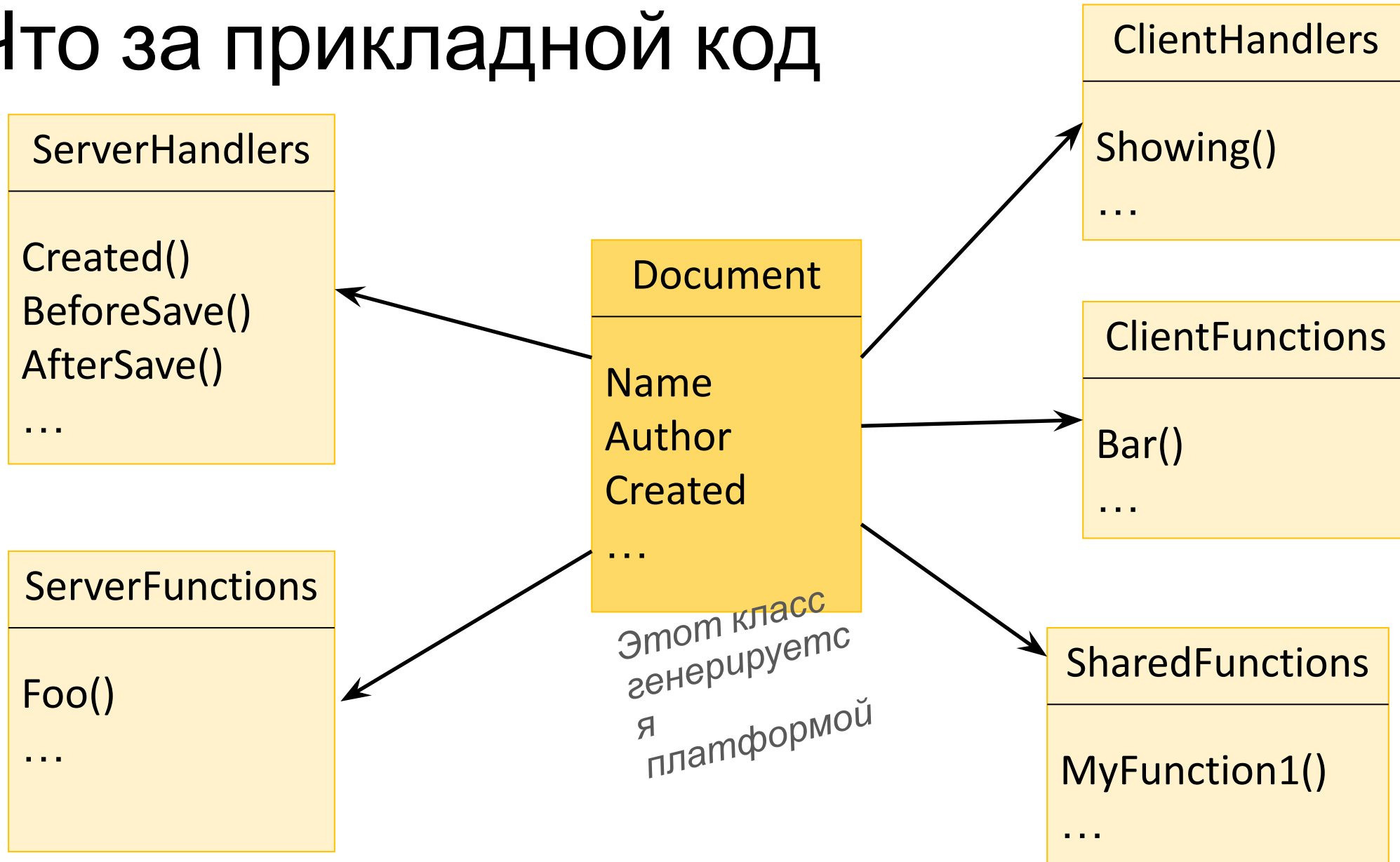
- Переопределение `compileMethod`,
- Сброс состояния Jitted с помощью `MethodDesc.Reset()`

# Method Inject v2. Проблемки

- Нормально заработало только на .NET Framework 3.5.
- Уж очень рискованно такое применять в продакшене.
- Поддерживать такое от версии к версии .NET – хз, как, потому что влезли конкретно внутрь кухни CLR

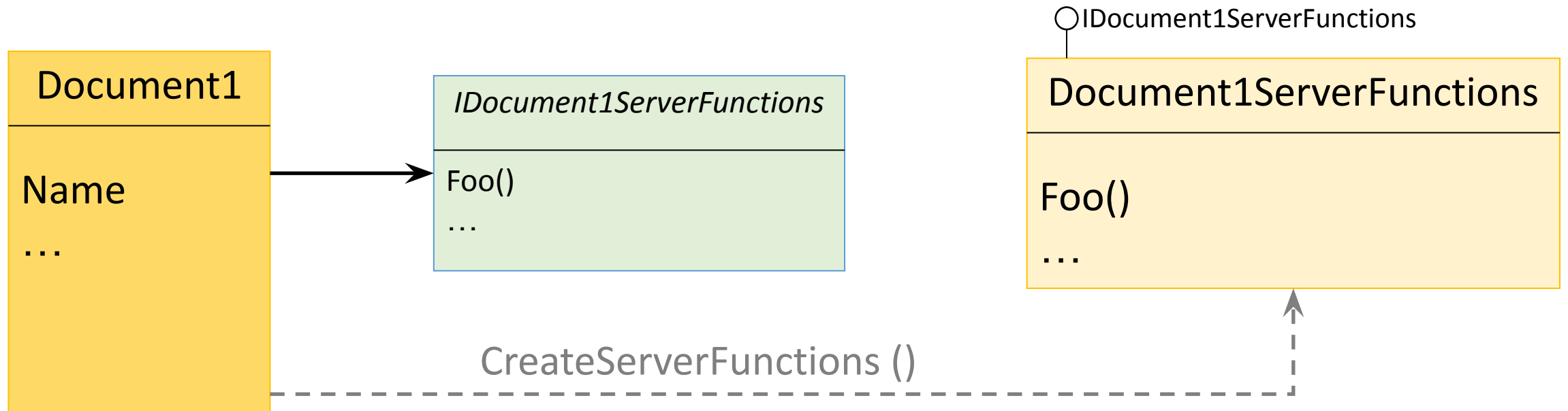


# Что за прикладной код



# Class swap

- Платформенный код создаёт объекты, содержащие прикладной код.
- Зачем менять изворачиваться с методами, если можно просто подсунуть объект



# Class swar. Работающий прототип

The screenshot shows the Visual Studio IDE with the following elements:

- Toolbar:** Includes icons for saving, undo, redo, and running. The status bar shows "Изменения (4)".
- File Explorer:** Shows the project structure with files like "Databook1ServerFunctions.cs", "Databook1Handlers.cs", and "Databook1 [Справочник]\*".
- Code Editor:** Displays the following C# code:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using Sungero.Core;
5 using Sungero.CoreEntities;
6 using Sungero.Module1.Databook1;
7
8 namespace Sungero.Module1
9 {
10     partial class Databook1ServerHandlers
11     {
12         public override void BeforeSave(Sungero.Domain.BeforeSaveEventArgs e)
13         {
14             string result = "привет";
15             e.AddError(result);
16         }
17     }
18 }
```
- Properties Window:** Shows a property named "Имя" with a text input field.
- History Window:** Shows a message "привет." with a red minus icon.
- Bottom Panel:** Shows "ОБЩИЕ ПАПКИ" and "ДЕЛОПРОИЗВОДСТВО".

# Мораль

## Исследования – офигенная тема

- Правильно, что не стали бросаться делать первое попавшееся.
- Нужно смотреть шире. Упёрлись в подмену кода на низком уровне.
- Документирование исследования – вообще тема.
- Фича не забыта и может быть когда-нибудь...

