



# MODELLALAPÚ SZOFTVERFEJLESZTÉS – 2. GYAKORLAT – ROSLYN SEGÉDLET

Diagnostic Analyzer és Code Fix Provider demo

Mezei Gergely

## Szerzői jogok

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot Modellalapú szoftverfejlesztés c. tantárgyat felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármely eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző előzetes engedélyével történhet.

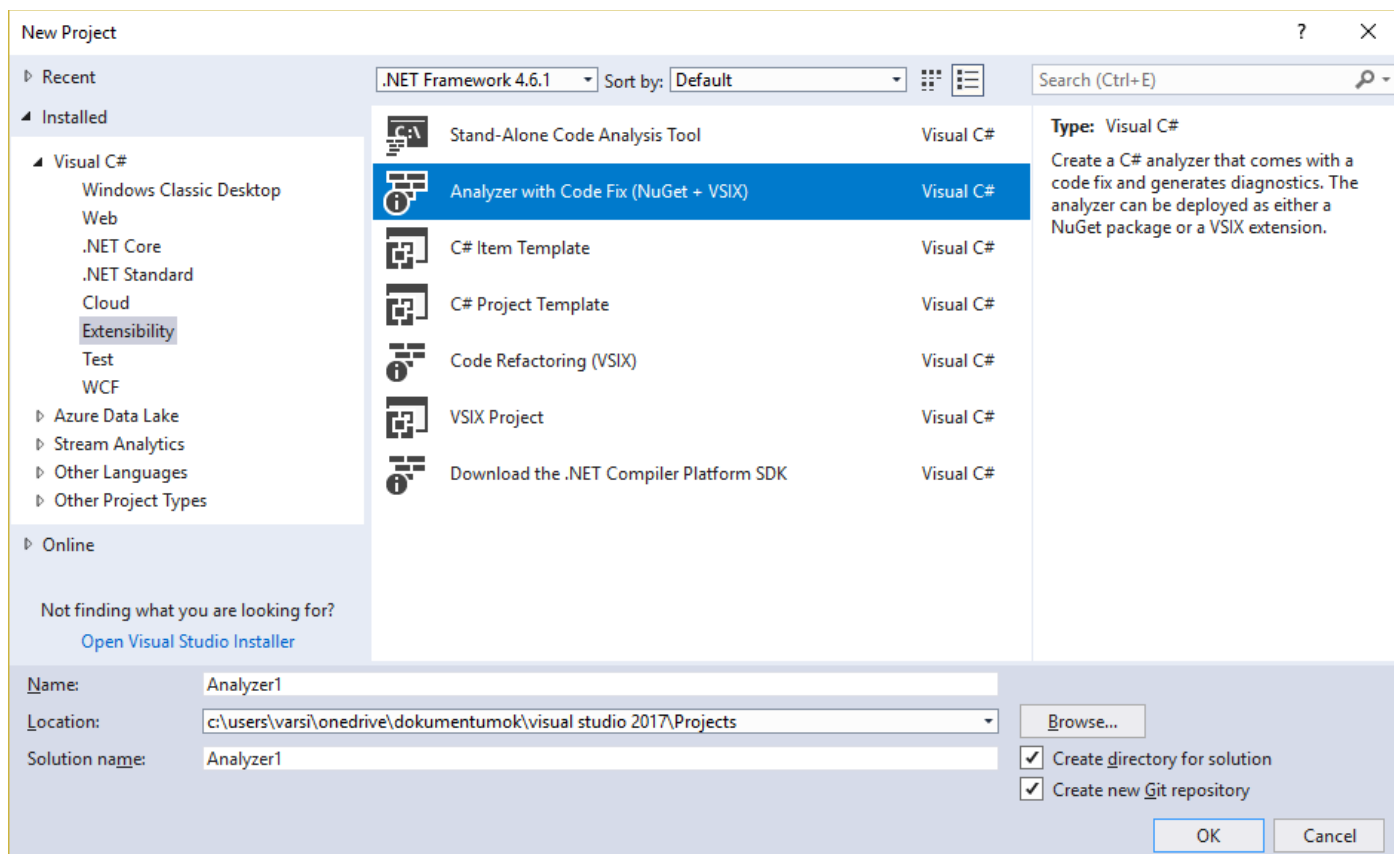


## I. BEVEZETÉS

A következőkben a Diagnostic Analyzer és a Code Fix Provider működését szemléltető demó főbb lépései kerülnek bemutatásra. A két eszköz célja a kapcsolódó gyakorlat előadásán hangzik el, ez a dokumentum csak a gyakorlati lépések tömör leírását tartalmazza.

## II. DIAGNOSTIC ANALYZER

Hozzunk létre egy új Diagnostic analyzert, amihez kapunk egy példa solutiont is, a következő fülön a New project alatt:



Figyeljünk arra, hogy a target framework minimum 4.6.1 legyen, ellenkező esetben régebbi Visual Studionál esetleg nem jelenik meg az Analyzer with Code Fix projekt típus!

A mintaprojekt lényege, hogy az osztályok neve csupa nagybetűs legyen, ellenkező esetben Warningot kapunk fordítás során. Mi ezt fogjuk átírni a következő példára: azokat a metódusokat, amelyiknek a visszatérési értékük Task, vagy Task<T> (tehát lehet őket „awaitelni”), azoknak a nevüknek a végén kötelező legyen az Async postfixet használni (és így a fejlesztő egyből látja, hogyha esetleg le hagyta az await kulcsszót, amire egyébként a fejlesztőkörnyezet figyelmeztet is).

Próbáljuk átnézni a projektstruktúrát és megfejteni, hogy mit csinál jelen formájában a projekt, majd módosítsuk úgy, hogy azt tegye, amire nekünk szükségünk van: a Rule statikus tagváltozó írja le a megjelenítendő hibaüzenetet, az Initialize metódus regisztrálja be az AnalyzeSymbol metódust a metódusok szemantikus elemzése után, és az AnalyzeSymbol metódust kell kiegészítenünk, hogy hozzon létre hibaüzeneteket a megfelelő alkalom esetén.

A megoldás lépései:

1. Resources.resx-ben aktualizáljuk a megfelelő megjelenő szövegeket:

	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. Módosítsuk az Initialize metódust úgy, hogy metódusokhoz regisztrálja be az AnalyzeSymbol metódust:

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

3. Módosítsuk az AnalyzeSymbol metódust, a következőre:

```
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.ToString().StartsWith("Task")
        && !methodSymbol.Name.EndsWith("Async"))
    {
        // For all such symbols, produce a diagnostic.
        var diagnostic = Diagnostic.Create(Rule, methodSymbol.Locations[0], methodSymbol.Name);

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

A hibaüzenet létrehozásához az első paraméter a hibaüzenetet leíró objektum, a második a helye a forráskódban, ezen kívül még a hibaüzenethez tudunk paramétereket csatolni, ez most a változó neve.

Indítsuk el az AnalyzerDemo.Vsix projektet! A Vsix a Visual Studio extension rövidítése, ha futtatjuk a projektet egy Experimental Visual Studio fog megnyílni, amibe az analyzer már fel van telepítve.

Hozzunk létre egy új projektet, és mutassuk be a működést egy egyszerű konzolos alkalmazáson!

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

    static async void TestMethod()
    {
    }

    static async Task TestMethod2()
    {
    }
}
```

```

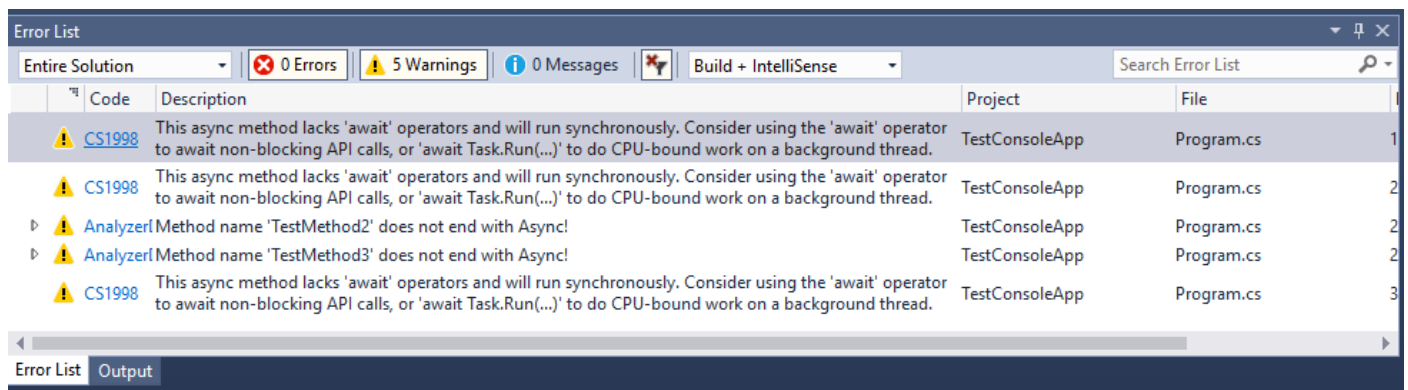
    }

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

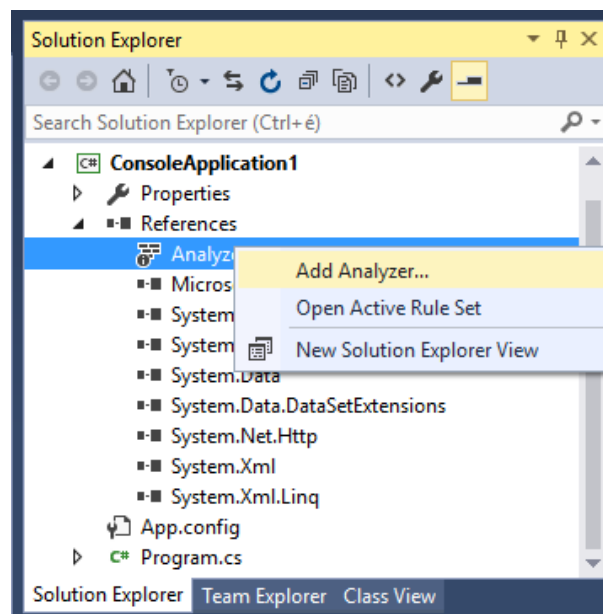
    static async Task TestMethod4Async()
    {
    }
}

```

TestMethod2 és TestMethod3 esetében kell figyelmeztetést kapnunk, tehát az elvárt eredmény az alábbihoz hasonló:



Érdekes lehet megmutatni, hogy mi történik akkor, hogyha nem Warningként, hanem Errorként definiáljuk a szabályt. Nem meglepő módon hibát fogunk kapni, viszont a várttal ellentétben a kódunk lefordul! A magyarázat a következő: a Diagnostic Analyzer csak akkor tudja megszakítani a build folyamatot, hogyha az a projekt része (hiszen ilyenkor más fejlesztőkörnyezetben máshogy fordulna a program, mint egy másikban). Ahhoz, hogy a build folyamat megszakadjon, fel kell vegyük a dll-t a referenciák közé!



### III. CODE FIX PROVIDER KÉSZÍTÉSE

Ha a Diagnostic Analyzer segítségével jeleztük a hibát a felhasználónak, akkor érdemes rögtön egy megoldást is felkínálni neki. A gyakorlatban ez a Ctrl+.-tal történő javítást jelenti. Ha megpróbáljuk, jelenleg még nem fog történni semmi. Írjuk meg a kapcsolódó kódot, amivel ki is lehet javítani az általunk rossznak ítélt kódot!

A **FixableDiagnosticIds** listával tudjuk megadni, hogy melyik hibákat tudjuk javítani. A **GetFixAllProvider** metódussal tudjuk megadni, hogyha esetleg a felhasználó nem csak ezt az egyet szeretné javítani, akkor milyen scope-ban tudja ezt megtenni (a három scope: dokumentum, projekt, solution). A megadott BatchFixer mindhármát lehetővé teszi. A **RegisterCodeFixesAsync** metódus fogja egy adott helyen beregisztrálni ténylegesen a kódjavító osztályt.

Módosítsuk az **AnalyzerDemoCodeFixProvider** osztályt a következőképpen:

1. Írjuk át a title-t:

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

2. Módosítsuk a RegisterCodeFixesAsync metódust így:

```
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. Töröljük a MakeUppercaseAsync metódust, majd írjuk meg az AddPostFixAsync metódust:

```
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;
}
```

Itt a Renamer osztály lesz az, aki nemcsak a metódus deklarációját cseréli ki a helyesre, hanem az összes, hozzá tartozó hivatkozást is.

#### 4. Teszteljük a kód javítását!