

# Approche Objet - Modèle de Conception

Marie Beurton-Aimar  
et  
Xavier Blanc (Youtube)

November 8, 2022

# Introduction

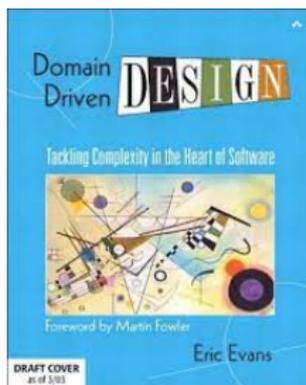
- ▶ Il existe plusieurs modèles de conception orientés objets **xDD** : test Driven Design, behaviour Driven Design (focalisent sur l'implémentation de bons comportements du logiciel), Model Driven Architecture, Data Domain Architecture, Domain Driven Design .....
- ▶ **DDD** : focalise sur le domaine de l'application et sur la logique associée. Le domaine est le métier adressé par le logiciel.
- ▶ Définie par les 3 points :
  - ▶ Attention portée sur le langage pour favoriser l'émergence d'un vocabulaire commun aux experts du fonctionnel (analyste) et à ceux du technique (programmeur).
  - ▶ Connexion très forte entre le modèle et l'implémentation.
  - ▶ Focalisation sur l'essence du métier - compétences spécifiques de l'entreprise.

# Comprendre le Métier

- ▶ Le développeur doit s'appropriier le domaine : lecture de documentation, discussion avec les experts, observer un expert métier, analyser d'autres logiciels existants relatifs au domaine.
- ▶ Les techniques de recueil de connaissances sont utiles pour capturer l'expertise :
  - ▶ Préparer l'interview pour cadrer les sujets abordés.
  - ▶ Mécanismes de double confirmation.
  - ▶ Demandes d'exemples variés et représentatifs à la fois des principes généraux et des exceptions.
  - ▶ Retour sur les interprétations de réponses précédentes afin de vérifier qu'il n'y a pas eu de malentendu.

# Modéliser le Domaine.

- ▶ Extraire le modèle ?
  - ▶ Eric Evans : *Domain Driven Design: Tackling Complexity in the Heart of Software*. 2003
- ▶ Concept principal : *Crunching or Distillation* c.à-d. prendre un problème dans tous les sens tout en demandant un feedback métier.



# Qu'est-ce qu'un modèle ?

- ▶ Un modèle est une représentation du monde mais il ne contient pas tout l'univers !
- ▶ Sélection des informations pertinentes, utiles à la résolution du problème.
- ▶ Des données peuvent être supprimées. Les *bruits* parasites doivent être supprimés.
- ▶ Un modèle doit être prédictif sans être fidèle à la réalité.

# Qu'est-ce qu'un modèle ?

- ▶ Le vocabulaire (lexique) du domaine doit être borné: un mot n'a de signification que dans un contexte donné - **zone linguistique**.
- ▶ Les différents services : marketing, comptabilité, livraison... n'ont pas le même point de vue sur le *client*.
  - ▶ Marketing : un profil de client est alimenté par la fréquence de ses visites ou son type d'achat.
  - ▶ Comptabilité : le client est avant tout un compte en banque.
  - ▶ Livraison : le client *est* une adresse postale.
- ▶ Le **modèle**, le **code** et le **lexique** sont liés : le code contient le vocabulaire, tout changement dans le code introduit un changement dans le modèle.
- ▶ En DDD le vocabulaire commun s'appelle : *ubiquitous language*.

# Exemple

- ▶ Un site de Ecommerce :
  - ▶ Concepts métiers : les produits du site et le panier.
- ▶ Questions :
  - ▶ qu'est ce qu'un produit ? un panier ?
  - ▶ quelles relations entre ces concepts ?
  - ▶ quel processus métier de bout en bout ?
- ▶ ==> Tests du domaine

# Exemple

- ▶ Partie d'échecs :
  - ▶ Concepts métiers : un pion, une dame, un roi ... un échiquier ... ne pas oublier la partie ! gagner ? perdre ?
  - ▶ qu'est ce qu'une partie ? comment bouger les pièces ?
  - ▶ quelles relations entre ces concepts ?
  - ▶ quel processus métier de bout en bout ?
- ▶ ==> Tests du domaine

# Méthode Domain Driven Design

- ▶ Coder le domain en objet.
- ▶ Aligner les objets sur le métier
- ▶ Utiliser les concepts/patterns de la méthode :
  - ▶ Value Object.
  - ▶ Entity.
  - ▶ Aggregate.
  - ▶ Repository.
  - ▶ Service.

# Value Object : Immutable Object

- ▶ Un objet métier qui représentent des données qui ne changent pas avec le temps.
- ▶ Les traitements sont de la lecture de données et du contrôle à la création.
- ▶ Exemple :
  - ▶ Un point GPS
  - ▶ Une couleur
  - ▶ Un produit d'un site de eCommerce
  - ▶ Un pion, une dame ... ..

# Codage d'un Value Object

- ▶ Principes :
  - ▶ Valeurs affectées à la création.
  - ▶ Impossibilité de modifier les valeurs après la création.
- ▶ Règles de codage :
  - ▶ Propriétés(attributs) `private` et `final` dont les types sont basiques ou des Value Object eux-mêmes.
  - ▶ Pas de *setter*.
  - ▶ Des *getter* public.
  - ▶ Affectation de toutes les valeurs dans le constructeur.
  - ▶ Surcharger `equals` pour faire en sorte que l'égalité soit une égalité de valeurs.

# Avantages des Value Objects

- ▶ Encapsulation garantie par construction (car *immutable*).
- ▶ L'échange des *Value Objects* est sans risque.
- ▶ Construction et destruction des *Value Objects* sans risque.
- ▶ Il est possible de factoriser les *Value Objects* afin d'avoir un *Value Object* par valeur.

# Favoriser les Value Objects

- ▶ Construire des *Value Objects* plutôt que d'utiliser les types de base:
  - ▶ ~~String phoneNumber;~~
  - ▶ `PhoneNumber phoneNumber;`
- ▶ Exploiter les *Value Objects* dans les structures complexes :
  - ▶ ~~int roadNumber; String roadName; String town;~~
  - ▶ `PostalAddress address;`
- ▶ Ne pas hésiter à remplacer un *Value Object* par un autre :
  - ▶ ~~translate(int x,int y) {for (Element e:canvas.getElements()) {e.x += x...}}~~
  - ▶ `translate(int x, int y) { for (Element e:canvas.getElements()) {canvas.add(new Element(e.x+x, e.y+y)); canvas.remove(e); } }`

# Codage d'une Entity

- ▶ C'est l'objet qui contient la logique *business* de l'application.
- ▶ Principes :
  - ▶ Affectation d'une identité à la création qui ne changera pas.
  - ▶ Un état qui changera et sera donc encapsulé.
  - ▶ Des traitements métiers dont les noms sont alignés sur le métier.
  - ▶ Très souvent liés à des *Value Objects*.

# Codage d'une Entity

- ▶ L'identité:

- ▶ Principale différence entre le *Value Object* et l'*Entity*.
- ▶ Peut être de type primitif ou d'un type prédéfini par un *Value Object* (recommandé).
- ▶ Exemple :

```
public class Customer{
    private String id;
    private String name;
}
public class Order{
    private String id;
    private String customerId;
}
```

# Codage d'une Entity

- ▶ L'identité:

- ▶ Principale différence entre le *Value Object* et l'*Entity*.
- ▶ Peut être de type primitif ou d'un type prédéfini par un *Value Object* (recommandé).
- ▶ Exemple :

```
public class Customer{
    private CustomerId id;
    private String name;
}

public class Order{
    private OrderId id;
    private CustomerId customerId;
}
```

# Codage d'une Entity

- ▶ L'identité:

- ▶ Principale différence entre le *Value Object* et l'*Entity*.
- ▶ Peut être de type primitif ou d'un type prédéfini par un *Value Object* (recommandé).
- ▶ Exemple :

```
public class Customer{
    private CustomerId id;
    private String name;
}
public class Order{
    private OrderId id;
    private Customer customer;
}
```

# Codage d'une Entity

- ▶ Règles de codage :
  - ▶ L'attribut `id` privé et final dont la valeur est donnée à la construction (un *getter* sans *setter*).
  - ▶ Des attributs/propriétés qui définissent l'état interne et pas de *setter*.
  - ▶ Des traitements métier publiques qui changent l'état.
- ▶ Possède un cycle de vie (continuité)  
*"Many objects are not fundamentally defined by their attributes, but rather by a thread of continuity and identity."*  
(Evans)

## Avantage des Entity

- ▶ L'id est présent dans le code (et pas généré par une plateforme technique).
- ▶ L'état est clairement identifié.
- ▶ Les traitements sont alignés avec le métier - pas de code amnésique.
- ▶ Utilisation de *Value Objects* qui favorise les échanges entre les objets.
- ▶ On favorise les *Entity* pour éviter les *bases de données sur pattes* et pour préciser les cas d'erreurs et d'exception.

# Aggregate : un Entity composite

- ▶ Un objet complexe qui contrôle des *Entity*.
- ▶ Définit un élément racine.
- ▶ Exemple :
  - ▶ Carnet d'adresse avec des fiches utilisateurs.
  - ▶ Une partie d'un jeu.
  - ▶ Un panier électronique.
  - ▶ Une partie d'échec.

# Codage d'un Aggregate

- ▶ Principes :
  - ▶ Comme pour l'*Entity* on doit définir un identifiant unique et non mutable.
  - ▶ Un état qui change et qu'on encapsule.
  - ▶ Les attributs qui définissent l'état sont des *Entity*.
  - ▶ Définition de traitements métiers dont le nom doit être fortement corrélés au métier/context.
  - ▶ Certains attributs peuvent aussi être des *Value Object*.

# Codage d'un Aggregate

- ▶ Règles de codage:
  - ▶ La valeur de l'identifiant est fournie ou générée à la construction - pas de *setter*, un seul *getter*.
  - ▶ Les attributs/propriétés n'ont pas de *setter*.
  - ▶ Les *Entity* qui appartiennent à l'*Agregate* sont créées par lui, il fournit l'identifiant.
  - ▶ Des traitements métiers publics qui gèrent l'état de l'*Agregate*.
  - ▶ Ne pas oublier de structurer l'*Agregate* grâce aux *Value Objects* pour diminuer la taille de l'*Agregate*.

# Avantage des Aggregates

- ▶ Présence d'un élément racine.
- ▶ C'est l'objet/le composant utilisé par l'utilisateur/le client.
- ▶ Gère son état à l'aide des *Entity*.
- ▶ Les traitements sont en synergie avec le métier et les interactions utilisateur.

# Avantage des Aggregates

- ▶ Présence d'un élément racine.
- ▶ C'est l'objet/le composant utilisé par l'utilisateur/le client.
- ▶ Gère son état à l'aide des *Entity*.
- ▶ Les traitements sont en synergie avec le métier et les interactions utilisateur.
- ▶ Pourquoi favoriser les *Aggregate* ?
  - ▶ Construction d'un unique objet
  - ▶ Préciser les cas d'erreur ou les exceptions des traitements métier.

# Synthèse sur la conception du Modèle

- ▶ Le métier au centre ==> la couche domaine.
- ▶ *Value Object* ==> objet non mutable.
- ▶ *Entity* ==> objet classique avec état.
- ▶ *Aggregate* ==> objet composite, racine.

# Aggregate et Sauvegarde

- ▶ L'accès au domaine depuis l'extérieur ==> *Repository*.
- ▶ L'*Aggregate* est visible hors du domaine ==> il sera utilisé par la couche application.
- ▶ L'extérieur doit pouvoir accéder à l'*Aggregate* ==> on doit pouvoir le retrouver, importance de l'identifiant.
- ▶ L'*Aggregate* ne donne pas accès à son état ==> les paramètres de ses méthodes ne doivent pas être des *Entity* - d'où l'importance des *Value Object*.

# Repository - Entrepôt d'Aggregates

- ▶ Un *Repository* est un objet du domaine qui stocke des *Aggregates*.
- ▶ Retrouver un *Aggregate* à partir de son identifiant ou sur d'autres critères.
- ▶ Sauver un *Aggregate* - soit mettre à jour les dépôts.
- ▶ ==> le *Repository* est visible hors du domaine.

# Gestion des Aggregates

- ▶ Un *Repository* doit proposer des méthodes pour retrouver des *Aggregate* :
  - ▶ `findById():Aggregate`
  - ▶ `findByAnyFeature:Aggregate`
- ▶ Dans le cas où le nombre d'*Aggregates* est grand il faut prévoir des accès par paquet (pagination/bufferisation).
  - ▶ **Attention** : les méthodes qui permettent d'explorer la totalité du *Repository* ne sont pas très utiles.
- ▶ ==> le *Repository* construit l'*Aggregate* en mémoire.

## Exemple du Jeu d'échec

```
public class GameRepository{  
    public Game findGameById(GameId gameId){  
    }  
    public Game[] findGameByPlayer(Player player){  
    }  
}
```

# Reconstruire un Aggregate

- ▶ Pour retrouver un *Aggregate* il faut le reconstruire en mémoire.
- ▶ Cela nécessite de re-construire l'intégralité de son état
  - ▶ il faut reconstruire les *Entity* qui le composent.
- ▶ Passer l' id à l'*Aggregate* lors de sa création.

## Exemple du Jeu d'échec

```
public Game findGameById(GameId gameId){
//lire les donnees sauvegardees... fichier, base de donn
    data= File.read();

    //reconstruire l' aggregate
    Game g= new Game(data.id);

    // restaurer l' etat
    g.moves = data....

    return g;
}
```

# Sauver un Aggregate

- ▶ Un *Repository* propose une méthode pour sauver l'*Aggregate*.
  - ▶ On passe en entrée l'*Aggregate* (objet en mémoire)
  - ▶ On n' obtient rien (`void`), on peut recevoir une exception si la sauvegarde a échoué.
  - ▶ ==> `public void save(Aggregate agg) // exception metier`
- ▶ Un *Repository* peut proposer des méthodes pour mettre à jour la sauvegarde.

## Exemple du Jeu d'échec

```
public class GameRepository{
    public void save (Game game){
    }
    public void update(Game game){
    }
}
```

# Sérialisation de l'état

- ▶ Le code de `save()` doit écrire l'état de l' *Aggregate* sur un support : fichier, base de données ...
- ▶ Il faut aussi obtenir l'état de l' *Aggregate* et/ou des *Entity* gouvernées par l' *Aggregate*.
- ▶ L' *Aggregate* doit fournir les méthodes permettant d'obtenir cet état: sérialization de l'objet.
  - ▶ Idéalement ces méthodes ne sont accessibles que par le *Repository*.

## Exemple du Jeu d'échec

```
public void save (Game g){
    data.id=g.getId();
    // retrouver l' emplacement des pieces
    data.location=g.getPieceLocations();
    //savoir si la partie est encore en cours
    data.isClosed=g.isClosed();
    // sauver sur un support
    file.write(data);
}
```

## Exemple GameRepositoryFile

```
public class GameRepositoryFile{

    public Game findGameById(GameId gameId){
        File f = new File(gameId);
        Game g = new Game(f.read() ...) ;
        return g;
    }
}

public void save(Game g){
    File f = new File(g.gameId);
    f.write( g ...);
}
}
```

## Exemple GameRepositoryInMemory

```
public class GameRepositoryInMemory{

    Set<Game> memory;

    public GameRepositoryInMemory(){
        memory=new HashSet<Game>();
    }
    public void save(Game g){
        memory.add(g);
    }
    public Game findGameById(GameId gameId){
        for (Game g:memory){
            if (g.id == gameId){
                return g;
            }
        }
    }
}
```

# Les couches du DDD - Rappel

- ▶ *Domain Layer* : entités du domaine - données et comportements.
- ▶ *Infrastructure Layer* : persistance, sauvegarde, implémentation du *Repository*. Ne doit pas contaminer la couche domaine. Les objets du domaine doivent rester ignorant des tâches accomplies par la couche *infrastructure*.
- ▶ *Application Layer* : service, interface utilisateur ...

# Synthèse

- ▶ L'*Aggregate* est visible hors du domaine.
- ▶ Besoin de sauver et/ou restaurer des *Aggregate* ==> le *Repository*.
- ▶ Le code du *Repository* est technique :
  - ▶ Lecture fichier, écriture fichier, base de données ... ==> comment sortir le code du domaine ?

# Repository et couche Infra

- ▶ La couche Infra ? quelle utilité ? quelle solution aux problèmes de la séparation des responsabilités (separation of concerns) ?

## Repository et Code Technique

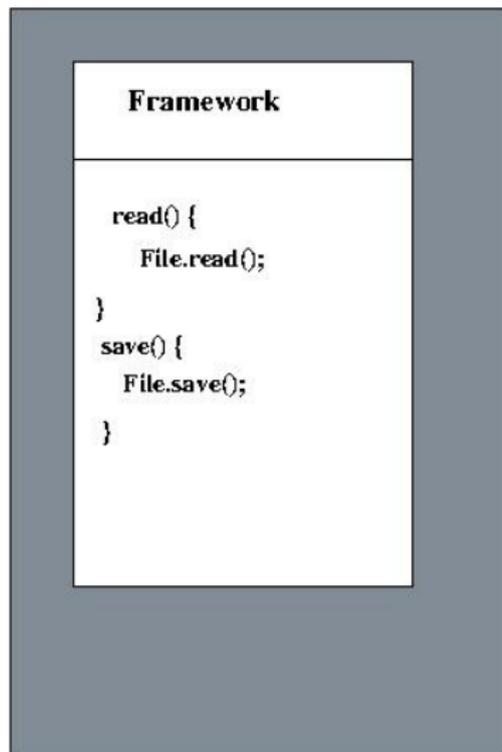
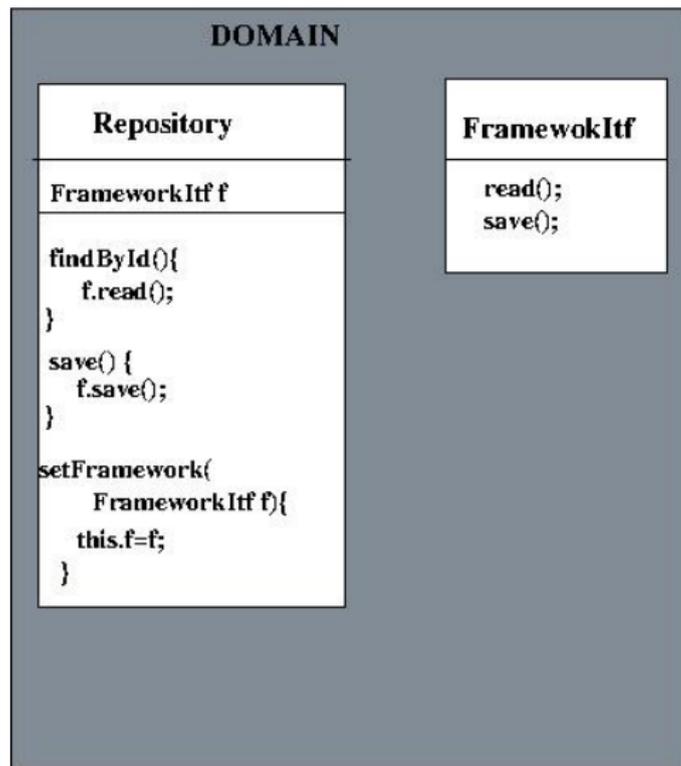
- ▶ Le Domaine doit rester abstrait.
- ▶ Exemple précédent : le jeu d' échecs - le Repository contenait du code technique.

```
public class GameRepositoryFile{
    public Game findGameById(GameId gameId){
        File f = new File(gameId);
        Game g = new Game(f.read() ... ) ;
        return g;
    }
}

public void save(Game g){
    File f = new File(g.gameId);
    f.write( g ...);
}
```

- ▶ La couche Domain ne devrait pas dépendre du framework technique.
- ▶ Corollaire : Le framework technique doit être au service du Domain

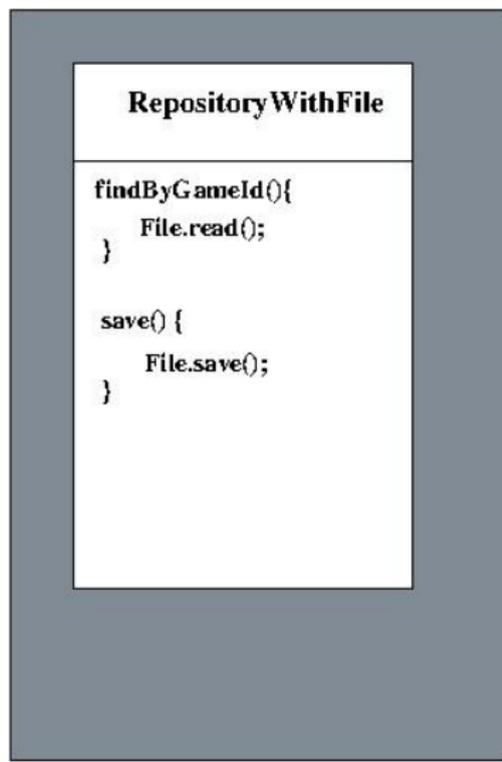
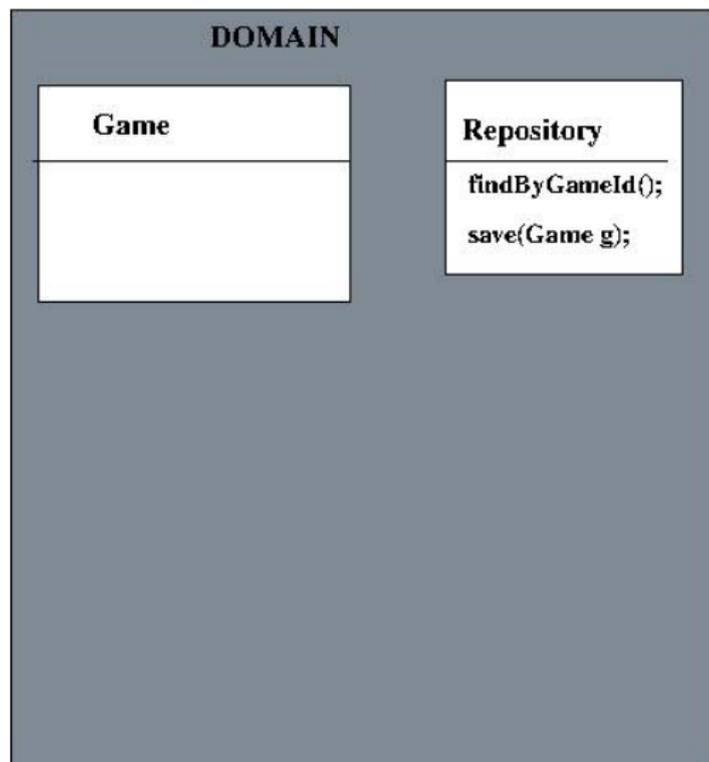
# Interface et inversion de dépendance



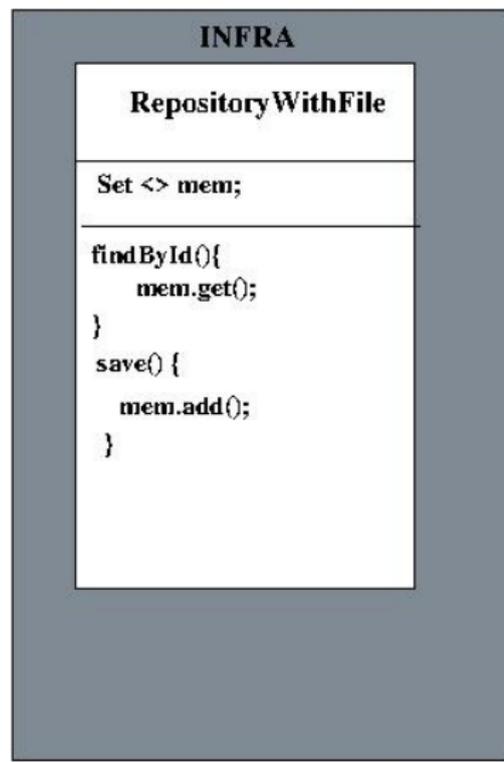
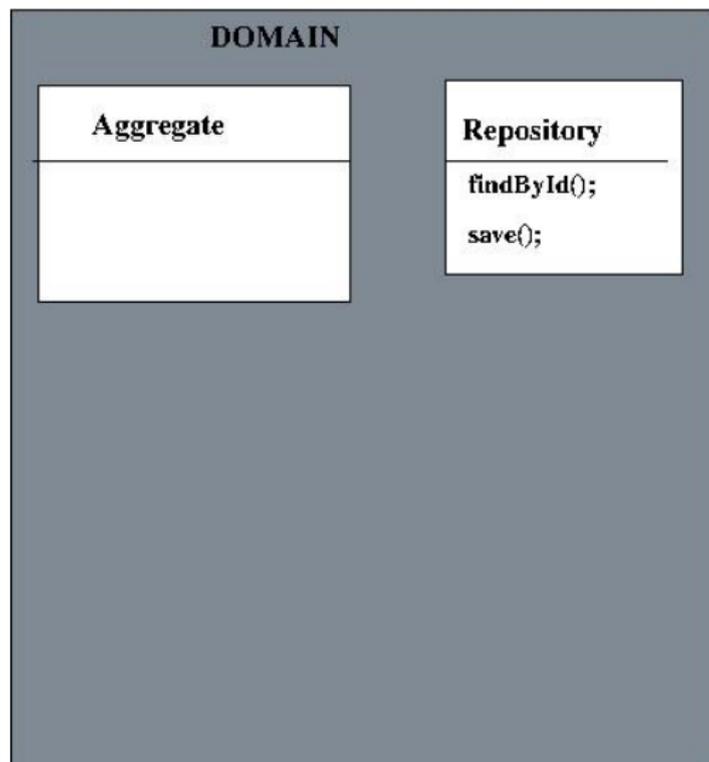
# Interface et inversion de dépendance

- ▶ L'exploitation d'une interface permet d'inverser les dépendances.
  - ▶ Définir le framework par une interface : `save()` `load()`
- ▶ L'appelant ne dépend que de l'interface :
  - ▶ L'interface appartient à la couche de l'appelant.
- ▶ L'appelé réalise l'interface :
  - ▶ L'appelé est dans une autre couche.
- ▶ A l'exécution on devra lier l'appelant à celui qui réalisera l'interface.

## Repository concret



## Repository concret en mémoire



# La Couche Infrastructure

- ▶ Cache tout ce qui est volatile, ou qui peut changer en fonction de la technologie, du déploiement, du système d'exploitation, de la méthode de transport des données ...
- ▶ Prend en charge la conversion des données en provenance d'autres domaines.
- ▶ **Important** : il est important de disposer d'infrastructure qui permette la réutilisation de fonctionnalités communes pour accélérer le développement de nouvelles applications :
  - ▶ Persistance
  - ▶ Logging, messaging
  - ▶ Domain events

# La Couche Application

- ▶ Une vision orientée **service** - sans état du Domain.
- ▶ Objectif : éviter que le client / User Interface manipule directement les Aggregate qui sont les points d'accès au Domain.
- ▶ La couche Application va s'occuper de définir les tâches à effectuer pour réaliser une tâche d'application donnée.
- ▶ Elle est responsable du mandat du travail du Domain nécessaire et interagit avec d' autres services - externes ou non.

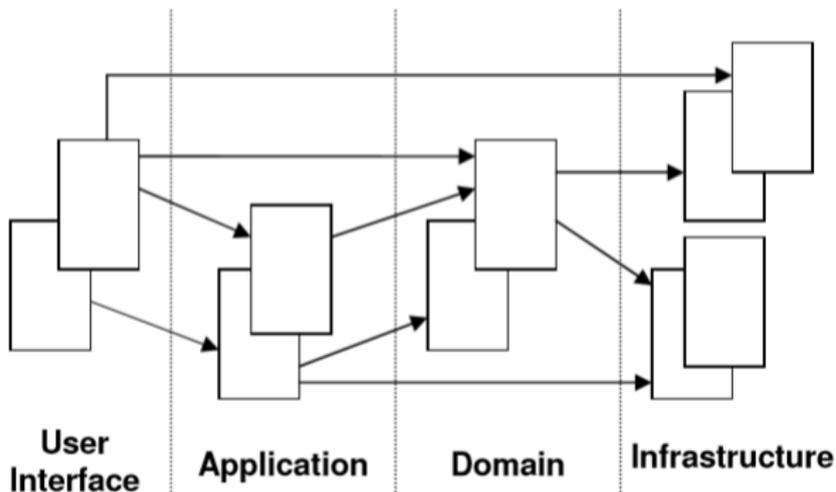
# La Couche Application

- ▶ Exemple : Application financière.
  - ▶ Implémentation de la capacité d'un utilisateur particulier à approuver une opération.
  - ▶ Quand cette opération est validée une entité du modèle change d'état (approuvée0).
  - ▶ D'autres acteurs doivent être informés par un message.
  - ▶ La couche Domain n'a pas à prendre en charge le service de messagerie.
- ▶ La couche Application est une couche d'isolation : les comportements de cette couche ne doivent pas être affectés par un changement de code dans les autres couches : Domain, Infra.

# Couche UI

- ▶ Aussi appelée couche *Présentation* dans les traductions en français.

## Layered Architecture



# Couche UI

- ▶ Accède à la couche Application
- ▶ Manipule les types rendus visibles par la couche Application.
- ▶ Autant de UI que de moyens d'accès à l'Application :  
interface web, interface graphique ...
- ▶ Exemple du jeu d'échecs graphiques :
  - ▶ les objets graphiques correspondant pièces ?
  - ▶ sont une vue des pièces,
  - ▶ pas métier, pas infra, pas application