



Привет!

# Я Георгий Круглов

Старший разработчик информационных систем  
группы разработки ТМЦ Обработка проблемных  
товаров

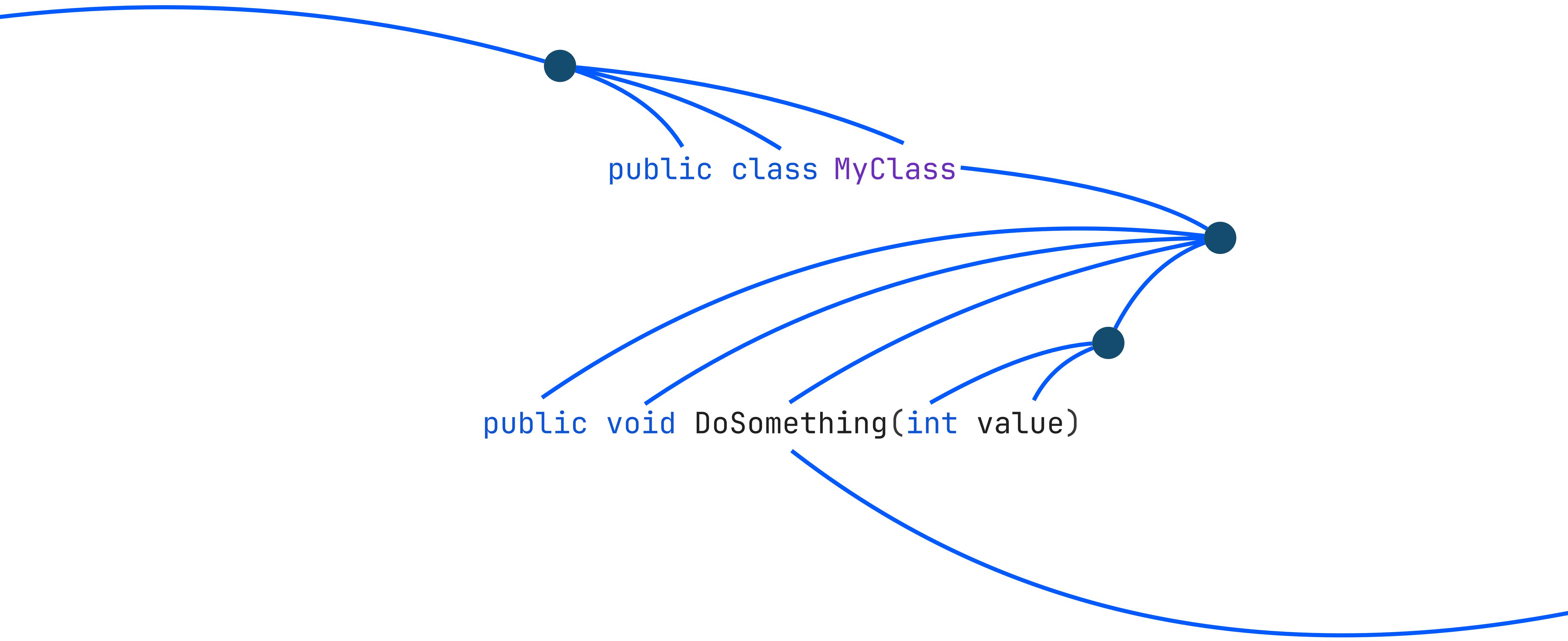
11:25 ✓



# Roslyn и плагины для компилятора C#

- API Roslyn
- Analyzer, CodeFixProvider
- SourceGenerator
- Публикация, отладка и тестирование

# Абстрактные синтаксические деревья





## Основы: Syntax API

# Узлы синтаксического дерева

01

**SyntaxTrivia**

- Комментарии
- Переносы строк
- Директивы препроцессора

02

**SyntaxToken**

03

**SyntaxNode**

# Узлы синтаксического дерева

01

**SyntaxTrivia**



- Ключевые слова
- Идентификаторы
- Спецсимволы

02

**SyntaxToken**

03

**SyntaxNode**



# Узлы синтаксического дерева

01

SyntaxTrivia



- Языковые структуры

- Неймспейсы

- Классы

- Методы

- Циклы

- ...

- Могут содержать другие SyntaxNode

- Могут содержать SyntaxToken'ы

- К ним могут быть привязаны SyntaxTrivia

02

SyntaxToken



03

SyntaxNode



# Создание узлов синтаксического дерева

```
SyntaxFactory.ClassDeclaration();
SyntaxFactory.MethodDeclaration();
SyntaxFactory.NamespaceDeclaration();
```

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));
var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));
var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Модификация синтаксических узлов

Код

```
public void MyMethod(int a)
```

With...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .WithParameterList(ParameterList(SingletonSeparatedList(parameter)));
```

Add...

```
var parameter = Parameter(Identifier("a")).WithType(PredefinedType(Token(SyntaxKind.IntKeyword)));

var method = MethodDeclaration(PredefinedType(Token(SyntaxKind.VoidKeyword)), Identifier("MyMethod"))
    .AddParameterListParameters(parameter);
```

# Roslyn Quoter

<https://roslynquoter.azurewebsites.net>

```
1 public class MyClass
2 {
3     public void MyMethod(int a)
4 }
```

Parse as: Regular-File ▾

- Open parenthesis on a new line
- Closing parenthesis on a new line
- Preserve original whitespace
- Keep redundant API calls
- Do not require 'using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;'
- Ready-To-Run (with compilation and diagnostics)

[Get Roslyn API calls to generate this code!](#)

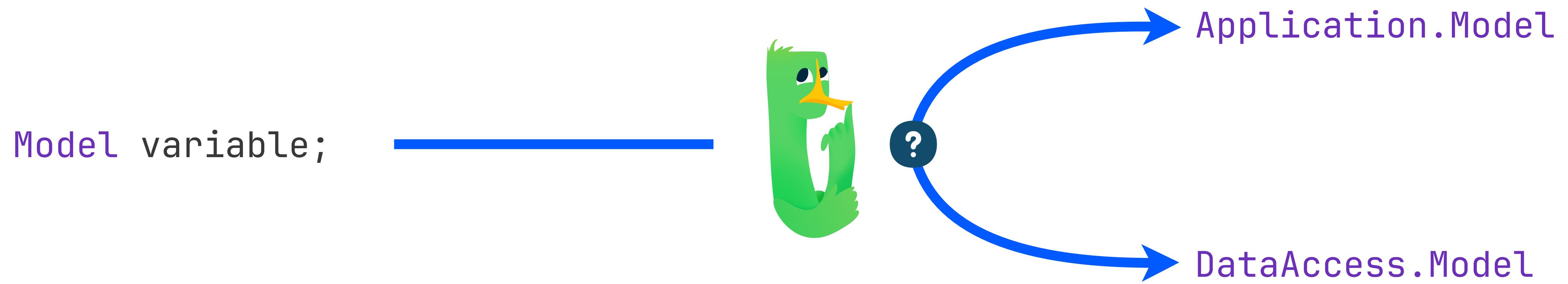
[Get Roslyn API calls to generate this code as LINQPad file!](#)

```
1 CompilationUnit()
2 .WithMembers(
3     SingletonList<MemberDeclarationSyntax>(
4         ClassDeclaration("MyClass")
5             .WithModifiers(
6                 TokenList(
7                     Token(SyntaxKind.PublicKeyword)))
8             .WithMembers(
9                 SingletonList<MemberDeclarationSyntax>(
10                    MethodDeclaration(
11                        PredefinedType(
12                            Token(SyntaxKind.VoidKeyword)),
13                            Identifier("MyMethod"))
14                    .WithModifiers(
15                        TokenList(
16                            Token(SyntaxKind.PublicKeyword)))
17                    .WithParameterList(
18                        ParameterList(
19                            SingletonSeparatedList<ParameterSyntax>(
20                                Parameter(
21                                    Identifier("a")))
22                            .WithType(
23                                PredefinedType(
24                                    Token(SyntaxKind.IntKeyword))))))))))))
25 .NormalizeWhitespace()
```



## Основы: Semantic API

# Проблема анализа синтаксиса



# Абстракции семантического API

## 01 Символы

- Представлены интерфейсом ISymbol
- Описывают форму программы
  - Классы
  - Свойства
  - Методы
  - ...
- Являются метаданными о будущей сборке

## 02 Операции

# Абстракции семантического API

01

Символы

...

02

Операции

- Представлены интерфейсом IOperation
- Описывают логику программы
  - Вызовы методов
  - Циклы
  - Объявления переменных
  - ...

# Различия Syntax и Semantic API

## Синтаксическое API

- То, что написано
- Использует классы для своих сущностей
- Сущности можно создать пользовательским кодом

## Семантическое API

- То, что это значит
- Использует интерфейсы для своих сущностей
- Сущности создаются самим Roslyn

# Получение дочерних объектов: символы

```
public interface INamedTypeSymbol : Symbol
{
    ImmutableArray<ISymbol> GetMembers();
}
```

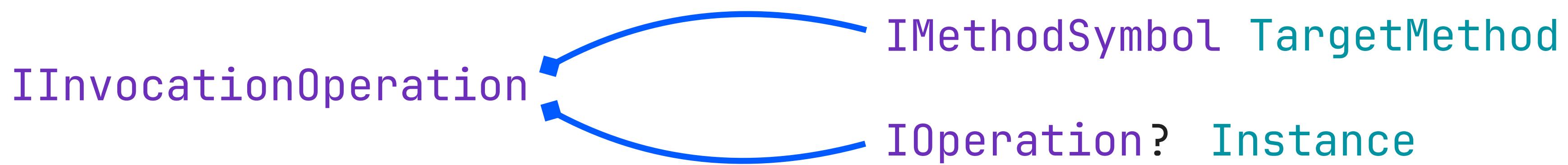
# Получение дочерних объектов: операции

```
public interface IOperation
{
    IEnumerable<IOperation> Descendents();
    OperationList ChildOperations { get; }
}
```

# Композиция семантических объектов

Операции могут хранить символы, но символы хранят операции

Операции по символам нужно получать через `SyntaxNode` и `SemanticModel`



# Compilation и SemanticModel

01 Compilation

- Информация для создания C# сборки
- Зависимости
- Исходные файлы
- Конфигурация компилятора
- ...
- Получение символов типов по их названию

02 SemanticModel

# Compilation и SemanticModel

## 01 Compilation

...

- Получение символов и операций по SyntaxNode
- DataFlow анализ
- ControlFlow анализ

## 02 SemanticModel

```
ISymbol? symbol = semanticModel.GetDeclaredSymbol(node);

if (symbol is INamedTypeSymbol)
{
    Console.WriteLine("It's a type!");
}
```



## Подводя итоги...

- Синтаксис представляет сам код, семантика – его значение
- Мы можем анализировать и создавать синтаксис, семантику – только анализировать
- С помощью SemanticModel можно получить семантику по синтаксису
- Используйте Semantic API, когда вам нужно иметь больше контекста анализируемого кода

# 03



## Анализ: Анализаторы

# Анализаторы

Флоу выполнения анализаторов

Roslyn

Ваш код



# Определение анализатора

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class MyAnalyzer : DiagnosticAnalyzer
{
    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get; }

    public override void Initialize(AnalysisContext context) { }
}
```

# Свойство SupportedDiagnostics

```
public static readonly DiagnosticDescriptor Descriptor = new DiagnosticDescriptor(  
    DiagnosticId,  
    Title,  
    MessageTemplate,  
    category: "Design",  
    DiagnosticSeverity.Error,  
    isEnabledByDefault: true);  
  
public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get; } = [Descriptor];
```

# Метод Initialize

Настройка работы анализатора

```
public override void Initialize(AnalysisContext context)
{
    context.EnableConcurrentExecution();
    context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
}
```

# Метод Initialize

Регистрация действий анализа

```
public override void Initialize(AnalysisContext context)
{
    ...

    context.RegisterSyntaxNodeAction>AnalyzeSyntaxNode, SyntaxKind.ClassDeclaration);
    context.RegisterSymbolAction>AnalyzeSymbol, SymbolKind.NamedType);
    context.RegisterOperationAction>AnalyzeOperation, OperationKind.MethodBody);
}
```

# Метод Initialize

Сигнатуры действий анализа

```
private void AnalyzeOperation(OperationAnalysisContext context) { }

private void AnalyzeSymbol(SymbolAnalysisContext context) { }

private void AnalyzeSyntaxNode(SyntaxNodeAnalysisContext context) { }
```

# Метод Initialize

Реализация действия анализа

```
private void AnalyzeSyntaxNode(SyntaxNodeAnalysisContext context)
{
    var node = (ClassDeclarationSyntax)context.Node;

    if (char.ToUpper(node.Identifier.Text[0]))
        return;

    var diagnostic = Diagnostic.Create(Descriptor, node.Identifier.GetLocation());
    context.ReportDiagnostic(diagnostic);
}
```

# Метод Initialize

Реализация действия анализа

```
private void AnalyzeSyntaxNode(SyntaxNodeAnalysisContext context)
{
    var node = (ClassDeclarationSyntax)context.Node;

    if (char.ToUpper(node.Identifier.Text[0]))
        return;

    var diagnostic = Diagnostic.Create(Descriptor, node.Identifier.GetLocation());
    context.ReportDiagnostic(diagnostic);
}
```

# Метод Initialize

Реализация действия анализа

```
private void AnalyzeSyntaxNode(SyntaxNodeAnalysisContext context)
{
    var node = (ClassDeclarationSyntax)context.Node;

    if (char.ToUpper(node.Identifier.Text[0]))
        return;

    var diagnostic = Diagnostic.Create(Descriptor, node.Identifier.GetLocation());
    context.ReportDiagnostic(diagnostic);
}
```

# Метод Initialize

Реализация действия анализа

```
private void AnalyzeSyntaxNode(SyntaxNodeAnalysisContext context)
{
    var node = (ClassDeclarationSyntax)context.Node;

    if (char.ToUpper(node.Identifier.Text[0]))
        return;

    var diagnostic = Diagnostic.Create(Descriptor, node.Identifier.GetLocation());
    context.ReportDiagnostic(diagnostic);
}
```

# AnalyzerReleases

Стоит вести учёт активных и неактивных анализаторов

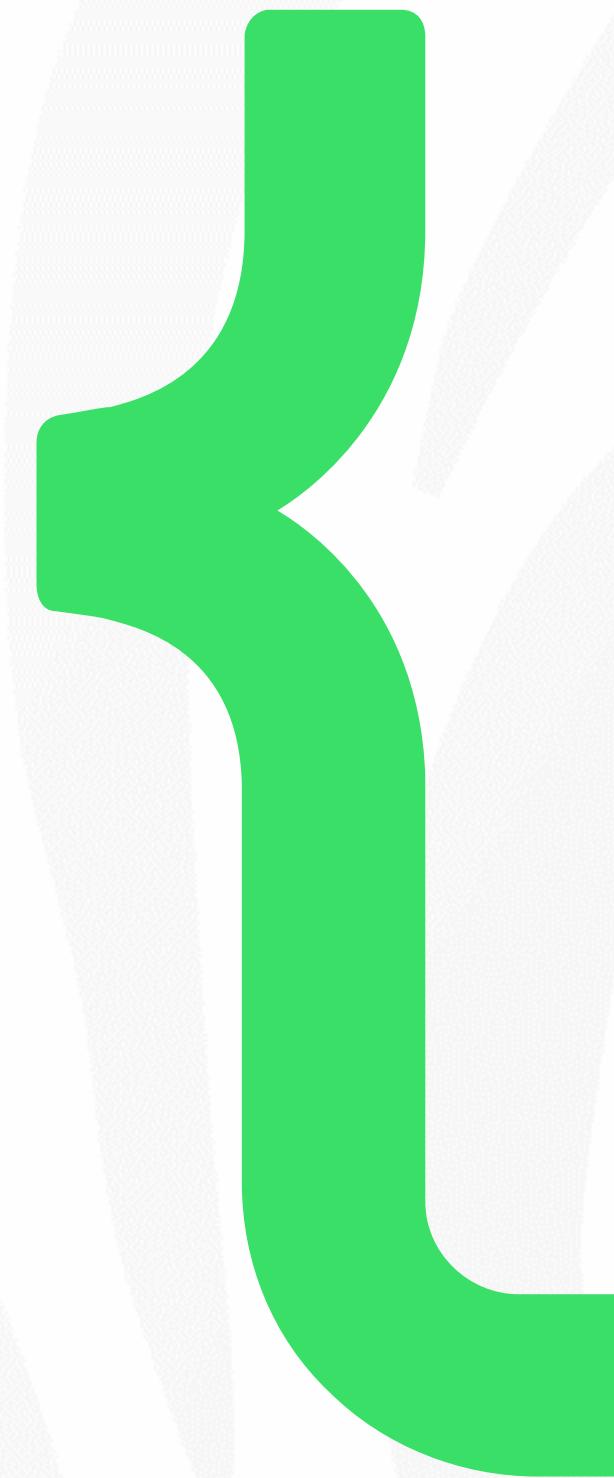
Активные – AnalyzerReleases.Shipped.md

Неактивные – AnalyzerReleases.Unshipped.md

Файлы должны находиться в корне проекта

Rule ID	Category	Severity	Notes
SK1301	Performance	Error	Do not chain LINQ methods after collection materialization

04



## Анализ: Кодфиксы

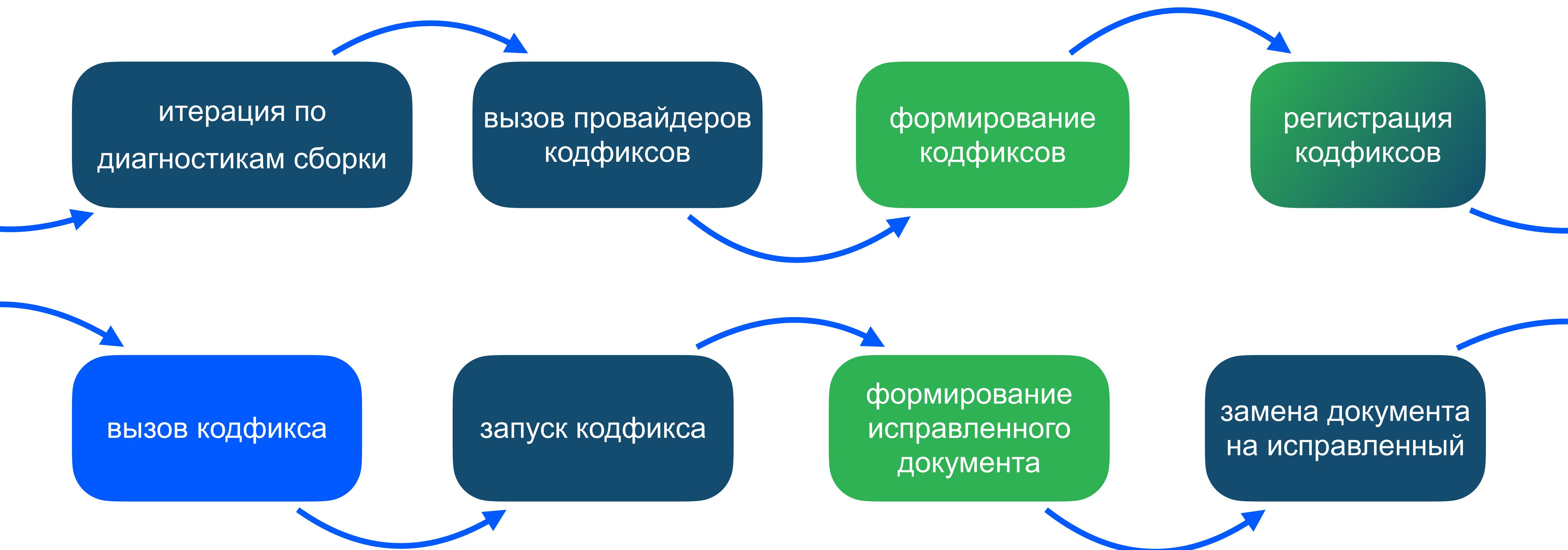
# Кодфиксы

Флоу выполнения кодфиксов

Roslyn

Ваш код

Юзер



# Определение провайдера кодфиксов

```
[ExportCodeFixProvider(LanguageNames.CSharp)]
public class MyCodeFixProvider : CodeFixProvider
{
    public override ImmutableArray<string> FixableDiagnosticIds { get; }

    public override async Task RegisterCodeFixesAsync(CodeFixContext context) { }
}
```

# Свойство FixableDiagnosticIds

Данное свойство задаёт диагностики, кодфиксы к которым может предоставить провайдер

```
public override ImmutableArray<string> FixableDiagnosticIds { get; } = [MyAnalyzer.DiagnosticId];
```

# Метод RegisterCodeFixesAsync

```
public override async Task RegisterCodeFixesAsync(CodeFixContext context)
{
    var root = await context.Document.GetSyntaxRootAsync();

    if (root is null)
        return;

    foreach (Diagnostic diagnostic in context.Diagnostics)
    {
        var node = (ClassDeclarationSyntax)root.FindNode(diagnostic.Location.SourceSpan);

        var codeAction = CodeAction.Create(
            title: "Capitalize class name",
            equivalenceKey: nameof(MyCodeFixProvider),
            createChangedDocument: ct => FixCodeAsync(context.Document, node, ct));

        context.RegisterCodeFix(codeAction, diagnostic);
    }
}
```

# Метод RegisterCodeFixesAsync

```
public override async Task RegisterCodeFixesAsync(CodeFixContext context)
{
    var root = await context.Document.GetSyntaxRootAsync();

    if (root is null)
        return;

    foreach (Diagnostic diagnostic in context.Diagnostics)
    {
        var node = (ClassDeclarationSyntax)root.FindNode(diagnostic.Location.SourceSpan);

        var codeAction = CodeAction.Create(
            title: "Capitalize class name",
            equivalenceKey: nameof(MyCodeFixProvider),
            createChangedDocument: ct => FixCodeAsync(context.Document, node, ct));

        context.RegisterCodeFix(codeAction, diagnostic);
    }
}
```

# Метод RegisterCodeFixesAsync

```
public override async Task RegisterCodeFixesAsync(CodeFixContext context)
{
    var root = await context.Document.GetSyntaxRootAsync();

    if (root is null)
        return;

    foreach (Diagnostic diagnostic in context.Diagnostics)
    {
        var node = (ClassDeclarationSyntax)root.FindNode(diagnostic.Location.SourceSpan);

        var codeAction = CodeAction.Create(
            title: "Capitalize class name",
            equivalenceKey: nameof(MyCodeFixProvider),
            createChangedDocument: ct => FixCodeAsync(context.Document, node, ct));

        context.RegisterCodeFix(codeAction, diagnostic);
    }
}
```

# Метод RegisterCodeFixesAsync

```
public override async Task RegisterCodeFixesAsync(CodeFixContext context)
{
    var root = await context.Document.GetSyntaxRootAsync();

    if (root is null)
        return;

    foreach (Diagnostic diagnostic in context.Diagnostics)
    {
        var node = (ClassDeclarationSyntax)root.FindNode(diagnostic.Location.SourceSpan);

        var codeAction = CodeAction.Create(
            title: "Capitalize class name",
            equivalenceKey: nameof(MyCodeFixProvider),
            createChangedDocument: ct => FixCodeAsync(context.Document, node, ct));

        context.RegisterCodeFix(codeAction, diagnostic);
    }
}
```

# Реализация исправления

```
private async Task<Document> FixCodeAsync(
    Document document,
    ClassDeclarationSyntax declaration,
    CancellationToken cancellationToken)
{
    var root = await document.GetSyntaxRootAsync(cancellationToken);

    if (root is null)
        return document;

    var identifier = SyntaxFactory.Identifier(Capitalize(declaration.Identifier.Text));
    var newDeclaration = declaration.WithIdentifier(identifier);

    var newRoot = root.ReplaceNode(declaration, newDeclaration);
    return document.WithSyntaxRoot(newRoot);
}
```

# Реализация исправления

```
private async Task<Document> FixCodeAsync(
    Document document,
    ClassDeclarationSyntax declaration,
    CancellationToken cancellationToken)
{
    var root = await document.GetSyntaxRootAsync(cancellationToken);

    if (root is null)
        return document;

    var identifier = SyntaxFactory.Identifier(Capitalize(declaration.Identifier.Text));
    var newDeclaration = declaration.WithIdentifier(identifier);

    var newRoot = root.ReplaceNode(declaration, newDeclaration);
    return document.WithSyntaxRoot(newRoot);
}
```

# Реализация исправления

```
private async Task<Document> FixCodeAsync(
    Document document,
    ClassDeclarationSyntax declaration,
    CancellationToken cancellationToken)
{
    var root = await document.GetSyntaxRootAsync(cancellationToken);

    if (root is null)
        return document;

    var identifier = SyntaxFactory.Identifier(Capitalize(declaration.Identifier.Text));
    var newDeclaration = declaration.WithIdentifier(identifier);

    var newRoot = root.ReplaceNode(declaration, newDeclaration);
    return document.WithSyntaxRoot(newRoot);
}
```

# Реализация исправления

```
private async Task<Document> FixCodeAsync(
    Document document,
    ClassDeclarationSyntax declaration,
    CancellationToken cancellationToken)
{
    var root = await document.GetSyntaxRootAsync(cancellationToken);

    if (root is null)
        return document;

    var identifier = SyntaxFactory.Identifier(Capitalize(declaration.Identifier.Text));
    var newDeclaration = declaration.WithIdentifier(identifier);

    var newRoot = root.ReplaceNode(declaration, newDeclaration);
    return document.WithSyntaxRoot(newRoot);
}
```

05



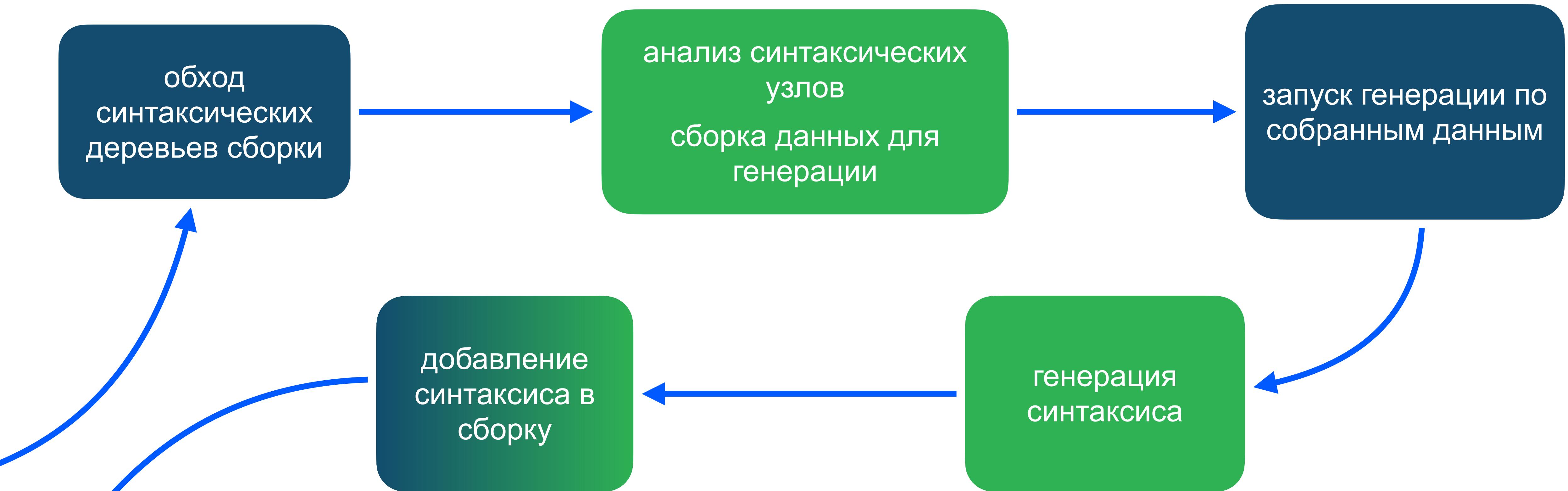
Генерация

# Source generators

Флоу выполнения генераторов

Roslyn

Ваш код



# Определение генератора

```
[Generator]
public class MyGenerator : ISourceGenerator
{
    public void Initialize(GeneratorInitializationContext context) { }

    public void Execute(GeneratorExecutionContext context) { }
}
```

# Контекст инициализации генератора

```
public readonly struct GeneratorInitializationContext
{
    public void RegisterForSyntaxNotifications(Func<ISyntaxReceiver> receiverCreator) { ... }

    public void RegisterForSyntaxNotifications(Func<ISyntaxContextReceiver> receiverCreator) { ... }

    ...
}
```

# Метод Initialize

Ресиверы синтаксиса

```
public interface ISyntaxReceiver
{
    void OnVisitSyntaxNode(SyntaxNode syntaxNode);
}
```

```
public interface ISyntaxContextReceiver
{
    void OnVisitSyntaxNode(GeneratorSyntaxContext context);
}
```

# Метод Initialize

Пример реализации ISyntaxContextReceiver

```
public class GenerateNameSyntaxReceiver : ISyntaxContextReceiver
{
    public List<INamedTypeSymbol> Symbols { get; } = [];

    public void OnVisitSyntaxNode(GeneratorSyntaxContext context)
    {
        if (context.Node is not ClassDeclarationSyntax declarationSyntax)
            return;

        INamedTypeSymbol? symbol = context.SemanticModel.GetDeclaredSymbol(declarationSyntax);

        INamedTypeSymbol? attributeSymbol = context.SemanticModel.Compilation
            .GetTypeByMetadataName("MyAnalyzers.Annotations.GeneratedNameFieldAttribute");

        if (symbol is null || attributeSymbol is null)
            return;

        bool hasAttribute = symbol.GetAttributes().Any(x => x.AttributeClass.Equals(attributeSymbol));

        if (hasAttribute is false)
            return;

        Symbols.Add(symbol);
    }
}
```

# Метод Initialize

## Пример реализации ISyntaxContextReceiver

```
public class GenerateNameSyntaxReceiver : ISyntaxContextReceiver
{
    public List<INamedTypeSymbol> Symbols { get; } = [];

    public void OnVisitSyntaxNode(GeneratorSyntaxContext context)
    {
        if (context.Node is not ClassDeclarationSyntax declarationSyntax)
            return;

        INamedTypeSymbol? symbol = context.SemanticModel.GetDeclaredSymbol(declarationSyntax);

        INamedTypeSymbol? attributeSymbol = context.SemanticModel.Compilation
            .GetTypeByMetadataName("MyAnalyzers.Annotations.GeneratedNameFieldAttribute");

        if (symbol is null || attributeSymbol is null)
            return;

        bool hasAttribute = symbol.GetAttributes().Any(x => x.AttributeClass.Equals(attributeSymbol));

        if (hasAttribute is false)
            return;

        Symbols.Add(symbol);
    }
}
```

# Метод Initialize

## Пример реализации ISyntaxContextReceiver

```
public class GenerateNameSyntaxReceiver : ISyntaxContextReceiver
{
    public List<INamedTypeSymbol> Symbols { get; } = [];

    public void OnVisitSyntaxNode(GeneratorSyntaxContext context)
    {
        if (context.Node is not ClassDeclarationSyntax declarationSyntax)
            return;

        INamedTypeSymbol? symbol = context.SemanticModel.GetDeclaredSymbol(declarationSyntax);

        INamedTypeSymbol? attributeSymbol = context.SemanticModel.Compilation
            .GetTypeByMetadataName("MyAnalyzers.Annotations.GeneratedNameFieldAttribute");

        if (symbol is null || attributeSymbol is null)
            return;

        bool hasAttribute = symbol.GetAttributes().Any(x => x.AttributeClass.Equals(attributeSymbol));

        if (hasAttribute is false)
            return;

        Symbols.Add(symbol);
    }
}
```

# Метод Initialize

## Пример реализации ISyntaxContextReceiver

```
public class GenerateNameSyntaxReceiver : ISyntaxContextReceiver
{
    public List<INamedTypeSymbol> Symbols { get; } = [];

    public void OnVisitSyntaxNode(GeneratorSyntaxContext context)
    {
        if (context.Node is not ClassDeclarationSyntax declarationSyntax)
            return;

        INamedTypeSymbol? symbol = context.SemanticModel.GetDeclaredSymbol(declarationSyntax);

        INamedTypeSymbol? attributeSymbol = context.SemanticModel.Compilation
            .GetTypeByMetadataName("MyAnalyzers.Annotations.GeneratedNameFieldAttribute");

        if (symbol is null || attributeSymbol is null)
            return;

        bool hasAttribute = symbol.GetAttributes().Any(x => x.AttributeClass.Equals(attributeSymbol));

        if (hasAttribute is false)
            return;

        Symbols.Add(symbol);
    }
}
```

# Метод Initialize

## Пример реализации ISyntaxContextReceiver

```
public class GenerateNameSyntaxReceiver : ISyntaxContextReceiver
{
    public List<INamedTypeSymbol> Symbols { get; } = [];

    public void OnVisitSyntaxNode(GeneratorSyntaxContext context)
    {
        if (context.Node is not ClassDeclarationSyntax declarationSyntax)
            return;

        INamedTypeSymbol? symbol = context.SemanticModel.GetDeclaredSymbol(declarationSyntax);

        INamedTypeSymbol? attributeSymbol = context.SemanticModel.Compilation
            .GetTypeByMetadataName("MyAnalyzers.Annotations.GeneratedNameFieldAttribute");

        if (symbol is null || attributeSymbol is null)
            return;

        bool hasAttribute = symbol.GetAttributes().Any(x => x.AttributeClass.Equals(attributeSymbol));

        if (hasAttribute is false)
            return;

        Symbols.Add(symbol);
    }
}
```

# Метод Initialize

## Пример реализации ISyntaxContextReceiver

```
public class GenerateNameSyntaxReceiver : ISyntaxContextReceiver
{
    public List<INamedTypeSymbol> Symbols { get; } = [];

    public void OnVisitSyntaxNode(GeneratorSyntaxContext context)
    {
        if (context.Node is not ClassDeclarationSyntax declarationSyntax)
            return;

        INamedTypeSymbol? symbol = context.SemanticModel.GetDeclaredSymbol(declarationSyntax);

        INamedTypeSymbol? attributeSymbol = context.SemanticModel.Compilation
            .GetTypeByMetadataName("MyAnalyzers.Annotations.GeneratedNameFieldAttribute");

        if (symbol is null || attributeSymbol is null)
            return;

        bool hasAttribute = symbol.GetAttributes().Any(x => x.AttributeClass.Equals(attributeSymbol));

        if (hasAttribute is false)
            return;

        Symbols.Add(symbol);
    }
}
```

# Метод Initialize

Регистрация ресивера

```
public void Initialize(GeneratorInitializationContext context)
{
    context.RegisterForSyntaxNotifications(() => new GenerateNameSyntaxReceiver());
}
```

# Метод Execute

Обработка данных из ресивера

```
public void Execute(GeneratorExecutionContext context)
{
    if (context.SyntaxContextReceiver is not GenerateNameSyntaxReceiver receiver)
        return;

    foreach (INamedTypeSymbol typeSymbol in receiver.Symbols)
    {
        context.CancellationToken.ThrowIfCancellationRequested();
        GenerateNameField(context, typeSymbol);
    }
}
```

# Метод Execute

## Генерация кода

```
private void GenerateNameField(GeneratorExecutionContext context, INamedTypeSymbol symbol)
{
    // Задаём инициализатор поля, присваивающий его значение
    EqualsValueClauseSyntax fieldValue = EqualsValueClause(LiteralExpression(
        SyntaxKind.StringLiteralExpression,
        Literal(symbol.Name)));

    // Задаём объявление "переменной"
    VariableDeclarationSyntax fieldDeclaration = VariableDeclaration(
        PredefinedType(Token(SyntaxKind.StringKeyword)))
        .AddVariables(VariableDeclarator(Identifier("Name")).WithInitializer(fieldValue));

    // Задаём объявление самого поля
    FieldDeclarationSyntax field = FieldDeclaration(fieldDeclaration)
        .AddModifiers(Token(SyntaxKind.PublicKeyword), Token(SyntaxKind.ConstKeyword));

    ...
}
```

# Метод Execute

## Генерация кода

```
private void GenerateNameField(GeneratorExecutionContext context, INamedTypeSymbol symbol)
{
    ...
    // Объявляем класс, добавляем в него поле
    ClassDeclarationSyntax classDeclaration = ClassDeclaration(symbol.Name)
        .AddModifiers(Token(SyntaxKind.PublicKeyword), Token(SyntaxKind.PartialKeyword))
        .AddMembers(field);

    // Объявляем идентификатор неймспейса
    IdentifierNameSyntax namespaceIdentifier = IdentifierName(
        symbol.ContainingNamespace.ToString(SymbolDisplayFormat.FullyQualifiedFormat));

    // Объявляем неймспейс, добавляем в него класс
    NamespaceDeclarationSyntax namespaceDeclaration = NamespaceDeclaration(namespaceIdentifier)
        .AddMembers(classDeclaration);

    // Создаём синтаксический корень
    CompilationUnitSyntax compilation = CompilationUnit().AddMembers(namespaceDeclaration);

    ...
}
```

# Метод Execute

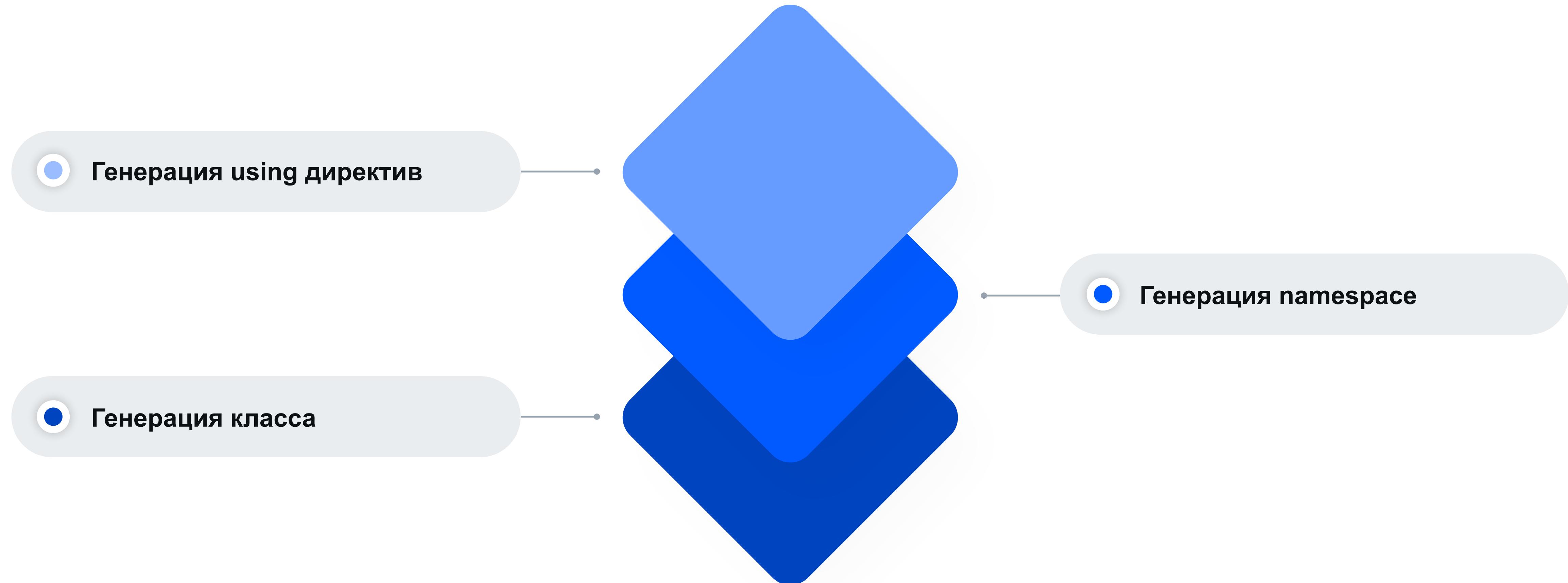
## Генерация кода

```
private void GenerateNameField(GeneratorExecutionContext context, INamedTypeSymbol symbol)
{
    ...

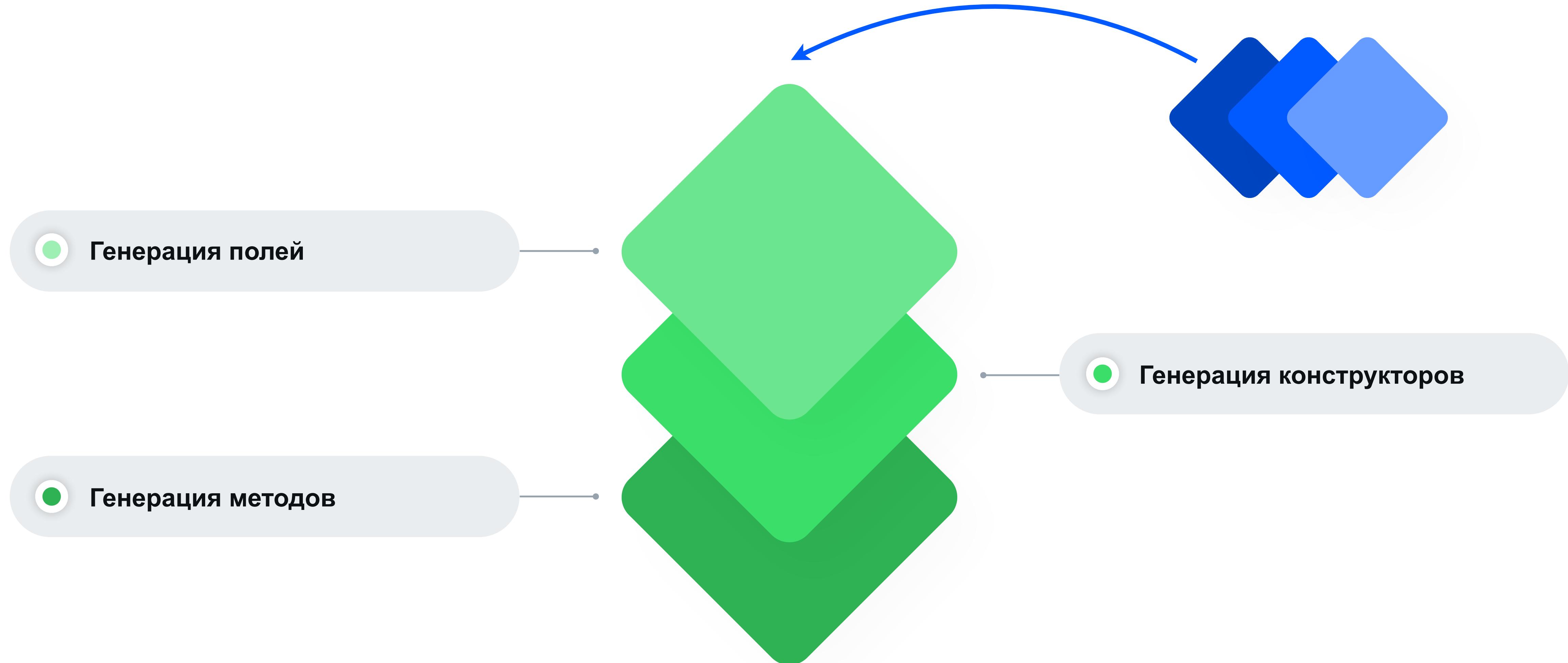
    var fileName = $"{symbol.Name}.NameField.cs";
    string text = compilation.NormalizeWhitespace(eol: "\n").ToFullString();

    context.AddSource(fileName, text);
}
```

# Декомпозиция генератора на цепочки обязанности



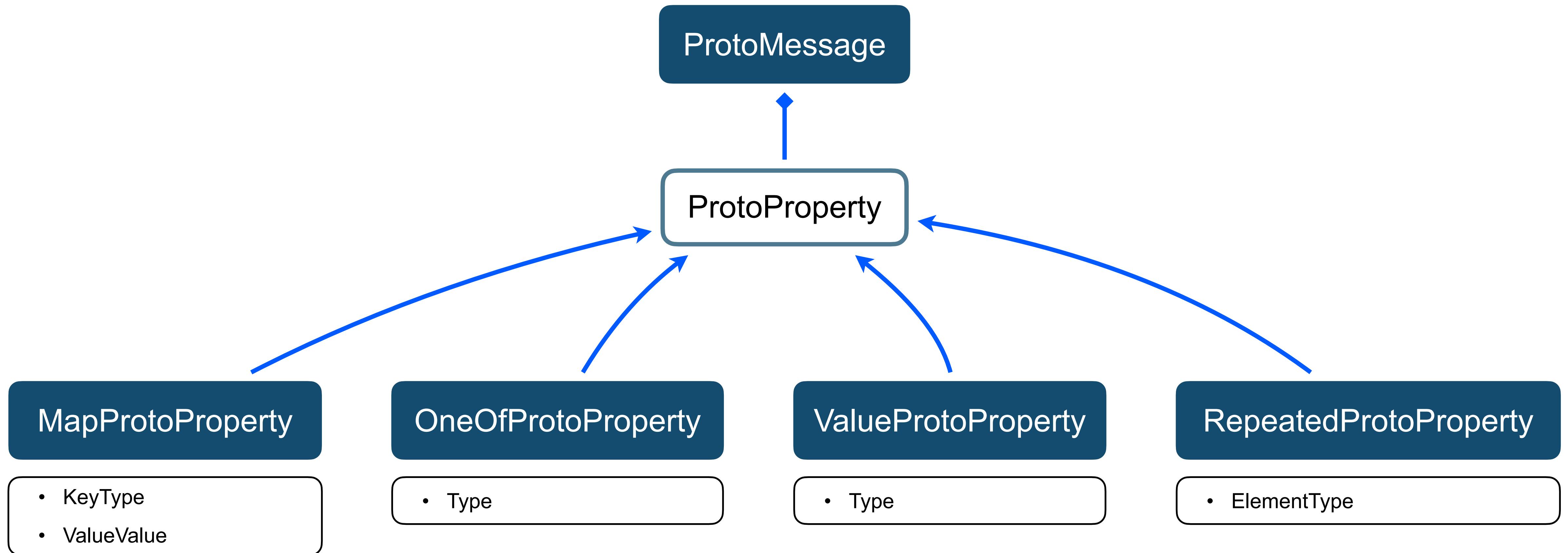
# Декомпозиция генератора на цепочки обязанности



# Доменные модели генераторов

- Выделяйте модели, описывающие домен вашего генератора
- Реализуйте создание объектов этой модели в ресивера
  - В нём реализуется полная обработка исходных данных
  - В нём содержится наиболее полная информация об исходных данных

# Доменные модели генераторов



06



Публикация

# Конфигурация csproj

## Свойства проекта

- Roslyn плагины работают только с netstandard2.0
- Версия языка не привязана к версии рантайма, используйте это
- Сборки Roslyn плагина не должны упаковываться в NuGet пакет как подключаемая библиотека

```
<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <LangVersion>latest</LangVersion>
    <IncludeBuildOutput>false</IncludeBuildOutput>
</PropertyGroup>
```

# Конфигурация csproj

## Упаковка сборок плагина

- Подключение сборок из NuGet зависимостей происходит только для файлов находящихся в analyzers/dotnet/cs
- Сборку нашего плагина в этот путь необходимо подключить вручную

```
<ItemGroup>
    <None Include="$(OutputPath)\$(AssemblyName).dll" Pack="true" PackagePath="analyzers/dotnet/cs" Visible="false" />
</ItemGroup>
```

# Подключение зависимостей

## Зависимости для выполнения плагинов

- Зависимости, используемые в реализации плагинов
- Не переходить к пользователям, устанавливающим плагинов
- Распространяются вместе со сборками плагинов, частью NuGet пакета

```
<ItemGroup>
    <PackageReference Include="Newtonsoft.Json"
        Version="13.0.3"
        PrivateAssets="all"
        GeneratePathProperty="true" />
</ItemGroup>

<ItemGroup>
    <None Include="$(PkgNewtonsoft.Json)\lib\netstandard2.0\*.dll"
        Pack="true"
        PackagePath="analyzers/dotnet/cs"
        Visible="false" />
</ItemGroup>
```

# Подключение зависимостей

## Зависимости генерируемого кода

- Зависимости, необходимые для сборки сгенерированного вашим плагином кода
- Должны быть транзитивными зависимостями относительно вашего NuGet пакета
- Подключаются стандартным способом

# 07



# ?

Отладка

# Подключение плагина как проект

```
<ItemGroup>
    <ProjectReference Include="path/to/analyzer.csproj"
        OutputItemType="Analyzer"
        ReferenceOutputAssembly="false" />
</ItemGroup>
```

# Подключение плагина как проект

## Недостатки

- Зависимости плагина не переходят к проекту-песочнице
- Такое подключение не соответствует способу, которым потребители будут подключать плагин

# Подключение плагина как NuGet пакет

Агрегация локальных NuGet пакетов

```
<PropertyGroup>
    <CustomPackageOutputPath>
        $([System.IO.Path]::Combine($(SolutionDir), 'bin', 'packages'))
    </CustomPackageOutputPath>
</PropertyGroup>

<Target Name="CopyPackageToGlobalPath" AfterTargets="Pack" Condition="'$(ShouldMovePackage)' = 'true'">
    <Copy SourceFiles="$(OutputPath)..\$(PackageId).$(PackageVersion).nupkg"
          DestinationFolder="$(CustomPackageOutputPath)" />
</Target>
```

# Подключение плагина как NuGet пакет

Агрегация локальных NuGet пакетов

```
<PropertyGroup>
    <CustomPackageOutputPath>
        $([System.IO.Path]::Combine($(SolutionDir), 'bin', 'packages'))
    </CustomPackageOutputPath>
</PropertyGroup>

<Target Name="CopyPackageToGlobalPath" AfterTargets="Pack" Condition="'$(ShouldMovePackage)' = 'true'">
    <Copy SourceFiles="$(OutputPath)..\$(PackageId).$(PackageVersion).nupkg"
          DestinationFolder="$(CustomPackageOutputPath)" />
</Target>
```

# Подключение плагина как NuGet пакет

Агрегация локальных NuGet пакетов

```
<PropertyGroup>
    <CustomPackageOutputPath>
        $([System.IO.Path]::Combine($(SolutionDir), 'bin', 'packages'))
    </CustomPackageOutputPath>
</PropertyGroup>

<Target Name="CopyPackageToGlobalPath" AfterTargets="Pack" Condition="'$(ShouldMovePackage)' = 'true'">
    <Copy SourceFiles="$(OutputPath)..\$(PackageName).$(PackageVersion).nupkg"
          DestinationFolder="$(CustomPackageOutputPath)" />
</Target>
```

# Подключение плагина как NuGet пакет

Агрегация локальных NuGet пакетов

```
<PropertyGroup>
    <CustomPackageOutputPath>
        $([System.IO.Path]::Combine($(SolutionDir), 'bin', 'packages'))
    </CustomPackageOutputPath>
</PropertyGroup>

<Target Name="CopyPackageToGlobalPath" AfterTargets="Pack" Condition="'$(ShouldMovePackage)' = 'true'">
    <Copy SourceFiles="$(OutputPath)..\$(PackageId).$(PackageVersion).nupkg"
          DestinationFolder="$(CustomPackageOutputPath)" />
</Target>
```

# Подключение плагина как NuGet пакет

Настройка NuGet.config

```
<configuration>
    <packageSources>
        <add key="local-roslyn" value="bin/packages" />
    </packageSources>
    <packageSourceMapping>
        <packageSource key="local-roslyn">
            <package pattern="MyAnalyzerName*" />
        </packageSource>
        <packageSource key="nuget.org">
            <package pattern="*" />
        </packageSource>
    </packageSourceMapping>
</configuration>
```

# Подключение плагина как NuGet пакет

Настройка NuGet.config

```
<configuration>
    <packageSources>
        <add key="local-roslyn" value="bin/packages" />
    </packageSources>
    <packageSourceMapping>
        <packageSource key="local-roslyn">
            <package pattern="MyAnalyzerName*" />
        </packageSource>
        <packageSource key="nuget.org">
            <package pattern="*" />
        </packageSource>
    </packageSourceMapping>
</configuration>
```

# Подключение плагина как NuGet пакет

Настройка NuGet.config

```
<configuration>
    <packageSources>
        <add key="local-roslyn" value="bin/packages" />
    </packageSources>
    <packageSourceMapping>
        <packageSource key="local-roslyn">
            <package pattern="MyAnalyzerName*" />
        </packageSource>
        <packageSource key="nuget.org">
            <package pattern="*" />
        </packageSource>
    </packageSourceMapping>
</configuration>
```



Не забывайте  
очищать кеши NuGet!

08



Тестирование

# Библиотеки для тестирования плагинов

“Microsoft.CodeAnalysis.CSharp.Analyzer.Testing.\*”

“Microsoft.CodeAnalysis.CSharp.CodeFix.Testing.\*”

“Microsoft.CodeAnalysis.CSharp.SourceGenerators.Testing.\*”

# Тестирование плагинов

## Анализаторы

```
var test = new CSharpAnalyzerTest<MyAnalyzer, XUnitVerifier>
{
    TestState =
    {
        Sources = { "public class myClass { }" },
        AdditionalReferences = { typeof(AnalyzerAttribute).Assembly },
    },
    ExpectedDiagnostics = { AnalyzerVerifier<MyAnalyzer>.Diagnostic(MyAnalyzer.DiagnosticId) },
};

await test.RunAsync();
```

# Тестирование плагинов

## Кодфиксы

```
var test = new CSharpCodeFixTest<ClassNameAnalyzer, ClassNameCodeFixProvider, XUnitVerifier>
{
    TestState =
    {
        Sources = { "public class a { }" },
        AdditionalReferences = { typeof(AnalyzerAttribute).Assembly },
    },
    FixedState =
    {
        Sources = { "public class A { }" },
    },
    NumberOfFixAllIterations = 1,
};

await test.RunAsync();
```

# Тестирование плагинов

## Генераторы

```
var test = new CSharpSourceGeneratorTest<ClassNameGenerator, XUnitVerifier>
{
    TestState =
    {
        Sources = { "[GenerateNameField]\npublic partial class A { }" },
        GeneratedSources =
        {
            ("generated/file/name.cs", "public partial class A { public const string Name = \"A\"; } "),
        },
    },
};

await test.RunAsync();
```

# 09



# Ссылки

# Ссылки

## SourceKit

<https://github.com/itmo-is-dev/SourceKit>

- NuGet пакет с общей логикой для написания плагинов
- Набор анализаторов
- Набор генераторов
  - SourceKit.Generators.Builder
  - SourceKit.Generators.Grpc
- Примеры концепций и подходов на реальном проекте
- Пример реализации теста анализатора с генератором



# Ссылки

- Краткое введение в принципы Roslyn

<https://github.com/dotnet/roslyn/blob/main/docs/wiki/Roslyn-Overview.md>

- Подробный гайд по тому как писать сурс-генераторы

<https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md>

<https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md#unit-testing-of-generators>

- Краткая документация по тестовым фреймворкам для Roslyn

<https://github.com/dotnet/roslyn-sdk/tree/main/src/Microsoft.CodeAnalysis.Testing>

- Сборник различных материалов по Roslyn

<https://github.com/ironcev/awesome-roslyn>

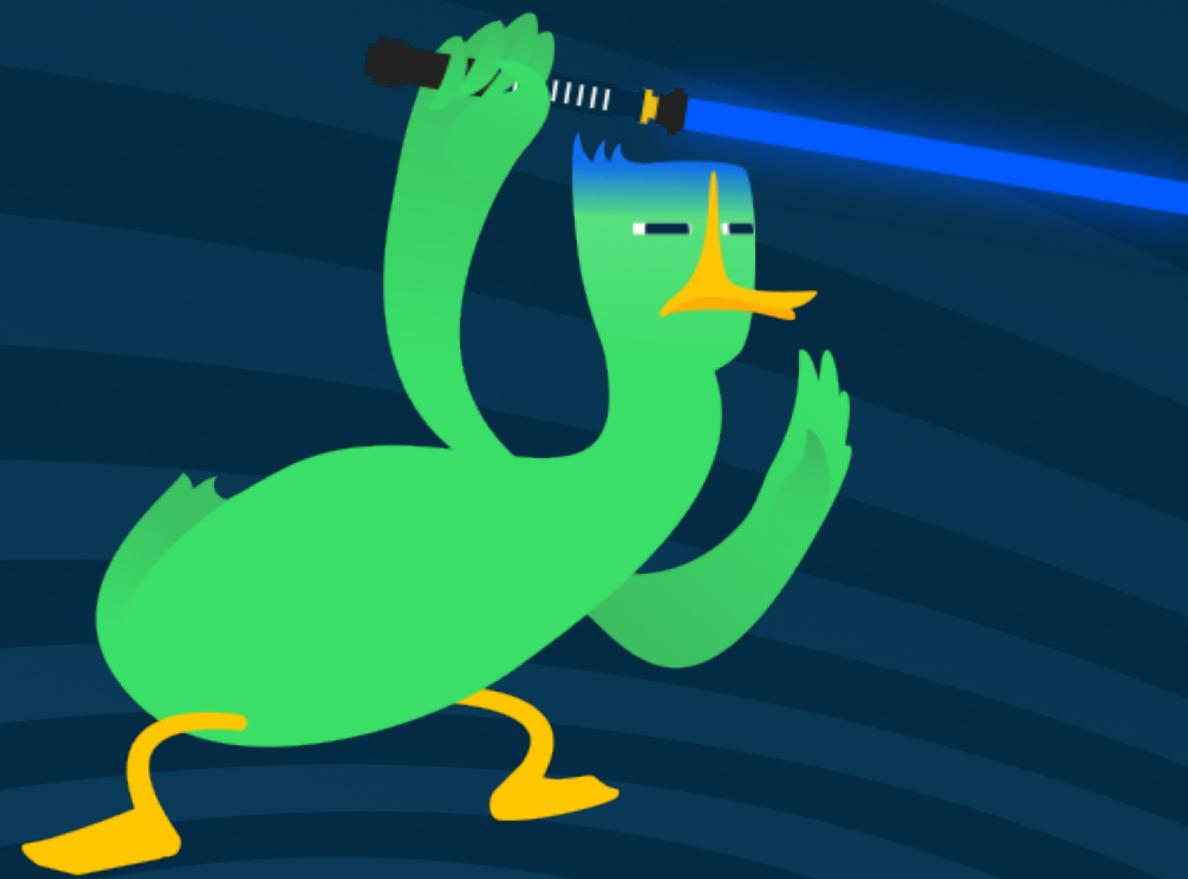
- Генератор вызовов к SyntaxFactory по C# коду

<https://roslynquoter.azurewebsites.net/>

- Библиотека с генераторами полифилов для C#

<https://github.com/Sergio0694/PolySharp>





Теперь вы  
знакомы с Roslyn!

## Что делать дальше?

- Попробовать себя в написании собственных плагинов
- Узнать про другие виды плагинов
  - Incremental Source Generators
  - Source Interceptors
  - Code Completion API
  - Code Refactoring API
  - ...
- Задать мне вопросы!



# Спасибо за внимание

Георгий Круглов  
[ronimizy@outlook.ru](mailto:ronimizy@outlook.ru)  
[github.com/ronimizy](https://github.com/ronimizy)