MODEL-DRIVEN SOFTWARE DEVELOPMENT - PRACTICE 4 - ROSLYN GUIDE

Diagnostic Analyzer és Code Fix Provider demo

Gergely Mezei

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

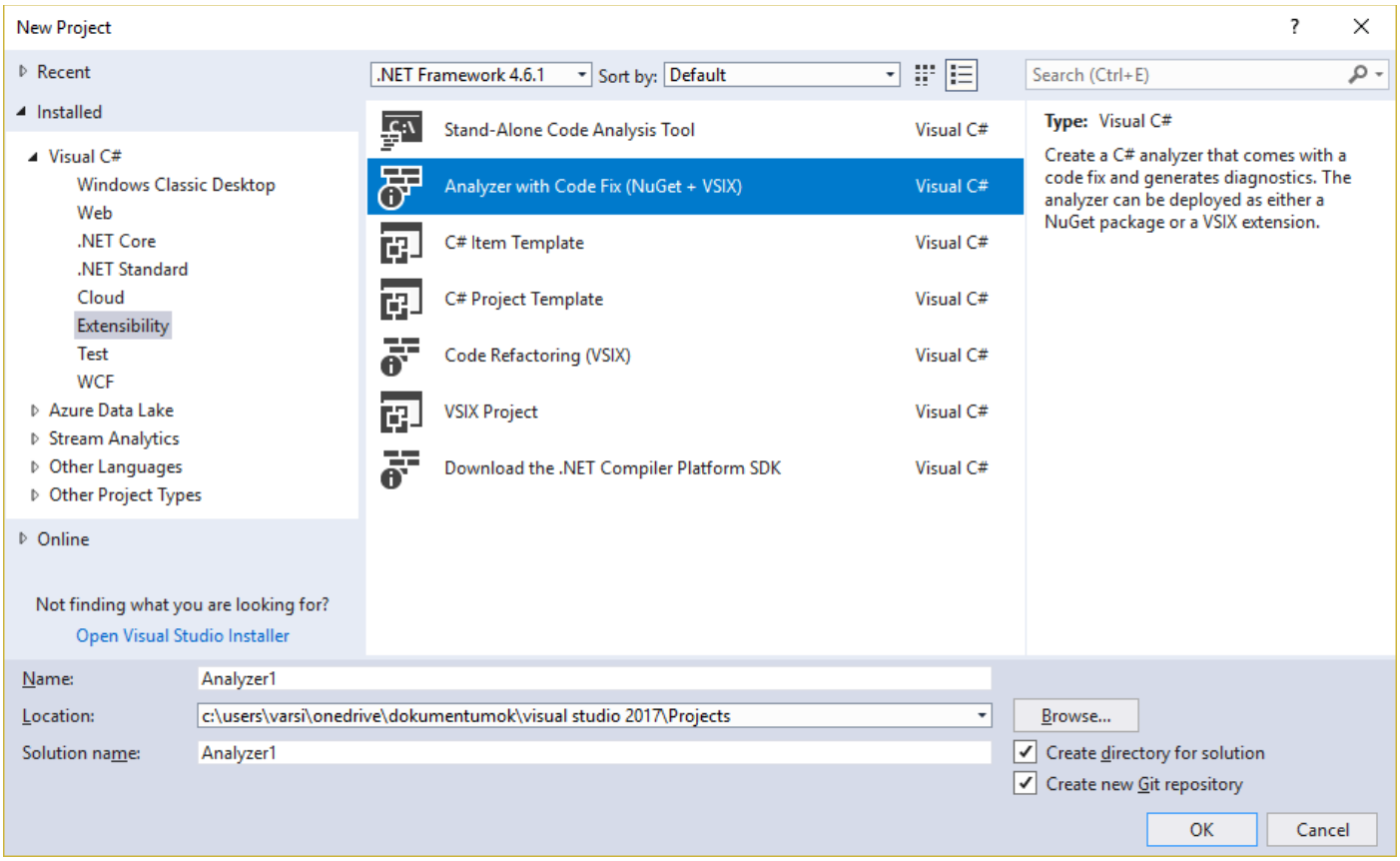
1117 Budapest, Magyar tudósok körútja 2.
Q. ép. B. szárny 207. • www.aut.bme.hu
Telefon: 463-2870 • Fax: 463-2871

I. Introduction

The following are the main steps of the demo that illustrate how the Diagnostic Analyzer and Code Fix Provider work. The aim of the two tools is presented in the presentation of the related practice, this document contains only a concise description of the practical steps.

II. Diagnostic analyzer

Let's create a new Diagnostic analyzer with an example solution in the following tab under New project:



Make sure that the target framework is at least 4.6.1, otherwise you may not see the Analyzer with Code Fix project type in older Visual Studio!

The goal of the sample project is that class names should be in all caps, otherwise we will receive a Warning during translation. We will change this behavior to the following example: methods that return Task or Task<T> (so they can be "awaited") should have an Async postfix at the end of their names (so the developers can see right away if they have omitted the await keyword, which the development environment warns them about).

Let's try to look at the project structure and figure out what the project is doing in its current form, then modify it to do what we need:

1. In Resources.resx, we update the corresponding text that appears:

	Name	Value
	AnalyzerDescription	Awaitable method names should end with Async postfix.
	AnalyzerMessageFormat	Method name '{0}' does not end with Async!
▶	AnalyzerTitle	Awaitable method names should end with Async postfix.
*		

2. Let's modify the Initalize method to register the AnalyzeSymbol method for methods:

```
public override void Initialize(AnalysisContext context)
{
    context. RegisterSymbolAction(AnalyzeSymbol, SymbolKind. Method);
}
```

3. Change the AnalyzeSymbol method to:

```
private static void AnalyzeSymbol(SymbolAnalysisContext context)
{
    var methodSymbol = (IMethodSymbol)context. Symbol;

    // Find just those named type symbols with names containing lowercase letters.
    if (
     methodSymbol.ReturnType.ToDisplayString(SymbolDisplayFormat.                MinimallyQualifiedFormat).
StartsWith("Task")
        && !Methotsimbol.name.endswit("asing"))
    {
        // For all such symbols, produce a diagnostic.
        var diagnostic = Diagnostic. Create(Rule, methodSymbol. Locations[0], methodSymbol. Name);

        context. ReportDiagnostic(diagnostic);
    }
}
```

To create an error message, the first parameter is the object describing the error message, the second is its location in the source code. We can also attach parameters to the error message (currently this is the name of the variable).

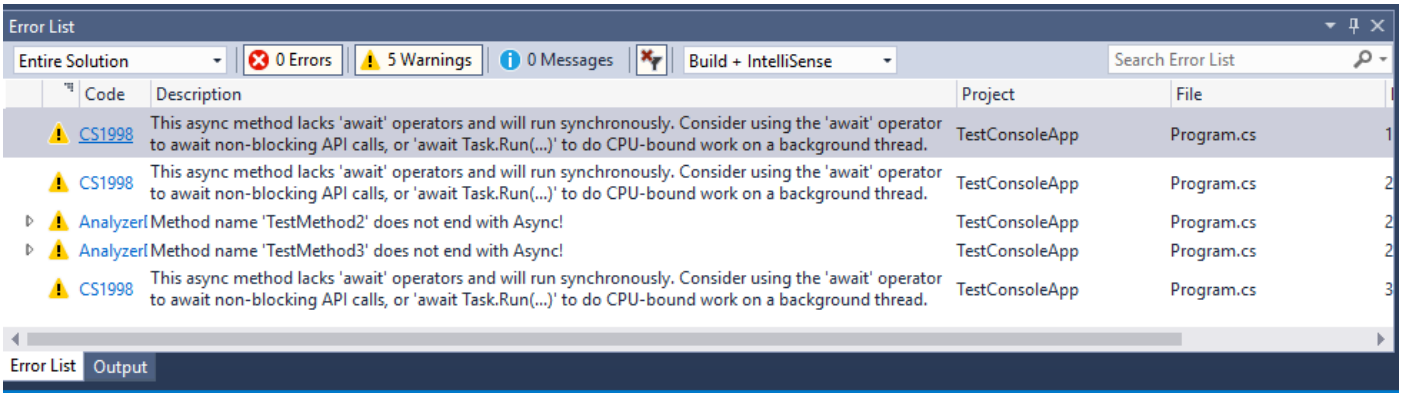
Let's start the AnalyzerDemo.Vsix project! Vsix stands for Visual Studio extension, if you run the project, an Experimental Visual Studio will open with the analyzer already installed.

Let's create a new project and demonstrate how it works on a simple Console app.

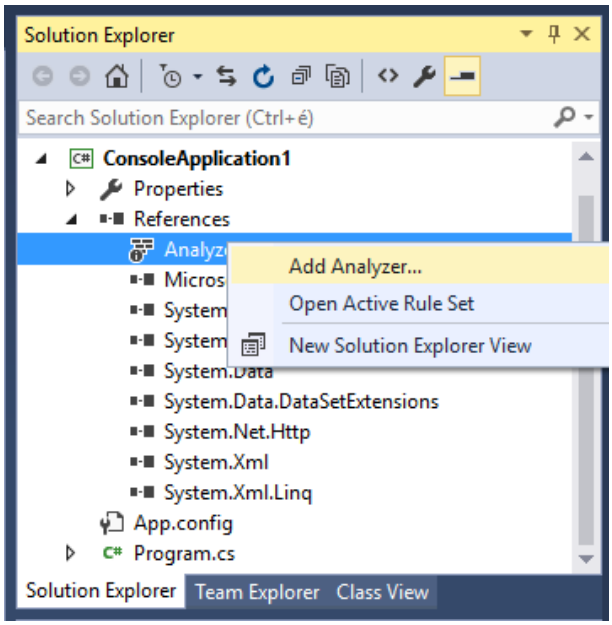
```
class Program
{
    static void Main(string[] args)
    {
    }

    static async void TestMethod()
    {

    }

    static async Task TestMethod2()
    {

    }
```

```
    static Task TestMethod3()
    {
        return Task. FromResult(0);
    }

    static async Task TestMethod4Async()
    {

    }
}
```

TestMethod2 and TestMethod3 should be warned, so the expected result is similar to the following:



Consider showing what happens if you define the rule as Error instead of Warning. Unsurprisingly, we will get an error, but contrary to expectations, our code will be compiled! The explanation: Diagnostic Analyzer can only cancel the build process if it is part of the project (because in this case the program would compiler differently in different development environments). For the build process to be interrupted, you need to add the .dll to the references!



III. Create a Code Fix Provider

If you have reported an error to the users using the Diagnostic Analyzer, it is worth offering them a solution immediately. In practice, this means fixing with Ctrl+. If we try, nothing will happen at the moment. We will write the related code to fix the wrong code.

With the **FixableDiagnosticIds** list we can specify which errors we can fix. The **GetFixAllProvider** method can be used to specify in what scope the user can fix more than just this one (the are three scopes: document, project, solution). The provided BatchFixer allows all three. The **RegisterCodeFixesAsync** method will actually register the code patch class at a specific location.

Modify the `AnalyzerDemoCodeFixProvider` class as follows:

1. Rewrite the title:

```
private const string title = "Add Async postfix!";
```

2. Modify the RegisterCodeFixesAsync method like this:

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

    // TODO: Replace the following code with your own analysis, generating a CodeAction for each fix to
suggest
    var diagnostic = context. Diagnostics.First();
    var diagnosticSpan = diagnostic. Location.SourceSpan;

    // Find the type declaration identified by the diagnostic.
    var declaration = root. FindToken(diagnosticSpan.Start). Parent.AncestorsAndSelf().
OfType<MethodDeclarationSyntax>(). First();

    // Register a code action that will invoke the fix.
    context. RegisterCodeFix(
        CodeAction. Create(
            title: title,
            createChangedSolution: c => AddPostFixAsync(context. Document, declaration, c),
            equivalenceKey: title),
        diagnostic);
}
```

3. Delete the MakeUppercaseAsync method, and then write the AddPostFixAsync method:

```
private async Task<Solution> AddPostFixAsync(Document document, MethodDeclarationSyntax declaration,
CancellationToken cancellationToken)
{
    //Compute the new name.
    var newName = declaration. Identifier.Text + "Async";

    // Get the symbol representing the type to be renamed.
    var semanticModel = await document. GetSemanticModelAsync(cancellationToken);
    var methodSymbol = semanticModel.GetDeclaredSymbol(declaration, cancellationToken);

    // Produce a new solution that has all references to that type renamed, including the declaration.
```

```
    var originalSolution = document. Project.Solution;
    var optionSet = originalSolution.Workspace.Options;
    var newSolution = await Renamer. RenameSymbolAsync(document. Project.Solution, methodSymbol, newName,
optionSet, cancellationToken). ConfigureAwait(false);

    // Return the new solution with the now-async-postfixed method name.
    return newSolution;
}
```

Here, it will be the Renamer class that replaces not only the method declaration with the correct one, but also all references associated with it.

4. Let's test how to fix the code!