



Roslyn Code Analysis

Кулаков Анатолий

Статический анализ

Позволяет добиться консистентного вида

- отступы
- комментарии
- наименования
- группировки
- скобки

...



Позволяет избежать популярных ошибок

- корректность вызовов
- дизайн библиотек
- интернационализация
- время жизни ресурсов
- производительность
- безопасность
- различные метрики кода

...



Минусы

- позднее время проверок
- бедные настройки (уровень критичности, контекст применения и т.д.)
- нет единой простой платформы для написания правил
- слабая система для «исправлений» найденных замечаний
- сложность использования, распространения и поддержки новых правил
- контекст анализа на уровне отдельной сборки
- множество багов и медленное развитие
- из-за анализа IL плохо работает с синтаксическим сахаром
- морально устарели



Features APIs



Refactorings

Code Fixes

...

Workspaces APIs



Code Formatting

Find All References

Expand/Reduce

...

Workspaces (Solutions/Projects/Documents)

Compiler APIs



Syntax Trees

Symbols

Binding and
Flow Analysis

Emit

Code Analysis API (aka Diagnostics API)

- все проверки происходят в реальном времени
- гибкая система уровня критичности
- стандартные инструменты для построения анализаторов и исправителей
- анализ исходных кодов, а не IL
- отставание инструментов от языка невозможно
- контекст анализа на уровне всего солюшена
- легко писать бизнес-правила (в среднем 50-100 строк на правило)
- Roslyn легко изучается, тестируется, портируется
- это модно 😊

```

DiagnosticAnalyzer.cs + x CodeFixProvider.cs
ImmutableAnalyzer - ImmutableAnalyzer.ImmutableAnalyzerAnalyzer - DiagnosticId

12 {
13     [DiagnosticAnalyzer(LanguageNames.CSharp)]
14     public class ImmutableAnalyzerAnalyzer : DiagnosticAnalyzer
15     {
16         public const string DiagnosticId = "ImmutableAnalyzer";
17
18         // You can change these strings in the Resources.resx file. If you do not want your an
19         internal static readonly LocalizableString Title = new LocalizableResourceString(nameo
20         internal static readonly LocalizableString MessageFormat = new LocalizableResourceStri
21         internal static readonly LocalizableString Description = new LocalizableResourceString
22         internal const string Category = "Naming";
23
24         internal static DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId, Tit
25
26         public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { retur
27
28         public override void Initialize(AnalysisContext context)
29         {
30             // TODO: Consider registering other actions that act on syntax instead of or in ad
31             context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
32         }
33
34         private static void AnalyzeSymbol(SymbolAnalysisContext context)
35         {
36             // TODO: Replace the following code with your own analysis, generating Diagnostic (v

```

```
DiagnosticAnalyzer.cs x CodeFixProvider.cs
ImmutableAnalyzer - ImmutableAnalyzer.ImmutableAnalyzerAnalyzer - Initialize(AnalysisContext context)

18 // You can change these strings in the Resources.resx file. If you do not want your ani
19 internal static readonly LocalizableString Title = new LocalizableResourceString(nameo
20 internal static readonly LocalizableString MessageFormat = new LocalizableResourceStri
21 internal static readonly LocalizableString Description = new LocalizableResourceString
22 internal const string Category = "Naming";
23
24 internal static DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId, Tit
25
26 public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { retur
27
28 public override void Initialize(AnalysisContext context)
29 {
30 // TODO: Consider registering other actions that act on syntax instead of or in ad
31 context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
32 }
33
34 private static void AnalyzeSymbol(SymbolAnalysisContext context)
35 {
36 // TODO: Replace the following code with your own analysis, generating Diagnostic
37 var namedTypeSymbol = (INamedTypeSymbol)context.Symbol;
38
39 // Find just those named type symbols with names containing lowercase letters.
40 if (namedTypeSymbol.Name.ToCharArray().Any(char.IsLower))
41 {
42 // For all such symbols, produce a diagnostic.
```



```

DiagnosticAnalyzer.cs* x CodeFixProvider.cs
ImmutableAnalyzer - ImmutableAnalyzer.ImmutableAnalyzerAnalyzer - Title

24 internal static DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId, Title,
25
26 public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { return
27
28 public override void Initialize(AnalysisContext context)
29 {
30     // TODO: Consider registering other actions that act on syntax instead of or in add
31     context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
32 }
33
34 private static void AnalyzeSymbol(SymbolAnalysisContext context)
35 {
36     // TODO: Replace the following code with your own analysis, generating Diagnostic
37     var namedTypeSymbol = (INamedTypeSymbol)context.Symbol;
38
39     // Find just those named type symbols with names containing lowercase letters.
40     if (namedTypeSymbol.Name.ToCharArray().Any(char.IsLower))
41     {
42         // For all such symbols, produce a diagnostic.
43         var diagnostic = Diagnostic.Create(Rule, namedTypeSymbol.Locations[0], namedType
44
45         context.ReportDiagnostic(diagnostic);
46     }
47 }
48 }
    
```


BuildDemo - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help

Debug - Any CPU Start

Roslyn Syntax Visualizer Syntax.dgml Program.cs*

Undo Show Related Layout Share 100% Legend Filters

Syntax Tree

- IdentifierName [751..754]
 - IdentifierToken [751..754]
- VariableDeclarator [755..785]
 - IdentifierToken [755..757]
 - Trail: WhitespaceTrivia [757..758]
 - EqualsValueClause [758..785]
 - EqualsToken [758..759]
 - Trail: WhitespaceTrivia [759..760]
 - ObjectCreationExpression [760..785]
 - NewKeyword [760..763]
 - GenericName [764..783]
 - ArgumentList [783..785]
 - SemicolonToken [785..786]

Properties

Type	ObjectCreationExpressionSyntax
Kind	ObjectCreationExpression
ContainsDiagnostics	False
ContainsDirectives	False
ContainsSkippedText	False
FullSpan	[760..785]
HasLeadingTrivia	False
HasStructuredTrivia	False
HasTrailingTrivia	False
Initializer	
IsMissing	False

```
graph TD;
    OCE[ObjectCreationExpression] --> new[new];
    OCE --> GN[GenericName];
    OCE --> AL[ArgumentList];
    GN --> WT[WhitespaceTrivia];
    GN --> IA[ImmutableArray];
    GN --> TAL[TypeArgumentList];
    TAL --> LT["<"];
    TAL --> PT[PredefinedType];
    TAL --> GT[>];
    PT --> int[int];
    AL --> LP["()"];
```

Test Explorer Roslyn Syntax Visualizer

Error List Output Find Results 1 Find Results 2 Find Symbol Results

ImmutableAnalyzer - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help

Debug - Any CPU - Start -

CodeFixProvider.cs x DiagnosticAnalyzer.cs

ImmutableAnalyzer - ImmutableAnalyzer.BuildCodeFixProvider - DoFix(ObjectCreationExpressionSyntax objectCreati

```
17 public class BuildCodeFixProvider : CodeFixProvider
18 {
19     public override ImmutableArray<string> FixableDiagnosticIds =>
20         ImmutableArray.Create(ImmutableAnalyzerAnalyzer.DiagnosticId);
21
22     public override async Task RegisterCodeFixesAsync(CodeFixContext context)
23     {
24         var root = await context.Document.GetSyntaxRootAsync(context.CancellationToken);
25         var objectCreation = root.FindNode(context.Span).FirstAncestorOrSelf<ObjectCreat
26
27         context.RegisterCodeFix(
28             CodeAction.Create("Hey! Use ImmutableArray<T>.Empty fool!",
29                 c => DoFix(objectCreation, context.Document, c)),
30             context.Diagnostics[0]);
31     }
32
33     private async Task<Document> DoFix(ObjectCreationExpressionSyntax objectCreation, Do
34     {
35         var generator = SyntaxGenerator.GetGenerator(document);
36         var memberAccess = generator.MemberAccessExpression(objectCreation.Type, "Empty"
37
38         var root = await document.GetSyntaxRootAsync(c);
39         var newRoot = root.ReplaceNode(objectCreation, memberAccess);
40
41         return document.WithSyntaxRoot(newRoot);
```

100 %

Error List Output Find Results 1 Find Results 2 Find Symbol Results

Program.cs

DemoFodder

DemoFodder.Program

Main(string[] args)

```
1 using System;
2 using System.Collections.Immutable;
3
4 namespace DemoFodder
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             var a1 = new ImmutableArray<int>();
11             Hey! Use ImmutableArray<T>.Empty fool!
12         }
13     }
14 }
15
```

Hey! Use ImmutableArray<T>.Empty fool!

ImmutableAnalyzer Don't use ImmutableArray<T> constructor (Build 2015)


```
...
{
    var a1 = new ImmutableArray<int>();
    var a1 = ImmutableArray<int>.Empty;
    Console.WriteLine("a1.Length = {0}", a1.Length);
    ...
}
```

Preview changes

Output

Show output from:

Спасибо ReSharper'у за наше счастливое детство



Знания о пользе статического
анализа кода

Малый порог вхождения

Удовольствие от
использования

СІ

IDE Features

ReSharper

- >1500 code inspections
- ~1000 quick fixes
- ~100 refactorings (50 for C#)
- >1000 feature actions

Roslyn

- <100 code analyzers
- ~20 code fixes



*Кирилл Скрыган, **JetBrains***

Code-Aware Library

Библиотеки, помогающие в design-time работать с другими библиотеками

- базируются на Roslyn
- распространяются через NuGet и VSIX
- нацелены на пропаганду лучших практик
- могут включать фиксы

Существующие примеры использования

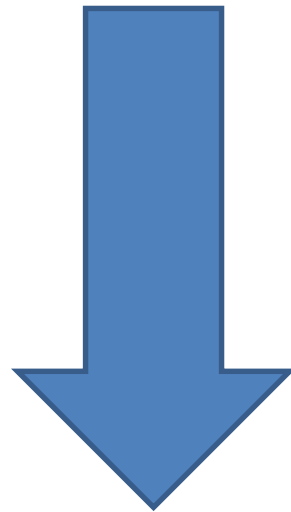
- FxCop
- StyleCop
- AzureAnalysis (работа с Azure)
- AsyncPackage (лучшие практики использования Task-based Asynchronous Pattern)
- MetaCompilation (обучение написанию своих анализаторов)
- AnalyzerPowerPack (общие правила работы с языком)
- RoslynDiagnostics (работа с Roslyn)
- C# Essentials (для перехода на синтаксис C# 6)
- TsqlAnalyzer (проверка T-SQL строк в коде C#)

Стоимость исправления ошибок

Момент выявления ошибки:

- до написания кода
- статические проверки
- unit-тесты
- code review
- интеграционные тесты
- ручные тесты
- ошибка при эксплуатации

Цена



Идеи для новых анализаторов

- **AutoMapper** может проверять соответствие всех свойств в преобразуемых типах, возможность создания конвертеров
- **Autofac** может проверять полноту регистраций, возможность создания компонента, захват зависимостей
- **RouteAttributes** должны содержать только url сегменты, которые есть в параметрах метода
- **ImmutableAttribute** – гарантирует, что состояние объекта не меняется после создания
- в **DTO** классах не должно быть бизнес-методов, все свойства открытые, коллекции проинициализированны, ссылки только на другие DTO или примитивы.
- объекты, участвующие в бинарной сериализации, должны иметь атрибут **Serializable**
- валидация **xml** конфигов на наличие ожидаемых настроек
- вся работа со **временем** должна быть только в UTC
- любой **ORM**, **WEB**, фреймворки переполнены неявными договорённостями

Любите статический анализ

- Если вы замечали в код-ревью, как ваша команда постоянно делает одни и те же ошибки. Теперь вы можете написать правила, позволяющие избежать надоедливых багов.
- Если ваши библиотеки содержат сложные бизнес-соглашения, требующие определённого порядка действий. Теперь правила позволят вам не забыть про неочевидные обязательства.
- Если у вас есть open source библиотеки или просто с public API, а пользователи упорно не хотят читать документацию по правильному их использованию. Теперь вы можете добавить корректирующие правила к вашим библиотекам и распространять их в виде NuGet пакета.