Formation Design Pattern, architectures et bonnes pratiques

Présentation

Module 0

Gaëtan GOUHIER

- Ingénieur en informatique
- Spécialisé dans le développement .NET
- Certifié Microsoft (70-483, 70-486, AZ-203) / Azure Developer Associate
- Développeur et formateur indépendant depuis 2016
- https://www.gaetan-gouhier.net/

Et vous ?

- Votre parcours ?
- Votre connaissance du sujet ?
- Vos attentes ?

Sommaire

Bonnes pratiques

Module 1

Bonnes pratiques

- Lors du déroulement d'un projet, la maintenance est la plus grosse partie
- Un projet dure généralement plusieurs années et évolue constamment
- Il va donc falloir respecter des bonnes pratiques pour pouvoir faire évoluer rapidement et sans problème son projet
  - DRY (Don't Repeat Yourself) : Si vous vous répétez c'est qu'il y a sûrement un meilleur moyen de faire
  - Factorisation : Extraire le code répété dans une méthode à part qui sera commune et donc plus maintenable
  - 1 classe = 1 fichier .cs
  - Convention de nommage : Respecter les mêmes conventions de nommage en équipe (ou même seul) vous permettra d'être plus cohérent et de lire plus rapidement votre code

Conventions

- Chaque langage / équipe a ses propres conventions mais voilà les prérequis de Microsoft
  - Les noms sont en PascalCase
  - Une classe commence par une majuscule
  - Une interface commence par un "I" majuscule
  - Les propriétés publiques commencent par une majuscule
  - Les propriétés privés, les variables locales et les paramètres de méthodes commencent par une minuscule
  - Les noms des listes finissent par un "s"
  - Il y a un retour à la ligne avant et après chaque bloc de code { ou }
  - Les noms des constantes sont tous en majuscule
  - Une classe contient dans l'ordre : les propriétés, les constructeurs puis les méthodes

Grasp

Module 2

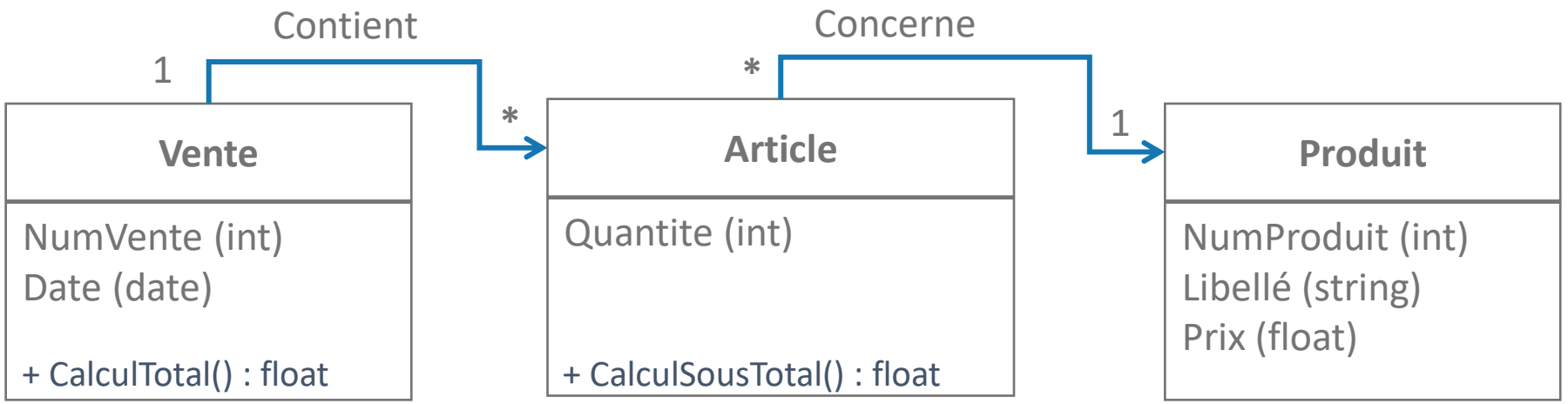
Principes « GRASP »

- General Responsability Assignement Software Patterns, ou Modèles Généraux d'Affectation de Responsabilités aux Logiciels.
- Il s'agit de patterns fondamentaux qu'il est nécessaire d'appliquer pour une bonne conception d'un projet
- Ces patterns sont, pour la plupart, directement issus du bon sens du concepteur, et leur représentation est tout aussi intuitive
  - Expert
  - Créateur
  - Faible couplage
  - Forte cohésion

Information Expert

- L'objectif de ce principe est d'affecter au mieux une responsabilité à une classe logicielle
- Afin de réaliser ses responsabilités, cette classe doit disposer des informations nécessaires
- Le modèle Expert est un pattern relativement intuitif, qui en général, est respecté par le bon sens du concepteur.
- Il s'agit tout simplement d'affecter à une classe les responsabilités correspondants aux informations dont elle dispose
- Une responsabilité n'est pas une méthode, mais des méthodes s'acquittent de responsabilités

Information Expert



Information Expert

```
foreach (var article in Vente.Articles) {
        prixTotal += article.Produit.Prix * article.Quantite;
}
```

```
foreach (var article in Vente.Articles) {
    prixTotal += article.CalculSousTotal();
}
```

Créateur (Creator)

- Ce principe consiste en la détermination de la responsabilité de la création d'une instance d'une classe par une autre
- Une classe A peut créer une instance d'une classe B si
  - La classe A contient des instances de la classe B
  - La classe A utilise des instances d'une classe B
  - La classe A possède les données nécessaires à l'initialisation des instance de classe B

Faible couplage

- Le faible couplage a pour objectif de faciliter la maintenance en minimisant les dépendances entre éléments
- Un élément faiblement couplé ne possède pas ou peu de dépendances vis a vis d'autres éléments
- Un couplage trop fort entre deux éléments peut entraîner lors de la modification d'un de ses éléments une modification d'un élément lié
- Les stratégies qui le permettent sont variées, mais reposent largement sur l'inversion de contrôle

Forte cohésion

- La cohésion mesure comment sont plus ou moins fortement liés et concentrés les responsabilités d'une classe
- La cohésion est une notion assez vague dont l'objectif est de déterminer si l'objet n'en fait pas "trop"
- Ainsi, un objet avec 100 méthodes, des milliers de lignes de codes, couvrant divers domaines n'est plus cohérent, il a perdu sa cohésion. Il vaut mieux déléguer, instituer une collaboration avec de nouveaux objets
- De plus, un objet trop gros (faible cohésion) risque d'introduire un fort couplage (car trop d'objets auront besoin de lui, trop d'interactions...)

SOLID

Module 2

Principes « SOLID »

- SOLID est un ensemble de concept pour rentre le code plus facile à lire et à maintenir
  - **S**ingle Responsibility Principle
  - **O**pen Closed Principle
  - **L**iskov Subtitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

Single Responsibility Principle

- Une classe doit absolument se concentrer sur une tâche, un but, ne prendre en charge qu’une seule responsabilité
- Une classe doit n'avoir qu'une seule raison de changer
- En accord avec le principe de "cohésion forte" (GRASP)

Single Responsibility Principle

```
public class Client {
    public int Id { get; set; }
    public string Name { get; set; }
    public int Insert() {
        try {
            // insertion en base de données, retourne l'ID
            return 0;
        }
        catch (Exception e) {
            System.IO.File.WriteAllText(@"c:\temp\log.txt", e.ToString());
            return -1;
        } } }
```

Single Responsibility Principle

```
public static class Log

{

    public static void Error(Exception ex)

    {

        System.IO.File.WriteAllText(@"c:\temp\log.txt",
ex.ToString());

    }

}
```

Single Responsibility Principle

```
try
{
    // insertion en base de données, retourne l'ID
    return 0;
}
catch (Exception e)
{
    Log.Error(e);
    return -1;
}
```

Single Responsibility Principle

```
public static class CrudClient {
        public static int Insert(Client client) {
            try {
                //accès DAL ou autre
                return 0;
            }
            catch (Exception e) {
                Log.Error(e);
                return -1;
            }
        }
    }
```

Single Responsibility Principle

```
public class Client
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Open Closed Principle

- Ce principe facilite à étendre un code sans avoir à revisiter le code existant
- Si on modifie notre code qui fonctionne en ajoutant une fonctionnalité, on risque de créer des problèmes sur le code qui fonctionne déjà (régression)

Open Closed Principle

```
public enum CustomerCategory { Normal, Silver, Vip }

// Client.cs

public CustomerCategory Category { get; set; }

public double GetDiscount()

{

        if (Category == CustomerCategory.Normal) return 0;

        else if (Category == CustomerCategory.Silver) return 5;

        else return 10;

} // Si on rajoute une catégorie plus tard, on devra modifier ce code
```

Open Closed Principle

```
public class Client {
    public int Id { get; set; }
    public string Name { get; set; }
    public virtual double Discount { get { return 0; } }
}
public class ClientSilver : Client {
    public override double Discount { get { return 5; } }
}
public class ClientVip : Client {
    public override double Discount { get { return 10; } }
}
```

Liskov Subtitution Principle

- Les méthodes qui utilisent des instances d'une classe A doivent pouvoir utiliser des objets dérivés de A sans même le savoir
- Notre méthode Insert peut recevoir n'importe quel type de client sans le savoir

Liskov Subtitution Principle

```
// Un propect ne doit pas pouvoir être enregistré en base
public class Prospect : Client
{
    public override double Discount { get { return 0; } }
}
```

Liskov Subtitution Principle

```
public static int Insert(Client client) {
    if (client is Prospect) return -1;
    try {
        //accès DAL ou autre
        return 0;
    }
    catch (Exception e) {
        Log.Error(e);
        return -1;
    }
}
```

Liskov Subtitution Principle

```
public interface IBase
{
    string Name { get; set; }
    double Discount { get; }
}
public interface IClient {
    int Id { get; set; }
}
```

Liskov Subtitution Principle

```
public class Client : IBase, IClient {
    public int Id { get; set; }
    public string Name { get; set; }
    public virtual double Discount { get { return 0; } }
}
public class ClientSilver : Client {
    public override double Discount { get { return 5; } }
}
public class ClientVip : Client {
    public override double Discount { get { return 10; } }
}
```

Liskov Subtitution Principle

```
public class Prospect : IBase
{
    public string Name { get; set; }
    public double Discount { get { return 0; } }
}
```

Liskov Subtitution Principle

```
public static int Insert(IClient client) {
try {
        //accès DAL ou autre
        return 0;
    }
    catch (Exception e) {
        Log.Error(e);
        return -1;
    }
}
```

Liskov Subtitution Principle

```
var all = new List<IBase>();
var client1 = new Client { Name = "client1" };
var vip1 = new ClientVip { Name = "vip1" };
var propect1 = new Prospect { Name = "propect1" };

all.Add(client1);
all.Add(vip1);
all.Add(propect1);

CrudClient.Insert(client1);
CrudClient.Insert(propect1); // ERROR
```

Interface Segregation Principle

- Aucune classe ne devrait pas avoir accès à des méthodes ou des propriétés qu’elle n'utilise pas
- Il faut donc diviser les interfaces volumineuses en plus petites plus spécifiques, de sorte que les classes n'ont accès qu'aux informations intéressantes pour elles

Dependency Inversion Principle - Inversion Of Control (IOC)

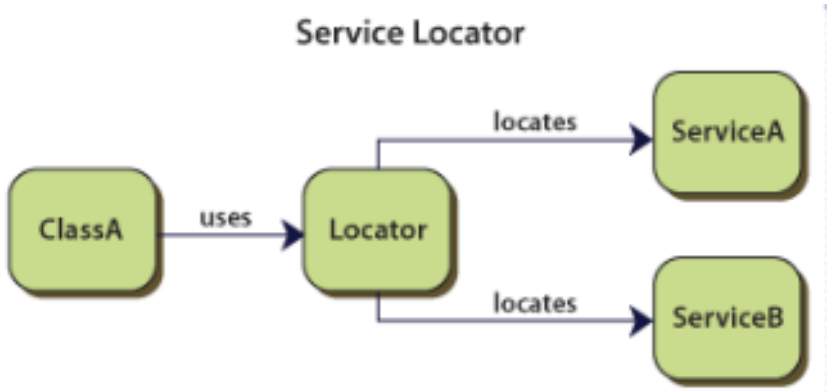
- L’inversion de contrôle est un patron d'architecture commun à tous les frameworks
- L'objectif va être de limité le couplage fort entre 2 classes
- L'IOC peut se faire sous 2 façon
  - Utiliser un locator
  - Passer par un outil d'injection de dépendance

IOC - Service Locator

```
try
{
    // insertion en base de données, retourne l'ID
    return 0;
}
catch (Exception e)
{
    Log.Error(e); // couplage fort, on doit passer par un intermédiaire
    return -1;
}
```

IOC - Service Locator

- Plutôt qu'une ClassA utilise directement un ServiceA ou ServiceB, elle va utiliser un Locator qui va se charger de retourner le ServiceA ou ServiceB
- L'avantage est que la ClassA ne connait pas directement le service qu'elle utilise, seul le Locator le sait. Si un changement de service se fait plus tard, la ClassA ne devra pas être modifié
- La ClassA ne connait que l'interface



IOC - Service Locator

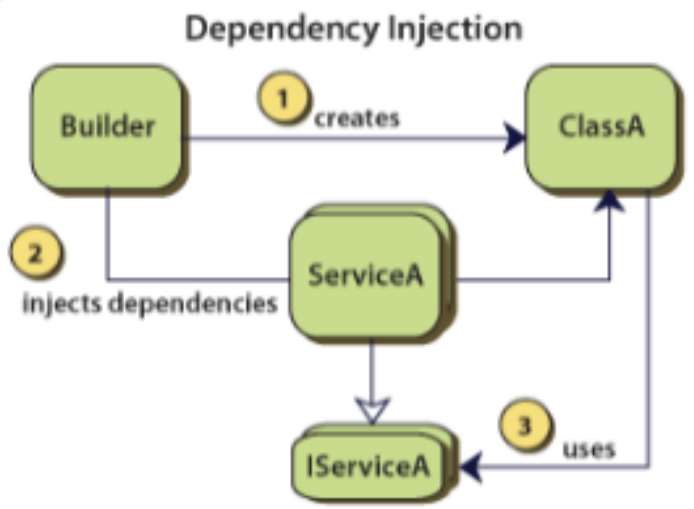
```
public interface ILog {
    void Error(Exception ex);
}
public class LogFile : ILog {
    public void Error(Exception ex) {
        System.IO.File.WriteAllText(@"c:\temp\log.txt", ex.ToString());
    } }
public class LogDb : ILog {
    public void Error(Exception ex) {
        // write in BDD ex
    } }
```

IOC - Service Locator

```
public class ServiceLocator {
    public static ILog GetLoggingService() {
        if (ConfigurationSettings.AppSettings.Get("Logging") == "file")
            return new LogFile();
        else
            return new LogDb();
    }
}
//utilisation
ServiceLocator.GetLoggingService().Error(e);
```

IOC - Injection de dépendance

- Plutôt qu'une ClassA utilise le Locator, c'est une autre classe (Builder) qui va mettre dans la ClassA la bonne implémentation du service
- La ClassA ne fait qu'exposer une propriété ou un paramètre de son constructeur dans lequel le builder va y mettre la bonne implémentation
- Un outil comme Unity est recommandé pour utiliser l'injection de dépendance



Unity

- Unity container vous permet de faire de l'injection de dépendance en prenant place du Builder
- Une première configuration d'Unity pour lié les interfaces aux implémentations est nécessaire
- Dès qu'on va demander une classe, l'arbre de dépendance sera créé automatiquement
- Unity pourra injecter les instances d'objets directement dans le constructeur, les méthodes ou les propriétés

Unity - Configuration

```
public class CrudClient {
    ILog Log { get; }
    public CrudClient(ILog log) {
        Log = log;
    }
    public int Insert(Client client) {
        try { return client.Id; }
        catch (Exception e) {
            Log.Error(e);
            return -1;
        } } }
```

Unity - Configuration

```
IUnityContainer unitycontainer = new UnityContainer();

// Lie les implémentations aux interfaces correspondantes
// Peut être fait en fichier de config XML
if (ConfigurationSettings.AppSettings.Get("Logging") == "file")
    unitycontainer.RegisterType<ILog, LogFile>();
else
    unitycontainer.RegisterType<ILog, LogDb>();

CrudClient crudClient = unitycontainer.Resolve<CrudClient>();
```

Design Pattern

Module 1

Introduction

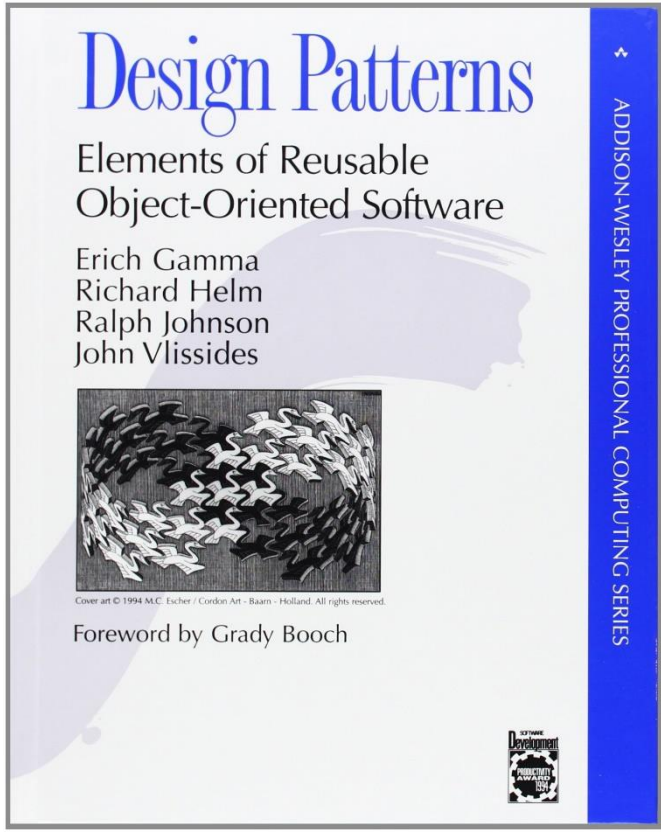
- Un Design pattern décrit à la fois
  - Un problème qui se produit très fréquemment dans un environnement
  - L'architecture de la solution à ce problème de telle façon que l'on puisse utiliser cette solution des milliers de fois
- Cela permet de décrire avec succès des types de solutions récurrentes à des problèmes communs dans des types de situations

Introduction

- Les design patterns offrent
  - Une documentation d'une expérience éprouvée de conception
  - Une identification et spécification d'abstractions qui sont au dessus du niveau des simples classes et instances
  - Un vocabulaire commun et aide à la compréhension de principes de conception
  - Un moyen de documentation de logiciels
  - Une aide à la construction de logiciels complexes et hétérogènes, répondant à des propriétés précises

Les débuts

- Le terme de Design Pattern est apparue en premier dans le livre "Elements of reusable object-oriented sofware" en 1995
- Ce livre a été écrit par le « Gang Of Four » (GoF, « la bande des quatres »)
  - Erich Gamma
  - Richard Helm
  - Ralph Johson
  - John Vlissides
- Ce livre est encore d'actualité et en vente
- Il regroupe l'ensemble des 23 patterns du GoF



Catégories de Design Patterns

- Création
  - Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
  - Isolation du code relatif à la création et à l'initialisation afin de rendre l'application indépendante de ces aspects
  - Exemples : Abstract Factory, Builder, Prototype, Singleton
- Structure
  - Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
  - Exemples : Adapter(objet), Composite, Bridge, Decorator, Facade, Proxy
- Comportement
  - Description de comportements d'interaction entre objets
  - Gestion des interactions dynamiques entre des classes et des objets
  - Exemples : Strategy, Observer, Iterator, Mediator , Visitor, State

Portée des Design Patterns

- Portée de Classe
  - Focalisation sur les relations entre classes et leurs sous-classes
  - Réutilisation par héritage (« Est un »)
- Portée d'Instance (Objet)
  - Focalisation sur les relations entre les objets
  - Réutilisation par composition (« A un »)

Présentation d'un Design Pattern

- Nom du pattern
  - Utilisé pour décrire le pattern, ses solutions et les conséquences
- Problème
  - Description des conditions d'applications.
  - Explication du problème et de son contexte
- Solution
  - Description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème, utilisation de diagrammes de classes, de séquences, …
  - Vision statique et dynamique de la solution
- Conséquences
  - Description des résultats de l'application du pattern sur le système (effets positifs et négatifs)

Exemples

Module 1

Pattern Singleton

- Le pattern Singleton permet de garantir la création d'une instance unique d'une classe durant toute la durée d'exécution d'une application

Pattern Singleton

- Catégorie
  - Création
- OBJECTIFS
  - Restreindre le nombre d'instances d'une classe à une et une seule
  - Fournir une méthode pour accéder à cette instance unique
- RAISONS DE L'UTILISER
  - La classe ne doit avoir qu'une seule instance. Cela peut être le cas d'une ressource système par exemple
  - La classe empêche d'autres classes de l'instancier. Elle possède la seule instance d'elle-même et fournit la seule méthode permettant d'accéder à cette instance
- RESULTAT
  - Le Design Pattern permet d'isoler l'unicité d'une instance
- RESPONSABILITES :
  - Singleton doit restreindre le nombre de ses propres instances à une et une seule. Son constructeur est privé : cela empêche les autres classes de l'instancier. La classe fournit une méthode statique qui permet d'obtenir l'instance unique

Pattern Singleton

Exemple

```
public sealed class Singleton {
        private static Singleton instance = null;
        private Singleton() { }
        public static Singleton Instance {
            get {
                if (instance == null)
                    instance = new Singleton();
                return instance;
            }
        }
    }
```

Exemple – Thread safe

```
public sealed class Singleton {
        private static Singleton instance = null;
        private static readonly object padlock = new object();
        Singleton() { }
        public static Singleton Instance {
            get {
                lock (padlock) {
                    if (instance == null)
                        instance = new Singleton();
                    return instance;
                } } } }
```

Exemple – Utilisation

```
Singleton singleton = new Singleton(); // ERROR

Singleton singleton = Singleton.Instance;
```

Pattern Proxy

- Catégorie
  - Structure
- Objectif du pattern
  - Fournir un intermédiaire entre la partie cliente et un objet pour contrôler les accès à ce dernier
  - Un proxy est utilisé principalement pour contrôler l'accès aux méthodes de la classe substituée
  - Il est également utilisé pour simplifier l'utilisation d'un objet « complexe » à la base : par exemple, si l'objet doit être manipulé à distance (via un réseau)
- Résultat
  - Le Design Pattern permet d'isoler le comportement lors de l'accès à un objet.

Exemple

```
public interface IMath {
    double Add(double x, double y);
    double Sub(double x, double y);
    double Mul(double x, double y);
    double Div(double x, double y);
}
```

Exemple

```
class Math : IMath {
    public double Add(double x, double y) { return x + y; }
    public double Sub(double x, double y) { return x - y; }
    public double Mul(double x, double y) { return x * y; }
    public double Div(double x, double y) { return x / y; }
}
```

Exemple

```
class MathProxy : IMath {
    private Math math = new Math();
    public double Add(double x, double y) => math.Add(x, y);
    public double Sub(double x, double y) => math.Sub(x, y);
    public double Mul(double x, double y) => math.Mul(x, y);
    public double Div(double x, double y) => math.Div(x, y);
}
```

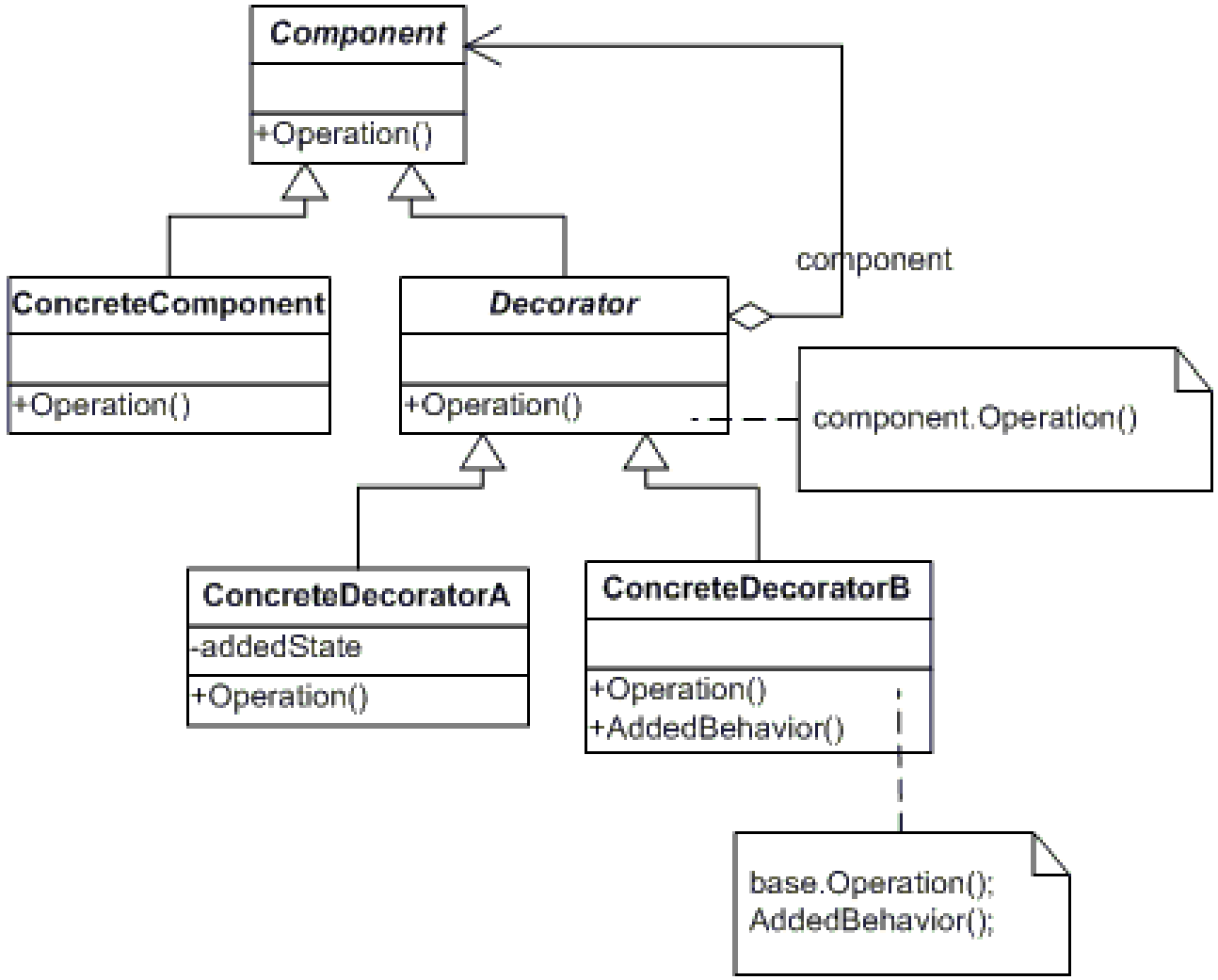
Exemple - Utilisation

```
// Create math proxy

MathProxy proxy = new MathProxy();

// Do the math

Console.WriteLine("4 + 2 = " + proxy.Add(4, 2));

Console.WriteLine("4 - 2 = " + proxy.Sub(4, 2));

Console.WriteLine("4 * 2 = " + proxy.Mul(4, 2));

Console.WriteLine("4 / 2 = " + proxy.Div(4, 2));
```

Pattern Decorateur

- Catégorie
  - Structure
- Définition
  - Le pattern Décorateur attache dynamiquement des responsabilités supplémentaires à un objet.
  - Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités
- Résultat
  - Le Design Pattern permet d'isoler les responsabilités d'un objet
  - Plutôt que de modifier un code existant qui fonctionne, on rajoute des fonctionnalités sans toucher le code existant

Exemple


Component
+Operation()
component
ConcreteComponent
+Operation()
Decorator
+Operation()
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior();

Exemple

```
abstract class HotDogsBase {
    protected string name;
    public virtual double GetName() {
        return name;
    }
    protected double price;
    public virtual double GetPrice() {
        return price;
    }
}
```

Exemple

```
class HotDogClassic : HotDogsBase {
    public HotDogClassic() {
        name = "Classic";
        price = 5.0;
    } }
class HotDogVege : HotDogsBase {
    public HotDogVege() {
        name = "Veggie";
        price = 6.0;
    } }
```

Ajout d’une fonctionnalité

- En V2 on aimerai gérer des extras, un hotdog pourra avoir un extra épicé par exemple.
- Cet extra aura lui aussi un nom et un prix en supplément

Exemple

```
abstract class Extra : HotDogsBase {
public HotDogsBase HotDogsComponent { get; set; }
protected Extra(HotDogsBase hotdog) {
     HotDogsComponent = hotdog;
}
public override string GetName() => $"{name} with {HotDogsComponent.GetName()}";
public override double GetPrice() => price + HotDogsComponent.GetPrice();
}
```

Exemple

```
class ExtraHoney : Extra {
    public ExtraHoney(HotDogsBase hotDogHoney) : base(hotDogHoney) {
        name = "honey";
        price = 1.5;
    }
}
class ExtraSpice : Extra {
    public ExtraSpice(HotDogsBase hotDogSpice) : base(hotDogSpice) {
        name = "spice";
        price = 1;
    }
}
```

Exemple

```
// V1
var hotdog = new HotDogVege();
Console.WriteLine(hotdog.GetName());

// V2
var hotdogAvecExtra = new ExtraSpice(new HotDogVege());
Console.WriteLine(hotdog.GetName());
```

Les Design Pattern du GoF

Portée \ Usage	Creation	Structurel	Comportement
Classe	Factory Method	Adapter	Interpreter
			Template Method
Objet	Builder	Bridge	Memento
	Abstract Factory	Composite	Observer
	Prototype	Decorator	State
	Singleton	Façade	Strategy
		Flyweight	Command
		Proxy	Visitor
			Chain of Responsibility
			Iterator
			Mediator

- https://www.dofactory.com/net/design-patterns
- http://www.e-naxos.com/Blog/post/Design-Patterns-ou-quand-comment-et-pourquoi-.aspx (FR)

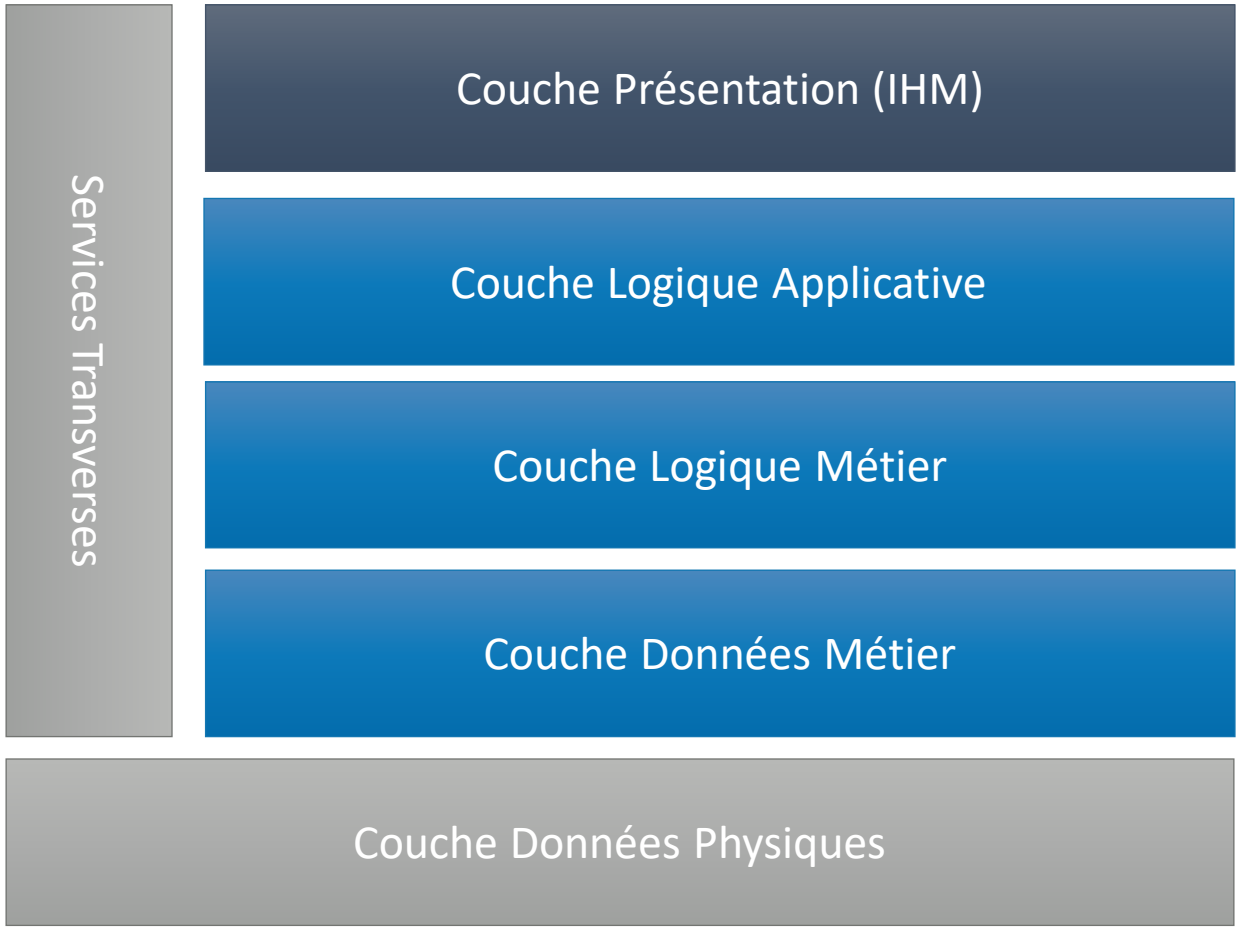
Architecture

Module 1

Introduction

- L'architecture est le regroupement des différents rôles nécessaires à la réalisation d’une application, en modules le plus autonomes possibles
- L'architecture d'un projet est défini sur un semble de couche qui va de l'interface utilisateur jusqu'à la base de données
- Une couche d'une application peut aussi être appelé un tiers
  - Une application 2 tiers aura un client lourd qui tape directement en base de données
  - Une 3 tiers aura un intermédiaire entre les deux couches IHM et BDD
  - Une 4 tiers ou n tiers aura plusieurs couches entre l'IHM et la BDD
- Concrètement une couche est un ensemble de classes, souvent regroupées dans un projet de bibliothèque de classe (DLL)

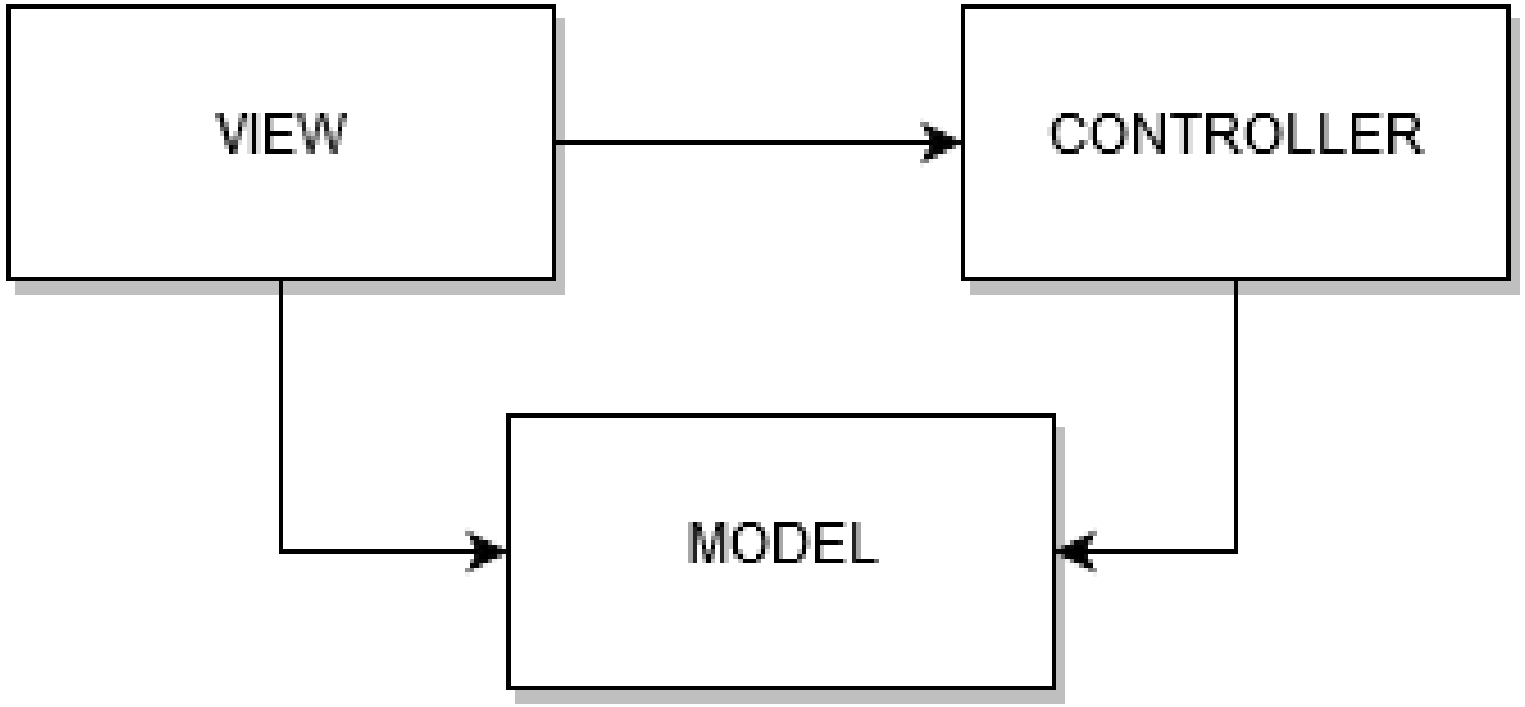
Introduction



L'architecture MVC

- L'architecture MVC permet de structurer une application en couches :
  - M (Model) : Il représente les données de l'application, à savoir les composants métiers et définit les interactions avec les supports de stockage (base de données, fichiers XML, etc.).
  - V (View) : Elle représente l'interface utilisateur. Elle n'effectue aucun traitement et se contente simplement d'afficher les données que lui fournit le modèle. Il peut y avoir plusieurs vues qui présentent les données d'un même modèle
  - C (Controller) : Il gère l'interface entre le modèle et le client. Il va interpréter la requête de ce dernier pour lui envoyer la vue correspondante. Il effectue la synchronisation entre le modèle et les vues
- Le MVC est la principale façon de développer des applications WEB quelle que soit la technologie

L'architecture MVC



L'architecture MVP

- Cette architecture est de moins en moins utilisé
- Elle contient aussi 3 couches: Model / View / Presenter
- La plus grande différence avec MVC est que le View ne connait pas le Model

L'architecture MVVM

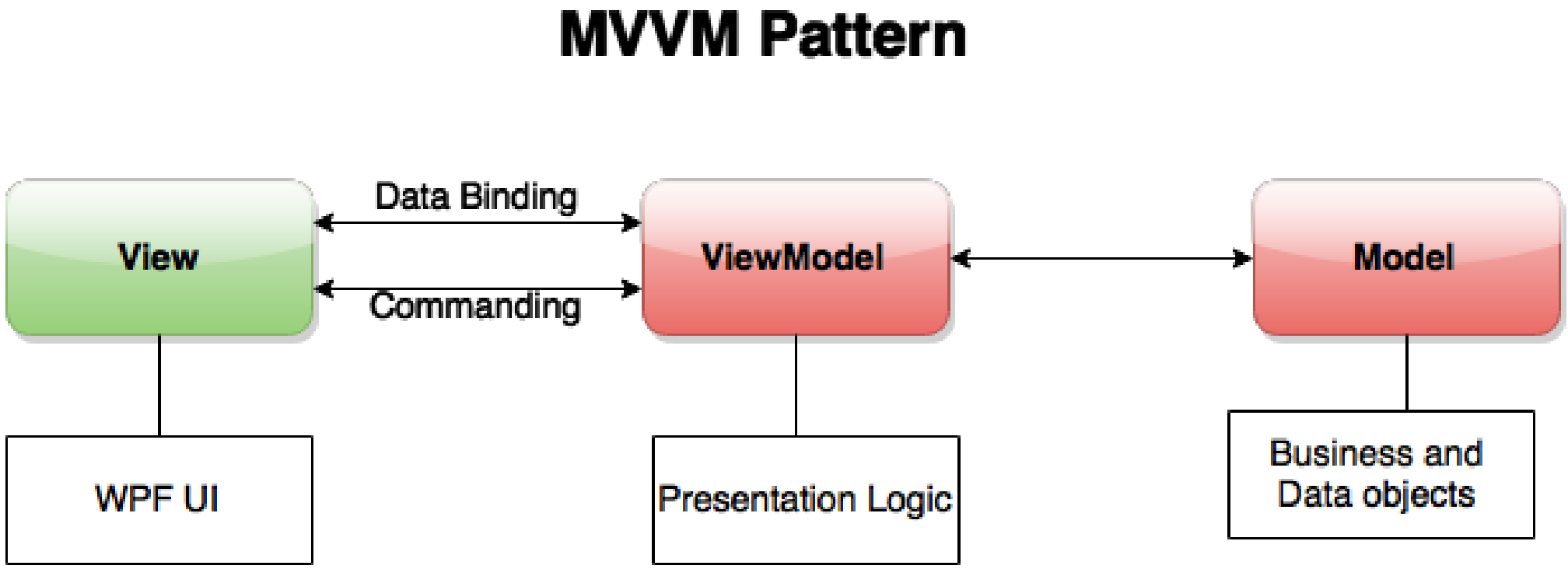
- Cette architecture est arrivé avec le problème survenu avec les applications WPF
- Ces applications reposent sur un binding entre une propriété C# et un élément de la vue
- Le problème est que la propriété C# doit lancer un évènement NotifyPropertyChanged pour mettre la vue à jour
- Si en WPF on fait du MVC, la vue connait le modèle et donc les bindings de la vue se font directement sur les models
- Problème si je modifie les models pour qu'ils lance l'évènement NotifyPropertyChanged, mes models ne seront plus utilisable dans d'autre type d'application tel quels
- La solution est de passer par un intermédiaire, le ViewModel
- La vue ne connait donc pas le model

L'architecture MVVM

- Le MVVM est une variation du patron de conception MVC
  - M comme Modèle : Il constitue la couche de données métiers et n'est lié à aucune représentation graphique précise. Le modèle est réalisé en code C# et ne repose pas sur l'utilisation de technologies propres à WPF.
  - V comme Vue : Elle permet de présenter les données en utilisant le DataBinding. Elle est constituée des éléments graphiques. Les contrôles sont rassemblés dans des UserControl.
  - VM comme Vue-Modèle : A pour but de servir la vue (XAML) et lui fournir les données ainsi que les commandes dont elle a besoin. Il peut être vu comme un adaptateur entre la Vue et Modèle.
- Le modèle n'est pas exploitable directement car il doit implémenter l'interface INotifyPropertyChanged afin de signaler les changements d'états.
- Les classes Vue-Modèle introduisent donc une abstraction entre la partie graphique et le reste

L'architecture MVVM

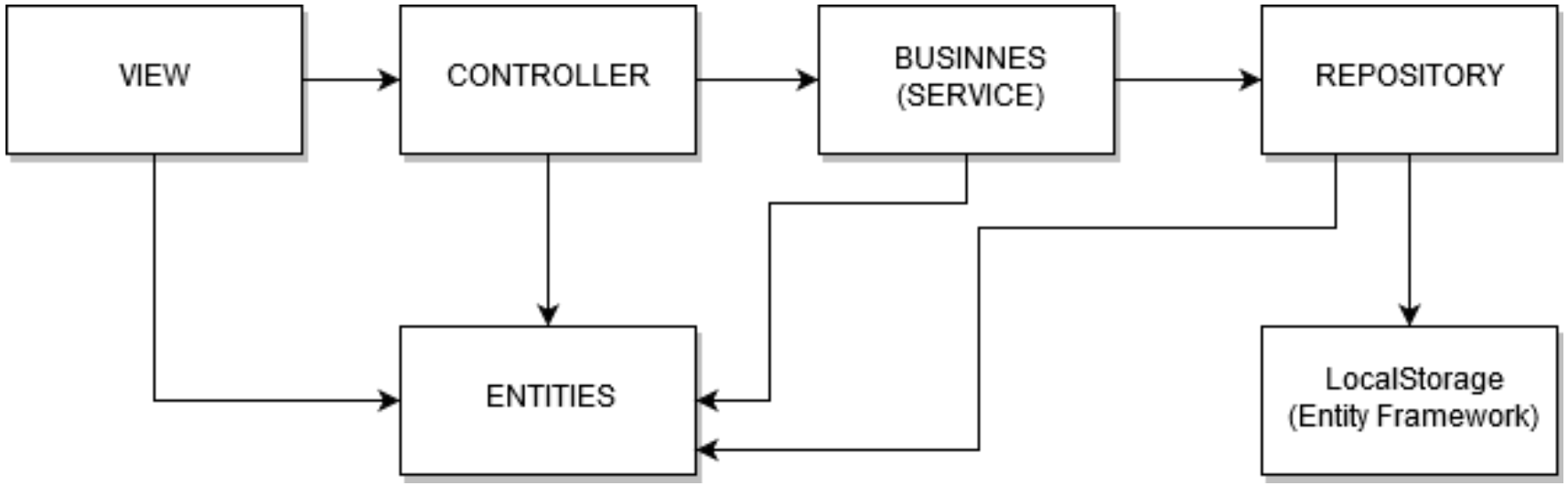
- Le modèle MVVM s’inspire du pattern MVC.



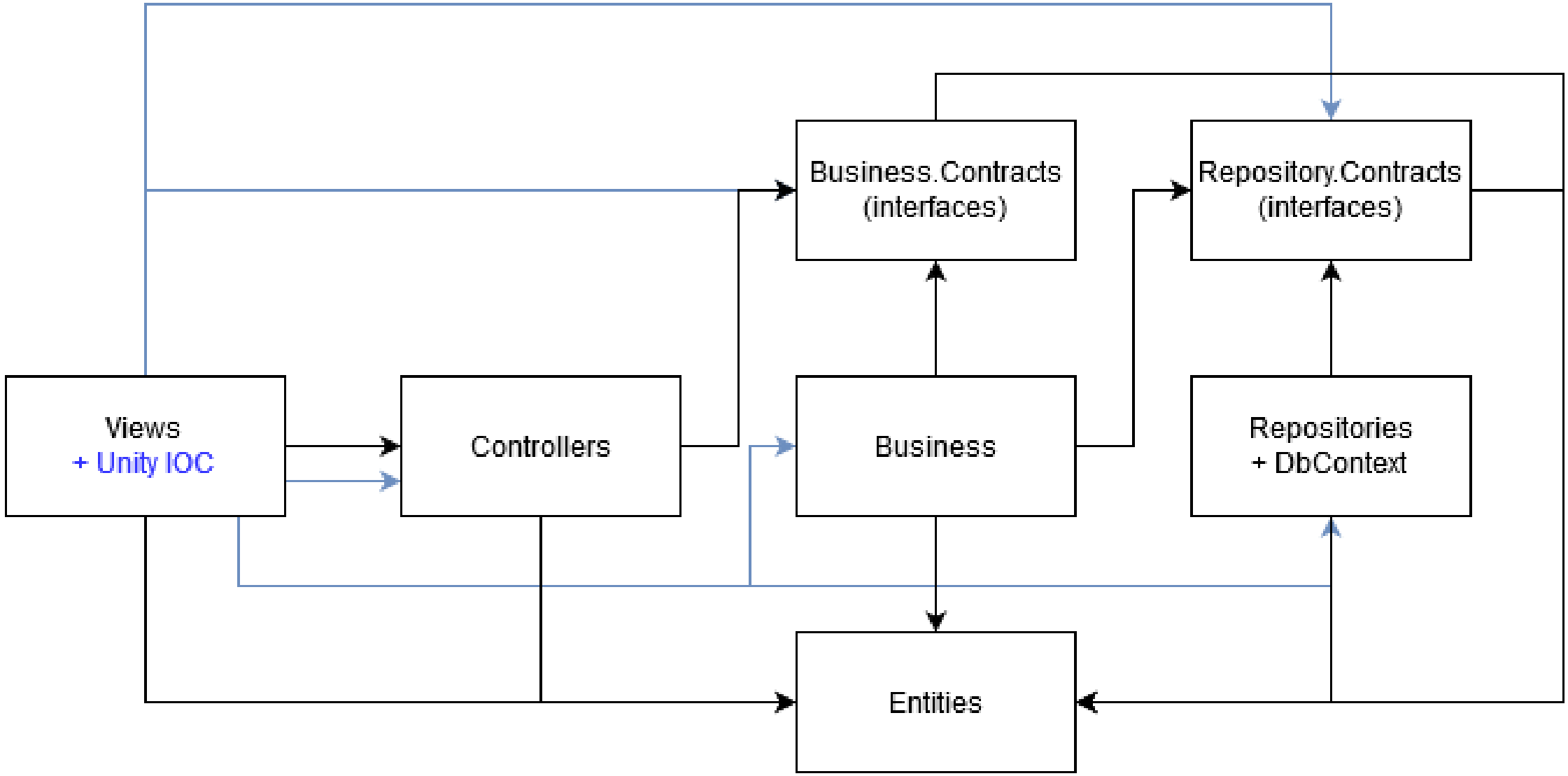
L'architecture MVC avancé

- Le problème de l'architecture MVC classique est que le contrôleur en fait trop
  - Il gère les données envoyées et récupérées de la vue
  - Il gère la partie de gestion métier
  - Il gère l'accès à la base de données
- La solution est de redécouper les couches pour que chacune n'est qu'une responsabilité
  - View : la partie vue qui ne change pas
  - Controlleur : Le contrôleur qui gère les données envoyées et récupérées de la vue
  - Business (Application Service) : La partie qui gère l'aspect métier / fonctionnel (Un électeur ne peut pas avoir moins de 18 ans)
  - Repository : Il gère la persistance des données en BDD
  - Entities (Models) : Contient toutes les classes métiers de l'application

L'architecture MVC avancé



L'architecture MVC avancé avec Unity



Fin de la formation