TP6 : Principe de Responsabilité Unique (SRP)

Objectif

Comprendre et appliquer le principe de Responsabilité Unique (Single Responsibility Principle), le premier des principes SOLID.

Rappel du Principe

"Une classe ne doit avoir qu'une seule raison de changer."

Ce principe stipule qu'une classe ne devrait avoir qu'une seule responsabilité, c'est-à-dire qu'elle ne devrait s'occuper que d'une seule partie de la fonctionnalité du logiciel. Si une classe a plusieurs responsabilités, elle devient couplée à ces responsabilités, ce qui la rend moins robuste et plus difficile à maintenir.

Problème

Le code suivant représente un gestionnaire de sauvegarde pour un jeu. Il est responsable de :

1. Sérialiser/désérialiser les données
2. Sauvegarder/charger les fichiers sur le disque
3. Chiffrer/déchiffrer les données
4. Gérer les erreurs et afficher des messages
5. Compresser/décompresser les données

```
// SaveManager.cs - Code problématique
using System;
using System.IO;
using System.Text;
using System.Security.Cryptography;
using UnityEngine;
using System.IO.Compression;

public class SaveManager : MonoBehaviour
{
    [SerializeField] private string saveFolderName = "SaveData";
    [SerializeField] private string encryptionKey = "GameSaveEncryptKey";
    [SerializeField] private bool useCompression = true;
    [SerializeField] private bool useEncryption = true;

    // Référence à l'UI pour afficher les messages
    [SerializeField] private GameObject saveLoadMessagePanel;
    [SerializeField] private UnityEngine.UI.Text messageText;

    private string SaveFolderPath => Path.Combine(Application.persistentDataPath,
saveFolderName);

    // Données du joueur (normalement cette classe serait séparée)
    [Serializable]
```

```
public class PlayerData
{
    public string playerName;
    public int level;
    public float health;
    public Vector3 position;
    public Quaternion rotation;
    public int[] inventory;
    public int currency;
    // etc.
}

private void Start()
{
    // Créer le dossier de sauvegarde s'il n'existe pas
    if (!Directory.Exists(SaveFolderPath))
    {
        Directory.CreateDirectory(SaveFolderPath);
    }
}

// Sauvegarde les données du joueur
public void SaveGame(PlayerData playerData, string fileName)
{
    try
    {
        // Sérialisation des données en JSON
        string jsonData = JsonUtility.ToJson(playerData, true);
        byte[] rawData = Encoding.UTF8.GetBytes(jsonData);

        // Compression des données si activée
        if (useCompression)
        {
            rawData = CompressData(rawData);
        }

        // Chiffrement des données si activé
        if (useEncryption)
        {
            rawData = EncryptData(rawData);
        }

        // Sauvegarde dans un fichier
        string filePath = Path.Combine(SaveFolderPath, fileName + ".sav");
        File.WriteAllBytes(filePath, rawData);

        // Affiche un message de réussite
        ShowMessage($"Sauvegarde réussie dans {fileName}.sav");
        Debug.Log($"Données sauvegardées dans {filePath}");
    }
    catch (Exception e)
    {
        // Gestion des erreurs
        ShowMessage($"Erreur lors de la sauvegarde : {e.Message}");
```

```
            Debug.LogError($"Erreur lors de la sauvegarde : {e.Message}");
        }
    }

    // Charge les données du joueur
    public PlayerData LoadGame(string fileName)
    {
        try
        {
            string filePath = Path.Combine(SaveFolderPath, fileName + ".sav");

            if (!File.Exists(filePath))
            {
                ShowMessage($"Fichier de sauvegarde {fileName}.sav introuvable");
                Debug.LogWarning($"Fichier de sauvegarde introuvable:
{filePath}");
                return null;
            }

            // Lecture du fichier
            byte[] rawData = File.ReadAllBytes(filePath);

            // Déchiffrement des données si nécessaire
            if (useEncryption)
            {
                rawData = DecryptData(rawData);
            }

            // Décompression des données si nécessaire
            if (useCompression)
            {
                rawData = DecompressData(rawData);
            }

            // Désérialisation des données
            string jsonData = Encoding.UTF8.GetString(rawData);
            PlayerData playerData = JsonUtility.FromJson<PlayerData>(jsonData);

            // Affiche un message de réussite
            ShowMessage($"Chargement réussi de {fileName}.sav");
            Debug.Log($"Données chargées depuis {filePath}");

            return playerData;
        }
        catch (Exception e)
        {
            // Gestion des erreurs
            ShowMessage($"Erreur lors du chargement : {e.Message}");
            Debug.LogError($"Erreur lors du chargement : {e.Message}");
            return null;
        }
    }

    // Supprime un fichier de sauvegarde
```

```
    public bool DeleteSaveFile(string fileName)
    {
        try
        {
            string filePath = Path.Combine(SaveFolderPath, fileName + ".sav");

            if (!File.Exists(filePath))
            {
                ShowMessage($"Fichier de sauvegarde {fileName}.sav introuvable");
                return false;
            }

            File.Delete(filePath);
            ShowMessage($"Sauvegarde {fileName}.sav supprimée");
            return true;
        }
        catch (Exception e)
        {
            ShowMessage($"Erreur lors de la suppression : {e.Message}");
            Debug.LogError($"Erreur lors de la suppression : {e.Message}");
            return false;
        }
    }

    // Liste toutes les sauvegardes disponibles
    public string[] ListSaveFiles()
    {
        try
        {
            if (!Directory.Exists(SaveFolderPath))
            {
                return new string[0];
            }

            string[] filePaths = Directory.GetFiles(SaveFolderPath, "*.sav");

            // Convertir les chemins complets en noms de fichiers sans extension
            for (int i = 0; i < filePaths.Length; i++)
            {
                filePaths[i] = Path.GetFileNameWithoutExtension(filePaths[i]);
            }

            return filePaths;
        }
        catch (Exception e)
        {
            Debug.LogError($"Erreur lors de la liste des sauvegardes :
{e.Message}");
            return new string[0];
        }
    }

    #region Compression
    // Compresse les données
```

```
    private byte[] CompressData(byte[] data)
    {
        using (MemoryStream output = new MemoryStream())
        {
            using (GZipStream gzip = new GZipStream(output,
CompressionMode.Compress))
            {
                gzip.Write(data, 0, data.Length);
            }
            return output.ToArray();
        }
    }

    // Décompresse les données
    private byte[] DecompressData(byte[] data)
    {
        using (MemoryStream input = new MemoryStream(data))
        {
            using (GZipStream gzip = new GZipStream(input,
CompressionMode.Decompress))
            {
                using (MemoryStream output = new MemoryStream())
                {
                    gzip.CopyTo(output);
                    return output.ToArray();
                }
            }
        }
    }
    #endregion

    #region Encryption
    // Chiffre les données
    private byte[] EncryptData(byte[] data)
    {
        using (Aes aes = Aes.Create())
        {
            byte[] key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(16,
'*').Substring(0, 16));
            aes.Key = key;
            aes.IV = new byte[16]; // IV simplifié pour l'exemple

            using (ICryptoTransform encryptor = aes.CreateEncryptor())
            {
                return PerformCrypto(data, encryptor);
            }
        }
    }

    // Déchiffre les données
    private byte[] DecryptData(byte[] data)
    {
        using (Aes aes = Aes.Create())
        {
```

```
            byte[] key = Encoding.UTF8.GetBytes(encryptionKey.PadRight(16,
'*').Substring(0, 16));
            aes.Key = key;
            aes.IV = new byte[16]; // IV simplifié pour l'exemple

            using (ICryptoTransform decryptor = aes.CreateDecryptor())
            {
                return PerformCrypto(data, decryptor);
            }
        }
    }

    // Effectue la transformation cryptographique
    private byte[] PerformCrypto(byte[] data, ICryptoTransform transform)
    {
        using (MemoryStream ms = new MemoryStream())
        {
            using (CryptoStream cs = new CryptoStream(ms, transform,
CryptoStreamMode.Write))
            {
                cs.Write(data, 0, data.Length);
                cs.FlushFinalBlock();
                return ms.ToArray();
            }
        }
    }
    #endregion

    #region UI Messages
    // Affiche un message à l'utilisateur
    private void ShowMessage(string message)
    {
        if (saveLoadMessagePanel != null && messageText != null)
        {
            messageText.text = message;
            saveLoadMessagePanel.SetActive(true);

            // Masquer le message après 3 secondes
            StopAllCoroutines();
            StartCoroutine(HideMessageAfterDelay(3f));
        }
    }

    // Masque le message après un délai
    private System.Collections.IEnumerator HideMessageAfterDelay(float delay)
    {
        yield return new WaitForSeconds(delay);
        if (saveLoadMessagePanel != null)
        {
            saveLoadMessagePanel.SetActive(false);
        }
    }
    #endregion
}
```

Problèmes avec ce code

1. La classe `SaveManager` a trop de responsabilités
2. Elle gère à la fois la sérialisation, le stockage, la compression, le chiffrement et l'interface utilisateur
3. Si l'une de ces responsabilités doit changer, toute la classe doit être modifiée
4. Le code est difficile à tester unitairement (comment tester uniquement la compression?)
5. La réutilisation du code est limitée (impossible de réutiliser juste le chiffrement)

Exercice

Refactorisez le code en appliquant le principe SRP :

1. Identifiez les différentes responsabilités dans la classe `SaveManager`
2. Créez des classes séparées pour chaque responsabilité
3. Assurez-vous que chaque classe a une seule raison de changer
4. Réorganisez la classe `SaveManager` pour qu'elle coordonne ces classes spécialisées

Avantages attendus

- Code plus facile à maintenir et à faire évoluer
- Chaque partie peut être testée indépendamment
- Les composants peuvent être réutilisés dans d'autres parties du projet
- Les modifications dans une fonctionnalité n'affectent pas les autres

Conseils

- Pensez en termes de responsabilités : un composant ne devrait faire qu'une seule chose, mais la faire bien
- Utilisez la composition pour assembler ces responsabilités
- Créez des interfaces claires entre les composants