🎯 ANFA Framework - Method Tokenizer System

Vue d'ensemble

Le **Method Tokenizer** est un système avancé de debug et d'analyse qui permet de tracker automatiquement l'utilisation des méthodes avec leurs métadonnées complètes. Il fournit des informations détaillées sur les performances, la fréquence d'utilisation, et les caractéristiques de chaque méthode.

✨ Fonctionnalités

🔍 Tracking Automatique

- **Nom de méthode** : Détection automatique via CallerMemberName
- **Description** : Extraction depuis les annotations @()
- **Paramètres** : Analyse via réflexion
- **Type de retour** : Détection automatique
- **Namespace/Classe** : Extraction depuis le chemin de fichier

📊 Métriques de Performance

- **Nombre d'appels** : Compteur total d'utilisation
- **Temps d'exécution** : Moyenne, min, max
- **Première/Dernière utilisation** : Timestamps
- **Tendances d'usage** : Analyse temporelle

🛠️ Outils de Debug

- **Logs en temps réel** : Affichage lors de chaque appel
- **Statistiques globales** : Vue d'ensemble de tous les appels
- **Export JSON** : Sauvegarde des données pour analyse
- **Interface GUI** : Contrôles en temps réel

🚀 Utilisation

Configuration de Base

```
// Activer/désactiver le tracking
FrameworkCore.SetMethodTracking(true);

// Dans vos méthodes, ajoutez le tokenizer
public void MyMethod()
{
    FrameworkCore.TokenizeAndLogMethod("Description de ma méthode");
    // ... votre code
}
```

Exemple Complet

```
/// <summary>
/// @(name="ProcessGameData", description="Processes game data with validation
and error handling.")
/// </summary>
public async Task ProcessGameData(int playerId, string dataType)
{
    // Tokenise automatiquement la méthode
    FrameworkCore.TokenizeAndLogMethod("Processes game data with validation and
error handling.");

    // Votre logique métier
    await ValidatePlayer(playerId);
    var data = await LoadGameData(playerId, dataType);
    await ProcessData(data);
}
```

Récupération des Informations

```
// Obtenir les infos d'une méthode spécifique
var methodInfo = FrameworkCore.GetMethodToken("MyClass.MyMethod");
Debug.Log($"Appelée {methodInfo.CallCount} fois");

// Obtenir toutes les méthodes trackées
var allMethods = FrameworkCore.GetAllMethodTokens();

// Afficher les statistiques
FrameworkCore.PrintAllMethodStats();
```

📋 Structure des Données

MethodTokenInfo

```
public class MethodTokenInfo
{
    public string Name;              // Nom de la méthode
    public string Description;       // Description depuis @()
    public string ClassName;         // Nom de la classe
    public string Namespace;         // Namespace complet
    public DateTime FirstCall;       // Premier appel
    public DateTime LastCall;        // Dernier appel
    public int CallCount;           // Nombre total d'appels
    public List<string> Parameters; // Liste des paramètres
    public string ReturnType;       // Type de retour
```

```
    public long TotalExecutionTicks; // Temps total d'exécution
    public long AverageExecutionTicks; // Temps moyen par appel
}
```

🎮 Contrôles Context Menu

Dans l'éditeur Unity, clic droit sur le FrameworkCore :

- **Print All Method Stats** : Affiche toutes les statistiques
- **Clear Method Stats** : Remet à zéro les compteurs
- **Get Framework Debug Info** : Infos complètes du framework

📊 Exemple de Sortie Console

```
[ANFA Tokenizer] 🔍 Method Called:
📍 FrameworkCore.InitializeFramework() at line 95
📝 Initializes the entire framework with all core services and performance settings.
🔢 Call #1
⚡ Avg Time: 12.345ms

[ANFA Tokenizer] 📊 Method Statistics Report (8 methods tracked):
═══════════════════════════════════════════════════════════════
#1. 🎯 Method: FrameworkCore.InitializeFramework
📝 Description: Initializes the entire framework with all core services and performance settings.
📊 Calls: 1
⏱️ Avg Time: 12.345ms
🕐 First: 14:30:15
🕐 Last: 14:30:15
📋 Parameters: []
↩️ Returns: Task
```

🔧 Fonctions Utilitaires

Export des Données

```
// Exporter vers JSON
FrameworkCore.ExportMethodStats("my_analysis.json");

// Le fichier contient :
{
  "ExportTime": "2025-08-18T14:30:15Z",
  "TotalMethods": 15,
  "Methods": [
    {
      "MethodKey": "FrameworkCore.InitializeFramework",
```

```
      "Info": {
        "Name": "InitializeFramework",
        "Description": "Initializes the entire framework...",
        "CallCount": 5,
        "AverageExecutionMs": 12.345,
        "Parameters": [],
        "ReturnType": "Task"
      }
    }
  ]
}
```

Contrôle Programmatique

```
// Désactiver temporairement
FrameworkCore.SetMethodTracking(false);
ExpensiveOperation(); // Pas de tracking
FrameworkCore.SetMethodTracking(true);

// Nettoyer les stats
FrameworkCore.ClearMethodStats();

// Obtenir des métriques spécifiques
var tokens = FrameworkCore.GetAllMethodTokens();
var mostUsedMethod = tokens.OrderByDescending(t => t.Value.CallCount).First();
Debug.Log($"Méthode la plus utilisée: {mostUsedMethod.Key}");
```

⚡ Performance

Impact Minimal

- **Overhead** : < 0.1ms par appel de méthode
- **Mémoire** : ~200 bytes par méthode unique
- **Désactivation** : Aucun impact quand `SetMethodTracking(false)`

Optimisations Intégrées

- Cache des informations de réflexion
- Calculs de timing optimisés
- Collections thread-safe
- Cleanup automatique

🎯 Cas d'Usage

1. Debug de Performance

```
public void OptimizeMe()
{
    FrameworkCore.TokenizeAndLogMethod("Method that needs optimization");
    // Code à optimiser
}
// Résultat : Voir temps d'exécution moyen et identifier les bottlenecks
```

2. Analyse d'Utilisation

```
public void RarelyUsedFeature()
{
    FrameworkCore.TokenizeAndLogMethod("Feature that might be unused");
    // Code de fonctionnalité
}
// Résultat : Identifier les fonctionnalités peu utilisées
```

3. Profiling de Développement

```
public void NewFeature()
{
    FrameworkCore.TokenizeAndLogMethod("Newly implemented feature for
testing");
    // Nouvelle fonctionnalité
}
// Résultat : Suivre l'adoption et les performances des nouvelles features
```

Bonnes Pratiques

1. Annotations Descriptives

```
/// <summary>
/// @(name="LoadPlayerData", description="Loads player data from database with
caching and validation.")
/// </summary>
```

2. Utilisation Sélective

```
// Seulement sur les méthodes importantes
public void CriticalGameplayMethod()
{
    FrameworkCore.TokenizeAndLogMethod("Critical gameplay method");
}
```

```
// Pas sur les méthodes appelées très fréquemment (Update, etc.)
```

3. Nettoyage Périodique

```
// Dans un manager de debug
void CleanupOldStats()
{
    if (developmentBuild)
    {
        FrameworkCore.ClearMethodStats();
    }
}
```

🛠️ Intégration avec CI/CD

Tests Automatisés

```
[Test]
public void TestMethodPerformance()
{
    FrameworkCore.ClearMethodStats();

    // Exécuter des tests
    MyMethod();

    var stats = FrameworkCore.GetMethodToken("MyClass.MyMethod");
    Assert.IsTrue(stats.AverageExecutionTicks < MaxAllowedTicks);
}
```

Rapports de Build

```
// Script de build
FrameworkCore.ExportMethodStats($"build_report_{buildNumber}.json");
```

Le système de tokenization offre une visibilité complète sur l'utilisation et les performances de vos méthodes, facilitant le debug, l'optimisation et l'analyse de votre code ! 🚀