

# Clean Architecture

## .Net Core

Nathanaël KLEIN





# Présentation du formateur

Nathanaël KLEIN

Plus de 7 ans dans le développement d'applications Backend en .net. dont 3 en tant que lead tech.  
Expert dans les technologies NoSQL. Formateur en .NET, ELK et Clean Architecture

- Application mobile Lyf Pay
- Outils de gestion comptable
- Développement D'IHM pour des société de robotiques
- Outil de détection de comportement Atypique pour la gestion de la fraude
- Process de paiement Bancaire
- Mise en place de stack ELK pour le monitoring et l'exploitation
- ...



# Sommaire

- ❖ **Introduction**
- ❖ Motivation à la **clean architecture**
  - Architecture
  - Cas pratique
  - Les erreurs systémiques
- ❖ Les principes **S.O.L.I.D**
- ❖ Notion sur les **composants**
  - Couplages
  - dépendances
- ❖ Tracer les **frontières**
- ❖ **Screaming Architecture**
- ❖ **Architectures d'application .Net**
  - N-tiers
  - Domain-Centric Design
  - principes de **Clean Architecture**
- ❖ **Application de la clean architecture en .Net**
  - Architecture de la solution
  - Librairies utiles
- ❖ **BDD TDD et Gherkin**

# Introduction



# Motivation à la Clean Architecture

1. C'est quoi l'architecture ?

## Motivation à la Clean Architecture

- 1. c'est quoi l'architecture ?**
- 2. Un projet type**
- 3. Quelles sont les erreurs ?**



# Qu'est ce que l'architecture ?

L'architecture, en son sens le plus général,  
est l'art de **concevoir des bâtiments**.

---

En informatique,

L'architecture est **l'expertise de concevoir des logiciels**.



# Quelle importance à l'architecture logicielle?

Tout comme un bâtiment se doit d'être  
**robuste et durer dans le temps,**

Une application doit aussi **faire preuve de pérennité.**

L'objectif, livrer un produit **fiable** et **économique** au client.

# Motivation à la Clean Architecture

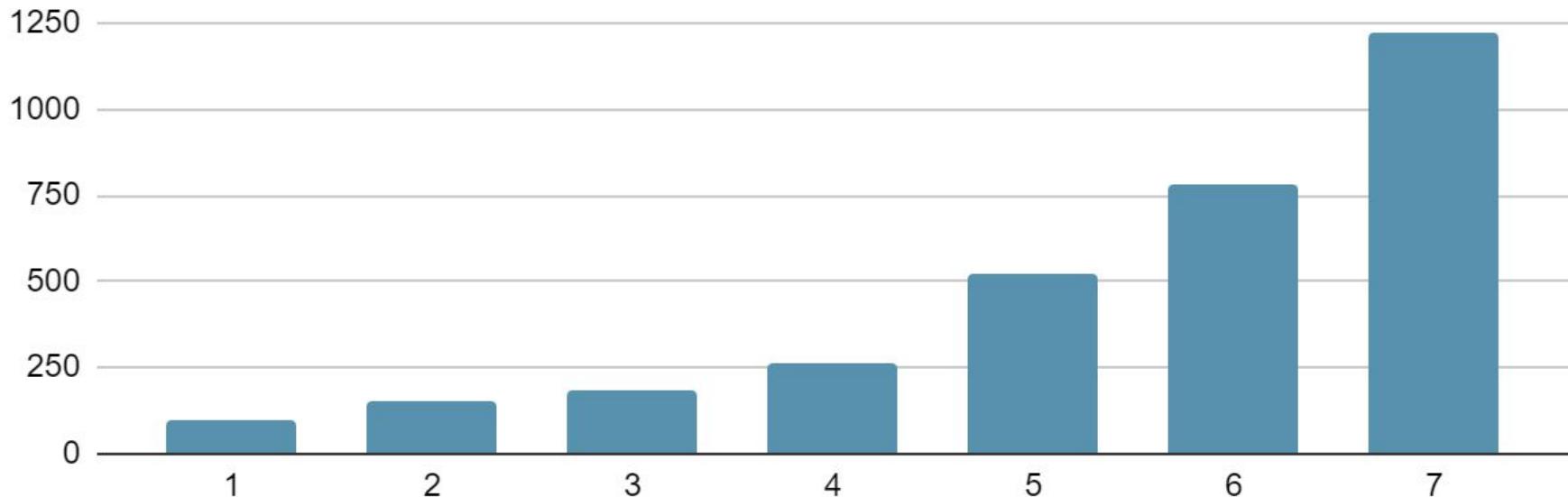
2. Un projet type





# Cycle de vie d'un logiciel leader du marché

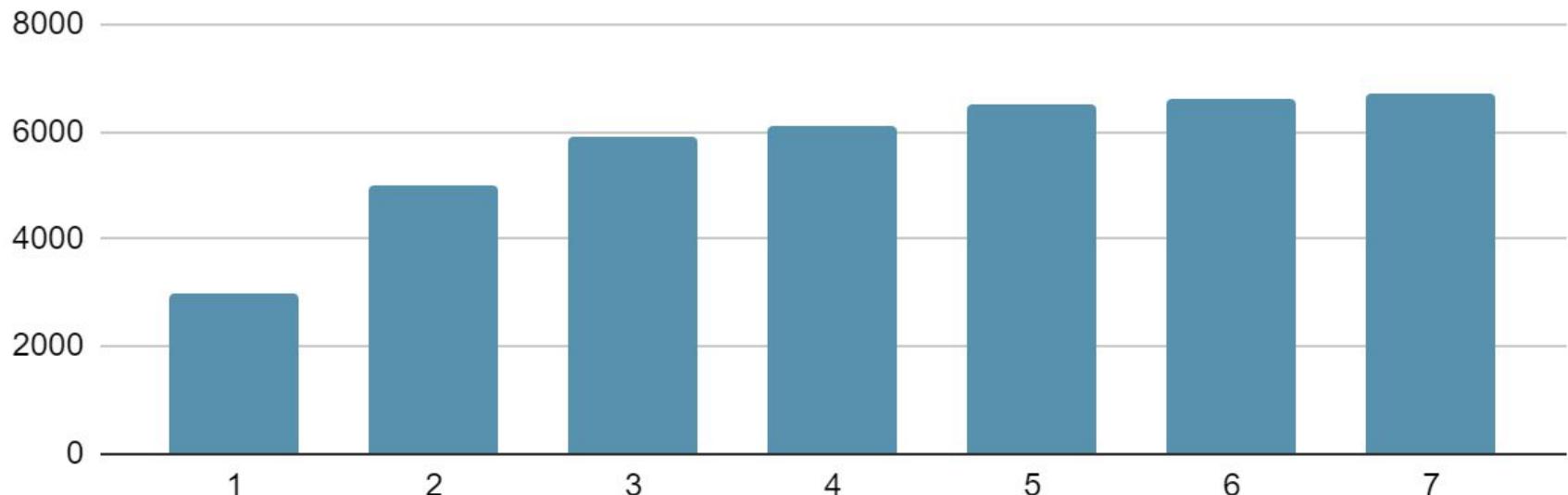
Nombre d'ingénieurs





# Cycle de vie d'un logiciel leader du marché

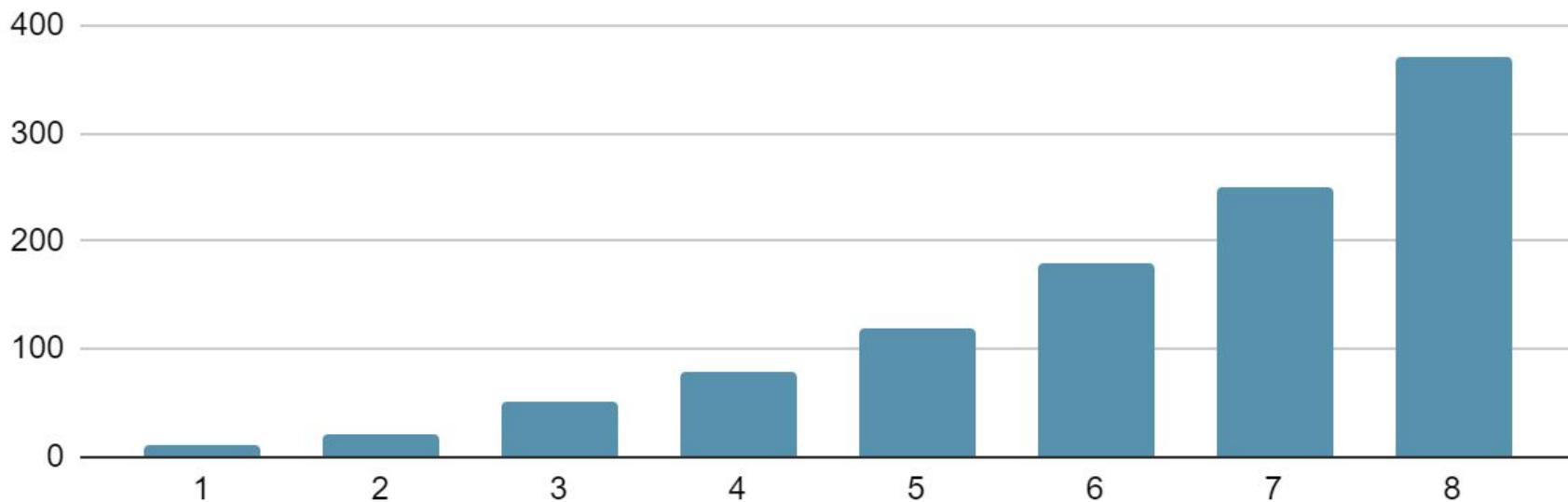
Taille du produit (CLOC lignes de code)





# Cycle de vie d'un logiciel leader du marché

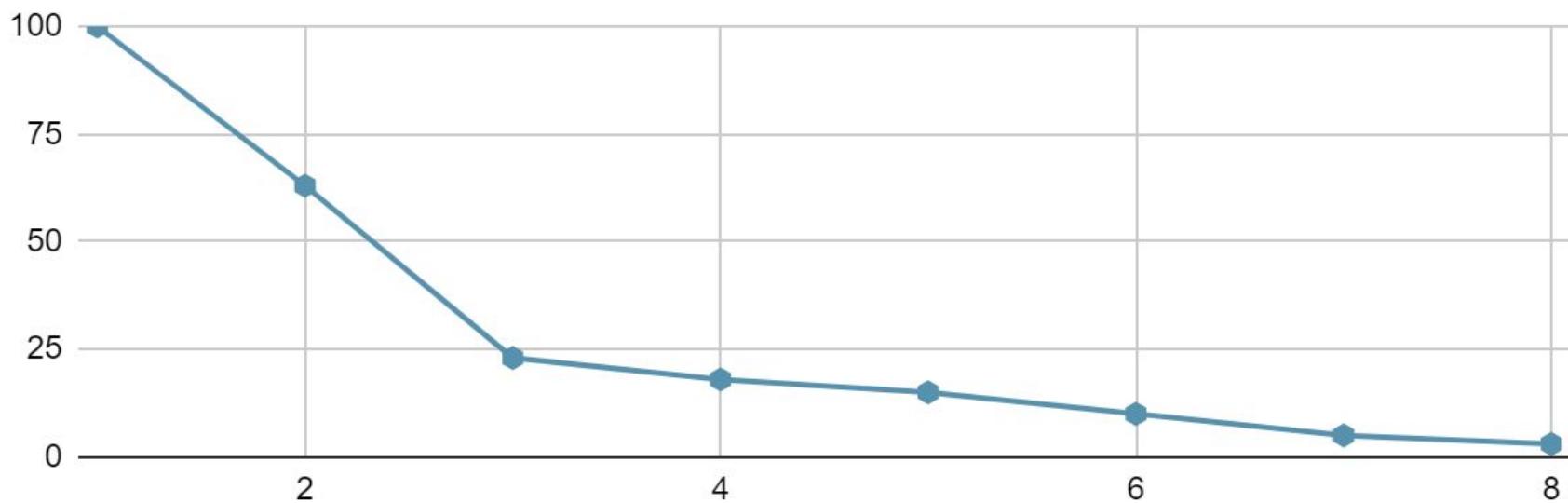
Cout par LOC





# Cycle de vie d'un logiciel leader du marché

Courbe de productivité





# Le cercle vicieux des rustine

Tous ces chiffres démontrent que l'application est devenue “Legacy”.

Le code base est alors plein de défaut qu'on ne peut réparer qu'avec des rustines.

La dette technique grossit, il va falloir payer !! 💸💸💸

Comme toute dette, plus on tarde **plus on a d'intérêts à payer**. Il se peut même qu'elle devienne **impossible à résorber**.





# Les impacts côté utilisateurs..

- UX dégradée
  - Bugs
  - Indisponibilité
  - Plateau de fonctionnalité
- Des Incidents parfois dramatiques
  - Fuite de données client
  - Perte de données
  - Cas grave voir mortel





# Etat français - 2014

## 400 millions d'euros

1850 régimes de paie pour 2,4 millions de fonctionnaires français, c'est l'imbroglio que le logiciel ONP, pour Opérateur national de paie, devait démêler en économisant au passage 190 millions d'euros par an. Pourtant après 7 ans de développement, et un planning qui s'étendait jusqu'à 2024 pour finir le projet, l'Etat a décidé de jeter l'éponge.

Pourquoi ? Le chantier était pharaonique, les objectifs mal définis et la coordination entre les ministères conflictuelle : les germes rêvés pour un échec.



# Institut National du Cancer (Panama) – 2001

**17 morts et 11 blessés**

Surexposés aux radiations émises par la machine qui devait éliminer leurs cellules cancéreuses en quelques séances, les patients panaméens ont en fait été victimes du logiciel gérant la machine.

Pourquoi ? Suivant l'ordre dans lequel les données du traitement étaient saisies par l'opérateur, la dose de radiation était calculée différemment. Les médecins furent traînés devant les tribunaux car la loi panaméenne impose une vérification du calcul par le corps médical. On comprend pourquoi.



# Impacts business

- Rentabilité
- Crédibilité
- productivité
- Compétitivité
- Pertes d'argent colossales
- Départ des développeurs
- Responsabilité légal engagé





# Impact côté développeurs ?

- Stress, angoisse
- Perte de motivation, travail peu gratifiant car peu productif
- Intégration de nouvelle recrue très difficile





# Coté Product manager

- Communication avec le client, la direction et le métier devient tendue et compliquée
- Plus de visibilité sur le projet



# Ce qui s'est passé

Lors des **évolutions fonctionnelles**  
l'application a **évolué techniquement**.

Elle est devenue **instable, difficile** et  
**coûteuse à maintenir**.





# Conséquences

C'est en général à ce moment, que l'on constate que le *coût de maintenance n'est plus acceptable* que l'on commence à parler de **refonte applicative** en pensant qu'elle réglera tous les problèmes.





## Mais ..

- Pourquoi payer pour une application qui fera **la même chose que l'ancienne ?**
- Plus grave, la refonte d'une application voit souvent les **mêmes erreurs se reproduire.**
- Ces erreurs conduiront certainement **à une énième refonte.**

# Motivation à la Clean Architecture

3. Quelles sont les erreurs ?





# Erreur fréquentes

- Besoin métier mal identifiés et non challengés
- User Stories mal rédigées voir pas rédigé du tout, sans règle métier et critères d'acceptation
- Dichotomie entre développeurs et le métier
- Peu ou pas de méthodologie





# Erreur fréquentes

- Violation des principes **S.O.L.I.D**
- **Couplage** entre métier et infra
- Pas ou peu de Test unitaires, car difficile à tester
- Code défensif
- Choix techniques au détriment du **métier**



# Petite parenthèse sur le code défensif

Maintenabilité

- **Complexité accrue** : code plus difficile à comprendre et à maintenir.
- **Manque de souplesse** : code moins flexible aux changements futurs.

Fiabilité

- **Fausses alertes** : erreurs indiquant un problème alors qu'il n'y en a pas.
- **Manque de couverture** : parties du code non protégées contre erreurs ou failles de sécurité.

# **Un discours à mettre en perspective**



# Un discours à mettre en perspective



Changer un robinet  
va demander  
beaucoup de travaux  
dans tout  
l'immeuble.





# Un discours à mettre en perspective



On ne peut pas  
changer la porte  
d'entrée sans prendre  
le risque que la  
maison s'effondre.





# Un discours à mettre en perspective



On ne peut pas  
tester le réseau  
électrique sans  
installer le four





# Un discours à mettre en perspective



Ajouter cette fonctionnalité va demander beaucoup de changement, ça va être très long à réaliser





# Un discours à mettre en perspective



Il va falloir tout  
refactorer pour  
ajouter cette  
fonctionnalité





# Un discours à mettre en perspective



Je ne peux pas  
tester sans  
appelle à la BDD  
ou envoyer un  
mail ...





# Un discours à mettre en perspective



Si on change de  
framework on  
risque de tout  
casser.





# Identification des problèmes

## Entrées

- Couplage au framework
- Couplage à un mode d'exécution
- Couplage à un mode d'exposition (rest, console..)

## Sorties

- Couplage à l'ORM
- Couplage à un modèle de persistance
- Couplage à un client mail, SMS
- etc ..

# Rappel des principes S.O.L.I.D





# S.O.L.I.D

- Responsabilité unique - **Single responsibility principle**
- Ouvert/fermé - **Open/closed principle**
- Substitution de Liskov - **Liskov substitution principle**
- Ségrégation des interfaces - **Interface segregation principle**
- Inversion des dépendances - **Dependency inversion principle**



# S: Single Responsibility Principle (SRP)

Une classe doit avoir **une seule et unique raison de changer**, ce qui signifie qu'une classe ne doit appartenir qu'à **une seule tâche**.



*A class should have one, and only one, reason to change. - Robert C.Martin*

# Que pensez-vous de cette classe ?

```
public class UserService
{
    public void Register(string email, string password)
    {
        if (!ValidateEmail(email))
            throw new ValidationException("Email is not an email");
        var user = new User(email, password);

        SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="Hello foo" });
    }

    public virtual bool ValidateEmail(string email)
    {
        return email.Contains("@");
    }

    public bool SendEmail(MailMessage message)
    {
        _smtpClient.Send(message);
    }
}
```



# Version corrigé

```
public class UserService
{
    EmailService _emailService;
    DbContext _dbContext;
    public UserService(EmailService aEmailService, DbContext aDbContext)
    {
        _emailService = aEmailService;
        _dbContext = aDbContext;
    }
    public void Register(string email, string password)
    {
        if (!_emailService.ValidateEmail(email))
            throw new ValidationException("Email is not an email");
        var user = new User(email, password);
        _dbContext.Save(user);
        emailService.SendEmail(new MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});
    }
}
```

```
public class EmailService
{
    SmtpClient _smtpClient;
    public EmailService(SmtpClient aSmtpClient)
    {
        _smtpClient = aSmtpClient;
    }
    public bool virtual ValidateEmail(string email)
    {
        return email.Contains("@");
    }
    public bool SendEmail(MailMessage message)
    {
        _smtpClient.Send(message);
    }
}
```

# Une autre ?

```
public class Book
{
    0 références
    public string Author { get; set; }

    8 références
    public string Title { get; set; }

    0 références
    public void Save()
    {
        // save to the database
    }
}
```

# Une autre ?

```
public class User
{
    0 références
    public void CreatePost(DbContext dbContext, string postMessage)
    {
        try
        {
            dbContext.Add(postMessage);
        }
        catch (Exception ex)
        {
            dbContext.LogError("Error:", ex.ToString());
            File.WriteAllText("Log.txt", ex.ToString());
        }
    }
}
```



# Exemple 2 : corrigé

```
1 public class Post
2 {
3     private ErrorLogger errorLogger = new ErrorLogger();
4
5     public void CreatePost(DbContext dbContext, string postMessage)
6     {
7         try
8         {
9             dbContext.Add(postMessage);
10        }
11        catch(Exception ex)
12        {
13            errorLogger.Log(ex.ToString());
14        }
15    }
16 }
17
18 public class ErrorLogger
19 {
20     private DbContext _dbContext;
21
22     public ErrorLogger(DbContext dbContext)
23     {
24         _dbContext = dbContext;
25     }
26
27     public void Log
28     {
29         _dbContext.LogError("Error:", ex.ToString());
30         File.WriteAllText("Log.txt", ex.ToString());
31     }
32 }
```



# Un 3ème pour la route ?

```
public class DataRetriever {
    public void RetrieveData() {
        // Code de récupération de données à partir d'une base de données
    }

    public void ProcessData(string data) {
        // Code de traitement de données
    }
}
```



# O: Open/Closed Principle

Les objets ou entités devraient être **ouverts à l'extension** mais fermés à la modification.





# EX : Calculer des airs

```
public class Rectangle{  
    public double Height {get;set;}  
    public double Wight {get;set; }  
}
```

```
public class AreaCalculator {  
    public double TotalArea(Rectangle[] arrRectangles)  
    {  
        double area;  
        foreach(var objRectangle in arrRectangles)  
        {  
            area += objRectangle.Height * objRectangle.Width;  
        }  
        return area;  
    }  
}
```

```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
public class Circle{
    public double Radius {get;set;}
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Rectangle objRectangle;
        Circle objCircle;
        foreach(var obj in arrObjects)
        {
            if(obj is Rectangle)
            {
                area += obj.Height * obj.Width;
            }
            else
            {
                objCircle = (Circle)obj;
                area += objCircle.Radius * objCircle.Radius * Math.PI;
            }
        }
        return area;
    }
}
```

```
public abstract class Shape  
{  
    public abstract double Area();  
}
```

```
public class Rectangle: Shape  
{  
    public double Height {get;set;}  
    public double Width {get;set;}  
    public override double Area()  
    {  
        return Height * Width;  
    }  
}  
public class Circle: Shape  
{  
    public double Radius {get;set;}  
    public override double Area()  
    {  
        return Radius * Radus * Math.PI;  
    }  
}
```

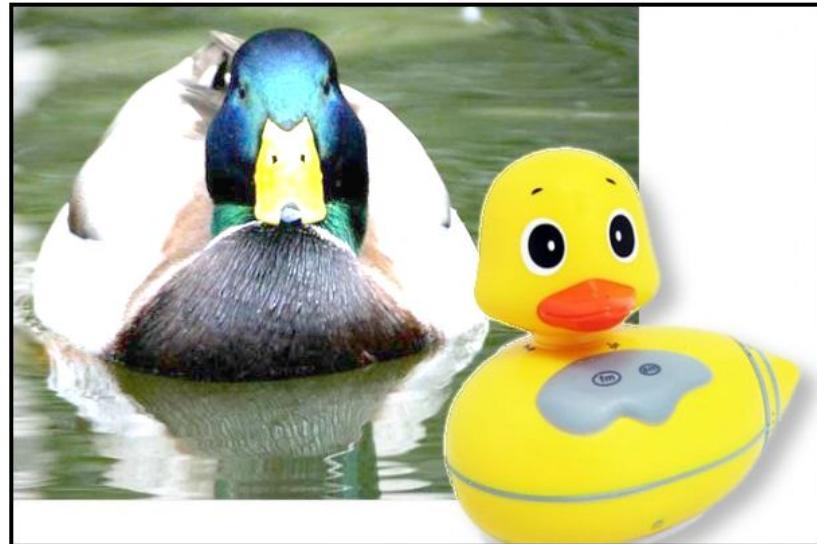
```
public class AreaCalculator  
{  
    public double TotalArea(Shape[] arrShapes)  
    {  
        double area=0;  
        foreach(var objShape in arrShapes)  
        {  
            area += objShape.Area();  
        }  
        return area;  
    }  
}
```



# L: Liskov Substitution Principle (LSP)



Si  $q(x)$  est une propriété démontrable pour tout objet  $x$  de type  $T$ , alors  $q(y)$  est vraie pour tout objet  $y$  de type  $S$  tel que  $S$  est un sous-type de  $T$ .



## LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction



# LSP ?

```
public class SumCalculator
{
    protected readonly int[] _numbers;

    1 référence
    public SumCalculator(int[] numbers)
    {
        _numbers = numbers;
    }

    0 références
    public int Calculate() => _numbers.Sum();
}
```

```
public class EvenNumbersSumCalculator : SumCalculator
{
    0 références
    public EvenNumbersSumCalculator(int[] numbers)
        : base(numbers)
    {

    }

    0 références
    public new int Calculate() => _numbers.Where(x => x % 2 == 0).Sum();
}
```

ÇA ME DÉRANGE PAS



## Pourquoi c'est mal ?

```
public class Program
{
    0 références
    static void Main(string[] args)
    {
        int[] numbers = new int[] { 5, 7, 9, 8, 1, 6, 4 };

        SumCalculator sum = new SumCalculator(numbers);
        Console.WriteLine($"The sum of all the numbers: { sum.Calculate()}");

        EvenNumbersSumCalculator evenSum = new EvenNumbersSumCalculator(numbers);
        Console.WriteLine($"The sum of all the even numbers: {evenSum.Calculate()}");
    }
}
```



# Pour ça !

```
SumCalculator evenSum = new EvenNumbersSumCalculator(numbers);|
```

- La class EvenNumberSumCalculator ne peut pas se substituer à la class SumCalculator car le comportement ne serait plus le même

# Correction

Création d'une classe abstraite  
Calculator.

```
public abstract class Calculator
{
    protected readonly int[] _numbers;

    2 références
    public Calculator(int[] numbers)
    {
        _numbers = numbers;
    }

    2 références
    public abstract int Calculate();
}
```



# Correction

Implémentation de cette classe abstraite.

Le nommage de la classe abstraite ne dit pas ce qu'elle calcule mais uniquement qu'elle possède une méthode qui calcul.

```
public class SumCalculator : Calculator
{
    0 références
    public SumCalculator(int[] numbers)
        : base(numbers)
    {
    }

    1 référence
    public override int Calculate() => _numbers.Sum();
}

1 référence
public class EvenNumbersSumCalculator : Calculator
{
    0 références
    public EvenNumbersSumCalculator(int[] numbers)
        : base(numbers)
    {
    }

    1 référence
    public override int Calculate() => _numbers.Where(x => x % 2 == 0).Sum();
}
```



# Correction

Ainsi le retour n'est plus inattendu.

```
public class Program
{
    static void Main(string[] args)
    {
        int[] numbers = new int[] { 5, 7, 9, 8, 1, 6, 4 };

        Calculator sum = new SumCalculator(numbers);
        Console.WriteLine($"The sum of all the numbers: {sum.Calculate()");

        Calculator evenSum = new EvenNumbersSumCalculator(numbers);
        Console.WriteLine($"The sum of all the even numbers: {evenSum.Calculate()}");
    }
}
```



# I: Interface Segregation Principle (ISP)

Un client ne doit **jamais être** **forcé à installer une interface** **qu'il n'utilise pas** et les clients ne doivent **pas être forcés à** dépendre de méthodes qu'ils **n'utilisent pas.**



Interface Segregation Principle

---

When more means less



# Pourquoi cette interface pose problème

```
public interface IShape
{
    public double TotalArea();
    public double TotalVolume();
}
```

```
public interface IShape
{
    public double TotalArea();
}

public interface IVolumeShape
{
    public double TotalVolume();
}
```

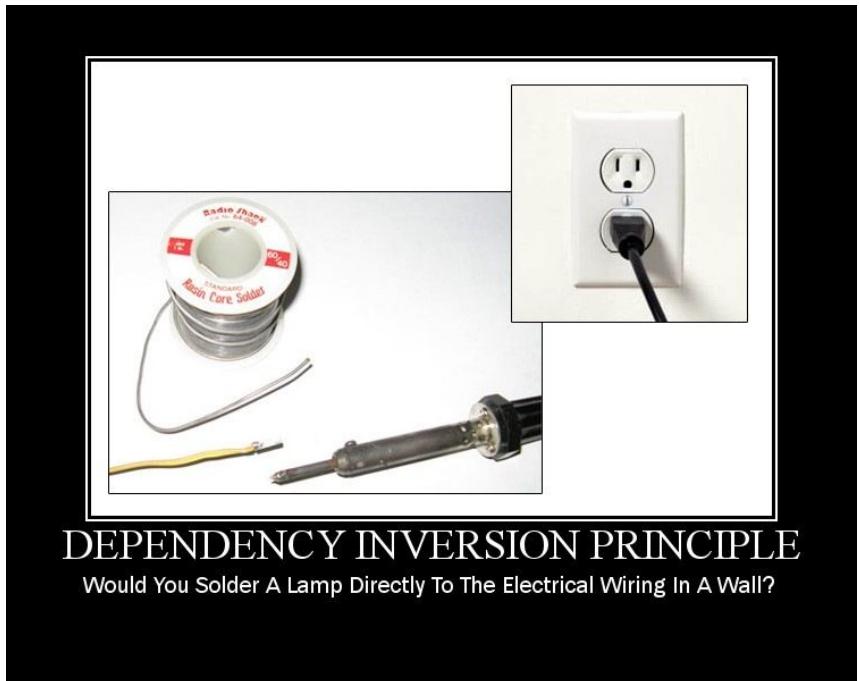
```
public class Cuboid : IShape, IVolumeShape
{
    public double TotalArea()
    {
        // Calcule l'air
    }

    public double TotalVolume()
    {
        // Calcule le volum
    }
}

public class Square : IShape
{
    public double TotalArea()
    {
        // Calcule l'air
    }
}
```

# D: Dependency Inversion Principle

Les entités doivent **dépendre des abstractions, pas des implémentations**. Il indique que le module de haut niveau ne doit pas dépendre du module de bas niveau, mais qu'ils doivent dépendre des abstractions.



# Exemple



```
1 public class Email
2 {
3     public string ToAddress { get; set; }
4     public string Subject { get; set; }
5     public string Content { get; set; }
6
7     public void SendEmail()
8     {
9         //Send email
10    }
11 }
12
13 public class SMS
14 {
15     public string PhoneNumber { get; set; }
16     public string Message { get; set; }
17
18     public void SendSMS()
19     {
20         //Send sms
21     }
22 }
23
24 public class Notification
25 {
26     private Email _email;
27     private SMS _sms;
28
29     public Notification()
30     {
31         _email = new Email();
32         _sms = new SMS();
33     }
34
35     public void Send()
36     {
37         _email.SendEmail();
38         _sms.SendSMS();
39     }
40 }
```

# Correction

```
public interface IMessage
{
    void SendMessage();
}

public class Email : IMessage
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }

    public void SendMessage()
    {
        //Send email
    }
}

public class SMS : IMessage
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }

    public void SendMessage()
    {
        //Send sms
    }
}
```

```
public class Notification
{
    private ICollection<IMessage> _messages;

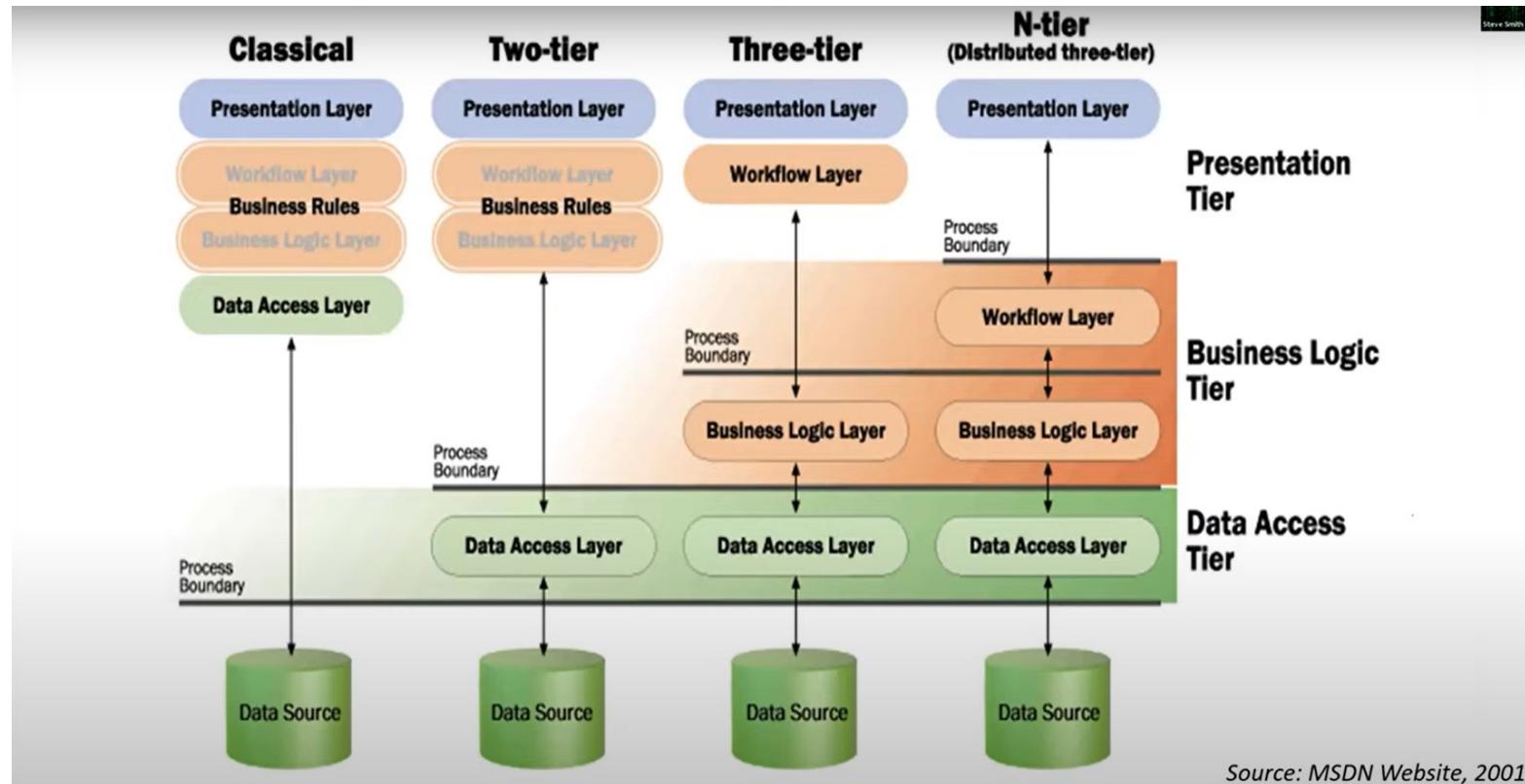
    public Notification(ICollection<IMessage> messages)
    {
        this._messages = messages;
    }

    public void Send()
    {
        foreach(var message in _messages)
        {
            message.SendMessage();
        }
    }
}
```

# Architectures d'applications .Net



# “Classic” N-tier Architecture

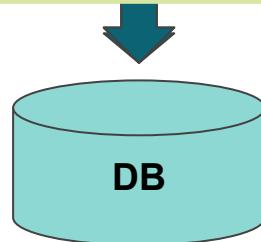


Source: MSDN Website, 2001

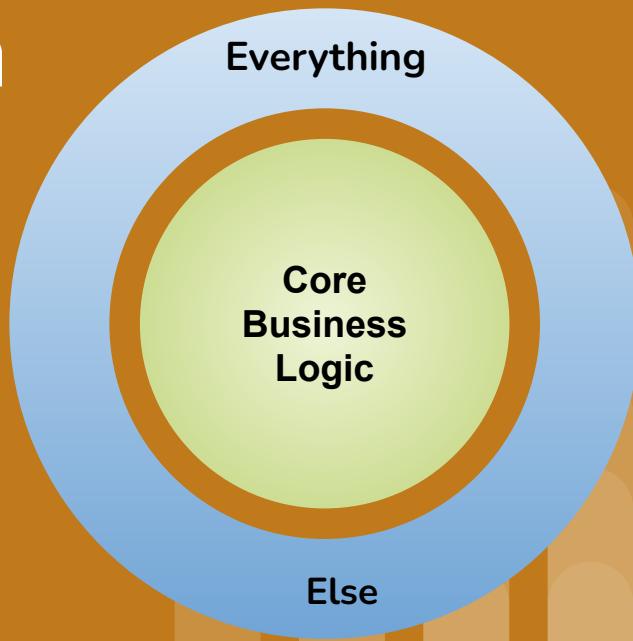
# Dépendance transitive

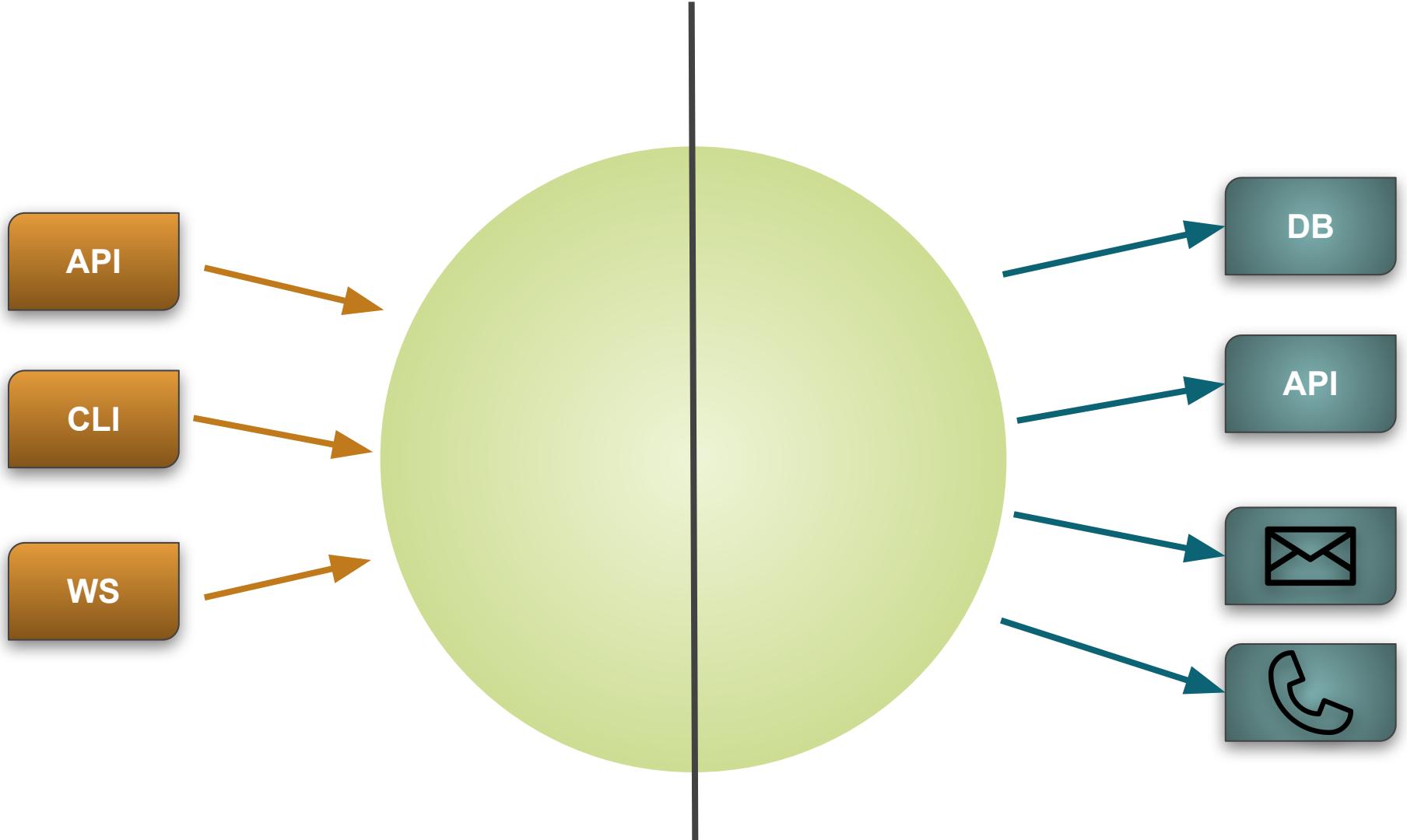
User Interface

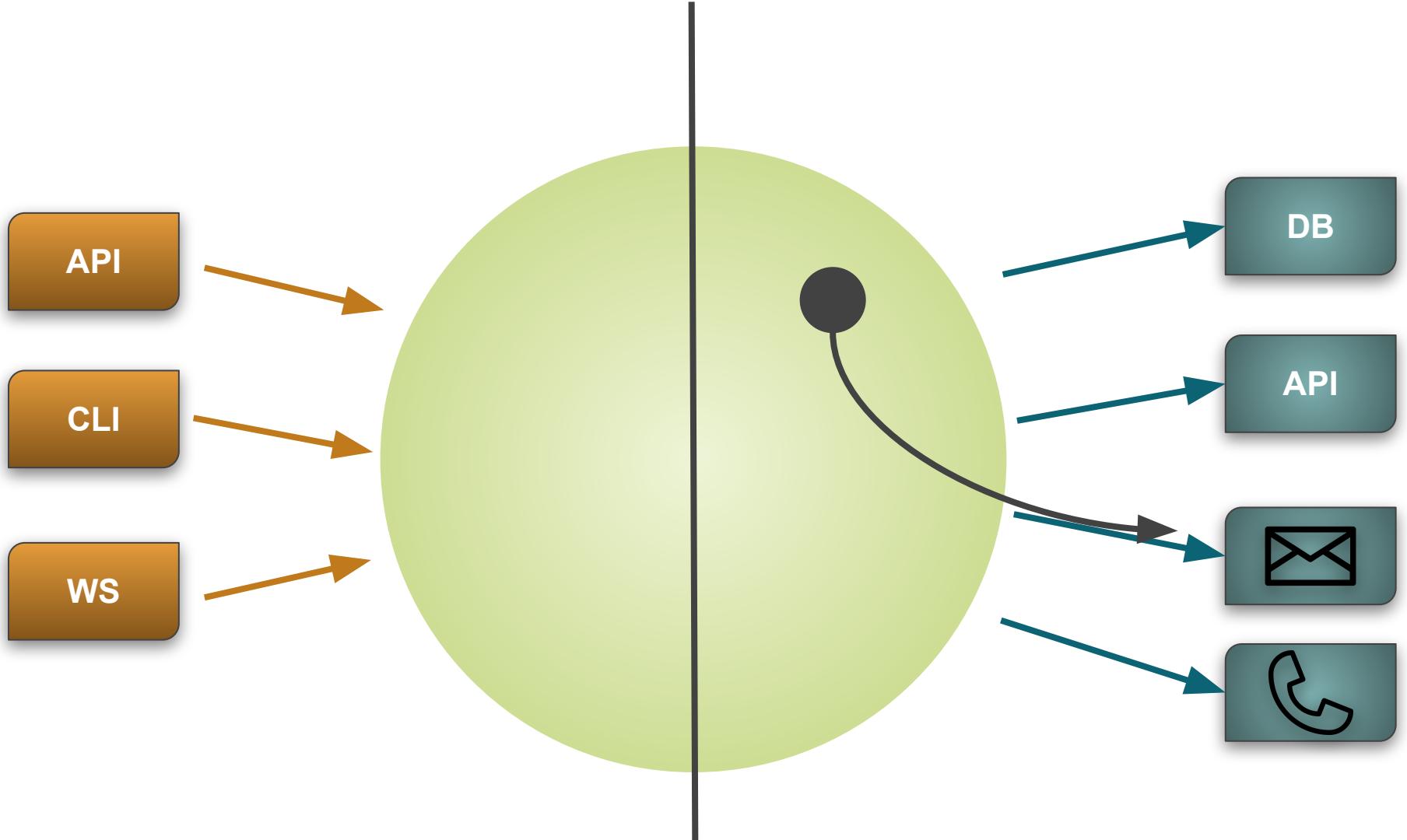
**TOUT A UNE DEPENDANCE A LA DB !!**

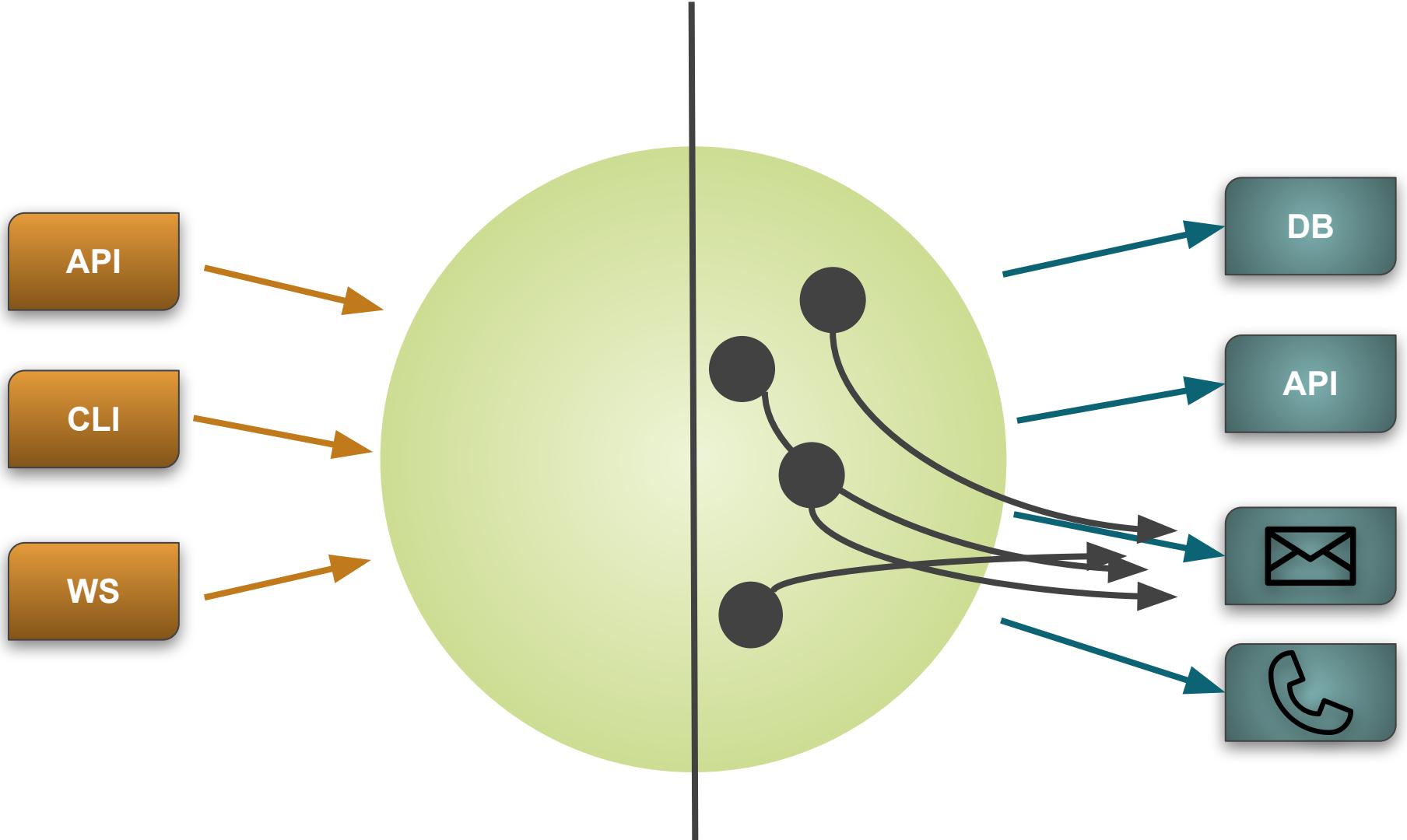


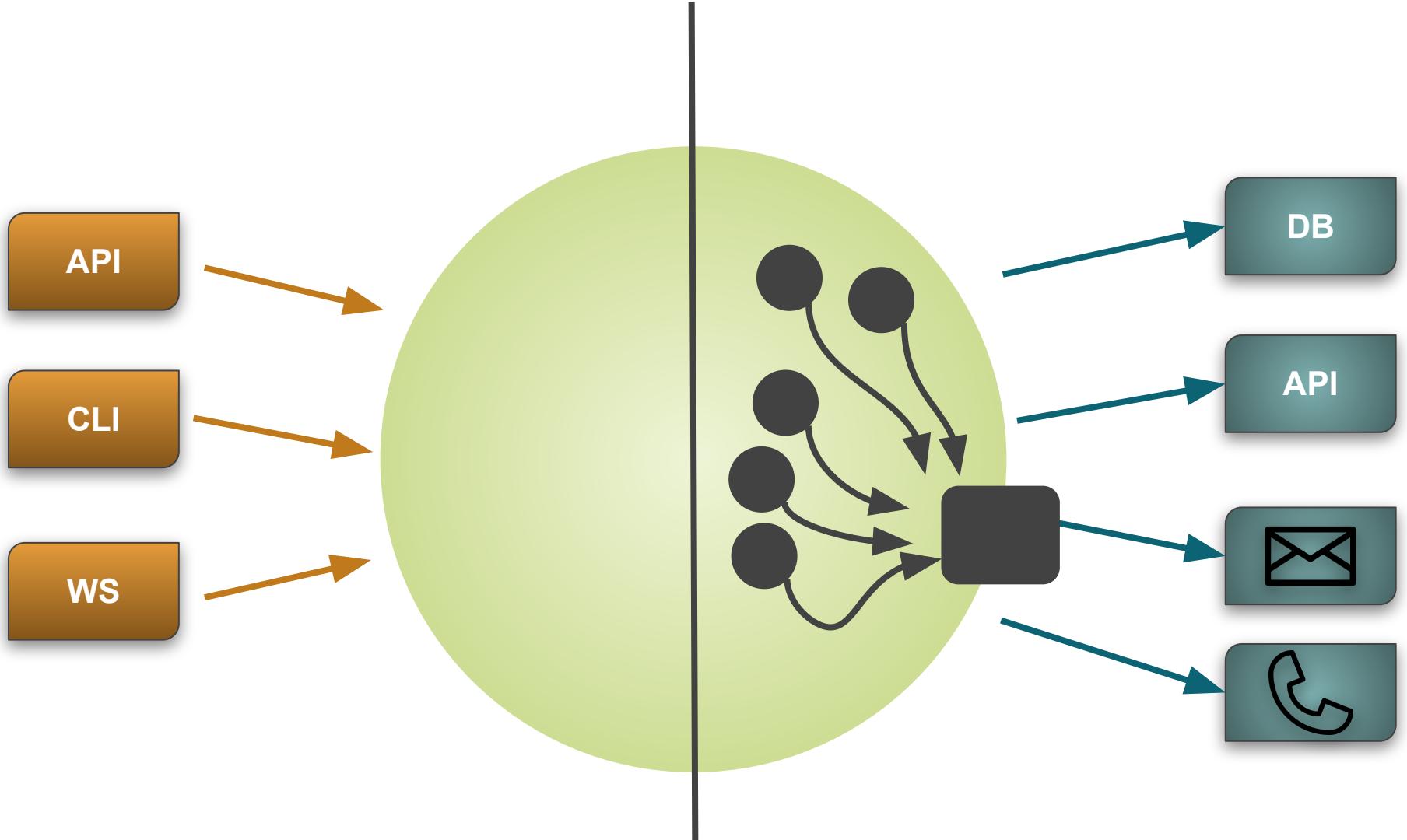
# Domain-Centric Design Et Clean Architecture

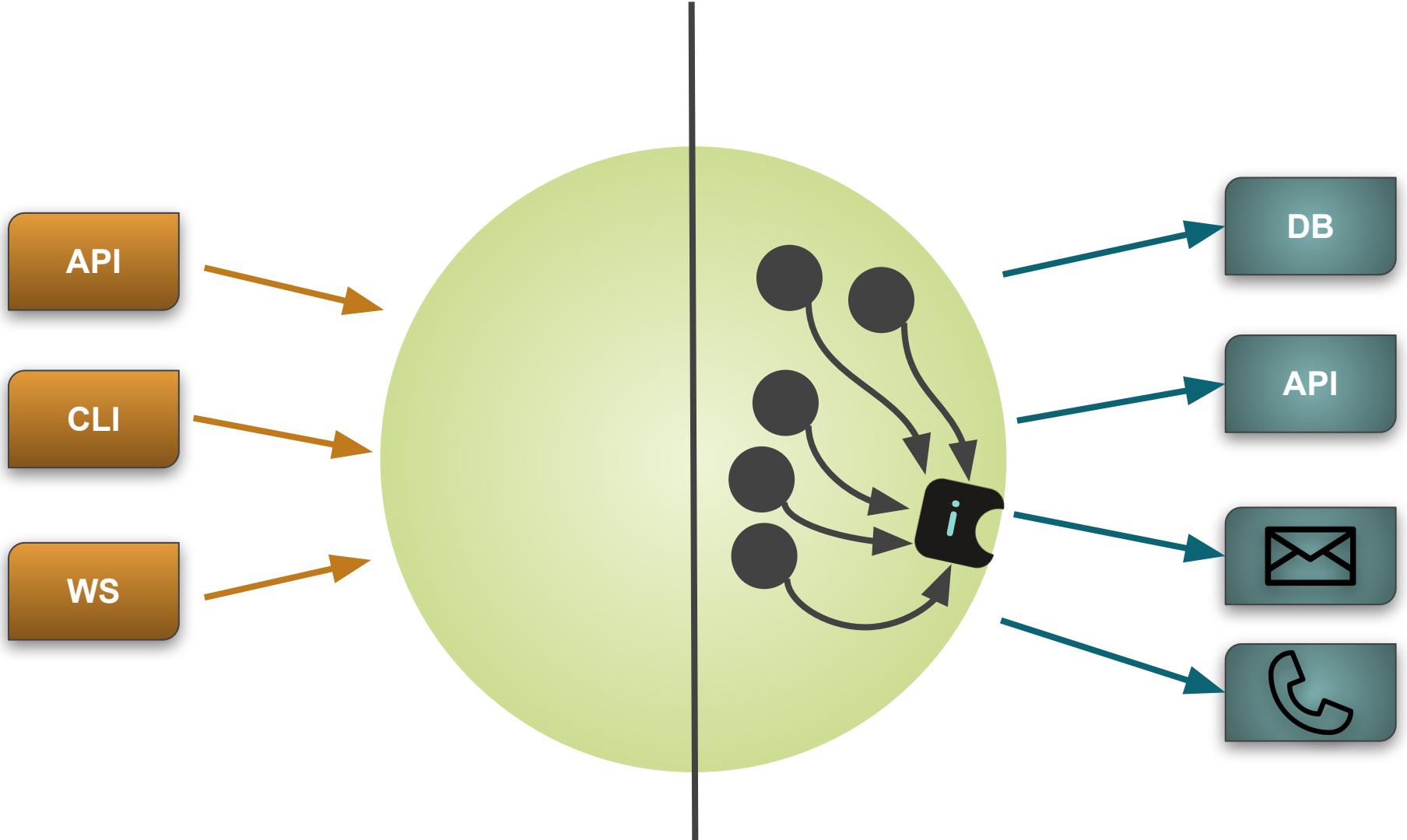


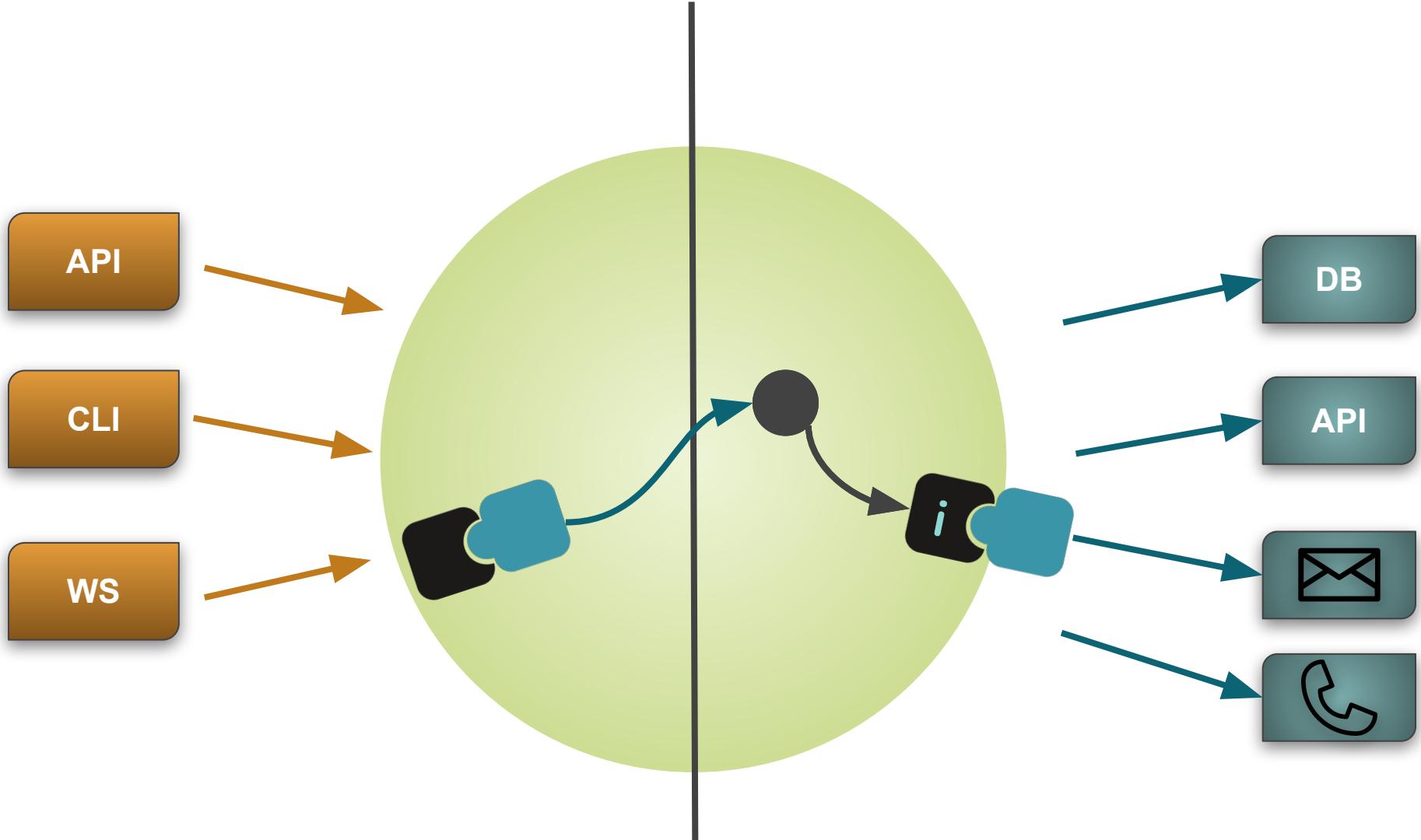










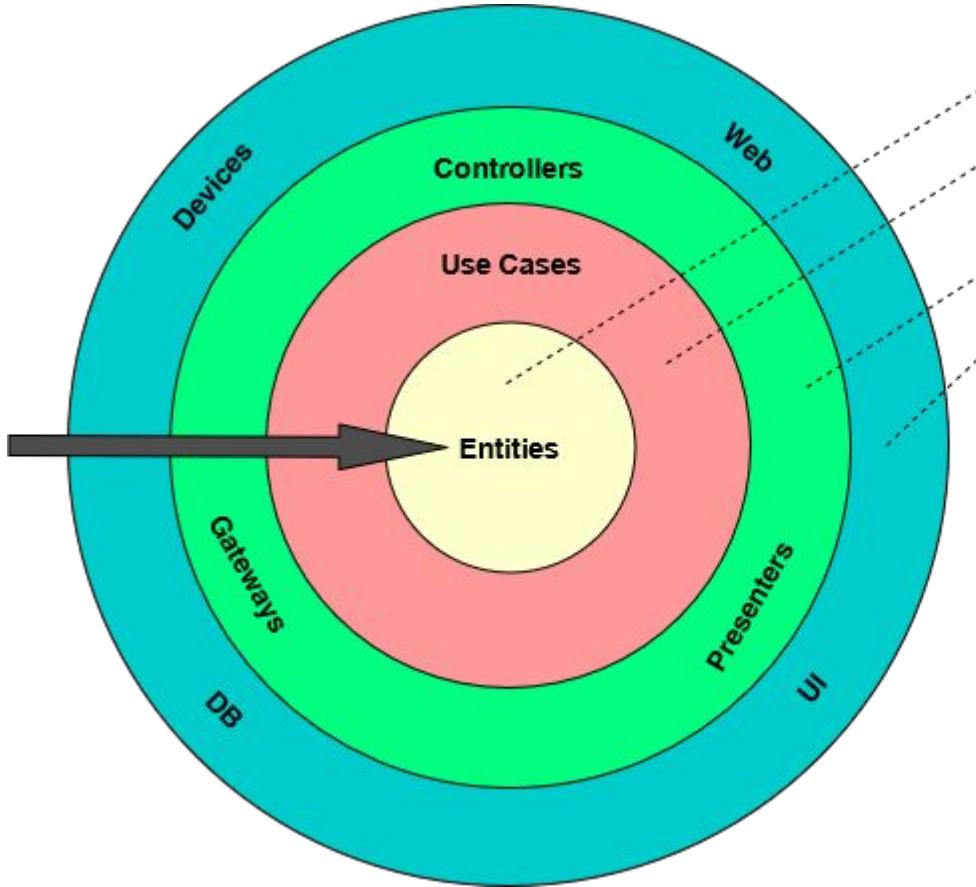


Primary actors



Secondary actors

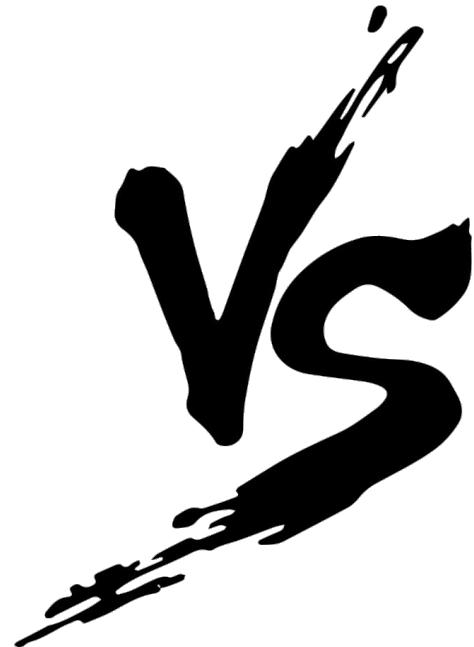
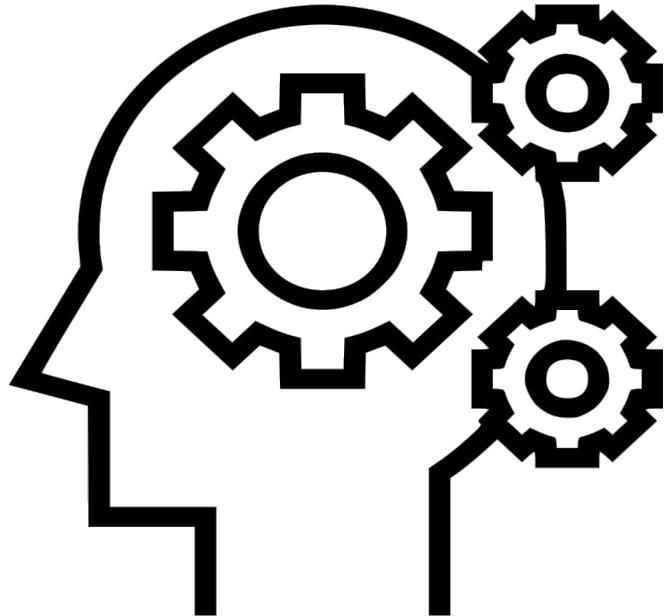




# Principes de Clean Architecture



# Quel est l'élément clé de votre application logiciel ?





# Les "Use case" : le cœur et l'âme de votre application !

*LOGIN*

*Register*

*Search*

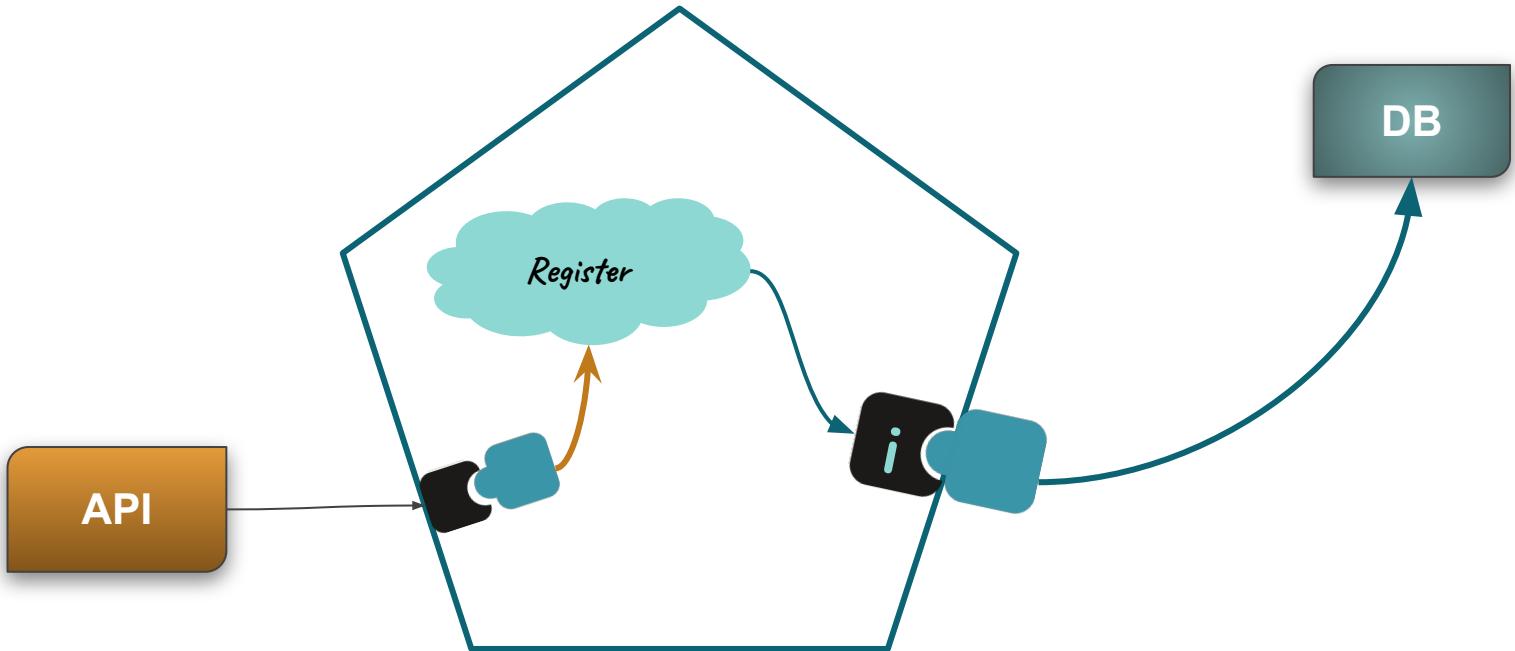
*Add to basket*

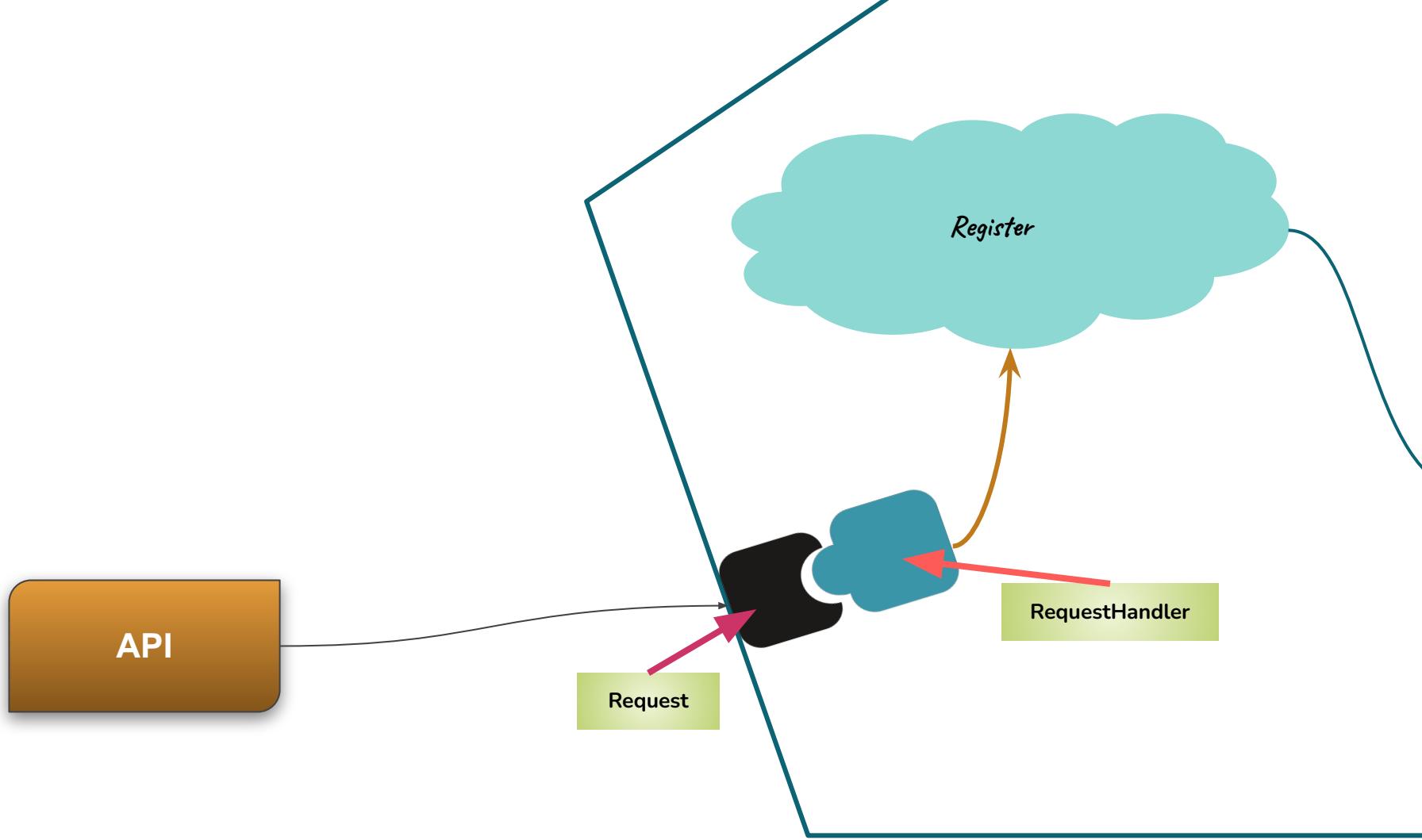
*Add credit card*

*Add product*

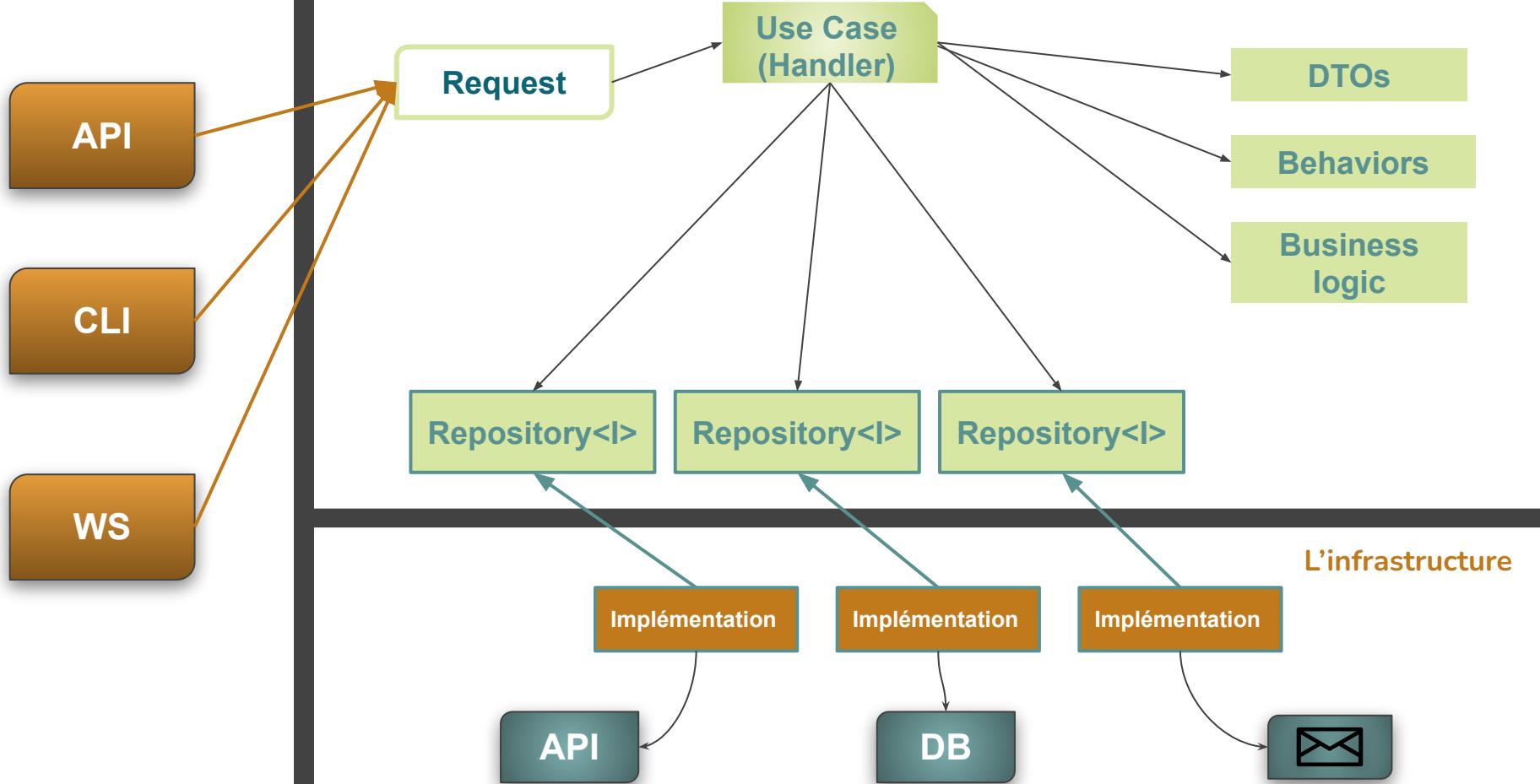
*Search*

# Exemple fonctionnel pour un Use Case





Les entrées à  
notre application





# Caractéristiques d'une Clean Architecture

- **Indépendance au Framework** : l'architecture ne dépend pas d'une librairie ou d'un framework et n'est donc pas contrainte à leurs limites
- **Testable** : Les règles métier peuvent être testé sans UX, DB, Web server, etc..
- **Indépendance à l'IUI** : indépendant de l'interface utilisateur on peut facilement passer d'une interface web à une interface console par exemple.
- **Indépendance à la DB** : On peut switcher d'un type de DB à une autre (SqlServer vers Mongo par exemple) sans souci car notre logique métier n'est pas liées au type de DB

**! Les règles métier ne connaissent rien de l'implémentation de leurs interfaces.**



# Le Domain

Le Domain est la couche qui contient la **logique métier pure**, contenue notamment dans les modèles du Domain (Domain models).

Un **Domain model** contient à la fois des comportements et des données contrairement à un **DTO** qui ne contient que des données ou une façade qui ne contient que du comportement.

*Exemple de logique métier pure: La formule pour calculer les intérêts sur un prêt est la même, qu'elle soit utilisée sur une feuille de papier ou implémentée dans une application.*



# Les use cases & le CQRS

Les use cases sont les **intentions** des **utilisateurs** dans le **système**.

Les use cases utilisent et dépendent des **modèles du Domain**.

*Exemple: En tant que banquier, je veux pouvoir simuler le coût d'un prêt immobilier sur N années.*

Une classe de use case contient une seule méthode publique généralement appelée execute, query ou command et qui interagit notamment avec des repositories.

Ces notions de Query et Command viennent d'un pattern appelé CQRS (Command Query Responsibility Segregation).



# Les use cases & le CQRS

Lecture

Ecriture

**Query**

**Command**

Pas de changement d'état du système

Changement d'état du système

Pas de changement d'état du système

Ne retourne rien



# Les adapters primaires / left

Les **adapters primaires** permettent à l'utilisateur d'**accéder au use case**. Ce sont nos points d'entrée de l'application.

Exemple d'adapters primaires:

- Une route (REST)
- Une commande shell
- Un cron



## Les adapters primaires / left

On ne retourne jamais un objet métier à un adaptateur mais uniquement un DTO.

Ceci afin d'éviter la mutation inopinée de nos dos données.



# Les adapters secondaires / right

Les **adaptateurs secondaires** sont les points de **sortie** de l'application. Ils permettent au système de communiquer avec l'extérieur.

Exemple d'adapters secondaires:

- Un repository
- Un service de référentiel
- Un client mail, sms

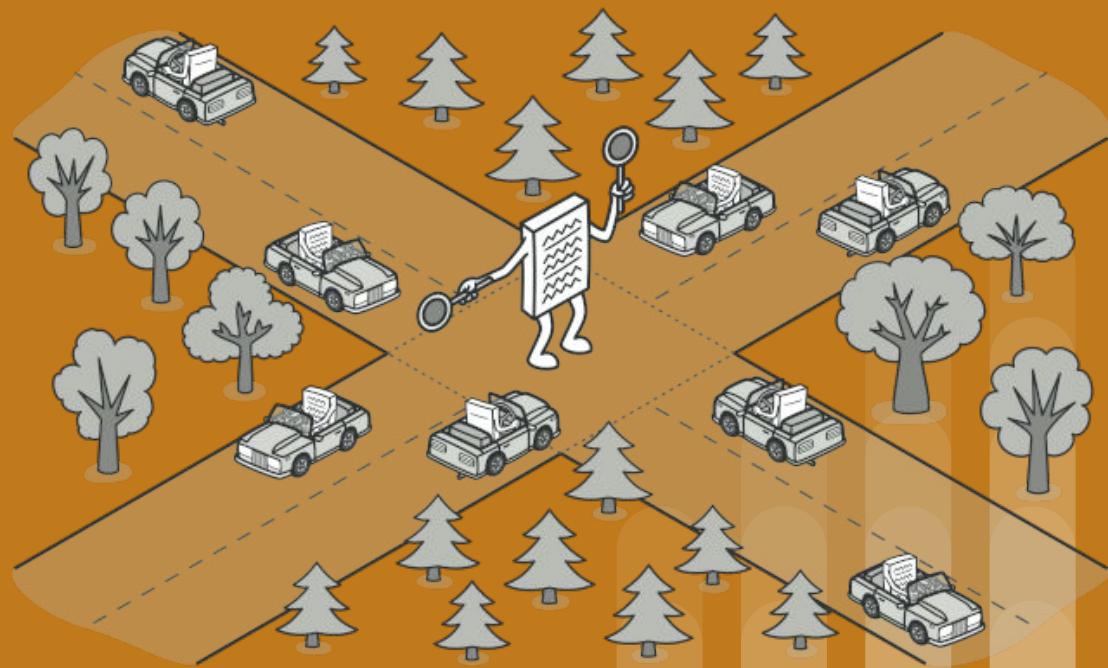
Les adapters (primaires comme secondaires) ne doivent pas avoir connaissance du Domain.



# Avantages et inconvénients de la Clean Architecture

- Plus de classes
- Plus de temps pour démarrer un projet
  
- + Indépendance au Framework
- + Indépendance à la BDD
- + Indépendance à l'UI
- + Indépendance à toutes librairies externe
- + Testable
- + Maintenable dans le temps
- + Lisible
- + Coût à la ligne de code stable au fil des releases

# Patron de conception Médiateur





# Introduction au pattern Mediator

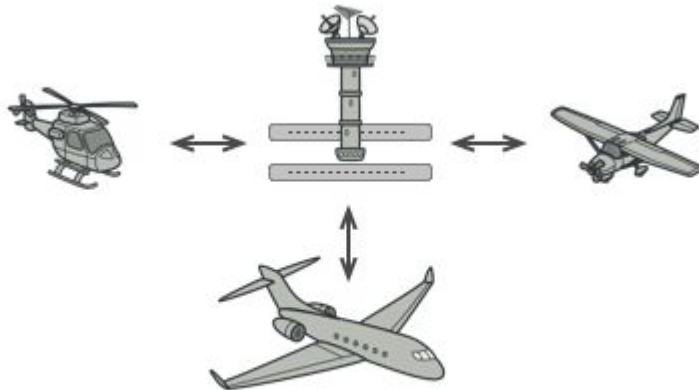
Le pattern **Mediator** est un type de design pattern comportemental qui est utilisé pour gérer les interactions complexes entre plusieurs objets.

Il fonctionne en introduisant un objet "**médiateur**" qui encapsule et contrôle comment ces objets interagissent entre eux.

Cela permet de **réduire le couplage entre les objets**, améliorant ainsi la flexibilité et la maintenabilité du logiciel.



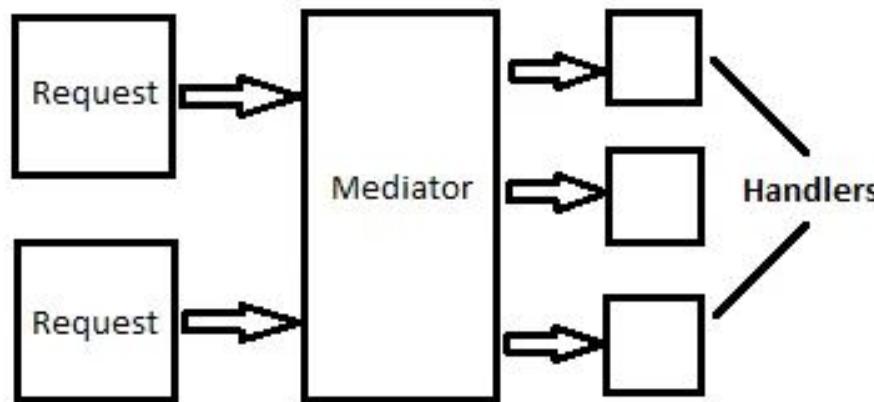
# Analogie



*Les pilotes d'avion ne communiquent pas directement ensemble pour déterminer qui sera le prochain à atterrir. Toute communication passe par la tour de contrôle.*



# Médiateur



```
public class AddSessionRequest : IRequest<int>
{
    4 références
    public DateTime StartDate { get; set; }
    3 références
    public DateTime EndDate { get; set; }
    3 références
    public string Label { get; set; }
    2 références
    public string Description { get; set; }
    3 références
    public int OwnerId { get; set; }
}

1 référence
public class AddSessionRequestHandler : IRequestHandler<AddSessionRequest, int>
{
    private readonly SessionRepository _repository;

    0 références
    public AddSessionRequestHandler(SessionRepository repository)
    {
        _repository = repository;
    }

    0 références
    public async Task<int> Handle(AddSessionRequest request, CancellationToken cancellationToken)
    {
        var session = new SessionDTO
        {
            Label = request.Label,
            Description = request.Description,
            StartDate = request.StartDate,
            EndDate = request.EndDate,
            Owner = new UserDTO { Id = request.OwnerId }
        };
        return await _repository.AddSession(session);
    }
}
```



# Appel au médiateur

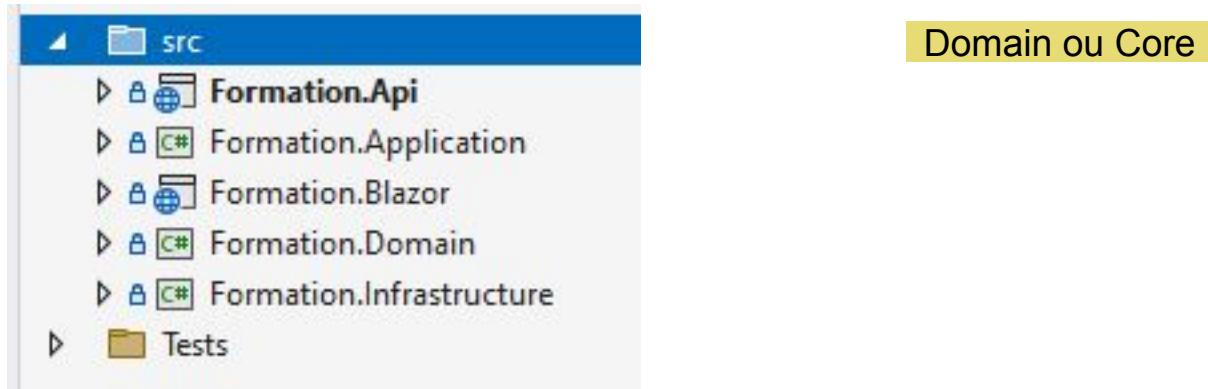
```
[HttpPost]  
0 références  
public async Task<IActionResult> AddSession([FromBody] AddSessionRequest request)  
{  
    return Ok(await _mediator.Send(request));  
}
```

# Application sur projet .Net



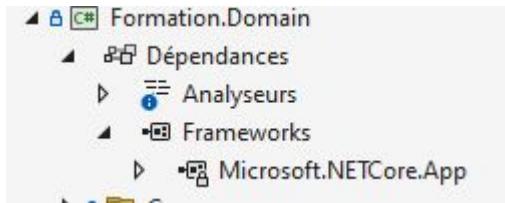
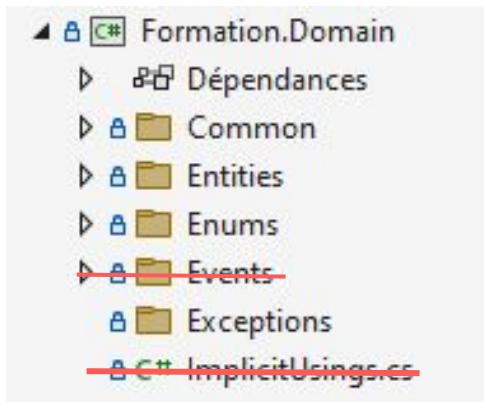
# Architecture de la solution

# Exemple d'arborescence



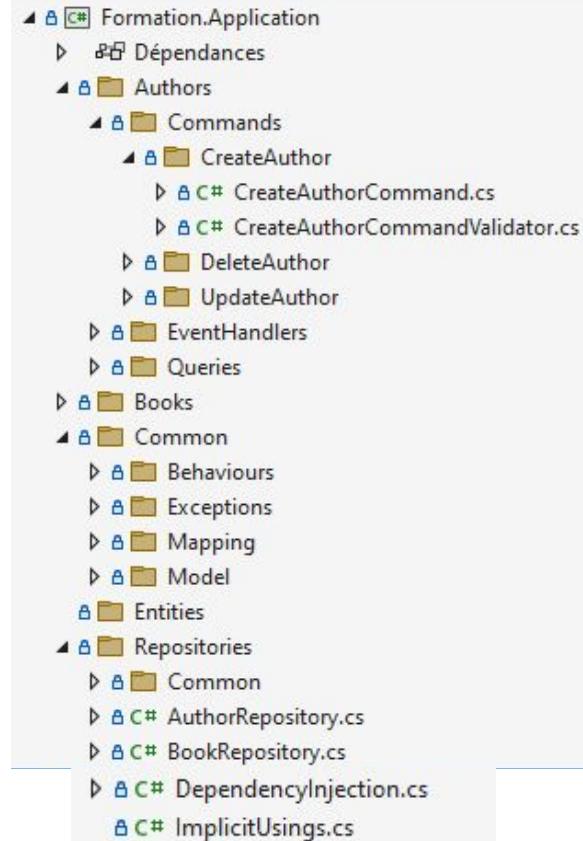
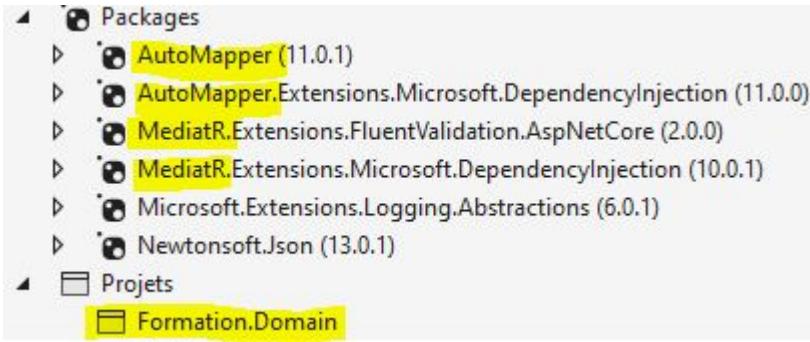


# Le domain



⚠ Aucune dépendances ⚠

# L'application





# L'application

```
namespace Formation.Application;
```

```
public static IServiceCollection AddApplication(this IServiceCollection services)
{
    var assembly = Assembly.GetExecutingAssembly();
    services.AddScoped<IDateTimeHelper, DateTimeHelper>();
    services.AddValidatorsFromAssembly(assembly);
    services.AddMediatR(c=> {
        c.RegisterServicesFromAssembly(assembly);
        c.AddBehavior(typeof(IPipelineBehavior<,>), typeof(UnhandledExceptionBehavior<,>));
        c.AddBehavior(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>)); });

    return services;
}
```

0 références

```
public static class DependencyInjection
```

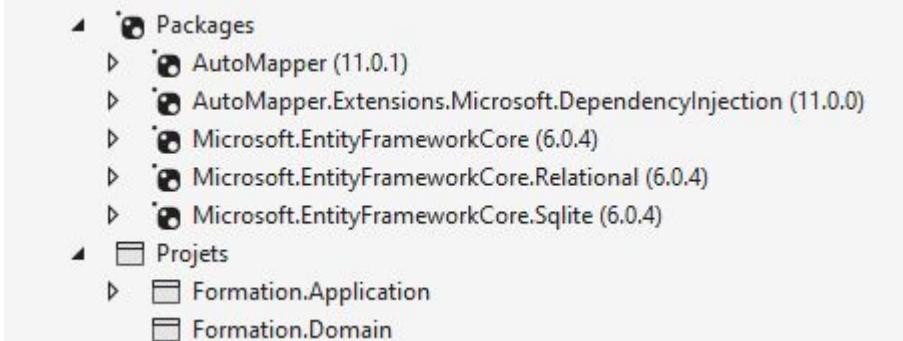
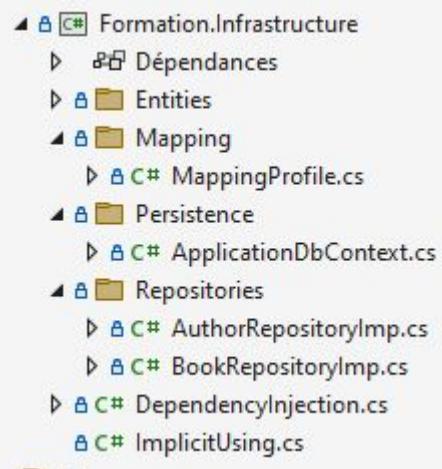
```
{
```

3 références

```
public static IServiceCollection AddApplication(this IServiceCollection services)
{
    var ass = Assembly.GetExecutingAssembly();

    services.AddAutoMapper(ass);
    services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
    services.AddMediatR(Assembly.GetExecutingAssembly());
    services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));
    services.AddTransient(typeof(IPipelineBehavior<,>), typeof(UnhandledExceptionBehaviour<,>));
    return services;
}
```

# L'infrastructure





# L'infrastructure : injection de dépendance

```
1  using System.Reflection;
2  using Formation.Infrastructure.Repositories;
3
4  namespace Formation.Infrastructure;
5  public static class DependencyInjection
6  {
7      public static IServiceCollection AddInfrastructure(this IServiceCollection services, IConfiguration configuration)
8      {
9          services.AddDbContext<ApplicationDbContext>(options => options
10              .UseSqlite(configuration.GetSection("Sqlite").Value));
11          services.AddAutoMapper(Assembly.GetExecutingAssembly());
12          services.AddScoped<AuthorRepository, AuthorRepositoryImp>();
13          services.AddScoped<BookRepository, BookRepositoryImp>();
14          return services;
15      }
16  }
```



Injection des repositories



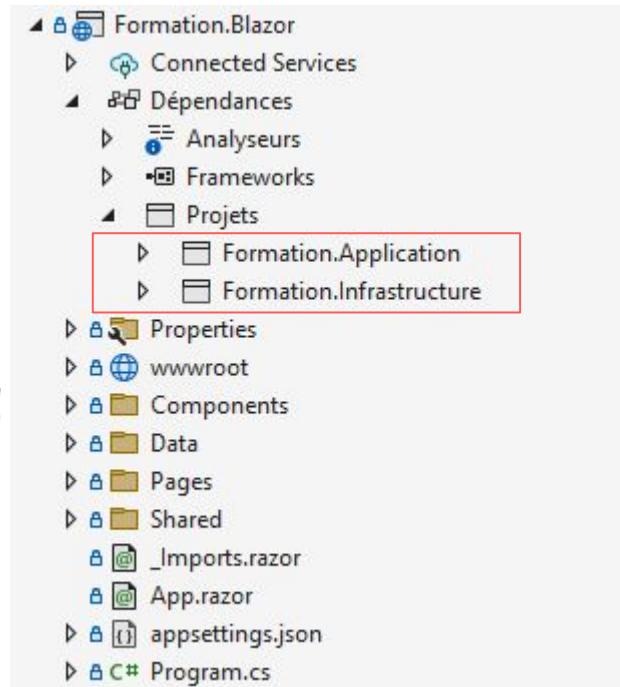
# L'UI

Infra uniquement pour l'injection de dépendance

```
global using Formation.Application;
global using Formation.Infrastructure;
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseWebRoot("wwwroot").UseStaticWebAssets();
// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddInfrastructure(builder.Configuration);
builder.Services.AddApplication();
builder.Services.AddAutoMapper(Assembly.GetExecutingAssembly());
```



# Librairie utilisées

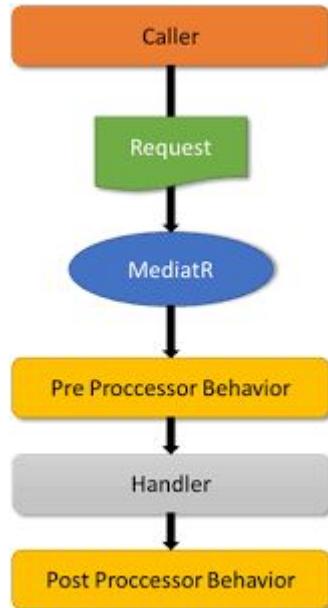


# MediatR



**Médiateur** est un **patron de conception** comportemental qui diminue les dépendances chaotiques entre les objets. Il restreint les communications directes entre les objets et les force à collaborer *uniquement via un objet médiateur*.

# MediatR : Behaviors





# MediatR : Behaviors



```
namespace Formation.Application;

0 références
public static class DependencyInjection
{
    3 références
    public static IServiceCollection AddApplication(this IServiceCollection services)
    {
        var ass = Assembly.GetExecutingAssembly();

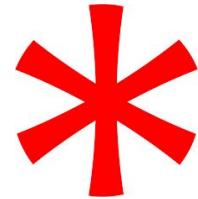
        services.AddAutoMapper(ass);
        services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());
        services.AddMediatR(Assembly.GetExecutingAssembly());
        services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));
        services.AddTransient(typeof(IPipelineBehavior<,>), typeof(UnhandledExceptionBehaviour<,>));
        return services;
    }
}
```



```
5 public class ValidationBehaviour<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
6     where TRequest : IRequest<TResponse>
7
8     private readonly IEnumerable<IValidator<TRequest>> _validators;
9
10    0 références
11    public ValidationBehaviour(IEnumerable<IValidator<TRequest>> validators)
12    {
13        _validators = validators;
14    }
15
16    0 références
17    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
18    {
19        if (_validators.Any())
20        {
21            var context = new ValidationContext<TRequest>(request);
22
23            var validationResults = await Task.WhenAll(
24                _validators.Select(v =>
25                    v.ValidateAsync(context, cancellationToken)));
26
27            var failures = validationResults
28                .Where(r => r.Errors.Any())
29                .SelectMany(r => r.Errors)
30                .ToList();
31
32            if (failures.Any())
33                throw new ValidationException(failures);
34        }
35    }
36
```



# Fluent validation: les validateur



La doc : <https://docs.fluentvalidation.net/en/latest/>

Lib Nuget : On utilisera celle de Mediatr qui permet l'injection dans le behaviors **MediatR.Extensions.Fluent Validation.Asp Net Core**



```
public class CreateBookCommandValidator : AbstractValidator<CreateBookCommand>
{
    private readonly BookRepository bookRepository;
    private readonly AuthorRepository authorRepository;

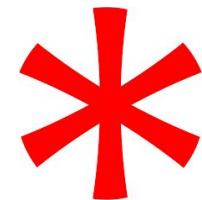
    0 références
    public CreateBookCommandValidator(BookRepository bookRepository, AuthorRepository authorRepository)
    {
        this.bookRepository = bookRepository;

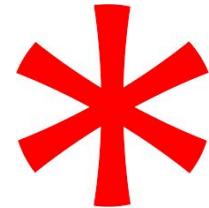
        RuleFor(b => b.AuthorId)
            .NotEmpty().WithMessage("{PropertyName} is required")
            .NotNull().WithMessage("{PropertyName} is required")
            .MustAsync(AuthorShouldExist).WithMessage("This author is unknow");

        RuleFor(b => b.Title)
            .NotEmpty().WithMessage("{PropertyName} is required")
            .NotNull().WithMessage("{PropertyName} is required")
            .MustAsync(BeUniqueTitle).WithMessage("The specified title already exist");
        this.authorRepository = authorRepository;
    }

    1 référence
    public async Task<bool> BeUniqueTitle(string title, CancellationToken cancellationToken) //Nope
    {
        return await bookRepository.GetByTitle(title) == null;
    }

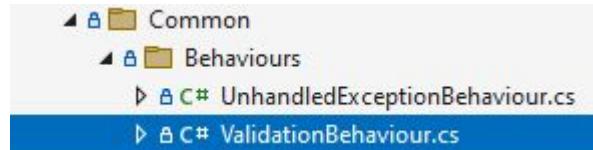
    1 référence
    public async Task<bool> AuthorShouldExist(int id, CancellationToken cancellationToken)
    {
        return await authorRepository.GetById(id) != null;
    }
}
```





# Fluent validation: les validateur

On n'appellera pas directement le validateur dans un Handler  
mais on le fera via les behaviors

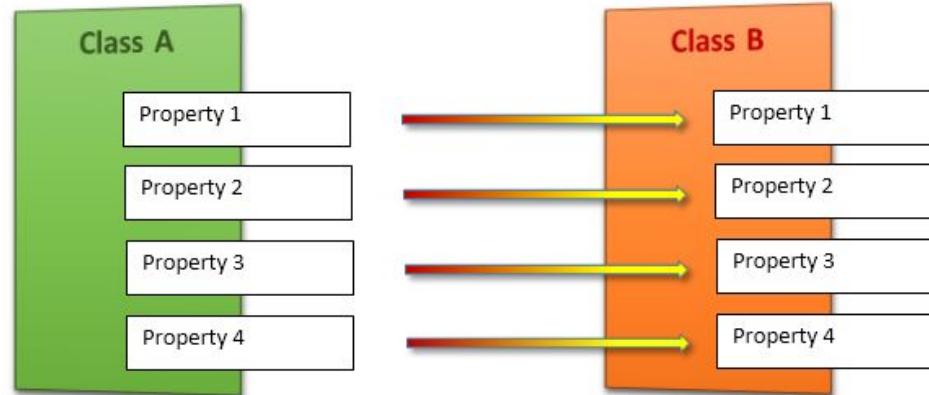




# AutoMapper : ou l'art de jouer avec un DTO

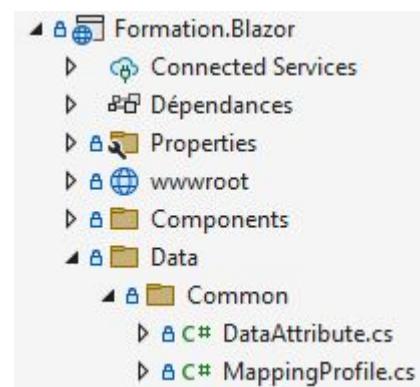
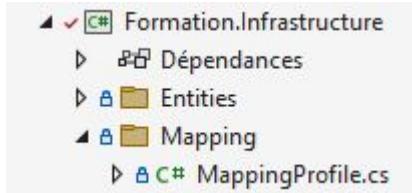
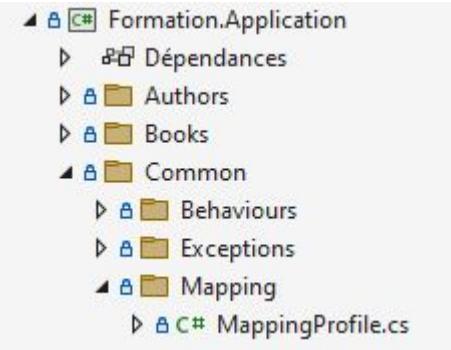


**En Bref** : permet de copier un objet d'une class vers un autre d'une autre class sans avoir besoin d'un converter.





# AutoMapper : ou l'art de jouer avec un DTO





# AutoMapper : ou l'art de jouer avec un DTO



```
public MappingProfile()
{
    CreateMap<CreateBookCommand, BookDTO>();
    CreateMap<CreateAuthorCommand, AuthorDTO>();
    CreateMap<UpdateAuthorCommand, AuthorDTO>();
}
```

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<BookDTO, Book>()
            .ReverseMap()
            .IncludeAllDerived()
            .ForMember(dest => dest.AuthorId, act => act.MapFrom(org => org.Author.Id))
            .ForMember(dest => dest.Author, act => act.MapFrom(org => new AuthorDTO
            {
                FirstName = org.Author.FirstName,
                LastName = org.Author.LastName
            }));
        CreateMap<AuthorDTO, Author>()
            .ReverseMap();
    }
}
```

```
CreateMap<AuthorDTO, Author>().ReverseMap();
CreateMap<CreateAuthorCommand, Author>();
CreateMap<UpdateAuthorCommand, AuthorDTO>().ReverseMap();
CreateMap<UpdateAuthorCommand, Author>();
CreateMap<PaginatedList<AuthorDTO>, PaginatedList<Author>>()
    .IncludeAllDerived()
    .ForPath(dest => dest.Items, act => act.MapFrom(org => org.Items))
    .ReverseMap();

CreateMap<BookDTO, Book>().ReverseMap();
CreateMap<PaginatedList<BookDTO>, PaginatedList<Book>>()
    .IncludeAllDerived()
    .ForPath(dest => dest.Items, act => act.MapFrom(org => org.Items));
```

# BDD TDD et Gherkin



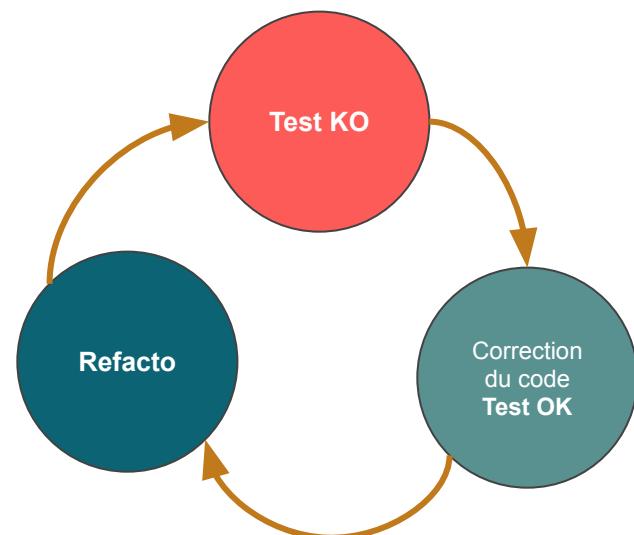
# TDD : Test Driven Development



# C'est quoi le TDD (Test Driven Development)

Ce sont les tests qui dirigent le développement d'une fonctionnalité.

- Le TDD est un cycle :
  - **Écrire un test**
    - Vérifier qu'il échoue (puisque n'y a pas de code correspondant)
  - **Écrire le code suffisant** pour que le test passe
    - Vérifier que le test passe
  - **Optimiser le code et vérifier qu'il n'y ait pas de régression.**





# les Avantages du TDD :

- Éviter des modifications de code **sans lien avec le but recherché**
- Éviter les accidents de parcours, où des tests échouent **sans qu'on puisse identifier le changement qui en est à l'origine**, ce qui aurait pour effet d'allonger la durée d'un cycle de développement.
- S'approprier plus facilement n'importe quelle partie du code en vue de le faire évoluer, car chaque test ajouté dans la construction du logiciel **explique et documente le comportement du logiciel**.
- Livrer une nouvelle version d'un logiciel avec un haut niveau de confiance dans la qualité des livrables, confiance justifiée par la couverture et la pertinence des tests à sa construction



# Exemple pratique FizzBuzz

Écrire un programme qui retourne les entiers de 1 à 108.

A prendre en compte :

- Pour les multiples de 3, remplacez le nombre par "Fizz".
- Pour les multiples de 5, remplacez le nombre par "Buzz".
- Un nombre multiple de 15 remplacez le nombre par "FizzBuzz".
- Sinon renvoyer le nombre tel quel

Exemple

"12Fizz4BuzzFizz78FizzBuzz. ...



A vos claviers ! 😊

**Objectif :** Écrire une fonction Add qui accepte une chaîne de caractères en entrée et en retourne un entier.

```
public int Add(string input)
```



# Calculatrice simple

La fonction doit répondre aux spécifications suivantes :

- La méthode doit être capable de traiter jusqu'à deux nombres dans la chaîne, séparés par des virgules, et renvoyer leur somme. Une chaîne vide doit retourner 0.
- Elle doit pouvoir traiter un nombre indéfini d'arguments.
- Elle doit accepter les sauts de ligne comme séparateurs en plus des virgules.
- La fonction ne doit pas autoriser un séparateur à la fin de la chaîne.
- Elle doit pouvoir gérer différents délimiteurs. Pour utiliser un délimiteur différent, la chaîne d'entrée doit commencer par une ligne spéciale : //\*[delimiteur]\*\n\*[numbers]\*.
- Si la méthode est appelée avec des nombres négatifs, elle doit renvoyer un message d'erreur : “Negative number(s) not allowed: <nombres négatifs>”.
- En cas de multiples erreurs, la fonction doit renvoyer tous les messages d'erreur séparés par des sauts de ligne.
- Les nombres supérieurs à 1000 doivent être ignorés lors de l'addition.

**TDD / BDD : une  
approche par les tests ...  
et l'usage**



# TDD / BDD : une approche par les tests ... et l'usage

Le **BDD** consiste à étendre le TDD en écrivant non plus du code compréhensible uniquement par des développeurs, mais sous forme de **scénario compréhensible par toutes les personnes impliquées dans le projet**.

*Impliquer le métier dans l'écriture des tests d'acceptances.*



# TDD / BDD : une approche par les tests ... et l'usage

Un test en BDD consiste à décrire une fonctionnalité selon un formalisme « **Given / When / Then** » :

- **Given** :  
*La description, il permet généralement de décrire l'état initial*
- **When** :  
*L'action, il s'agit des actions effectuées (appel d'une méthode,...)*
- **Then** :  
*La vérification, elle est présente pour vérifier que le résultat attendu (par exemple que le message affiché contient bien tel message*

# Gherkin / Cucumber / Specflow





# C'est quoi ?

*Scenario : affichage du prix du produit*

*Etant donné que je navigue sur le site ecommerce*

*Quand j'affiche une fiche produit*

*Et que ce produit a une promotion de « 0 » %*

*Alors le prix affiché du produit ne change pas*

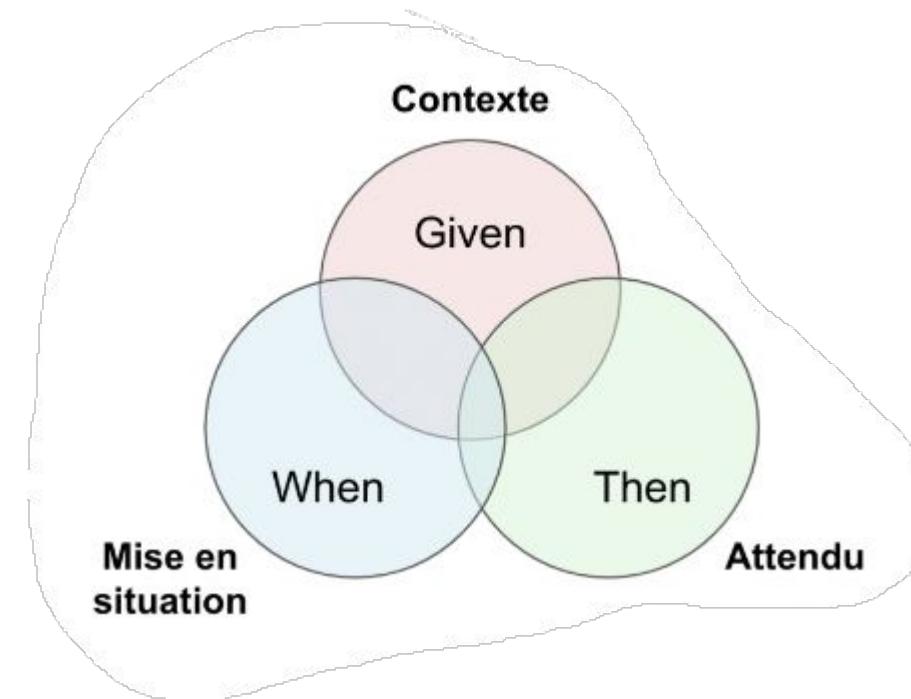
*Etant donné que je navigue sur le site ecommerce*

*Quand j'affiche une fiche produit*

*Et que ce produit a une promotion de « 20 » %*

*Alors le prix affiché du produit est de « -20% »*

*Et une mention « promotion -20% » rappelle la promotion*



# Pourquoi Gherkin?

Sans :

We would like to encourage new users to buy in our shop.  
Therefore we offer 10% discount for their first order.

```
public void CalculateDiscount(Order order)
{
    if (order.Customer.IsNew)
        order.FinalAmount =
            Math.Round(order.Total * 9/10);
}
```



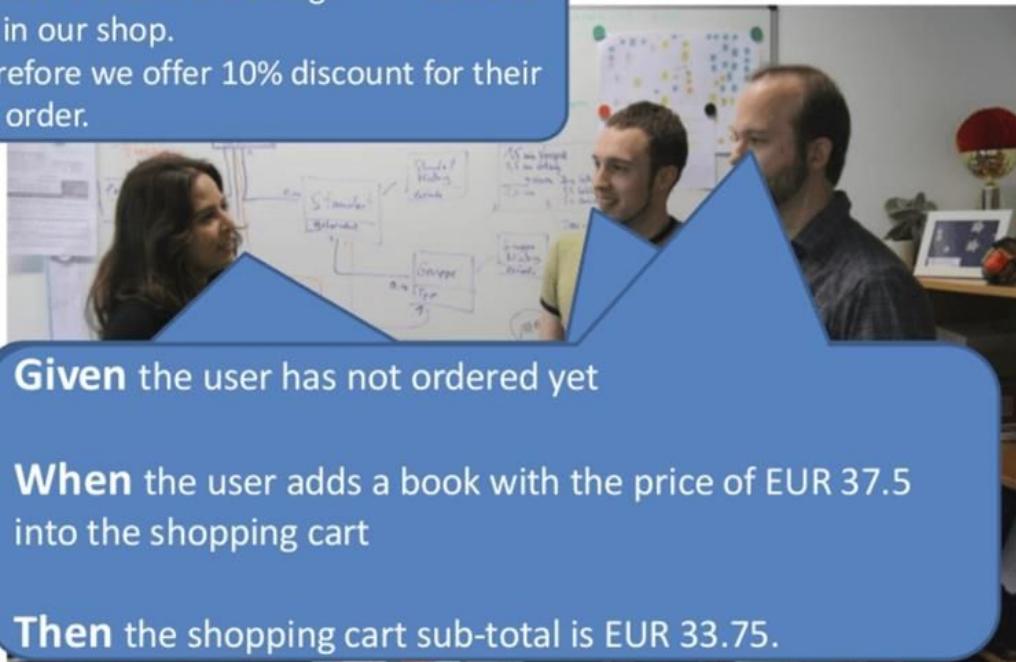
Register as “bart\_bookworm”  
Go to “/catalog/search”  
Enter “ISBN-0955683610”  
Click “Search”  
Click “Add to Cart”  
Click “View Cart”  
Verify “Subtotal” is “\$33.75”



# Pourquoi Gherkin?

Avec :

We would like to encourage new users to buy in our shop.  
Therefore we offer 10% discount for their first order.





# L'importance de la spécification

- Des cas d'usages compris par toutes le partie prenante du projet
- Un langage commun
- Plus d'efficacité
- Moins de frustration



# Les mots clés

<https://cucumber.io/docs/gherkin/reference/>

- Feature
- Example or Scenario
- Given, When, Then, And, But for steps (or \*)
- Background
- Scenario Outline (or Scenario Template)
- Examples (or Scenarios)



# SpecFlow



**Specflow** est un framework de test prenant en charge les pratiques BDD dans le framework .NET. C'est un framework open source hébergé sur GitHub.

Avec Specflow on pourra définir des scénarios simples défini par le langage Gherkin qui est clairement compréhensible par n'importe qui.



# Installation



Pour utiliser SpecFlow dans nos projets avec Visual Studio, on doit rajouter l'extension Specflow :

Grâce à cette extension on va pouvoir :

- Créer des projets de tests de type SpecFlow dans notre solution.
- Editer des fichiers de type « feature » contenant les tests mis en place avec le langage Gherkin
- Utiliser le framework « SpecFlow+ Runner » pour exécuter nos tests.

The screenshot shows the 'Manage Extensions' window in Visual Studio. The search bar at the top right contains the text 'specflow'. The results list shows three items:

- SpecFlow for Visual Studio 2019**: SpecFlow integration for Visual Studio 2019. It has a green checkmark icon, indicating it is installed. Details: Created By: SpecFlow Team, Version: 2019.0.81.23124, Downloads: 162535, Pricing Category: Free, Rating: ★★★★☆ (26 Votes). Buttons: More Information, Report Extension to Microsoft.
- Deveroom for SpecFlow (Visual Studio 2019)**: Visual Studio extension for working with SpecFlow projects and Gherkin feature files. It has a green circle icon. Details: spex, Spec [Preview], SpecFlow (Features) sync to Azure DevOps (TestCases).
- Roaming Extension Manager**: A link to the Roaming Extension Manager.



# SpecFlow



**Specflow** est un framework de test prenant en charge les pratiques BDD dans le framework .NET. C'est un framework open source hébergé sur GitHub.

Avec Specflow on pourra définir des scénarios simples défini par le langage Gherkin qui est clairement compréhensible par n'importe qui.



# SpecFlow



**Specflow** est un framework de test prenant en charge les pratiques BDD dans le framework .NET. C'est un framework open source hébergé sur GitHub.

Avec Specflow on pourra définir des scénarios simples défini par le langage Gherkin qui est clairement compréhensible par n'importe qui.

# Composants





# Définition

Les composants sont les unités de déploiement. Ce sont les plus petites entités qui peuvent être déployées dans le cadre d'un système. En Java, ce sont des fichiers jar. En Ruby, ce sont des fichiers gem. **En .Net, ce sont des DLL.**

Ils peuvent être déployés indépendamment en tant que plug-ins séparés chargés dynamiquement, tels que des fichiers .jar ou .dll ou .exe. Indépendamment de la manière dont ils sont finalement déployés, **les composants bien conçus conservent toujours la capacité d'être indépendamment déployables et, par conséquent, indépendamment développables.**

# Couplage de composants



KOHNKE



# Principe de dépendance Acyclique

Avez vous déjà développé toute une journée, obtenu **une solution fonctionnelle** et en revenant le lendemain constaté que **ça ne fonctionne plus** ?

**Le syndrome du lendemain !! (The morning after syndrome)**

*Quelqu'un est resté plus tard que vous et a changé quelque chose dont votre développement dépend.*





# Solution 1 : Le merge hebdomadaire

Chaque développeur travaille sur sa branche et le merge est fait **une fois par semaine**.

- + le Développeur travaille indépendamment pendant 4 jours
- Le prix à payer au moment de l'intégration





## Solution 2 : éliminer les dépendance cyclique

### ADP (Acyclic Dependencies Principle)

**Partitioner** l'environnement de développement en **composants** livrable. Les composants deviennent des unit of work qui peuvent être la **responsabilité d'un développeur ou d'une seule team**.

Lorsqu'un composant est fonctionnel, il est mis à disposition des autres équipes.

On peut alors continuer à développer son composant **sans impacter les autres équipes**.

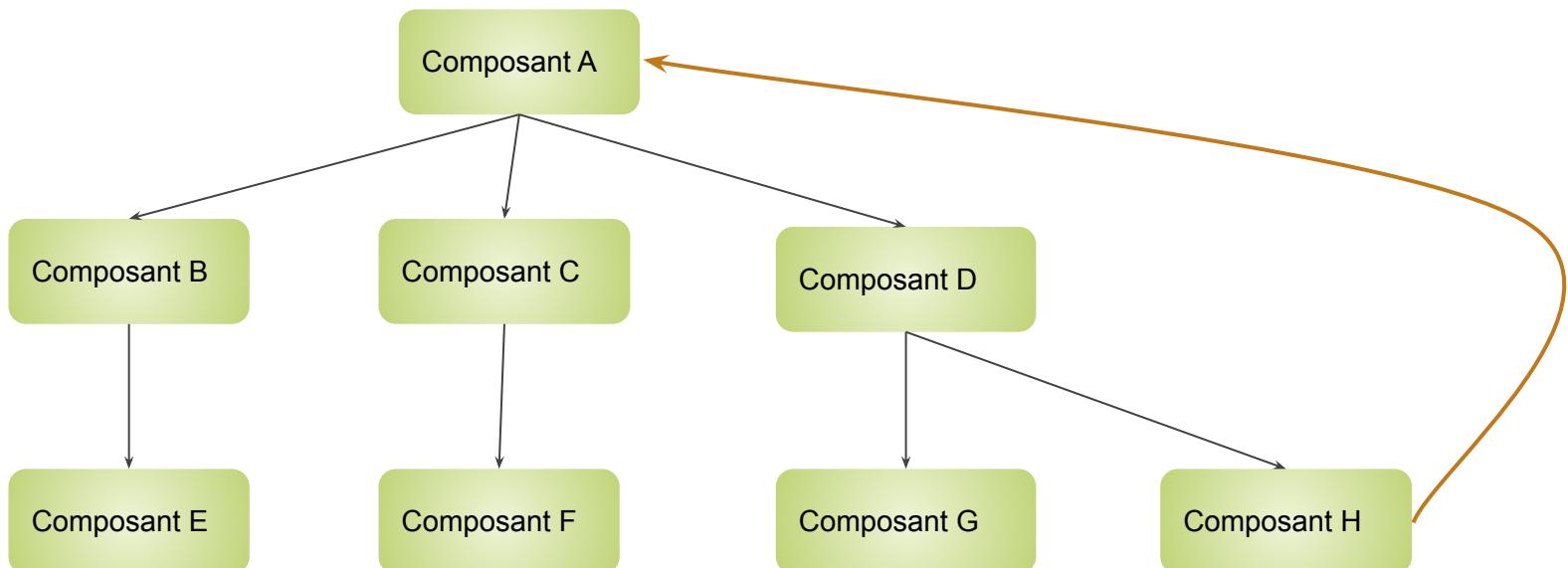
*Lors d'une livraison, les autres équipes ont le choix d'utiliser la nouvelle version ou de continuer à utiliser les anciennes.*

**Aucune équipe n'est à la merci d'une autre.**



# Solution 2 : éliminer les dépendance cyclique

## ADP (Acyclic Dependencies Principle)

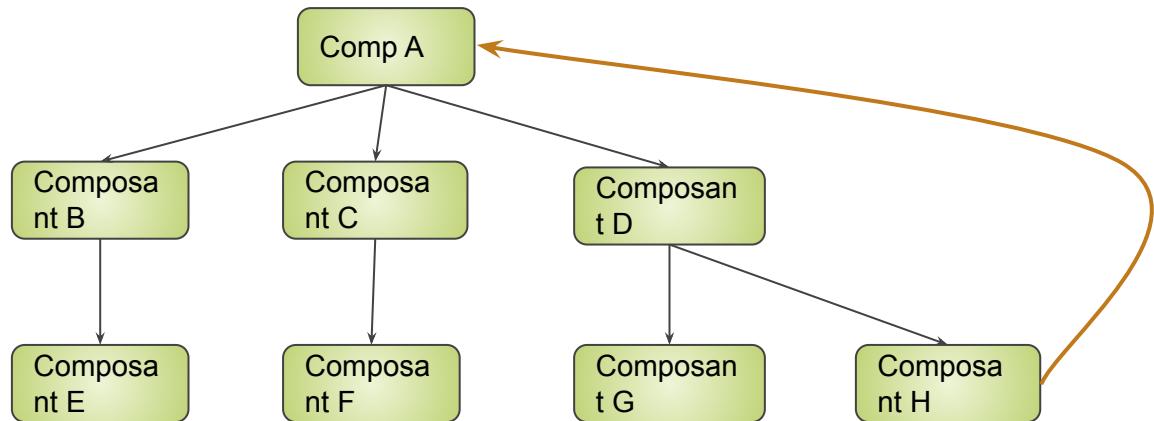




## Solution 2 : éliminer les dépendance cyclique ADP (Acyclic Dependencies Principle)

Le composant D qui dépend de A à une dépendance commune avec A => H

! Il y a violation de l'ADP





## Solution 2 : éliminer les dépendance cyclique ADP (*Acyclic Dependencies Principle*)

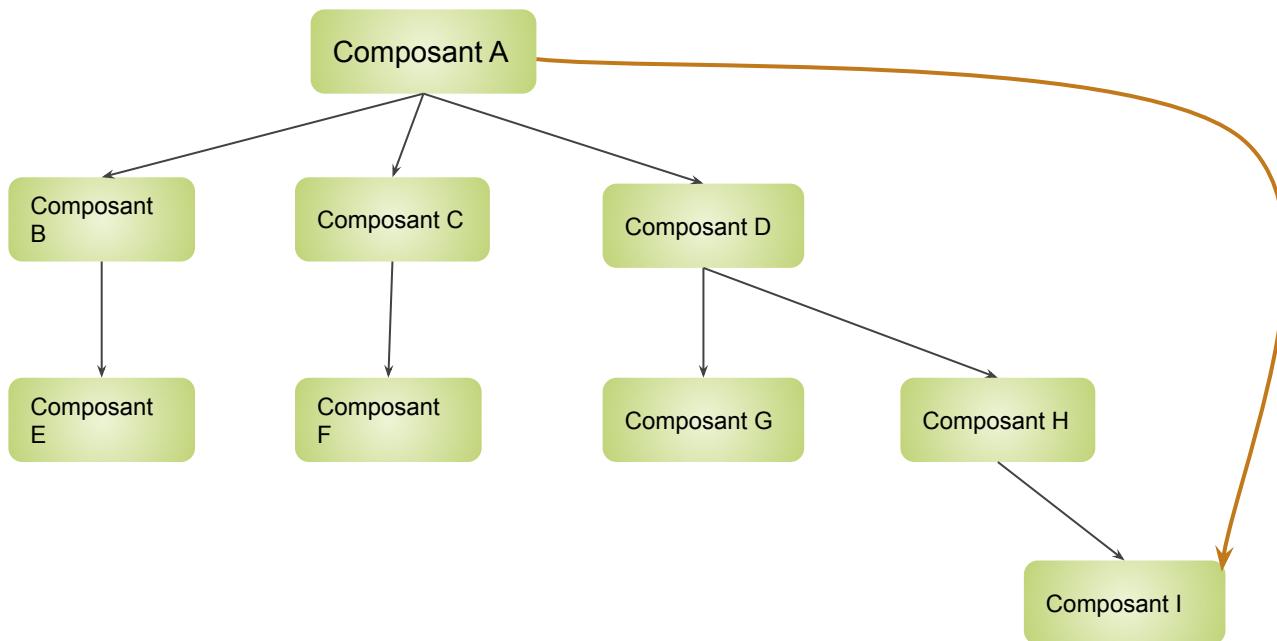
### Résolution de l'ADP :

1. Appliquer le principe d'inversion de dépendance  
(le D de **S.O.L.I.D** DIP)
  - a. Créer une **classe abstraite** ou une **Interface** dans le composant H
  - b. Implémenter cette interface ou class abstraite das A
2. Créer un nouveau composant **I** dont vont dépendre les 2 classes



# Solution 2 : éliminer les dépendance cyclique

## ADP (Acyclic Dependencies Principle)



# Frontières

Tracer les limites





# Tracer les limites

Les limites séparent les éléments logiciels les uns des autres et **empêchent ceux d'un côté de connaître ceux de l'autre.**

Certaines frontières doivent être tracées très tôt dans la vie d'un projet.

D'autres doivent être repoussées au maximum pour éviter qu'elles ne polluent les logiques métier de base.



# Tracer les limites

*Le but d'un architecte logicielle est de minimiser les ressources humaines nécessaires pour construire et maintenir le système requis.*

Ce qui peut saper ce noble but :

- Couplage
- et surtout couplage à des décisions prématurées



# Tracer les limites

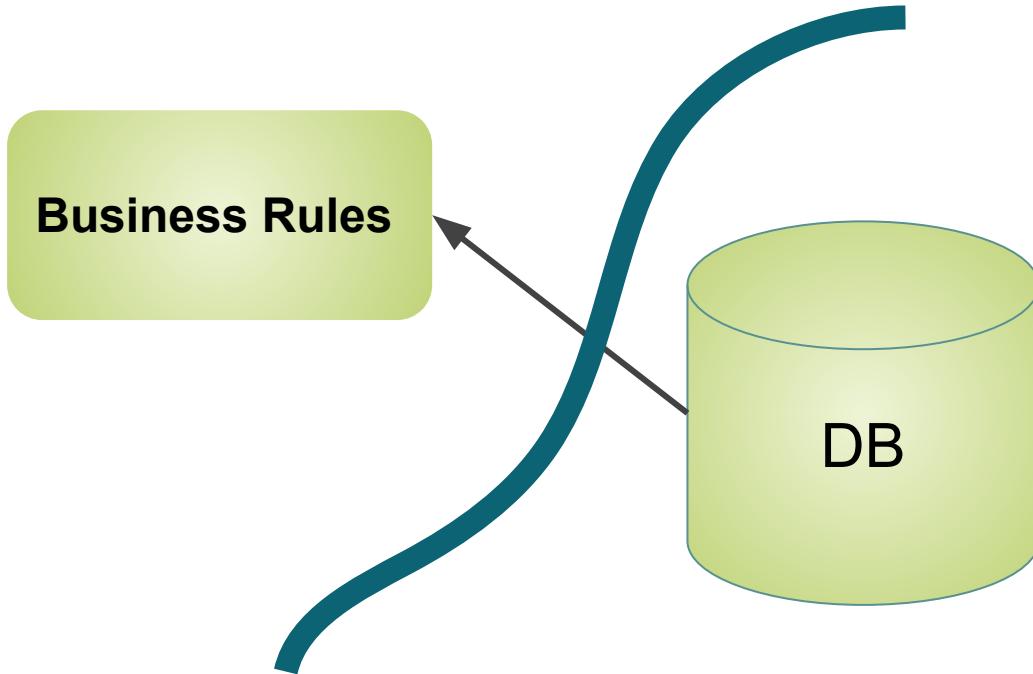
Une **bonne architecture de système** est une architecture dans laquelle des décisions comme celles-ci sont rendues accessoires et reportables.

Une **bonne architecture de système** ne dépend pas de ces décisions.

Une **bonne architecture de système** permet de prendre ces décisions au plus tard sans impact significatif.

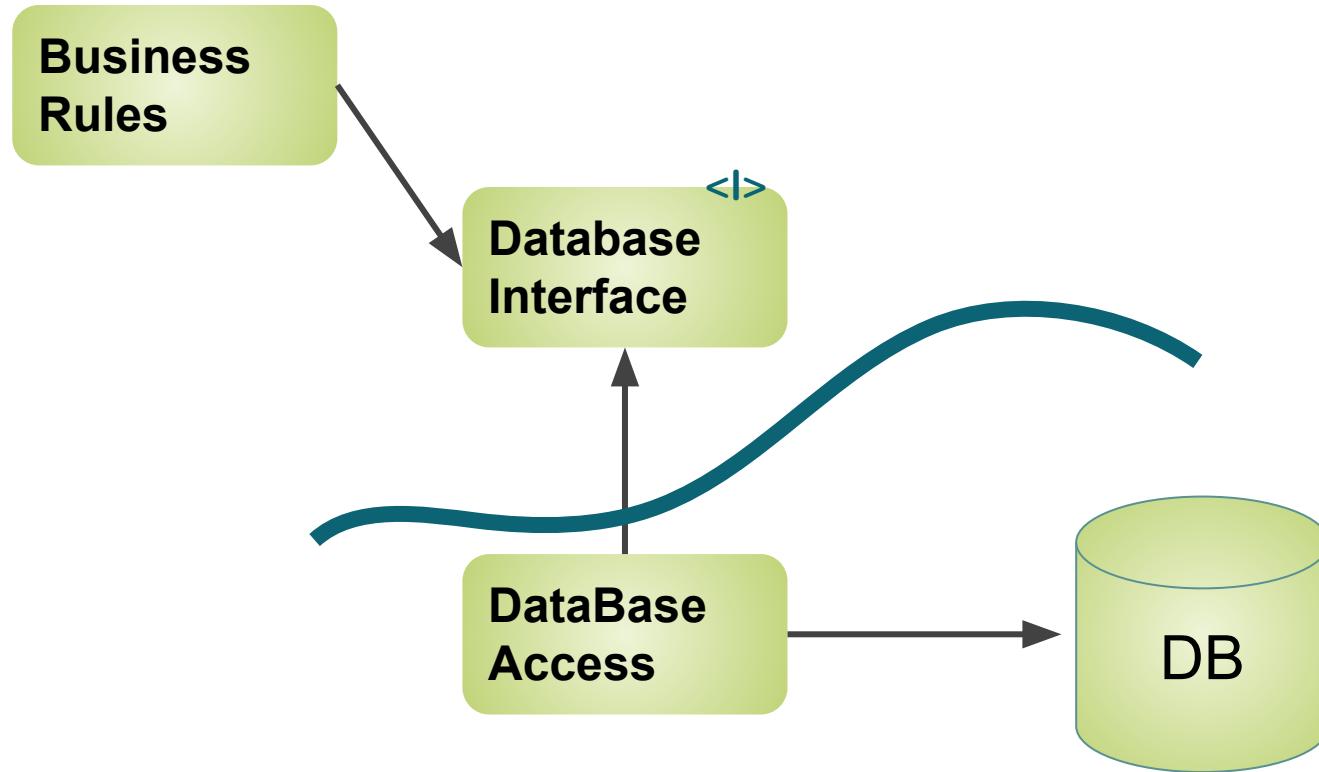


# Tracer les limites





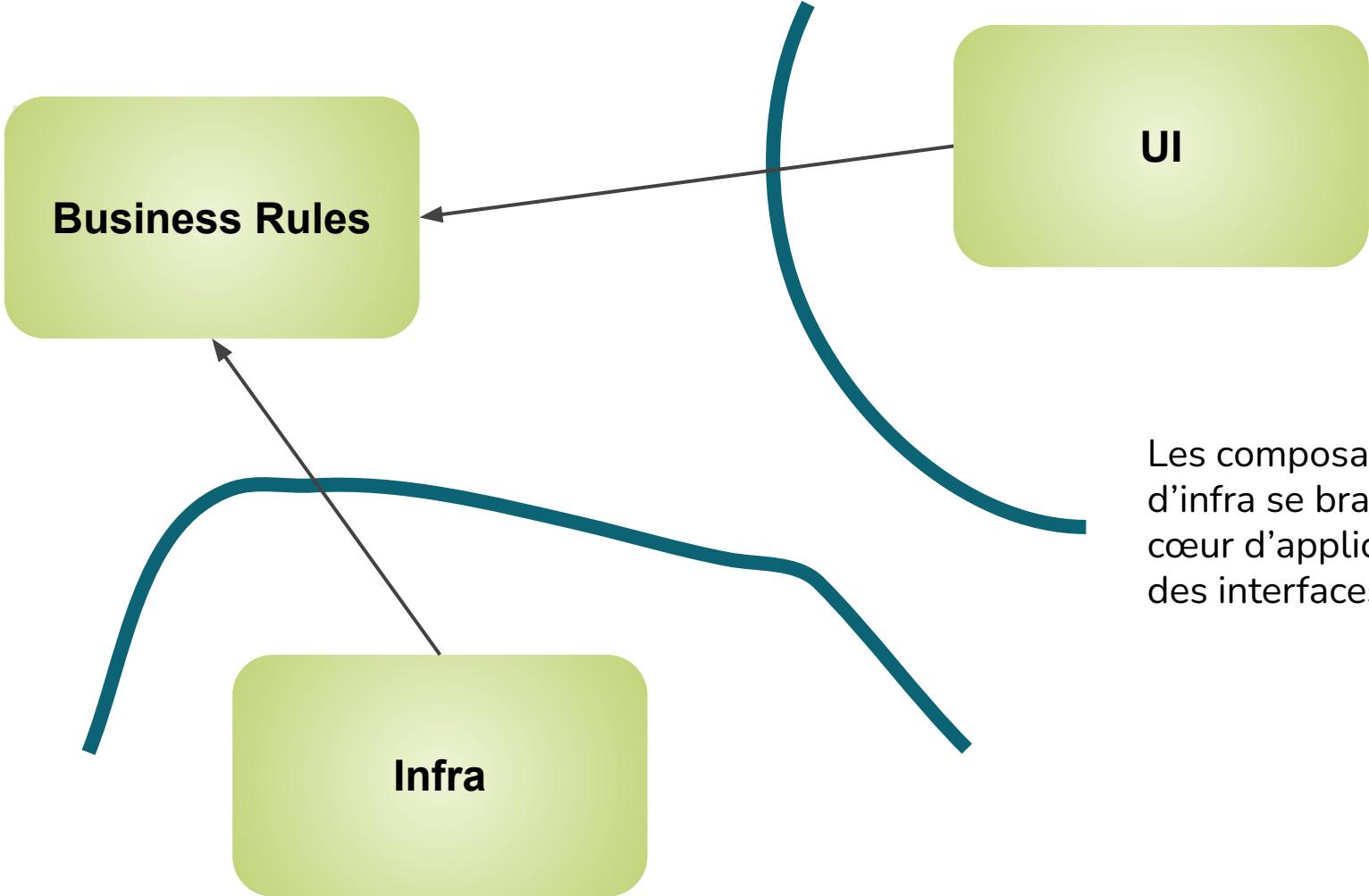
# Tracer les limites





# Plugin Architecture

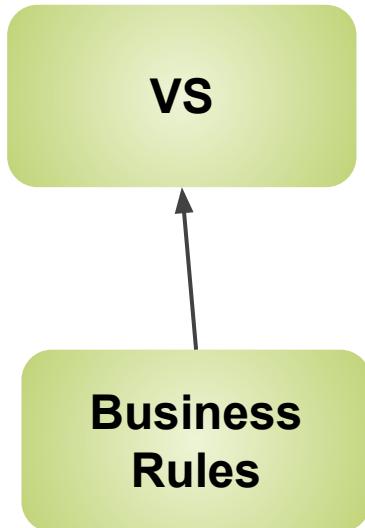




Les composants D'UI et d'infra se branchent au cœur d'application via des interfaces.



# Exemple : Resharper et Visual Studio



ReSharper dépend du code source de Visual Studio  
ReSharper ne peut rien faire pour perturber l'équipe Visual Studio  
L'équipe Visual Studio pourrait complètement désactiver l'équipe ReSharper si elle le souhaite  
**Relation profondément asymétrique**



# Tracer les frontières : Conclusion

Pour tracer les frontières dans une architecture applicative :

- **Partitionner** le système en composant
- Définir parmis ces composant :
  - Lesquels sont le **coeur du métier**
  - lesquels sont des **plugin** (pas en rapport direct avec la logique métier)
- Arranger le code dans ces composants pour que **les dépendances entre les composants aient une seule direction vers le Coeur d'application.**

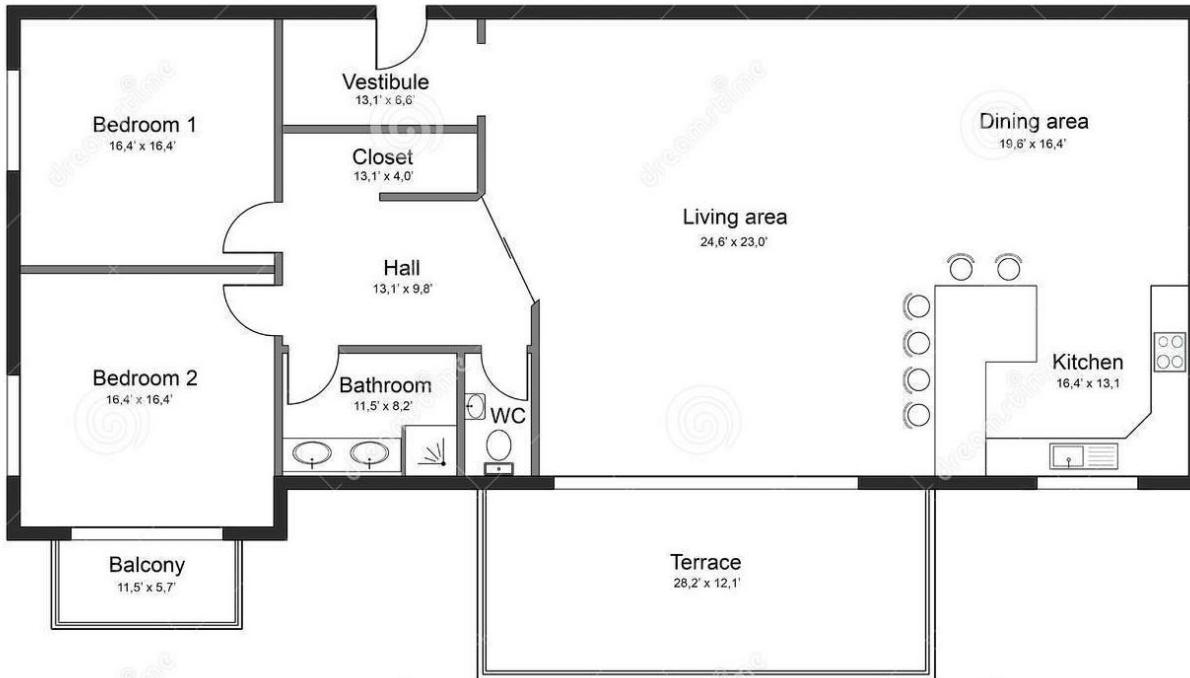
Votre application respectera alors le principe d'inversion des dépendances DIP et les flèches de dépendance sont agencées pour pointer du low-level vers des abstractions du high-level.

# Screaming Architecture

Une architecture qui parle !

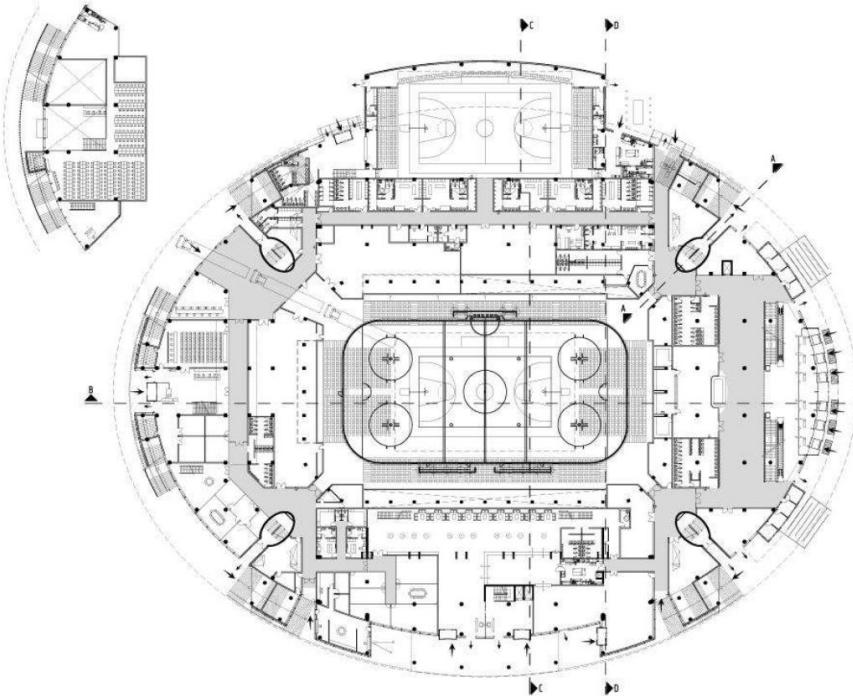


# Architecture d'une maison





# Architecture d'un stade





# Architecture d'application

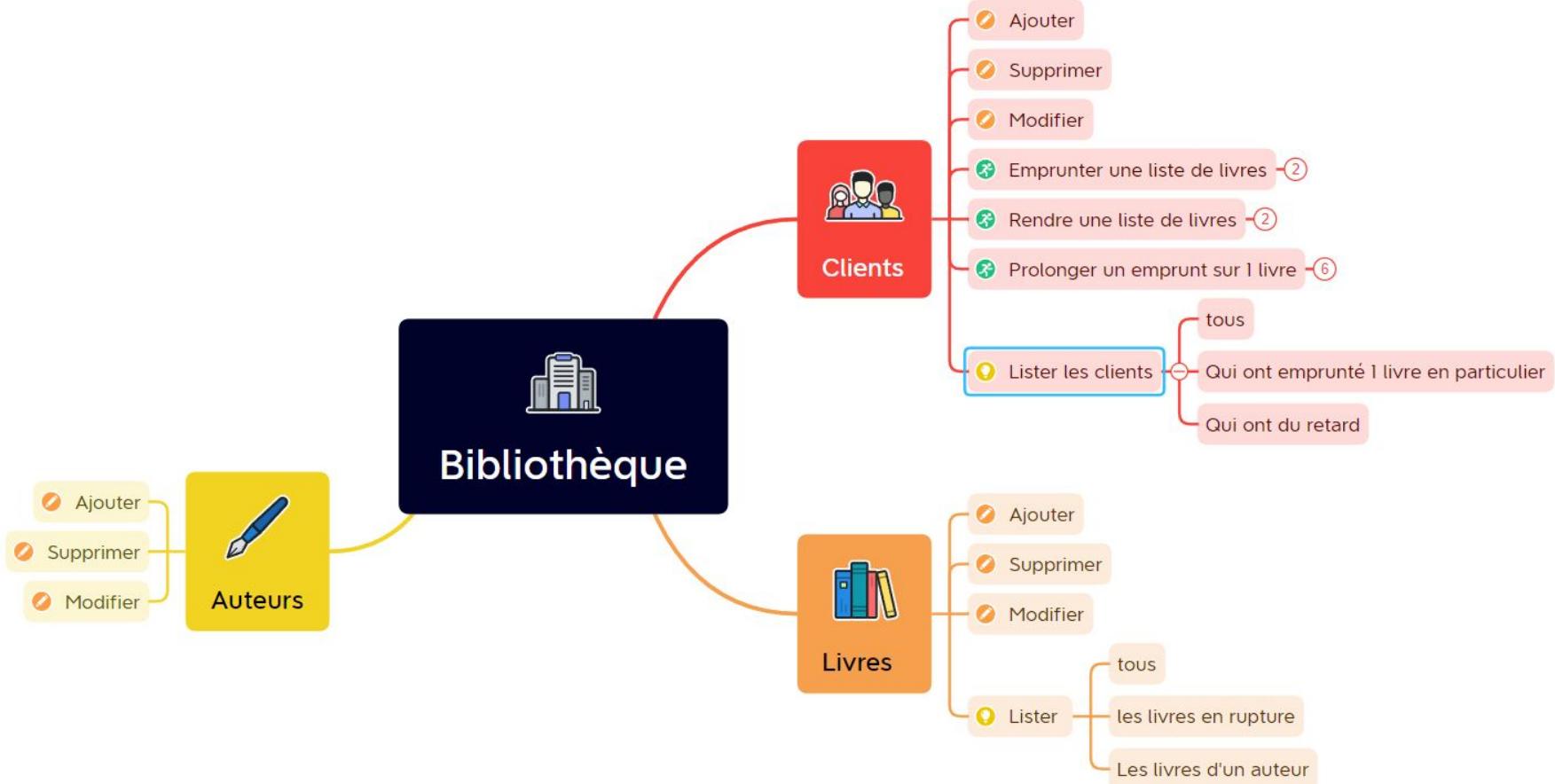
-  Content
-  Controllers
-  Models
-  Scripts
-  Views



-  Customers
-  Employees
-  Products
-  Sales
-  Vendors

# TP

Crud bibliothèque





# TP : Gestion bibliothèque

Les entités:

Un livre:

- un titre
- une description
- un **auteur**
- une date de publication
- un nombre de pages

Un Auteur :

- Nom
- prénom
- Date de naissance
- **livres**



# TP : Gestion bibliothèque

## Les Use Cases:

- Ajouter / modifier / supprimer un **auteur**
  - Voir un **auteur**
  - Voir tous les **auteurs** dans une liste paginé et filtrable selon le nom prénom et date de naissance
- 
- Ajouter / modifier / supprimer un **livre**
  - Voir un livre
  - Voir tous les **Livres** dans une liste paginé et filtrable selon le titre la date de publication le nom prénom et date de naissance de l'auteur



# TP Bonus : Gestion des emprunt

**Emprunteur :**

- Nom, prénom, date de naissance, Id
- **emprunts**

**Emprunt:**

- Id
- **emprunteur**
- Date d'emprunt
- livre
- rendu (booléen)



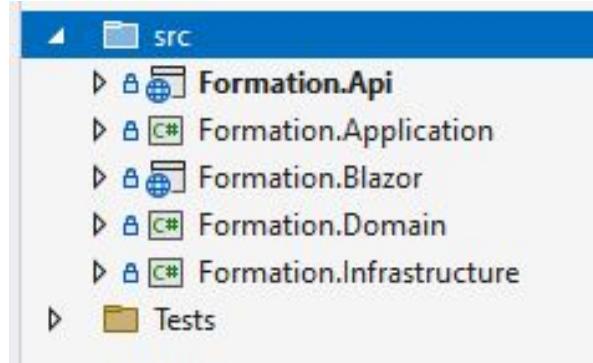
# TP Bonus : Gestion des emprunt

Use Cases:

- Voir si un livre est emprunté (On suppose ici qu'il n'y a qu'un seul exemplaire pour chaque livre)
- ajouter un emprunt
- terminer un emprunt
- Lister les emprunts en cours pour un emprunteur



# étape 1: création de la solution et de sa structure





## Etape 2 : Création du projet de test de validation

- Utilisation de **SpecFlow** pour spécification **Gherkin**
- Création de la solution de test avec Specflow
- Choix **Nunit** ou **Xunit**
- Rédaction d'un premier test de d'acceptance



## Etape 2.1 : Rédaction des test de de validation

- **Rédaction d'un premier test de d'acceptance**
- Début de l'implémentation des Domain et Application pour faire passer notre test
- On reboucle avec un second test



## **Etape 2: Domain et Application**

- Début de l'implémentation des Domain et Application pour faire passer notre test
- On reboucle avec un second test