ARCHITECTURE (1h30min)

AU MENU CE JOUR

Importance de l’architecture

Qui est concerné ?

Rôle de l’architecte

Qualité & Principes de développement

SOLID

DRY, YAGNI…

Architecture applicative

Hexagonale, Clean Architecture, DDD

Complexité & Micro-services

Importance de l’architecture

QUI EST CONCERNÉ ?

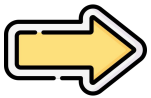
Chaque membre de l'équipe à son compte à y trouver !

Rendre la **DX agréable**

Faciliter la maintenance applicative

Minimiser le coût de développement

Livraison + fréquente car + de confiance



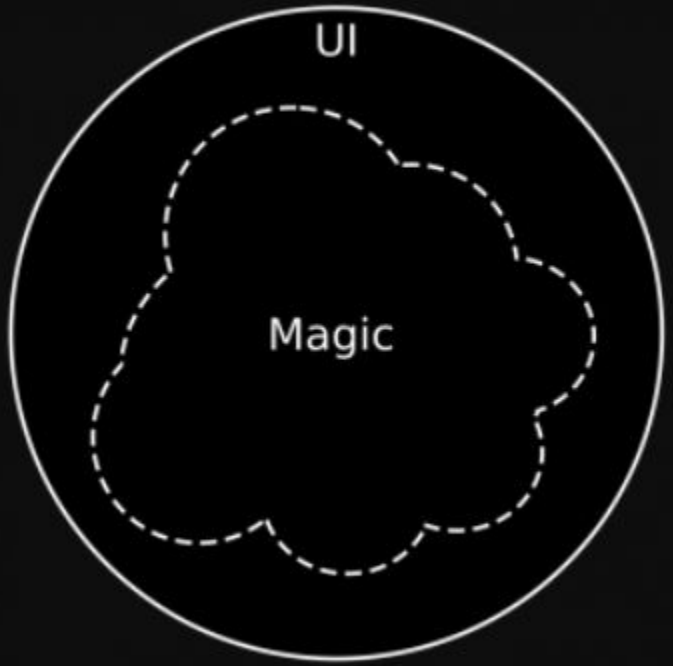
Amélioration de la satisfaction d'équipe et sa productivité

Amélioration de la satisfaction client

Spoiler : on peut avoir un produit qui fonctionne bien et avoir une architecture bancale

Spoiler^2 : on va le payer à l'évolution du produit

What other see



What you see

RÔLE DE L'ARCHITECTE

GARDIEN DE LA QUALITÉ À DIFFÉRENT NIVEAU

Concevoir l'architecture

Définir les fondations techniques

Anticiper et guider

Prévoir les évolutions et garder l'adaptabilité du produit

Aligner technique et métier

Faire le pont entre besoins business et solutions techniques

Garantir la qualité

Performance, sécurité, maintenabilité et évolutivité

Architecte applicatif ou technique, architecte d'entreprise, architecte solution...

VOUS ÊTES DES ARCHITECTES !

Principes de développement

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Single Responsibility

```
class Employee {
  calculatePay()
  reportHours()
}
```

Fais une seule chose, mais fais-la bien !

Une classe qui fait X choses différentes est une bombe à retardement.

Ici, nous cassons le SRP car ***Employee*** est responsable de plusieurs actions à destinations de plusieurs acteurs (DAF, manager...)

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Open Closed ❌

```
class Player {
  printRole() {
    switch(this.role) {
        case "elf":
            console.log("Player is elf !");
            break;
        case "warrior":
            console.log("Player is warrior !");
            break;
        case "..."
    }
  }
}
```

Ajoute des options, ne change pas le moteur...

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Open Closed ✅

```
class Player {
    constructor(private readonly role: PlayerRole) {}
    printRole() {
        console.log("Player is ", this.role.getRole());
    }
}

interface PlayerRole {
    getRole(): string
}

class RoleElf implements PlayerRole {
    getRole() {
        return 'elf';
    }
}

class RoleWarrior implements PlayerRole {
    getRole() {
        return 'warrior';
    }
}

const player = new Player(new RoleWarrior());
player.printRole();
```

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Liskov Substitution ❌

```
class Player {
    items: string[];
    flying: boolean;

    fly(): {
        this.flying = true;
    }
}

class Warrior extends Player {
    fly(): void {
        throw new Error("Warriors can't fly, only Elves can !!!!");
    }
}
```

Une classe fille doit pouvoir remplacer un parent sans casser

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Liskov Substitution ✅

```
class Player {
    items: string[];
}

class Elf extends Player {
    items: string[];
    flying: boolean;

    fly(): void {
        this.flying = true;
    }
}
```

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Interface Segregation ❌

```
interface CharacterActions {
    throwMagicPower(): number;
    speak(): string;
}
class PlayableCharacter implements CharacterActions {
    throwMagicPower(): number {
        return 100;
    }
    speak(): string {
        return "I am a playable character";
    }
}
class NonPlayableCharacter implements CharacterActions {
    throwMagicPower(): number {
        // not applicable to NPC...
        return 0;
    }
    speak(): string {
        return "I am a non playable character";
    }
}
```

Personne ne doit dépendre de méthodes qu'il n'utilise pas

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Interface Segregation ✅

```
interface Speaker {
    speak(): string;
}

interface Magician {
    throwMagicPower(): Damage;
}

class PlayableCharacter implements Magician, Speaker {
    throwMagicPower(): Damage {
        return 100;
    }
    speak(): string {
        return "I am a playable character";
    }
}
class NonPlayableCharacter implements Speaker {
    speak(): string {
        return "I am a non playable character";
    }
}
```

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Dependency Inversion ❌

```
class NotifyUserUseCase {
    constructor(private readonly emailSender = new EmailSender()) {}
    execute({order, buyer}) {
        const text = `An order of ${order.price}${order.currency} have be
        this.emailNotificationSender.send({
            text,
            to: buyer
        })
    }
}
```

Dépendre d'abstraction, pas de classes concrètes

PRINCIPES DE DÉVELOPPEMENT

S.O.L.I.D - Dependency Inversion ✅

```
interface NotificationProvider {
    send(): void;
}

class EmailNotificationProvider implements NotificationProvider {
    send(): {
        // stuff to send email
    }
}

class SMSNotificationProvider implements NotificationProvider {
    send(): {
        // stuff to send SMS
    }
}

class FakeNotificationProvider implements NotificationP
    send(): {}
}
```

```
class NotifyUserUseCase {
    constructor(private readonly notificationProvider: NotificationProvid
    execute({order, buyer}) {
        const text = `An order of ${order.price}${order.currency} have be
        // business logic don't care about notification being a SMS or Ma
        this.notificationProvider.send({
            text,
            to: buyer
        })
    }
}
```

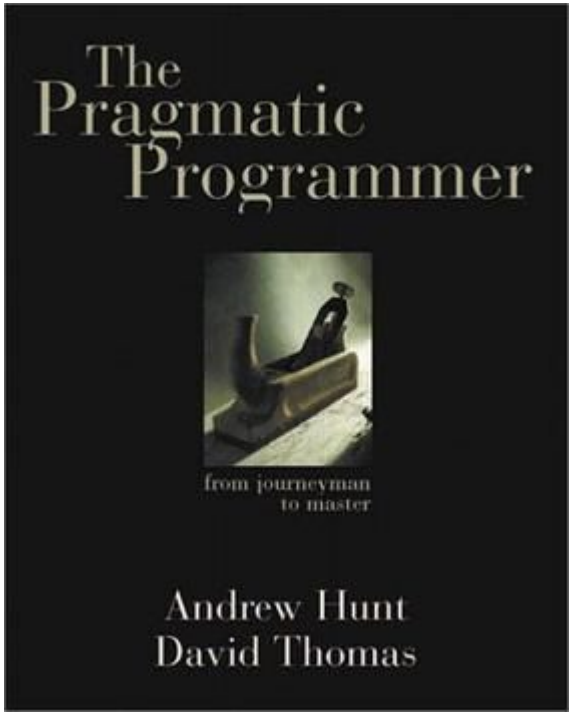
PRINCIPES DE DÉVELOPPEMENT

D.R.Y - Don’t Repeat Yourself

“Dans un système, toute connaissance doit avoir une représentation unique, non ambiguë, faisant autorité”

La **modification** d'un **élément** d'un système **ne change pas** les autres **éléments non liés** logiquement. De plus, tous les **éléments liés** logiquement **changent uniformément**, de manière **prévisible.**

⚠️ Ne pas se répéter, sauf quand il le faut.



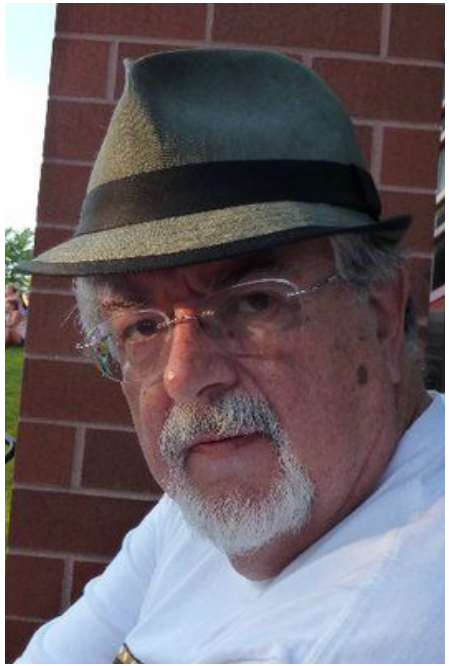
PRINCIPES DE DÉVELOPPEMENT

Y.A.G.N.I - You Ain't Gonna Need It

“Mettez toujours en œuvre les choses quand vous en avez effectivement besoin, pas lorsque vous prévoyez simplement que vous en aurez besoin”

Avez vous réellement besoin d’implémenter un CRUD complet ?

N’ajouter **pas de fonctionnalités “pour prévoir”**.
Réflexion **valable** aussi pour vos **backlogs** produits ;)



CE NE SONT PAS DES RÈGLES UNIVERSELLES, PLUTÔT DES GUIDES

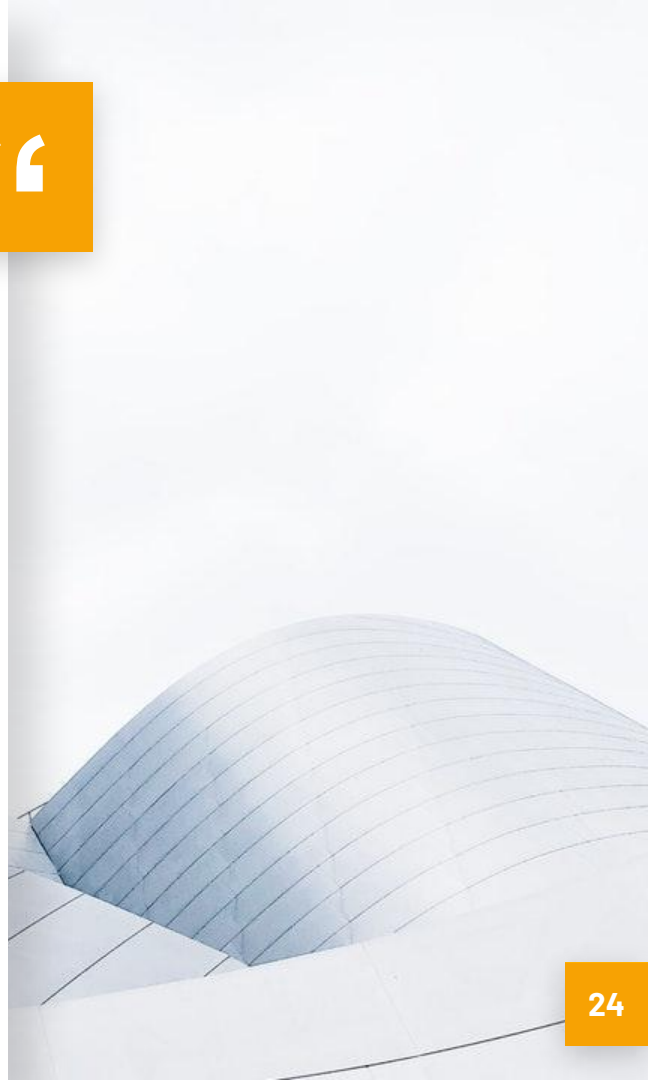
Architecture Applicative

Me on Google: “Hexagonal architecture for dummies”

Everybody: “Left/Right, DDD, Clean archi, Ports/Adapters, SOLID .....”

Also me:

Permettre à une **application** d'être **pilotée aussi bien par des utilisateurs** que [...] par des **tests automatisés**, et d'être **développée** et **testée** en **isolation** de ses **éventuels systèmes d'exécution** et **BDD**.



Séparation des responsabilités

User Side

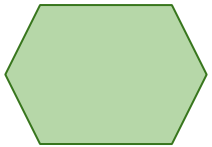
Ce que l'on présente

On parle aussi de Left-side, ou d'application

On y trouve l'API, le code IHM...
Ici, on pilote le Business !

Business

Le métier et ses règles



On parle aussi de Center ou encore d'Hexagone, de "domaine"...

On cherche à isoler cette partie du reste... C'est le coeur de l'application, pas besoin de savoir coder pour la comprendre.

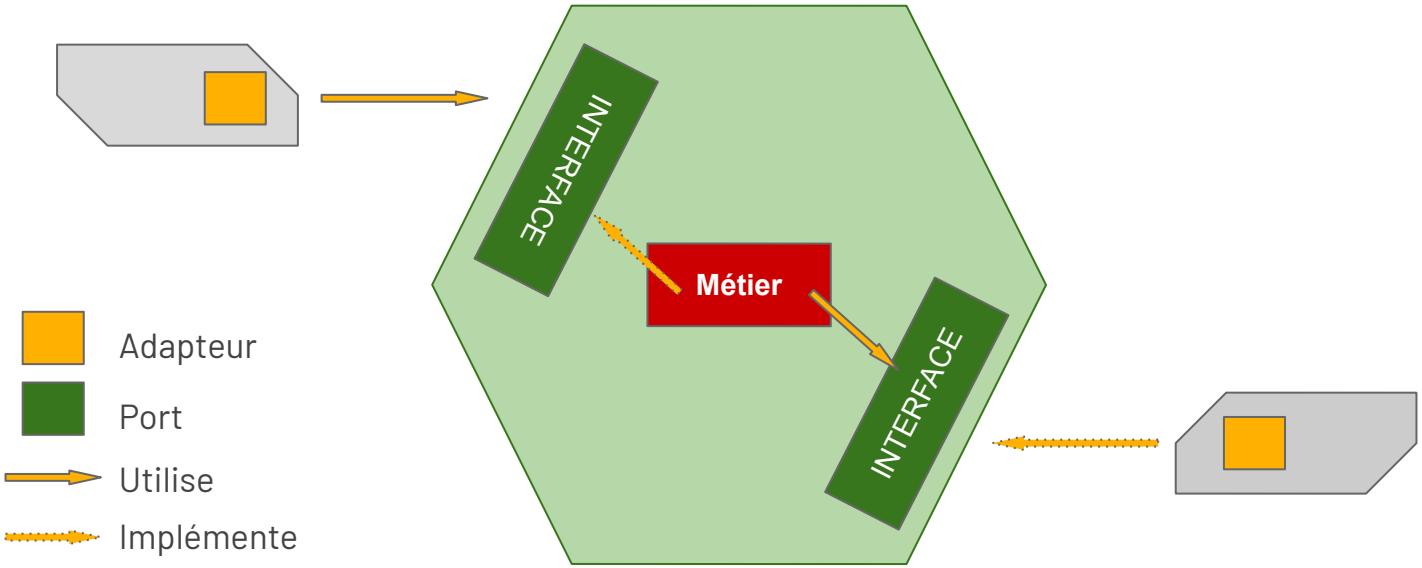
Server Side

Ce auquel on dépend

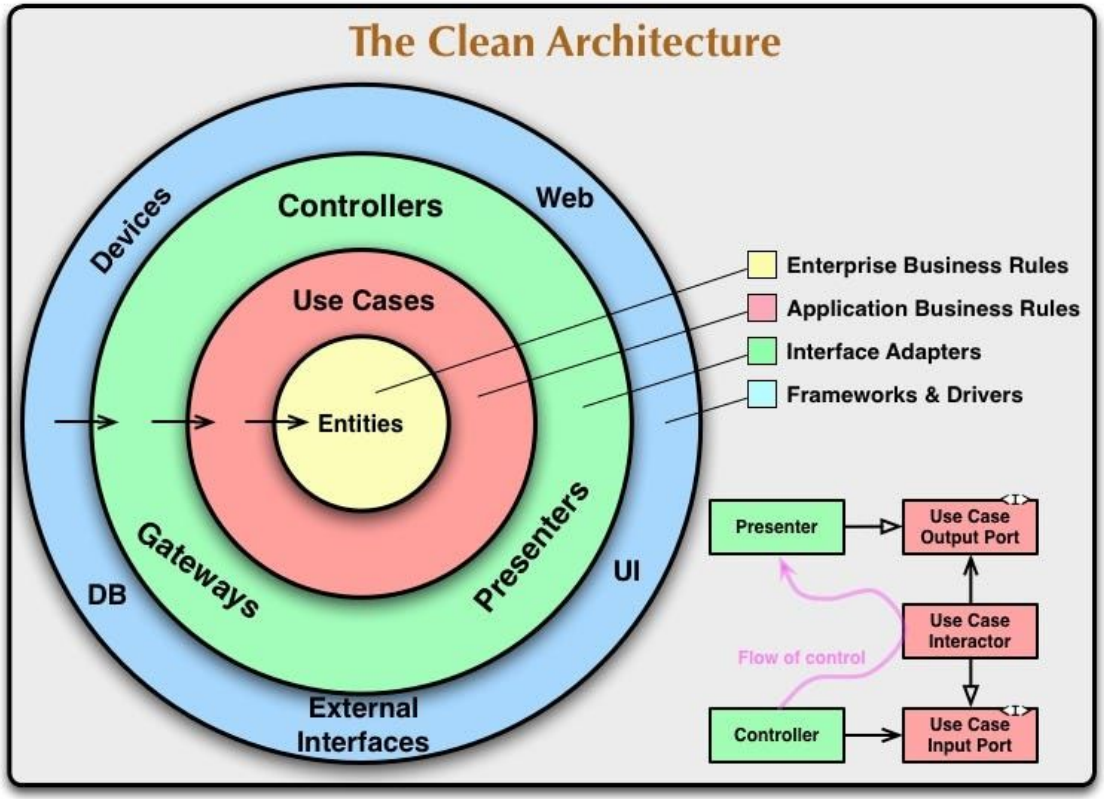
On parle aussi de Right-side, d'infrastructure

Ce dont l'application à besoin, des autres WS, une BDD, des fichiers...

Inversion de dépendances grâce aux interfaces



CLEAN / OIGNON



Objectifs sont d'avoir un logiciel :

- Indépendant des frameworks, **sans les contraindre**
- Testable
- Indépendant de l'UI
- Indépendant de la DB
- Indépendant des API externes

Une application complète des principes SOLID !
Et + de structure dans la nomenclature

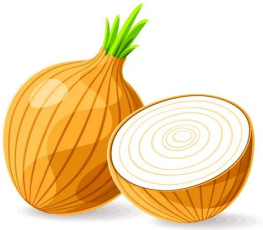
Clean Hexa Oignon...

*Sont des **architectures** où l'on va chercher la **modularité**, le **découplage** et la **testabilité**.*

*Le **business** est **naturellement isolé** du reste.*

DES ARCHITECTURES EN COUCHES...

Où l'on multiplie les couches, en y ajoutant l'inversion de dépendances.



L'architecture Hexagonale est une architecture en couche, c'est juste qu'**il y en a deux** : l'**infrastructure et le domaine**.

La Clean Architecture / Oignon, c'est **une multiplication de ces couches** où les niveaux les + à l'extérieur dépendent de la couche en dessous.

Comparé à une architecture n-tiers classique, on y a juste **appliqué les principes SOLID**, et particulièrement l'**inversion de dépendances**.

IL N'Y A PAS DE SILVER BULLET

Trop complexe pour des cas simples...

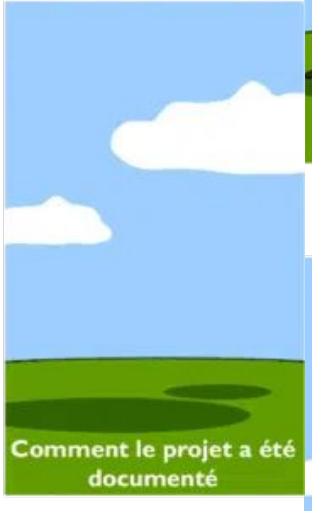
Un vrai et simple CRUD n'a pas besoin de ces principes, une séparation des responsabilités est déjà très bien.

Trop simple pour des cas complexes...

Un applicatif d'un grand groupe comme celui de D4, Norauto etc, a besoin de plus d'isolation, plus de découpage, plus de SOLID


Ce que le développeur a fait


Comment les commerciaux l'ont décrit

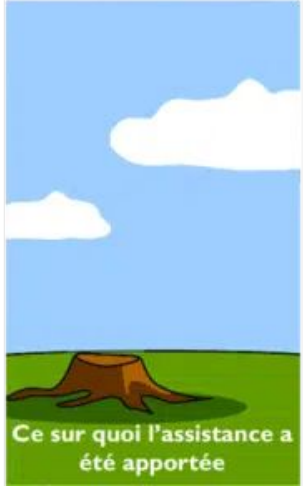

Comment le projet a été documenté


Ce que le client a expliqué


Ce qu'a compris le chef de projet


Ce que l'analyste a conçu


Ce pourquoi le client a été facturé


Ce sur quoi l'assistance a été apportée


Ce dont le client avait réellement besoin

DDD – Domain Driven Design

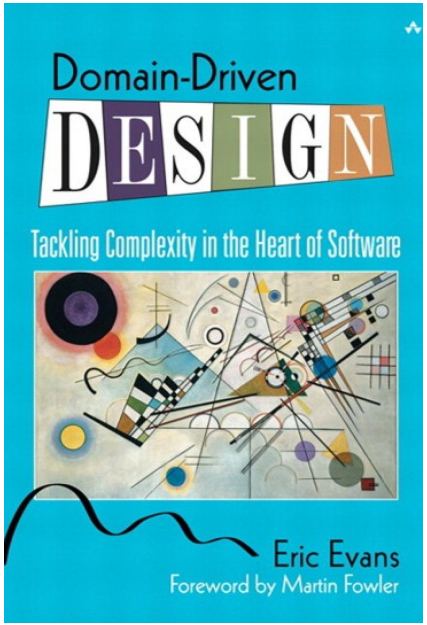
Conception **centrée sur le domaine**
Collaboration constante avec les **experts métiers**
Language ubiquitaire omni-présent !

3 concepts :

- Domain & Model = Business
- **Bounded Context** = diviser le domaine en **sous-domaines cohérents et indépendants**
- Les couches de l'oignon / clean si on aborde la partie technique

Avantages :

- **Réduire les incompréhensions** technique / business
- **Faciliter un passage en micro-service** grâce à la modularité et indépendance des bounded context
- **Maintenabilité**, due à l'organisation notamment !



DDD – Domain Driven Design

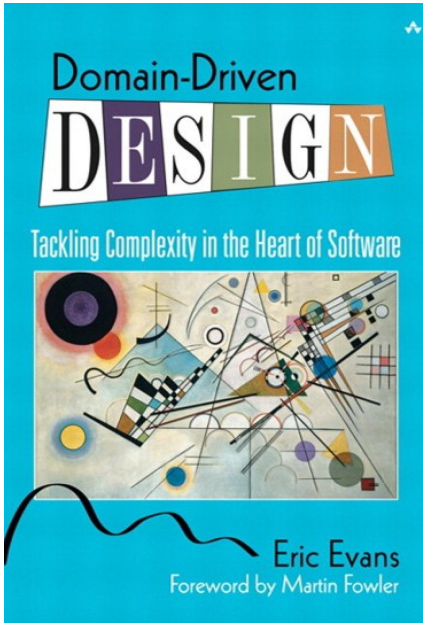
Attention aux inconvénients

Le DDD est un **outil très puissant** si il est **correctement maîtrisé**.
Et **il faut du temps** pour le maîtriser.

À utiliser plutôt quand le **métier est riche** (au sens complexe)
Si il y a d'**importants enjeux sur les règles métiers** (banque…)

Mais pas quand…

Un simple **CRUD**
S'il s'agit de gérer du **contenu** (CMS & Cie)
Pas ou **peu de maintenance**, de besoin d'évolution
Une application **techniquement simple**
Un **domaine très générique** ou **peu de complexité** métier
« Time To Market » rapide (il faut **livrer rapidement**)
Une équipe de **développeurs trop jeunes**



DDD – Domain Driven Design

Par où commencer ?

Lire et se former, créer des side project

Définir un langage métier commun, documenté

Communiquer avec les experts du métiers (ateliers)

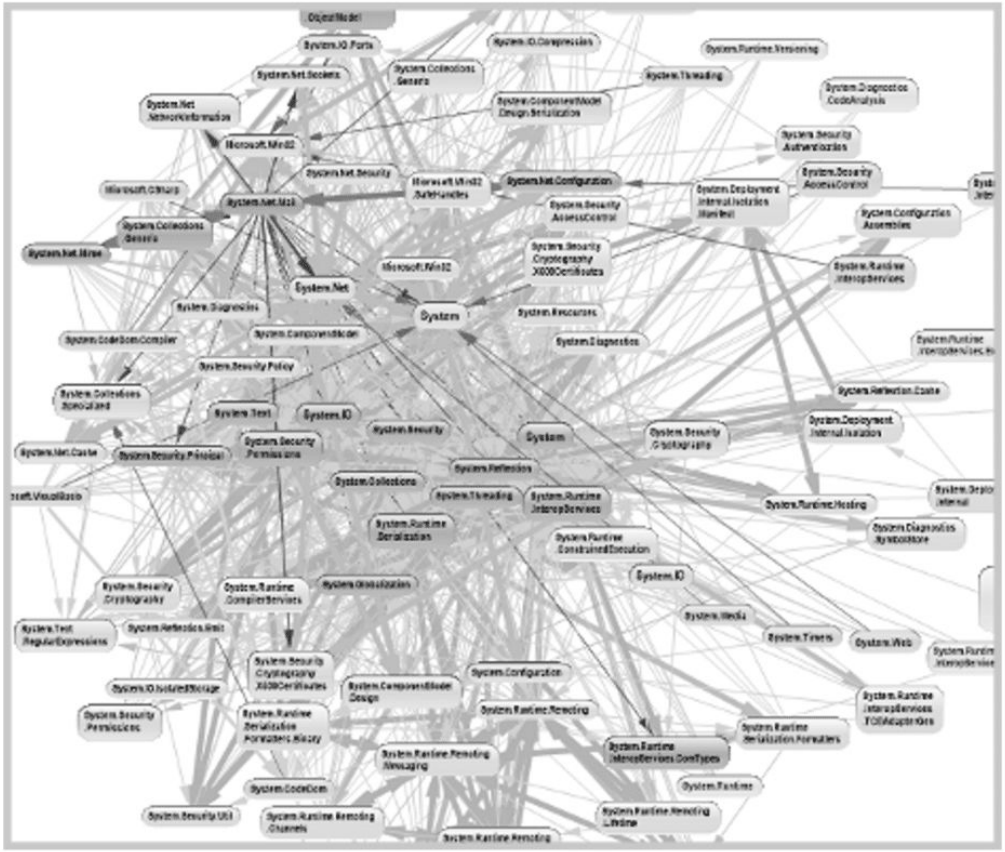
Mettre le business au centre de son code

Isoler les règles de gestion

Découper en module métier

Écrire du code métier et compréhensible par le métier

ET NOS MONOLITHES ALORS ?



Le monolithe **Big Ball Of Mud**...
Tout est **couplé**...
Un controller utilise parfois une DB,
Parfois un service...
Qui parfois envoie un message à une API tierce...
Entité en DB = Entité sur le réseau
“Tester c’est douter, yolo”
“Il faudrait changer de librairie pour les dates, celle qu’on utilise **change son licencing**, on en a pour un bout de temps...”

Bon courage pour **comprendre le business** et le **faire évoluer** !

LA TRANSITION MICRO-SERVICE : L'IDÉE

LA COMPLEXITÉ ESSENTIELLE

COMPLEXITÉ ESSENTIELLE

Inhérente au problème que le logiciel cherche à résoudre.
Elle découle de la nature même du domaine d'application et des besoins des utilisateurs.
Difficile de réduire ou d'éliminer cette complexité, car elle fait partie intégrante du problème.

Par exemple :
Les règles métiers d'une application bancaire
La gestion de commandes
La législation…

LA COMPLEXITÉ ACCIDENTELLE

COMPLEXITÉ ACCIDENTELLE

La complexité accidentelle **provient des choix techniques** dans le développement d'un logiciel.
Elle **n'est pas intrinsèque au problème**, mais résulte de la manière dont nous tentons de le résoudre.
Peut être **réduite ou évitée avec de meilleures pratiques** ou une conception plus simple.

Par exemple :
Code mal architecturé
Choix d'un framework inadéquat
Bugs causés par une mauvaise compréhension...

LA COMPLEXITÉ OBLIGATOIRE

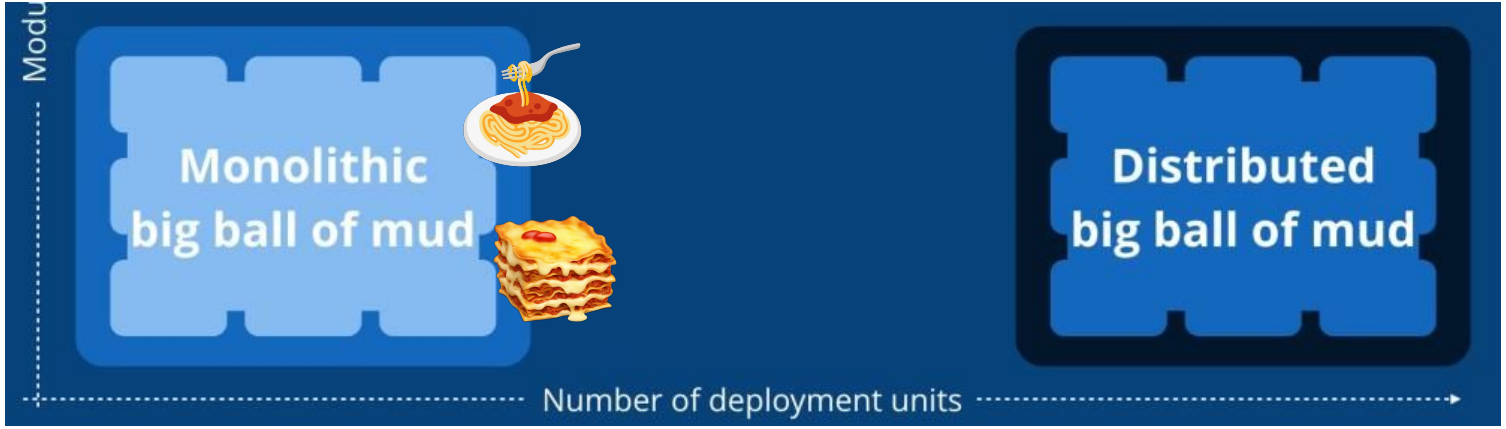
COMPLEXITÉ OBLIGATOIRE

La c**omplexité obligatoire** est la complexité que l'on doit ajouter afin de r**endre le service qui est demandé**.

Elle ne fait **pas partie du problème** à résoudre mais **sans elle**, votre solution n'a **aucune valeur ajoutée**.

Par exemple :
Mise en place d'un serveur HTTP
Configuration de la base de données
Création de l'infrastructure…

LA TRANSITION MICRO-SERVICE : LA RÉALITÉ



COMPLEXITÉ ESSENTIELLE

COMPLEXITÉ ACCIDENTELLE

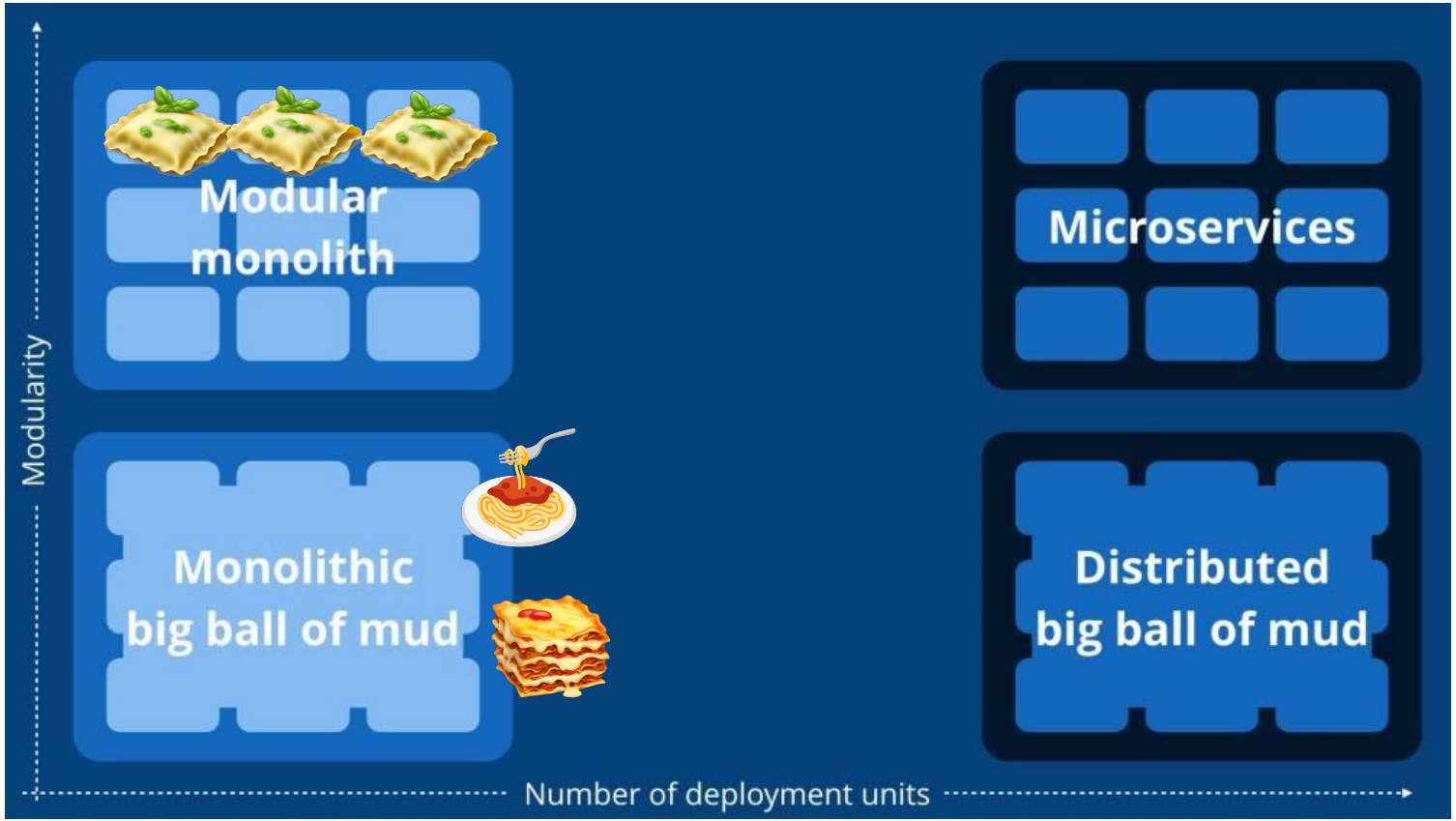
COMPLEXITÉ OBLIGATOIRE

Migrer du monolithe au microservice,

Ce n'est pas prendre une lasagne et l'enrober de pâte pour en faire un ravioli...

Plus qu'une solution technique, il faut se pencher sur les ***bordures fonctionnelles.***

VERS DE LA MODULARITÉ



À VOUS DE JOUER

RESSOURCES

Alistair in the Hexagone - talk YT en 3 parties

Des microservices aux migroservices - Talk YT

Model Mitosis : ne plus se tromper entre les microservices et le monolithe - Talk YT

DDDViteFait - Livre

DDD (Blue book, stratégique) - Livre

DDD (Red Book, tactique) - Livre

TDD By example - Livre

Clean Architecture - Livre

Architecture Hexagonale : trois principes et un exemple d'implémentation - Octo Article

https://github.com/VaughnVernon/IDDD_Samples