Visual Studioを使用した AIを使わない C#コード生成

株式会社ONE WEDGE
山岸理雄

自己紹介

山岸理雄（やまぎしりゅう）

- C# エンジニア
- 経歴
  - 2015/04～2023/12 愛知県豊橋市 製造業向けシステム開発
  - 2024/01～2024/11 無職 超楽しかった
  - 2024/12 **株式会社ONE WEDGE** 現職4か月目
- Qiita書いてます（最近さぼってる）→ **@ry18847**

@ry18847

224 Contributions

17	0	6
投稿	フォロー	フォロワー

開発効率を上げるために、SourceGeneratorやT4テンプレートを作ったりしています。

最推しのVTuber
でびでび・でびる
↑かわいい

いらんコードごとコピペ

バグごとコピペ
（最悪セキュリティホールごと）

規約を守らずコピペ

コピペは悪！！

貼り付けミス

意味を理解せずコピペ

ボイラープレート

同じコードの乱立
（せめて関数化しろ）

著作権とか
大丈夫そ？

人はなぜコピペをするのか

わからない実装を検索してコピペ

ボイラープレートコードをコピペ

だが・・・今は違う！！

- 生成AIで動的コード生成
- 静的コード生成で自動化

AIコード生成

- ChatGPT
- Copilot
- Claude　...など

C始まり多くない？

使ってみると、便利だが

苦労することも多い

ChatGPTに作らせたコードの例 :FizzBuzz

```
for (int i = 1; i <= 100; i++)
{
    if (i % 3 == 0 && i % 5 == 0)
        Console.WriteLine("FizzBuzz");
    else if (i % 3 == 0)
        Console.WriteLine("Buzz");
    else if (i % 5 == 0)
        Console.WriteLine("Fizz");
    else
        Console.WriteLine(i);
}
```

逆

コードが正しくない！
修正に時間がかかる！

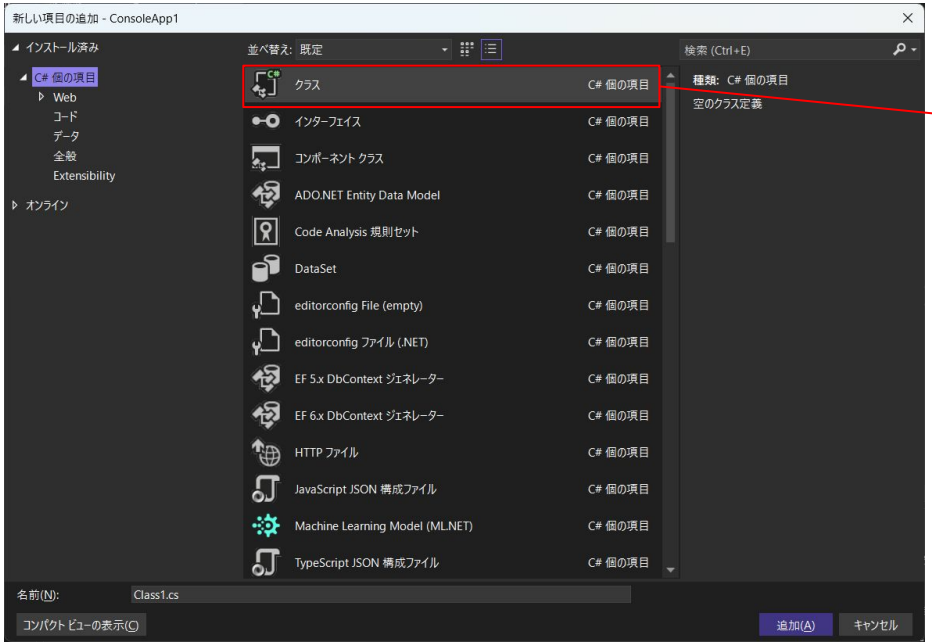
同じ質問をしたのに
毎回違うコードになる！

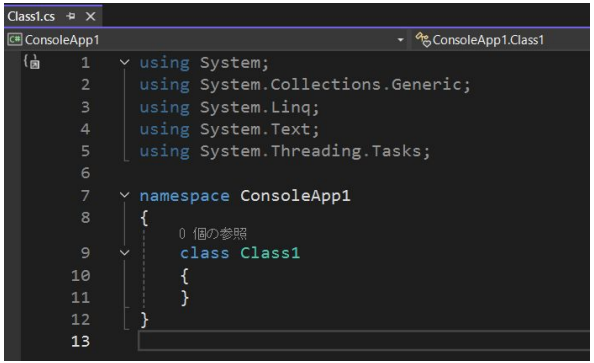
AIを使わない静的コード生成 in Visual Studio

- 項目テンプレート
- プロジェクトテンプレート
- T4テンプレート
  - デザイン時
  - ランタイム（もう使わない）
- Source Generator

項目テンプレート

慣れれば1～2時間程度





追加されるファイルに
毎回必要な内容を
設定しておける
・継承関係
・コンストラクタ引数など

項目テンプレート

慣れれば1～2時間程度

活用したいケース

- 画面など、同じようなものを　繰り返し作る場合
- 継承するクラスや実装するインターフェイスが決まっている場合
- 必ず実装するメソッドがある場合
- 実装手順をコメントに記述して　おきたい場合

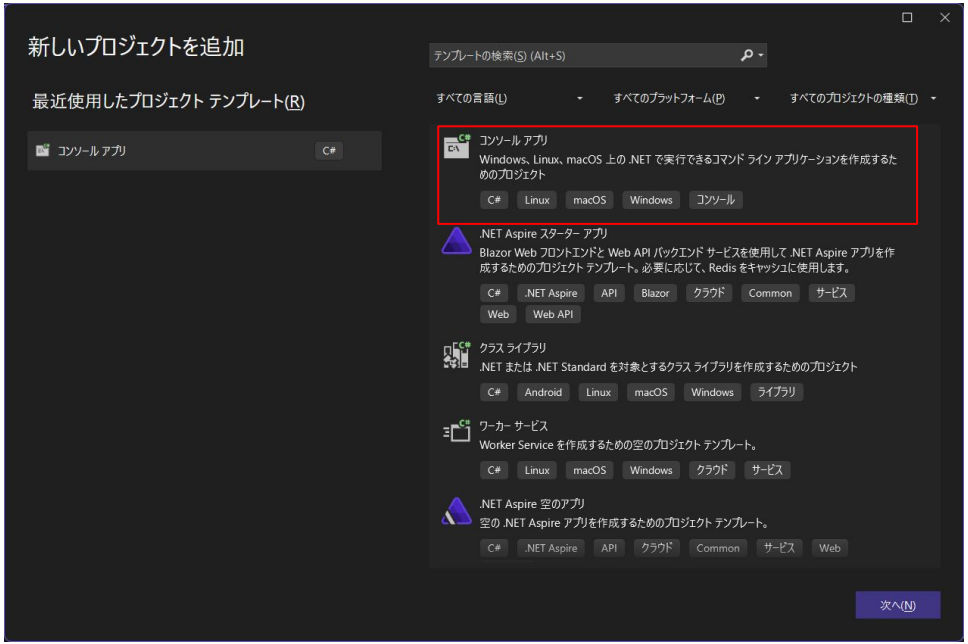
できること

- 追加されるファイルの内容を自由に設定可能
- 複数ファイルの一括追加
- 新規ファイルへのパラメータ埋め込み

詳しくはMSDNを参照
https://learn.microsoft.com/ja-jp/visualstudio/ide/how-to-create-item-templates?view=vs-2022

プロジェクトテンプレート

慣れれば1～2時間程度

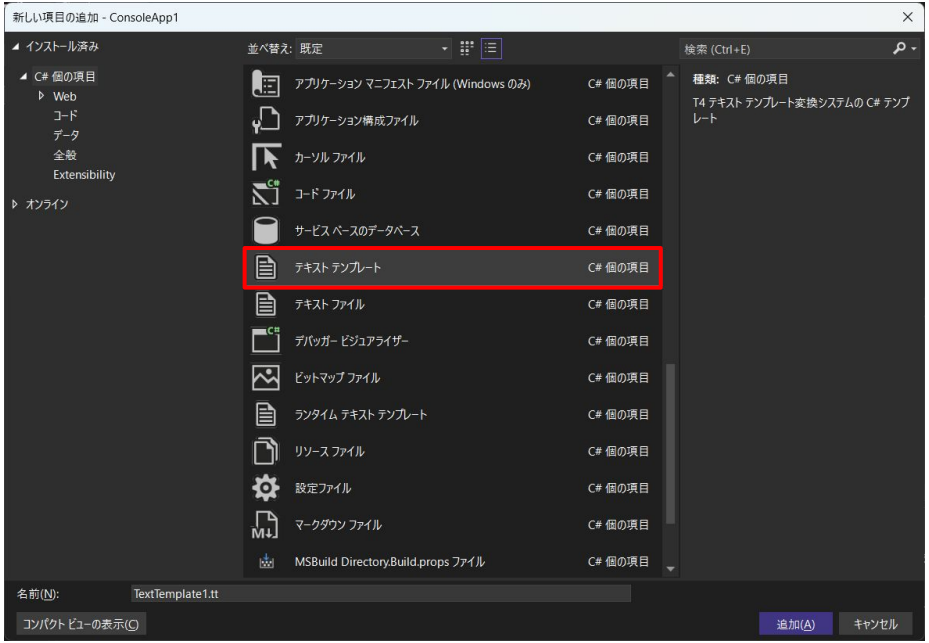


活用例

- 参照すべきプロジェクトやNugetパッケージが決まっている場合
- フォルダ構成がややこしい場合
- 必ず含めたいファイルがある場合

デザイン時T4テンプレート

安定まで3日程度



Visual Studioに埋め込まれた
コード生成ツール

- テンプレート変更時、自動でコードを生成する
- 生成されたコードはファイルとして存在するため変更も(一応)可能
  ＊再作成すると変更が消えるので非推奨
- ブレークポイントも置ける

デザイン時T4テンプレート

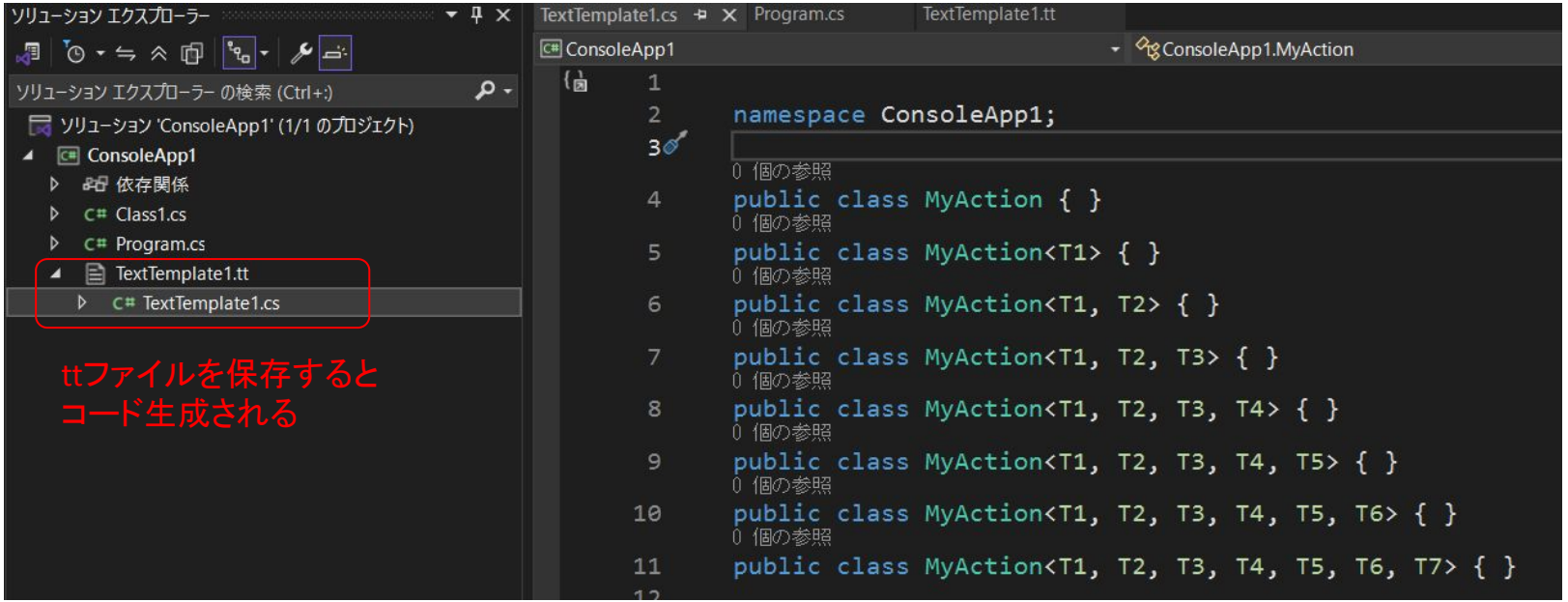
安定まで3日程度

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ output extension=".cs" #>  出力ファイルの拡張子

namespace ConsoleApp1;

public class MyAction { }
<# for(int i = 1; i < 8; i++) { #>  T4テンプレートの制御構文
public class MyAction<<#= string.Join(", ", Enumerable.Range(1, i).Select(x => $"T{x}")) #>> { }
<# } #>
```

※ T4テンプレートは Visual Studioでシンタックスハイライトされないため Carbonで色付け

デザイン時T4テンプレート

安定まで3日程度



活用例：メッセージを定義したファイルから定数化

```
MessageBox.ShowDialog(MessageHelper.GetMessage("I001"));
```

同じメッセージを出す場所は　全ファイルを全文検索しないとわからない

誤って存在しないIDを指定するなど間違えやすい

定義から消した場合実行するまでエラーにならない

このメッセージが何なのか正しいメッセージなのか定義ファイルを見ないとわからない

```
[
  {
    "Id": "I001",
    "Body": "登録しました。"
  },
  {
    "Id": "W001",
    "Body": "編集中です。破棄してもよろしいですか？"
  },
  {
    "Id": "W002",
    "Body": "破棄しました。"
  }
]
```

活用例：メッセージを定義したファイルから定数化

1行目で hostspecific=true にすると
同一ソリューションにあるファイルを
T4テンプレートから読み込んで処理できる

```json
[
  {
    "Id": "I001",
    "Body": "登録しました。"
  },
  {
    "Id": "W001",
    "Body": "編集中です。破棄してもよろしいですか？"
  },
  {
    "Id": "W002",
    "Body": "破棄しました。"
  }
]
```

```
<#@ template debug="false" hostspecific="true" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="System.Text.Json" #>
<#@ assembly name="System.Memory" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Text.Json" #>
<#@ import namespace="System.Text.Json.Serialization" #>
<#@ import namespace="System.Text.Json.Nodes" #>
<#@ output extension=".cs" #>
<#
    var jsonFilePath = this.Host.ResolvePath("Messages.json");
    var jsonContent = File.ReadAllText(jsonFilePath);
    var data = JsonSerializer.Deserialize<JsonNode>(jsonContent).AsArray();
#>
namespace ConsoleApp1;
public static class Messages
{
<# for (int i = 0; i < data.Count; i++) {
    var id = data[i]["Id"];
    var body = data[i]["Body"];
#>
    /// <summary><#= body #></summary>
    public static string <#= id #> => "<#= body #>";

<# } #>
}
```

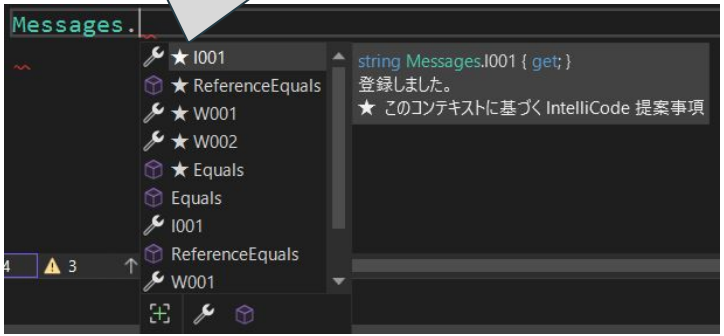
活用例：メッセージを定義したファイルから定数化

```
1   namespace ConsoleApp1;
2   public static class Messages
3   {
4       /// <summary>登録しました。</summary>
5       public static string I001 => "登録しました。";
6
7       /// <summary>編集中です。破棄してもよろしいですか？</summary>
8       public static string W001 => "編集中です。破棄してもよろしいですか？";
9
10      /// <summary>破棄しました。</summary>
11      public static string W002 => "破棄しました。";
12
13  }
14
```

定義から消えた場合
定数も消えるので
コンパイルエラー

※ 定義ファイル変更時は再出力の操作が必要

定数化されていれば
IntelliSenseから「選ぶ」の
で間違えにくい



同じメッセージを出すところは
「参照を検索」ですぐわかる

```
MessageBox.ShowDialog(Messages.I001);
```

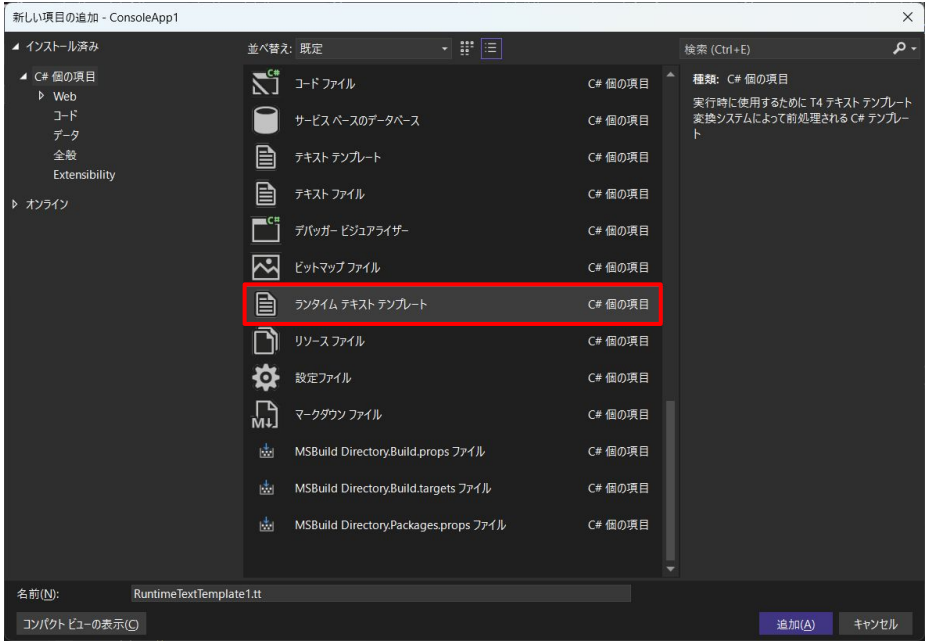
string Messages.I001 { get; }
登録しました。

ここでは、'I001' は null ではありません。

XMLドキュメントにより
内容がすぐわかる

ランタイムT4テンプレート

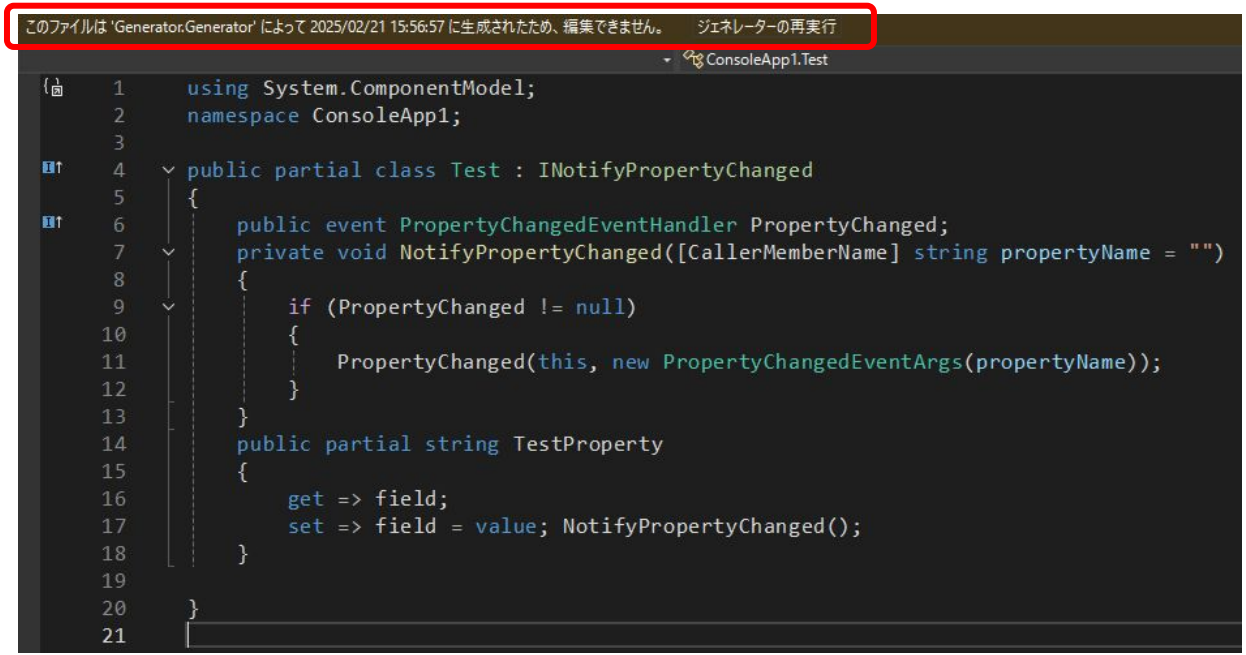
こいつはもう死んだ！
使う必要なし



「文字列を作るクラス」
を作るテンプレート
クラスなのでnewして使う

C# 11から**生文字リテラル** が
使えるようになった
あえてT4を使う理由はない

Source Generator 安定まで3～5日程度



静的なコードをコンパイル時に自動生成する

- 元コードを編集するとすぐ反映
- ファイルとしては存在しない
- 編集不可能
- ブレークポイントは置ける

Source Generator 安定まで3～5日程度

```
1   using Generator;
2   namespace ConsoleApp1;
3
4   [PropertyChanged]
5   public partial class Test
6   {
7       public partial string TestProperty { get; set; }
8   }
9
```

生成条件に合致するものをコンパイル時に検出してソースコード生成

実装方法が決まっている場合に、それを自動実装する

例：

- 通信前に全必須項目の nullをチェックする
- SQL実行結果を列名と同名のプロパティにマップする
- プロパティ変更時にイベント送出したりログ出力したり
- コピーコンストラクタ実装
- ログ出力やデバッグ用の ToString実装 …など

今までコピペして作ってきた
頭を使わない面倒くさいコード
がすべて自動実装できる

Source Generator 安定まで3～5日程度

作るのは**そこそこ**大変 （というかかなり面倒）

1. 生成結果のコードをべたに実装し、方向性を確認しておく
2. Generatorプロジェクトを TargetFramework=.net standard 2.0 で作る
3. IsRoslynComponent=true 設定を追加
4. Generatorを使用する側はAnaylzerプロジェクトとして参照する
5. SourceGeneratorのデバッグのためにlaunchSettings.jsonを設定
6. 自動生成対象を見つけるためのルールを確定させる
7. ルールに基づいて**構文解析をC#で実装する**
8. 構文解析結果をもとに生成ソースコードを文字列で作る
9. Generatorプロジェクトを修正するたびにVisual Studioを再起動する
10. 生成されたコードが想定通り動作するかユニットテスト実装する

```
// ソース自動生成の対象を「特定の属性を持つもの」として定義
var source = context.SyntaxProvider.ForAttributeWithMetadataName(
    "Generator.PropertyChangedAttribute", // 属性の正式名称
    static (node, token) => true, // 属性に加えてさらに条件があるかどうか
    static (context, token) => context);

context.RegisterSourceOutput(source, Generate);
```

```
private static void Generate(SourceProductionContext context, GeneratorAttributeSyntaxContext source)
{
    var typeSymbol = (INamedTypeSymbol)source.TargetSymbol;
    var ns = typeSymbol.ContainingNamespace.IsGlobalNamespace ? ""
        : $"namespace {typeSymbol.ContainingNamespace};";

    // 対象のプロパティを構文解析で引っこ抜く
    var props = typeSymbol.GetMembers()
        .Where(x => x is IPropertySymbol
        {
            IsStatic: false, // 非static
            DeclaredAccessibility: Accessibility.Public, // public以外は対象外
            IsPartialDefinition: true, // partial でないと部分実装できない
            // public get, private set とかは認めない
            SetMethod.DeclaredAccessibility: Accessibility.Public,
            GetMethod.DeclaredAccessibility: Accessibility.Public,
        })
        .Cast<IPropertySymbol>()
        .Select(x => new Data(x.Type.ToString(), x.Name)).ToArray();
    var code = Create(ns, typeSymbol.Name, props);
    context.AddSource($"{typeSymbol.Name}.Generator.g.cs", code);
}
```

コード生成のユニットテスト

T4テンプレート

→テストしない

- テンプレートを変更すると テスト対象が消えるかもしれないため
- 親クラス等をテストして、生成コードは最小限にする

SourceGenerator

→専用クラスを作ってテスト

- テストプロジェクトからGeneratorプロジェクトを参照
- テスト用クラスをテストプロジェクト内で作り、そのクラスの自動生成結果を検証する

まとめ

1. 自動生成を活用するとコピペを減らせる
   - 項目テンプレート
   - デザイン時T4テンプレート
   - SourceGenerator
2. コピペを減らせると開発効率があがる
3. 開発効率があがると早く帰れる
4. **早く帰れるとでび様の配信をムに見られる(重要)**

Visual Studioを活用して
ソースコードを自動生成しよう！！

備考：VS Codeでも動くの？

Visual Studioを使えるのに
VS Codeを使ってる人