

Réflexivité dans Unity et Post Processor

Dans Unity, nous avons la possibilité d'utiliser la classe **AssetPostProcessor** pour modifier les Assets importés. On peut automatiser des modifications spécifiques à exécuter sur les Assets avant ou après son importation mais la difficulté majeure est de réécrire un Script pour tout nouvel Asset ce qui peut devenir très fastidieux. Il serait très utile de pouvoir les gérer à un seul endroit, et pour réaliser cette tâche nous aurons besoin de la Réflexivité.

La réflexivité est un outil puissant en C# et dans d'autres langages comme Java, Python ou C++ : Son principe est de permettre d'examiner et manipuler son propre code pendant l'exécution. On peut interroger les types, les méthodes et les propriétés à l'intérieur du programme et de les manipuler dynamiquement.

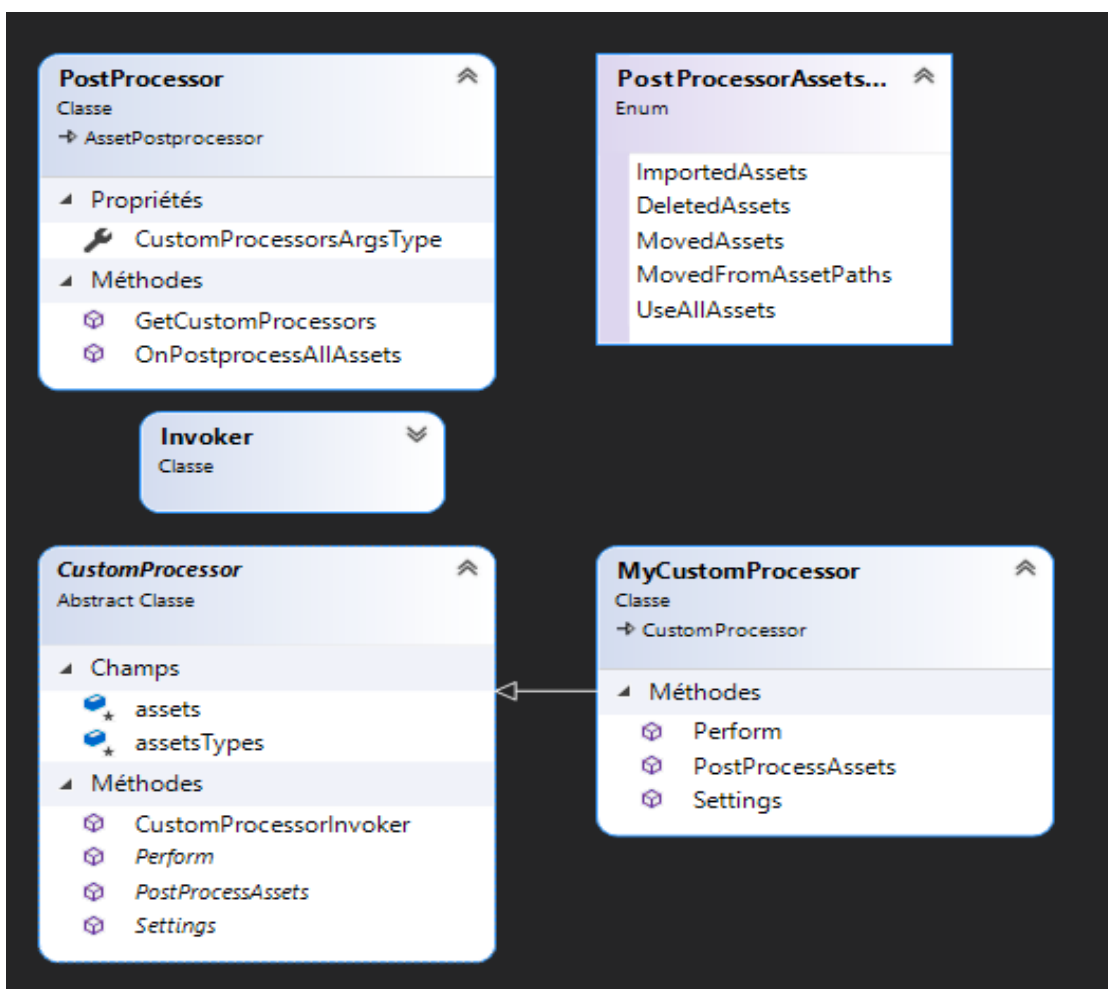
La réflexivité en C# permet, en récupérant les métadonnées contenues dans les Assemblies, de récupérer les informations sur les types et les membres d'un programme.

L'idée est la suivante: nous allons rediriger les appels Unity Aux PostProcessors qui propose plusieurs Fonctions bien utiles à un seul endroit et utiliser la réflexivité pour avoir autant de script que l'on désire.

Les Fonctions des AssetProcessors:

- **OnPreprocessAsset()** appelé avant que l'asset ne soit importé dans le projet.
- **OnPostprocessAsset()** appelé après que l'asset a été importé dans le projet.
- **OnPostprocessAllAssets()** appelé après que tous les assets aient été importés dans le projet.

notez qu'il est possible récupérer les autres appels déclenchés, mais nous allons nous concentrer sur ce **OnPostprocessAllAssets** puisqu'il se déclenche après l'importation et ce pour tous les Assets ajoutés, déplacés, modifiés ou supprimés dans Unity.



Notre **PostProcesseur** principal va appeler tous les **CustomProcessors** que nous allons rajouter, et ce de manière dynamique. Plus besoin de vérifier un à un les Script, et une maintenance facilitée du code.

Pour nous aider, dans **System.Reflection**, la classe **Invoker** va nous permettre d'appeler des méthodes sur des objets en utilisant la réflexivité, d'appeler dynamiquement les méthodes mais aussi d'appeler les constructeurs les classes voulues.

```
public class PostProcessor : AssetPostprocessor
{
    public static Type[] CustomProcessorsArgsType => new Type[] { typeof(string[]), typeof(string[]), typeof(string[]), typeof(string[]) };

    public static void OnPostprocessAllAssets(string[] importedAssets, string[] deletedAssets,
        string[] movedAssets, string[] movedFromAssetPaths, bool didDomainReload)
    {
        Type[] customProcessors = GetCustomProcessors();

        foreach (Type type in customProcessors)
        {
            CustomProcessor instance = Activator.CreateInstance(type) as CustomProcessor;
            MethodInfo method = type.GetMethod("CustomProcessorInvoker", CustomProcessorsArgsType);

            if (method != null)
            {
                method.Invoke(instance, new object[] { importedAssets, deletedAssets, movedAssets, movedFromAssetPaths });
            }
        }
    }

    public static Type[] GetCustomProcessors()
    {
        Type[] classes = Assembly.GetExecutingAssembly().GetTypes();
        return classes.Where(t => t.Namespace == "AssetPostProcessors.CustomProcessors" && t.IsClass && t.IsPublic).ToArray();
    }
}
```

GetCustomProcessors récupère toutes les classes d'un Sous Domaine particulier ici **AssetPostProcessors.CustomProcessors** ou sont rassemblées les Classes dérivées.

Nous Instancions ensuite les classes présentes dans **AssetPostProcessors.CustomProcessors** (La classe abstraite **CustomProcessor** n'en fera pas partie, seul les classes filles seront ainsi instanciées).

Il nous faudra aussi récupérer la fonction voulue (**MethodInfo**) via **GetMethod()**.

L'objet **CustomProcessorsArgType** renvoie un Type représentant la signature de la méthode qui sera Invoquée. Ce type correspond à la signature du **()**, nous en avons besoin pour que Chaque classe récupère la liste des Assets modifiés.

La classe mère des Custom Processor est simple: **CustomProcessorInvoker()** sera appelé via **Invoke()** dans PostProcessor, le reste sera implémenté dans les classes filles.

```
public abstract class CustomProcessor
{
    protected List<string> assets = new();
    protected PostProcessorAssetsTypes assetsTypes;

    public void CustomProcessorInvoker(string[] importedAssets, string[] deletedAssets,
        string[] movedAssets, string[] movedFromAssetPaths)
    {
        Settings();

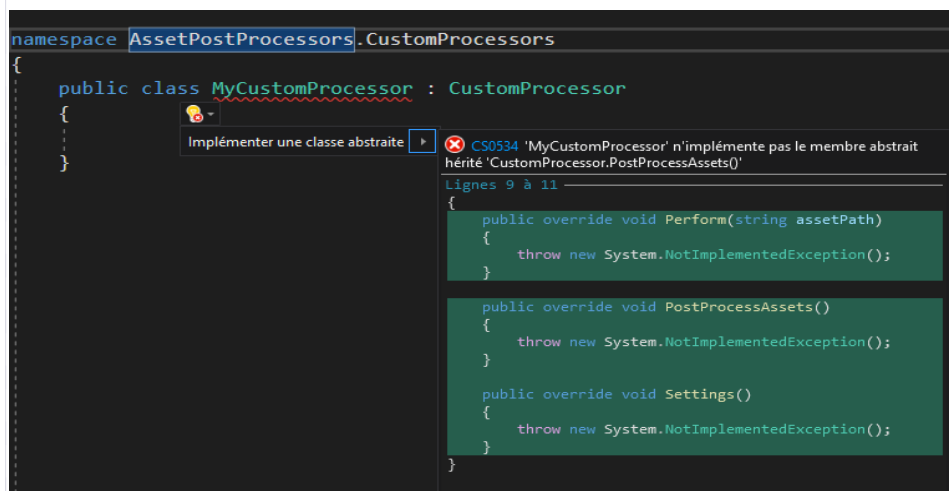
        if (assetsTypes.HasFlag(PostProcessorAssetsTypes.ImportedAssets)) assets.AddRange(importedAssets);
        if (assetsTypes.HasFlag(PostProcessorAssetsTypes.DeletedAssets)) assets.AddRange(deletedAssets);
        if (assetsTypes.HasFlag(PostProcessorAssetsTypes.MovedAssets)) assets.AddRange(movedAssets);
        if (assetsTypes.HasFlag(PostProcessorAssetsTypes.MovedFromAssetPaths)) assets.AddRange(movedFromAssetPaths);

        PostProcessAssets();
    }

    public abstract void Settings();
    public abstract void PostProcessAssets();
    public abstract void Perform(string assetPath);
}
```

Enfin pour Ajouter un Custom Processor rien de plus simple.

Il suffit de créer une nouvelle classe dérivant de **CustomProcessor** clic-droit sur le nom de la classe pour l'implémenter.



```
namespace AssetPostProcessors.CustomProcessors
{
    public class MyCustomProcessor : CustomProcessor
    {
    }
}
```

CS0534 'MyCustomProcessor' n'implémente pas le membre abstrait hérité 'CustomProcessor.PostProcessAssets()'

Lignes 9 à 11

```
{
    public override void Perform(string assetPath)
    {
        throw new NotImplementedException();
    }

    public override void PostProcessAssets()
    {
        throw new NotImplementedException();
    }

    public override void Settings()
    {
        throw new NotImplementedException();
    }
}
```

- **Settings()** est ici utilisé pour modifier assetTypes (c'est optionnel bien entendu nous pouvons plus simplement modifier les variables directement, l'utilité est d'avoir pour plus tard une méthode que l'on peut modifier à loisir).
- **PostProcessAsset()** permet de parcourir les fichiers (et plus tard nous pourrons filtrer ceux-ci).
- **Perform()** fera le traitement pour chaque fichier. Pour l'exemple nous allons juste les afficher sur la console.

```
public class MyCustomProcessor : CustomProcessor
{
    public override void Settings() => assetsTypes = PostProcessorAssetsTypes.ImportedAssets;

    public override void PostProcessAssets()
    {
        foreach (string assetPath in assets)
            Perform(assetPath);
    }

    public override void Perform(string assetPath)
    {
        Debug.Log("File found :" + assetPath);
    }
}
```

Remarques :

Même si la réflexivité est un outil puissant notez qu'il est plutôt destiné à être utilisé Dans l'Editor uniquement ou au chargement par exemple quand la situation le permet.

La classe Abstraite CustomProcessor sera amené à être améliorée: PostProcessAsset est ici exposée mais dans l'idéal nous voudrions que celle-ci ne le soit pas: le traitement de la liste d'assets est la responsabilité de cette Classe et devrait gérer un système de filtrage pour que seul Perform() ne soit à implémenter dans les divers Scripts. Par exemple: filtrer seulement les images pour changer leur DPI dès qu'un utilisateur en ajoute dans Unity.