C# AVANCÉ

#1 - Design Patterns

Argann BONNEAU - 2023/2024

01 **Un *Design Pattern* ?**

La boite à outils par excellence

02 **Design Pattern & Jeu-vidéo**

Vous en utilisez déjà…

03 **Autres bonnes pratiques**

SOLID, DRY, KISS, et autres acronymes bizarres

TP **TP 1 : Bonnes pratiques**

01

Un *Design Pattern* ?

Comment passer d’un besoin à du code

Un peu d’histoire

À la fin des années 80, le métier de développeur est de plus en plus commun en entreprise.

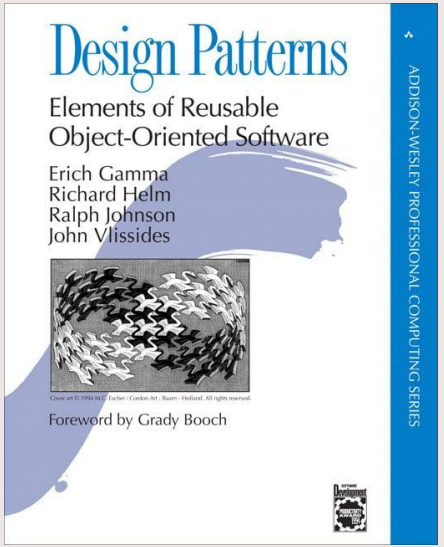
L’avènement de la “programmation orienté objet” apporte un nouveau lot de problématiques plus abstraites, on essaye alors de ne pas juste “coder”, mais “*bien* coder”.

On commence à parler “d’architecture” de code : on construit soigneusement un programme, choisissant avec minutie comment répartir son code tel un.e architecte prévoit la construction d’un bâtiment.

Un peu d'histoire

Pendant un congrès portant sur la "programmation orienté objet", Erich Gamma et Richard Helm se rencontrent. Intéressés par l'architecture logicielle, ils décident de collaborer sur l'écriture d'un recueil de bonnes pratiques. Rejoint ensuite par Ralph Johnson et John Vlissides, ils formeront le *Gang of Four*, et publieront en 1994 un des livres les plus importants sur le sujet :

Design Patterns: Elements of Reusable Object-Oriented Software



Patron de Conception

Un Design Pattern (ou patron de conception) est une **solution** à un problème informatique **courant** considérée comme “**bonne pratique**”.

“Oh non, j’ai plein de route qui se rejoignent à une gigantesque intersection, y’a des accidents, comment je peux faire ?!”

C’est un problème *courant* qui a une solution *courante* :

Ressources

Où trouver ces patrons de conception ?

Un peu partout sur le web, mais je vous conseille :

https://refactoring.guru/design-patterns

Chaque design pattern y est accompagné du problème qu'il résout, de la solution qu'il apporte, de sa structure, d'exemples de code, de conseils et même de ses limites !

Exemple

“OK, j’ai une classe AudioManager qui me permet de gérer les sons de mon jeu.

Je veux être sûr.e qu’il n’existe qu’un seul AudioManager dans ma scène.

Et ça pourrait être cool que cette instance soit accessible de n’importe où facilement…

Mais je ne peux pas rendre ma classe statique, parce que je veux pouvoir l’attacher à un GameObject !”

=> C’est un problème très courant !

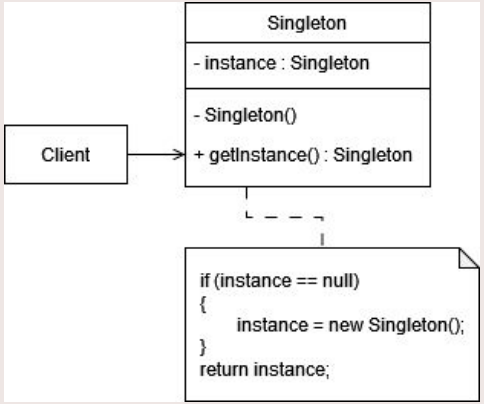
Exemple : Singleton

Il existe un *Design Pattern* parfait pour ce problème : le Singleton.

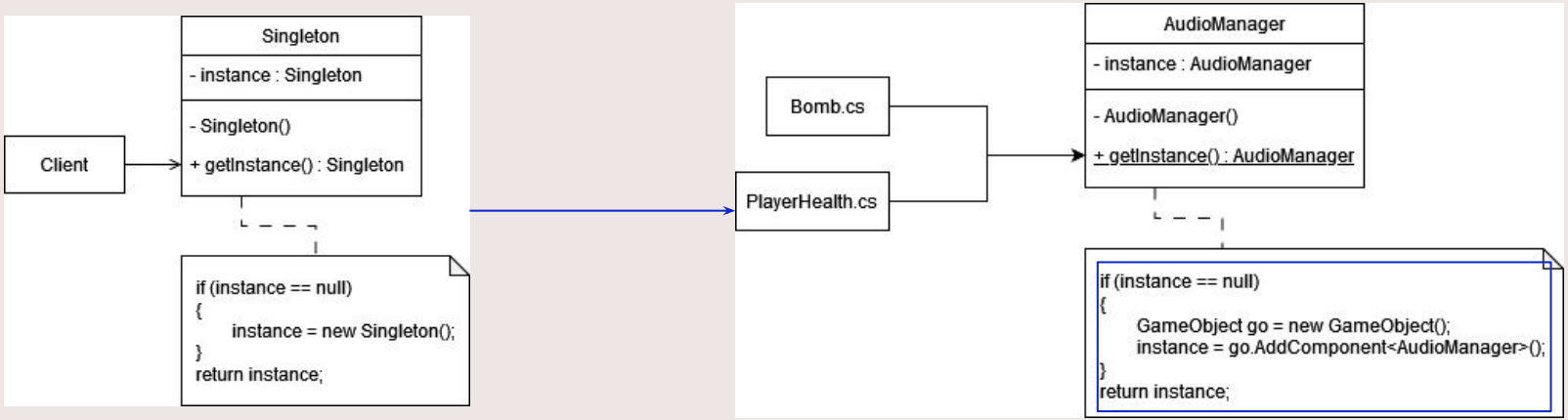
Il permet de nous assurer :

- Qu'une classe n'ai qu'une seule instance ;
- Que cette instance soit accessible globalement ;

Parfait pour nous !



Exemple : Singleton



On adapte le code à Unity !

Exemple : Singleton

Rien ne nous empêche d'adapter encore plus le code aux spécificités de Unity et de C#, par exemple en utilisant une propriété plutôt qu'une méthode 'getInstance' :

```
using UnityEngine;

Script Unity | 3 références
public class AudioManager : MonoBehaviour
{
    private static AudioManager instance;

    0 références
    public static AudioManager Instance
    {
        get
        {
            if (instance == null)
            {
                GameObject go = new GameObject("Audio Manager");
                instance = go.AddComponent<AudioManager>();
            }
            return instance;
        }
    }

    0 références
    public void PlayOuch()
    {
        // On charge le fichier "ouch.mp3", et on le joue.
    }
}
```

Grâce à ça, on a accès facilement à notre instance depuis n'importe quel script:

```
using System.Collections;
using UnityEngine;

Script Unity | 0 références
public class Bomb : MonoBehaviour
{
    Message Unity | 0 références
    IEnumerator Start()
    {
        Debug.Log("Bomb exploding in 10 seconds...");

        yield return new WaitForSeconds(10);

        AudioManager.Instance.PlayOuch();
    }
}
```

Exemple : Singleton

Mais, ce design pattern à ses limites !

- Il nécessite d'être grandement modifié pour fonctionner dans un contexte complètement asynchrone
- Il donne l'accès à l'instance à n'importe qui
- Il rend complexe le fait d'écrire des tests unitaires de la classe

Catégories

Trois catégories de design patterns :

- **Création** : propose différentes façons de créer/instancier des objets suivant les problématiques
- **Structure** : permet d'assembler des classes, des objets, des méthodes pour former des structures plus complexes et puissantes
- **Comportement** : permet d'ajuster nos algorithmes et nos traitements de données pour que tout se passe bien.

Singleton est un pattern de type "Création", vu sa façon de créer l'instance de la classe.

Exemple : Observer

Quel est le problème de ce code ?

```
Script Unity | 1 référence
public class ScoreUI : MonoBehaviour
{
    [SerializeField]
    private TMP_Text _scoreText;

    Message Unity | 0 références
    private void Update()
    {
        _scoreText.text = "" + ScoreManager.Instance.Score;
    }
}
```

Exemple : Observer

Le changement de score est quelque chose de **ponctuel**, actualiser le texte *à chaque frame* ne sert à rien !

Il faudrait pouvoir appeler ce bout de code *uniquement lorsque le score est modifié…*

Exemple : Observer

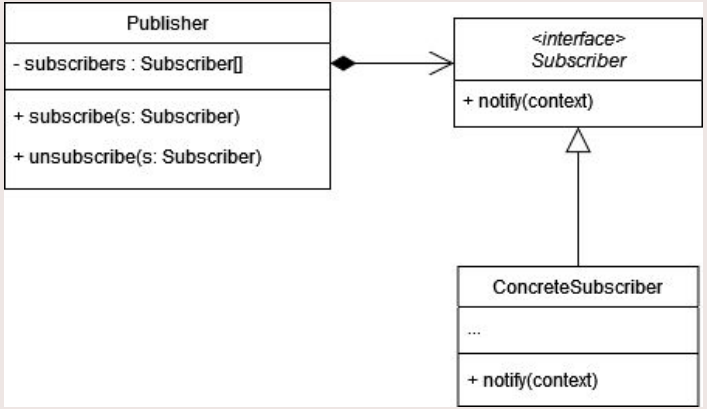
```
Script Unity | 3 références
public class ScoreManager : MonoBehaviour
{
    private static ScoreManager instance;

    0 références
    public static ScoreManager Instance ...

    [SerializeField]
    private ScoreUI _ui;

    2 références
    public int Score { get; private set; }

    0 références
    public void SetScore(int newScore)
    {
        Score = newScore;

        _ui.SetScoreText("" + Score);
    }
}

Script Unity | 1 référence
public class ScoreUI : MonoBehaviour
{
    [SerializeField]
    private TMP_Text _scoreText;

    1 référence
    public void SetScoreText(string scoreText)
    {
        _scoreText.text = scoreText;
    }
}
```

Non !

ScoreManager ne devrait se préoccuper que du calcul du score, pas de son affichage ! Si d'autres classes veulent réagir au changement de score, alors ScoreManager deviendra tentaculaire…

Idéalement, il faudrait que ScoreUI puisse *détecter* le changement de score, pour ne modifier le texte qu'à ce moment-là.

Exemple : Observer



Le Design Pattern *Observer* est là pour ça !

Publisher est la classe responsable de l'évènement que d'autres classes veulent écouter.

Ces classes sont appelées des **Subscribers**, et doivent implémenter une interface spécifique contenant uniquement une méthode de notification.

Publisher va alors stocker une liste de **Subscribers** et les notifier lorsque l'évènement a lieu.

Exemple : Observer

```
public interface IScoreChangeSubscriber
{
    2 références
    public void Notify(int newScore);
}
```

```
public class ScoreUI : MonoBehaviour, IScoreChangeSubscriber
{
    [SerializeField]
    private TMP_Text _scoreText;

    Message Unity | 0 références
    private void Awake()
    {
        ScoreManager.Instance.Subscribe(this);
    }

    2 références
    public void Notify(int newScore)
    {
        _scoreText.text = "" + newScore;
    }
}
```

```
public class ScoreManager : MonoBehaviour
{
    private static ScoreManager instance;

    1 référence
    public static ScoreManager Instance...

    2 références
    public int Score { get; private set; }

    private List<IScoreChangeSubscriber> subscribers = new();

    1 référence
    public void Subscribe(IScoreChangeSubscriber subscriber)
    {
        subscribers.Add(subscriber);
    }

    0 références
    public void Unsubscribe(IScoreChangeSubscriber subscriber)
    {
        subscribers.Remove(subscriber);
    }

    0 références
    public void SetScore(int newScore)
    {
        Score = newScore;

        foreach (IScoreChangeSubscriber subscriber in subscribers)
        {
            subscriber.Notify(Score);
        }
    }
}
```

Exemple : Observer

Qu'est-ce que c'est lourd de l'implémenter !

Si on veut pouvoir différencier les évènements "ScoreAdded" et "ScoreRemoved", ça veut dire créer deux nouvelles interfaces, deux nouvelles listes dans ScoreManager, deux nouvelles méthodes…

Si seulement il existait quelque chose en C# pour faire ça plus facilement…

Les évènements C#

Le mot-clé **event** est l'implémentation *directement dans le langage* du pattern Observer !

Vous ne pouvez **invoquer** un évènement que depuis la classe qui le contient.

Vous pouvez ajouter ou retirer une méthode d'écoute avec les opérateurs "+=" et "-=".

"Action<int>" signifie que seules des méthodes ne renvoyant rien et ne prenant qu'un seul paramètre de type int peuvent écouter l'évènement.

```
public class ScoreManager : MonoBehaviour
{
    private static ScoreManager instance;

    1 référence
    public static ScoreManager Instance...

    2 références
    public int Score { get; private set; }

    public event Action<int> ScoreChanged;

    0 références
    public void SetScore(int newScore)
    {
        Score = newScore;

        ScoreChanged?.Invoke(Score);
    }
}
```

```
public class ScoreUI : MonoBehaviour
{
    [SerializeField]
    private TMP_Text _scoreText;

    Message Unity | 0 références
    private void Awake()
    {
        ScoreManager.Instance.ScoreChanged += Notify;
    }

    1 référence
    public void Notify(int newScore)
    {
        _scoreText.text = "" + newScore;
    }
}
```

Les évènements C#

Si vous voulez changer le type de retour de la méthode, le nombre de paramètres, leurs types, etc, vous pouvez utiliser le mot clé **delegate**

```
public delegate void ScoreDelegate(int score);

public event ScoreDelegate ScoreChanged;

0 références
public void SetScore(int newScore)
{
    Score = newScore;

    ScoreChanged?.Invoke(Score);
}
```

Ici, j'ai remplacé "Action<int>" (qui est un type delegate déjà existant en C#) par mon propre type delegate… qui fait ici la même chose.

02

Design Pattern & Jeu-vidéo

Vous en utilisez déjà…

Design Patterns originaux & JV

Vous voulez développer un système de sauvegarde, mais les éléments à sauvegarder sont des champs privés répartis dans plein de classes ?

⇒ **Design Pattern “Memento” !**

Vous voulez manipuler des millions d’instances partageant des données fixes sans faire ramer votre pc ?

⇒ **Design Pattern “Flyweight” !**

Vous voulez pouvoir changer le comportement d’ennemis à la volée dans votre jeu ?

⇒ **Design Pattern “Strategy” !**

Limites des Design Patterns

Les Design Patterns créés en 94 sont bien, mais de nouveaux ont été trouvés, parfois plus spécifiques au monde du jeu-vidéo.

Un excellent livre gratuit, consultable en ligne, qui en liste une bonne quantité :

https://gameprogrammingpatterns.com/contents.html

Ayez TOUJOURS ce site sous la main !

J'insiste !

Les Design Patterns de Unity

Quasiment tous les moteurs de jeu fonctionnent sur le principe d'une **boucle infinie** qui permet d'actualiser l'état et l'affichage du jeu régulièrement.

Dans Unity, elle est visible à travers les méthodes Update et FixedUpdate, par exemple.

Ce principe fondamental, c'est le Design Pattern *Game Loop*.

⇒ https://gameprogrammingpatterns.com/game-loop.html

Les Design Patterns de Unity

Comme vous le savez, dans Unity vous développez des *composants*, c'est-à-dire des comportements que vous allez assembler sur des objets de jeu pour leur donner une consistance.

Unity n'a pas inventé ce principe.

Eh oui, c'est *encore* un design pattern : le pattern *Component*.

⇒ https://gameprogrammingpatterns.com/component.html

Et encore (encore) d'autres…

Le pattern **State** est fondamental dans la plupart des IA de jeux.

Une **Event Queue** est généralement utilisée dans le traitement des inputs du joueur.

Le **Service Locator** peut vous permettre de rassembler des singletons sous une seule bannière.

Les patterns **Object Pool** et **Spatial Partition** sont des techniques d'optimisation que quasiment tous les jeux utilisent.

Allez les lire, les découvrir, et utilisez-les dans votre code !

Ne réinventez pas la roue !

03

Autres bonnes pratiques

SOLID, DRY, KISS, et autres acronymes bizarres

“Code propre”

Écrire un script avec une syntaxe correcte, qui s’exécute normalement, c’est relativement facile.

Le rendre *compréhensible* pour d’autres développeur.euse.s, c’est plus difficile.

Réussir à produire du code qui fonctionne, simple à comprendre, et qui n’ajoute pas de *dette technique*, c’est **très complexe**.

La première catégorie est obligatoire.

La seconde vous permettra de bosser en équipe.

La troisième vous octroiera des louanges de votre équipe.

Principes S.O.L.I.D.

S - Single Responsibility Principle (Responsabilité Unique)

Chaque classe, fonction ou méthode ne doit avoir qu'une seule responsabilité unique.

```
public class Player : MonoBehaviour
{
    private float _speed;

    private int _health;

    private int _maxHealth;

    private bool _isJumping;

    0 références
    public void MovePlayer(Vector3 direction, float speed) { }

    0 références
    public void ApplyDamage(int damage) { }

    0 références
    public void Jump(float jumpForce) { }

    0 références
    public void ApplyHeal(int heal) { }
}
```

```
public class PlayerHealth : MonoBehaviour
{
    private int _health;

    private int _maxHealth;

    0 références
    public void ApplyHeal(int heal) { }

    0 références
    public void ApplyDamage(int damage) { }
}
```

```
public class PlayerMovement : MonoBehaviour
{
    private float _speed;

    private bool _isJumping;

    0 références
    public void MovePlayer(Vector3 direction, float speed) { }

    0 références
    public void Jump(float jumpForce) { }
}
```

Principes S.O.L.I.D.

O - Open/Closed Principle (Ouvert / Fermé)

Toute classe/méthode/etc. doit être **fermée** à la modification de son comportement et **ouverte** à l'extension.

```
public class EnemyAttack : MonoBehaviour
{
    0 références
    public void Attack(Player player)
    {
        // Attaquer le joueur au corps à corps
    }
}
```

Ajout de l'attaque à distance

```
public interface IEnemyAttack
{
    2 références
    public void Attack(Player player);
}

Script Unity | 0 références
public class EnemyMeleeAttack : MonoBehaviour, IEnemyAttack
{
    1 référence
    public void Attack(Player player)
    {
        // Attaquer le joueur au corps à corps
    }
}

Script Unity | 0 références
public class EnemyRangeAttack : MonoBehaviour, IEnemyAttack
{
    1 référence
    public void Attack(Player player)
    {
        // Attaquer le joueur à distance
    }
}
```

Principes S.O.L.I.D.

L - Liskov Substitution Principle (Substitution de Liskov)

Si B est un sous-type de A, alors une instance de A peut être remplacée par une instance de B sans changer le programme.

```
2 références
public class Apple
{
    2 références
    public virtual void LogColor()
    {
        Debug.Log("Red");
    }
}

1 référence
public class Orange : Apple
{
    2 références
    public override void LogColor()
    {
        Debug.Log("Orange");
    }
}

0 références
public class Program
{
    0 références
    public void Main()
    {
        Apple apple = new Orange();

        apple.LogColor();
        // Affiche "orange", bizarre pour une pomme !
    }
}
```

```
3 références
public abstract class Fruit
{
    3 références
    public abstract void LogColor();
}

0 références
public class Apple : Fruit
{
    2 références
    public override void LogColor()
    {
        Debug.Log("Red");
    }
}

1 référence
public class Orange : Fruit
{
    2 références
    public override void LogColor()
    {
        Debug.Log("Orange");
    }
}
```

Principes S.O.L.I.D.

I - Interface Segregation Principle (Ségrégation des interfaces)

Une classe ne doit pas dépendre de méthodes qu'elle n'utilise pas.

```
3 références
public interface IUnit
{
    1 référence
    public void Attack(IUnit target);

    1 référence
    public void Move(Vector3 direction);
}

0 références
public class Watchtower : IUnit
{
    1 référence
    public void Attack(IUnit target)
    {
        // La tour de garde attaque l'ennemi
    }

    1 référence
    public void Move(Vector3 direction)
    {
        // La tour ne peut pas bouger,
        // ça sert à quoi ?!
    }
}
```

```
0 références
public interface IMovable
{
    0 références
    public void Move(Vector3 direction);
}

1 référence
public interface IAttacker
{
    1 référence
    public void Attack(IUnit target);
}

0 références
public class Watchtower : IAttacker
{
    1 référence
    public void Attack(IUnit target)
    {
        // La tour de garde attaque l'ennemi
    }
}
```

Principes S.O.L.I.D.

D - Dependency Inversion Principle (Inversion des dépendances)

(pour faire simple) Une classe doit idéalement dépendre d'une **abstraction** plutôt que d'une **implémentation**.

Ce point-là est un peu compliqué à illustré, amène à un *terrier de lapin* qui mériterait son propre cours, et son utilité est débattue dans le contexte du développement sur Unity.

Principe D.R.Y

DRY - Don’t Repeat Yourself (Ne vous répétez pas)

N’ayez pas la même information ou le même algorithme à plusieurs endroits de votre code.

⇒ La vitesse de déplacement de votre personnage est définie à plusieurs endroits ? Changez ça !

⇒ Le bout de code permettant de savoir si un objet se trouve dans le viseur du joueur est copié-collé partout dans votre code ? Pareil, changez ça !

Principe K.I.S.S

KISS - Keep it simple, stupid (Garde ça simple, idiot)

Ayez toujours en tête de faire quelque chose de *simple*, ne cherchez pas la complexité par plaisir de chercher la complexité.

Principes du langage Python :

- Préfère le *simple* au *complexe* et le *complexe* au *compliqué*.

Antoine de Saint-Exupéry :

- Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

TP

01 - Bonnes pratiques

Sources

Diapo 6 : Par Atelier Tinga — Travail personnel, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15454621