

# DDD – DOMAIN DRIVEN DESIGN

FORMATION A LA CONCEPTION ORIENTÉE DOMAINE MÉTIER



**BNP PARIBAS**

La banque d'un monde qui change



# O CONCEPTION LOGICIELLE AVEC LE DOMAIN DRIVEN DESIGN (DDD)



# Domain Driven Design (DDD)

- Une méthode de conception logicielle qui favorise une conception proche de la réalité
- Focalisation sur le métier (au lieu d'uniquement des parties techniques)
  - On utilise explicitement des noms et des verbes du domaine dans le code
  - La connaissance métier est le noyau de l'application
- Le DDD est un ensemble de principes et éléments de conception
  - Design Objet et Séparation des responsabilités
  - Catalogue de concepts: les Building Blocks (Entités, Value Objects, Services, Repository, etc)
  - Ubiquitous Language



# Objectifs de la formation

- Fournir le matériel nécessaire pour comprendre ce qu'est le **Domain Driven Design (DDD)** et ce qu'il peut vous apporter
- La conception (au sens générale) peuvent être complexes
  - Seule la pratique de la conception logicielle et des différentes briques DDD vous permettront d'être à l'aise avec l'ensemble des concepts
- A l'issue de la formation, certains concepts et principes seront directement applicable à vos projets sans faire du DDD à part entière
  - Valider rapidement une conception, valider rapidement un développement
  - Refactorer au fur et à mesure une conception existante vers un design DDD
    - Ex: Utilisation uniquement des Value Object pour avoir un **fort** ROI



# Programme - Journée 1

- **Fondamentaux de la modélisation et pourquoi le DDD**
- **Prérequis DDD**
  - Fondamentaux de la qualité
    - Dépendance, couplage, et cohésion
  - Fondamentaux de la programmation OO
    - SOLID, DIP, IoC, DI, Design Patterns
  - Autres principes indispensables
    - Yagni, Dette Technique
- **DDD Tactique**
  - Introduction aux briques de bases
    - Value Object



# Programme - Journée 2

---

- **DDD Tactique (Suite)**
  - Introduction aux briques de bases
    - Entité
  - Poursuite du Catalogue DDD
    - Application Service, Domain Service, Infrastructure Service



# Programme - Journée 3

---

- Cohérence transactionnelle et cohérence éventuelle
- DDD Tactique Avancé
  - Agrégat
  - Domain Event



# Bibliographie et lectures recommandées

- **DDD**
  - Domain driven design
  - Implementing Domain driven design
  - Sandro Mancuso: Crafted Design
- **Craftsmanship**
  - Refactoring
  - Design patterns
  - Patterns of enterprise application architecture
  - Analysis patterns
  - Enterprise integration patterns
  - Growing object-oriented software, guided by tests
  - BDD in action
  - Working effectively with legacy code
- **Java**
  - Effective Java
  - Java puzzlers
  - Java concurrency in practice
  - The well-grounded Java developer
  - Java performance
  - Java collections and generics
  - Java 8 in action
  - Mastering lambdas
- **Divers**
  - The mythical man-month



# Logistique

- Planning
  - 9H30 – 17H30
  - 2 pauses de 15min (matin et après-midi)
  - Déjeuner en groupe (11H45-13H)
- Déroulement
  - Présentation théorique sous forme de diapositives
  - Atelier de conception (fil rouge)
- Ressources
  - Crayon et papier requis pour les ateliers
  - Fourniture de la présentation à l'issue de la formation
- Interactions
  - Permanente
- Autres questions?





# FONDAMENTAUX DE LA MODÉLISATION



# Objectifs



## 1 Domaine métier et l'approche du DDD

- 2 Les différents modèles
- 3 Des problèmes de conception constatés
- 4 Première approche de modélisation
- 5 Le contenu de la méthode DDD
- 6 Processus de modélisation préconisé

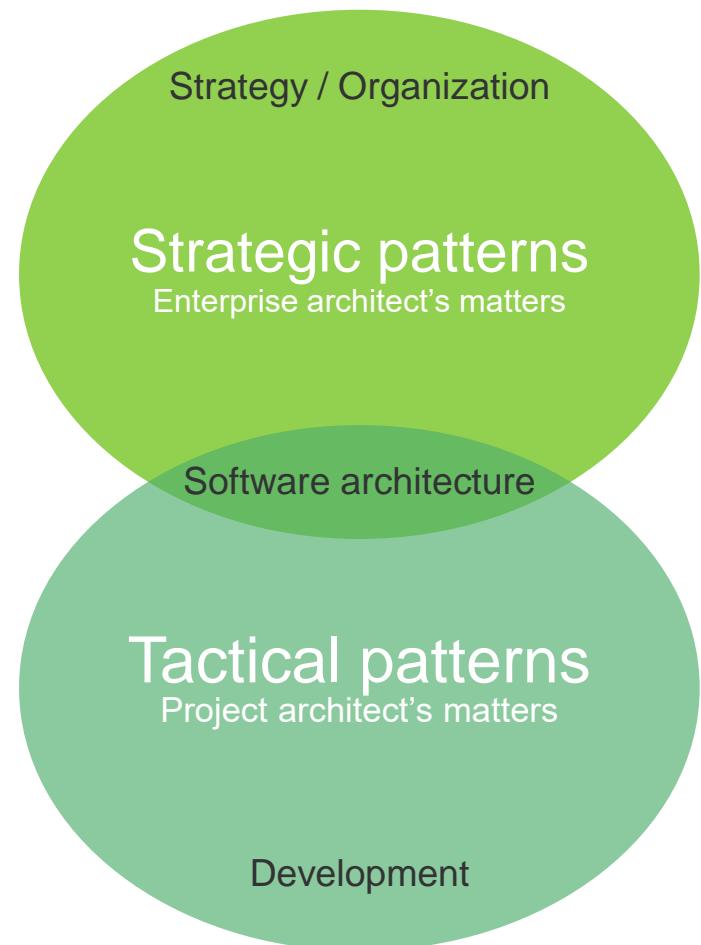
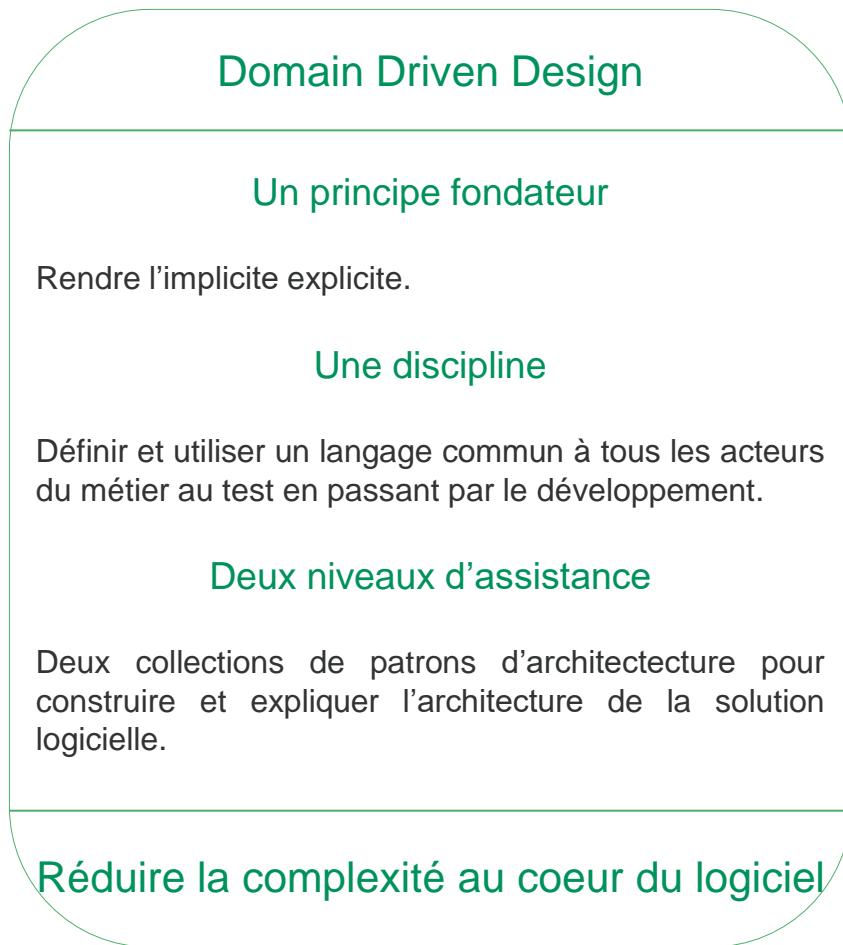


# Le domaine

- **Le domaine métier est une sphère de connaissance et d'activités propres à un métier**
  - Un vocabulaire spécialisé (une signification particulière pour chaque terme)
  - Des experts
  - Des sphères d'influence
- Exemples de domaines métier:
  - Banque
  - Transport (réseaux ferrés)
  - Gestion des utilisateurs
  - ...
- Le logiciel orchestre tous les concepts du domaine pour apporter une solution à un problème donné
  - Exemple: Solution de réservation de salle



# Le contenu du DDD en quelque mots...



# La conception

---

- La conception est à la fois une activité et un livrable (artefact)
- Concevoir, c'est prendre des décisions durant toutes les phases du logiciel
  - **Faire des choix et assumer ces choix**
  - Prendre des décisions **en adéquations avec les contraintes**
    - Exigences de qualité
    - Ressources mises à notre disposition
    - Scope du logiciel
    - Compétences des équipes
    - Temps de réalisation alloué
- Tous les acteurs du logiciel sont des concepteurs (mais à différents niveaux)
  - Business Analyst (BA)
  - Concepteur fonctionnel et/ou technique
  - Développeur



# Objectifs

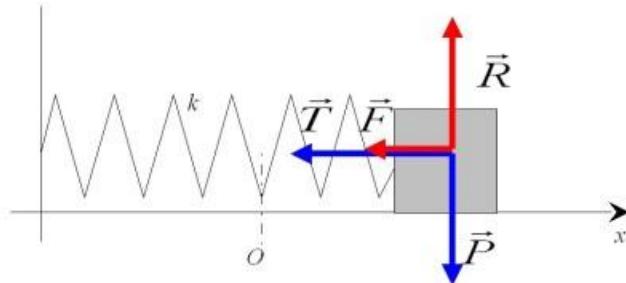


- 1 Domaine métier et l'approche du DDD
- 2 Les différents modèles**
- 3 Des problèmes de conception constatés
- 4 Première approche de modélisation
- 5 Le contenu de la méthode DDD
- 6 Processus de modélisation préconisé



# Le modèle

- Les modèles décrivent certains aspects du domaine en donnant une description simplifiée (mais réaliste) de la réalité
  - Exemple: Un oscillateur harmonique



- Un modèle forme les bases d'un *langage spécialisé* pour un domaine
- Les modèles sont créés et exploités par les différents acteurs de la chaîne logicielle. Ils constituent une activité de collaboration pour
  - Communiquer
  - Faciliter l'exploration du métier
  - Gérer la complexité
  - Générer du code (de moins en moins)

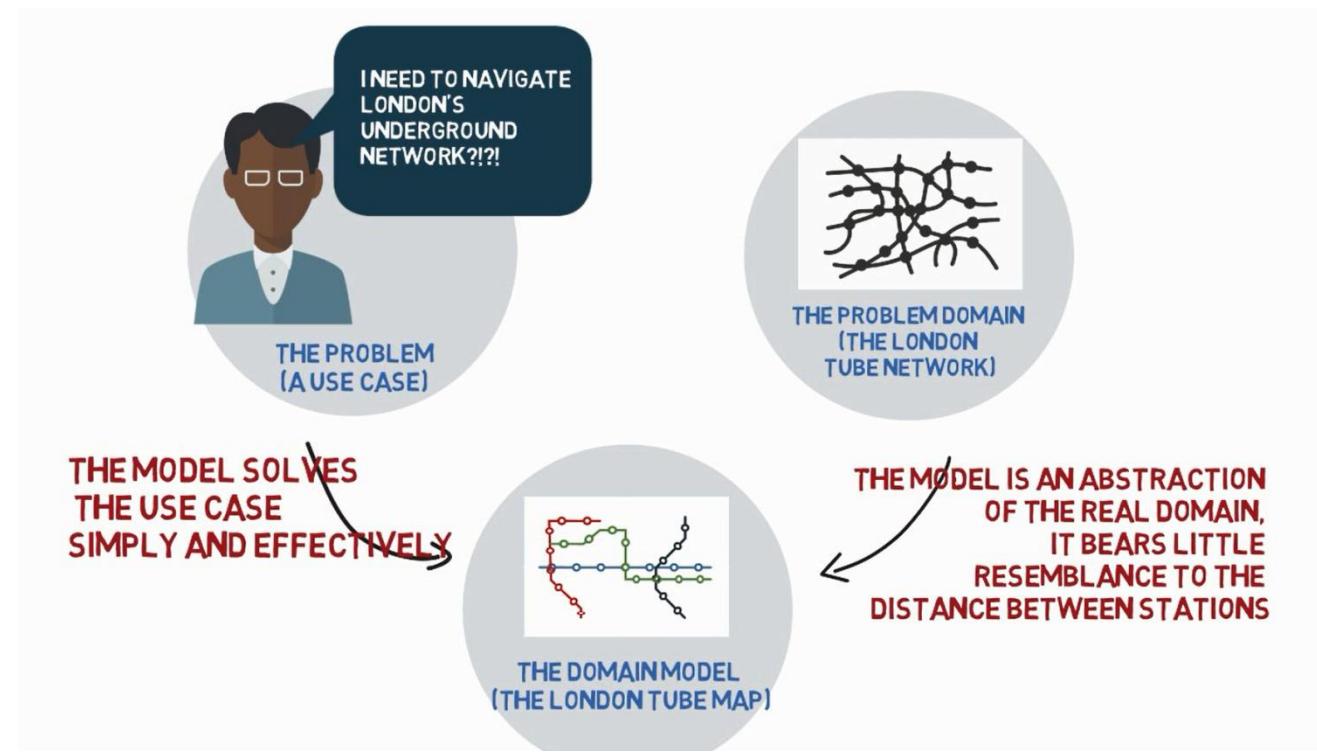
# Trouver le bon modèle?

« All models are wrong, some are useful »  
*G.Box, statistician*



# A quoi sert un modèle ?

Quand on modélise dans le but de concevoir, on ne fait pas du modélisme, c'est-à-dire chercher à reproduire tous les détails, mais on recherche quelles sont les caractéristiques minimales nécessaires à la résolution du problème.



Arrêt sur image d'une vidéo de Scott Millett



# Un bon modèle



- Une représentation exacte de la réalité**  
Impossible et inutile



**Une abstraction utile**

Un modèle composé d'abstractions simples mais composable s en un tout reflétant la réalité



**Résoud des problèmes**

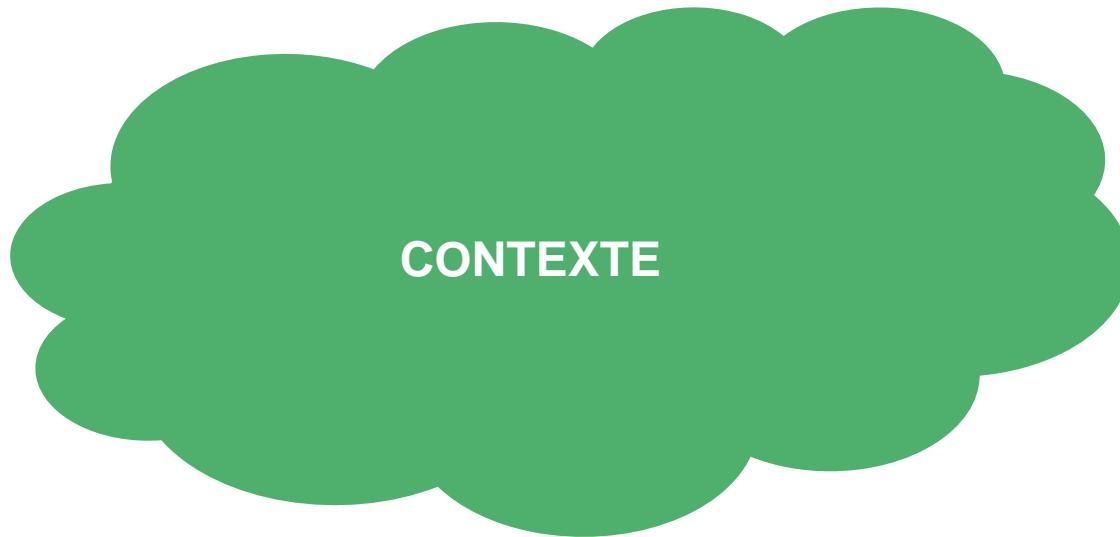
Compréhensible et univoque, sans ambiguïtés sur les concepts du domaine.

Partagé par tous

Ouvert aux nouveaux éléments (évolutif)

# Notion de Contexte

*Un cadre dans lequel un mot  
ou une phrase a un sens précis et sans ambiguïté*



Sens = Terme + Contexte

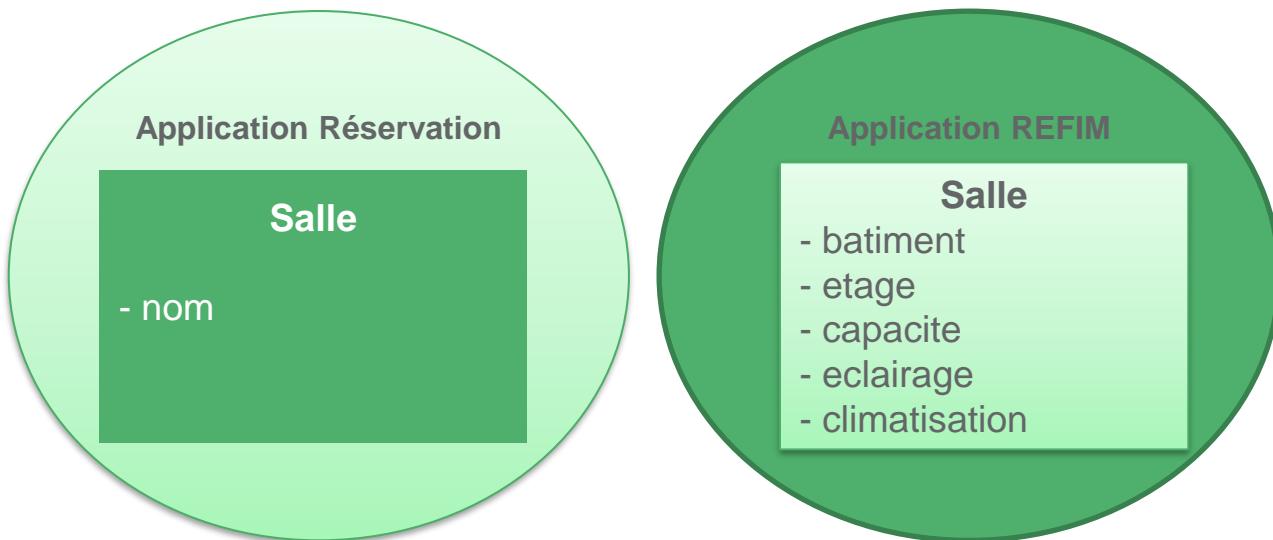


**BNP PARIBAS**

La banque d'un monde qui change

# Le contexte d'un modèle

- Chaque modèle s'applique dans un contexte
- Il s'agit d'un cadre dans lequel un concept métier (un mot ou une phrase) a une signification précise et sans ambiguïté
- Les contextes sont délimités par des **frontières linguistiques**
  - Ex: Le nom « Salle » n'a pas la même signification dans ces deux contextes:



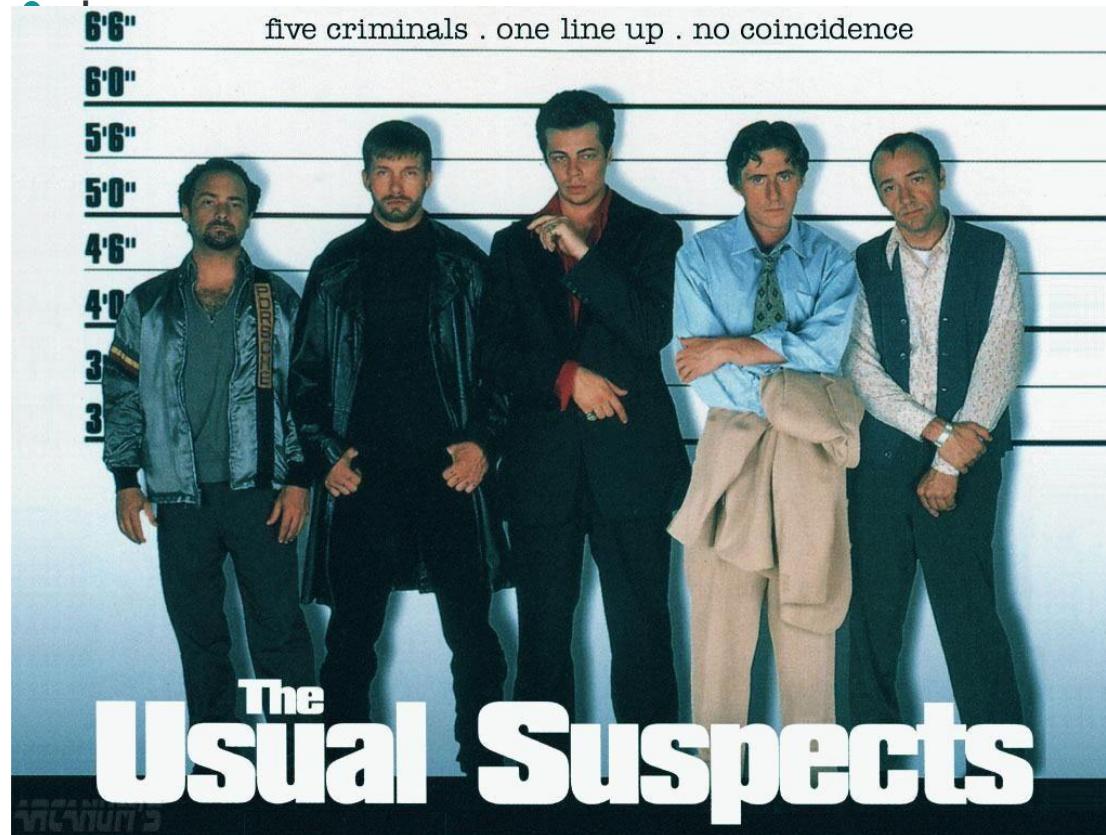
# Objectifs



- 1 Domaine métier et l'approche du DDD
- 2 Les différents modèles
- 3 Des problèmes de conception constatés**
- 4 Première approche de modélisation
- 5 Le contenu de la méthode DDD
- 6 Processus de modélisation préconisé



# Les principaux problèmes de conception



On constate toujours les mêmes problèmes de conception:

- Un code aveugle
- Un modèle anémique/squelettique
- Smart Client et ActiveRecord
- Une mauvaise utilisation de MVC
- Un langage pas complètement commun entre la MOE et le métier



BNP PARIBAS

La banque d'un monde qui change

# Un code aveugle

```
1  @Transactional
2  public void saveCustomer(
3      String customerId,
4      String customerFirstName, String customerLastName,
5      String streetAddress1, String streetAddress2,
6      String city, String stateOrProvince,
7      String postalCode, String country,
8      String homePhone, String mobilePhone,
9      String primaryEmailAddress, String secondaryEmailAddress) {
10
11     Customer customer = customerDao.readCustomer(customerId);
12
13     if (customer == null) {
14         customer = new Customer();
15         customer.setCustomerId(customerId);
16     }
17     if (customerFirstName != null) {
18         customer.setCustomerFirstName(customerFirstName);
19     }
20     if (customerLastName != null) {
21         customer.setCustomerLastName(customerLastName);
```

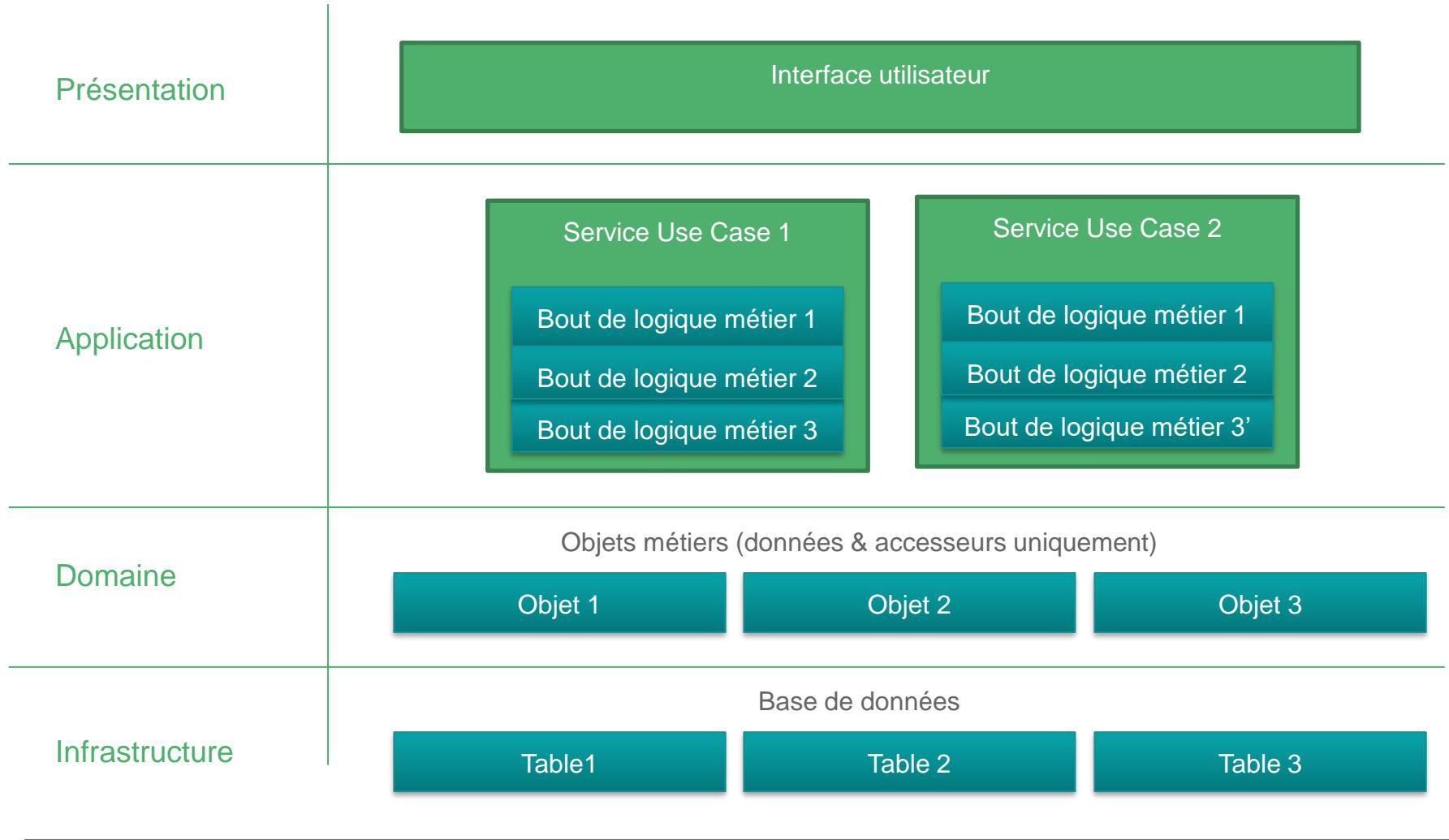


# Un modèle anémique

- Un modèle anémique se caractérise par des entités avec seulement des structures de données, sans comportement
- Un modèle anémique favorise:
  - La naissance de procédures (souvent dans un package "services")
  - Des contrôleurs épais (Transaction Script)
- Ainsi, le modèle anémique ne respecte pas les principes de la programmation **Orientée Objet**
  - Pas d'encapsulation
  - Pas de colocalisation des traitements et des données



# Illustration modèle anémique



# Les conséquences du modèle anémique

1. Dispersion et répétition de la logique métier dans le code de chaque cas d'utilisation (use case)
2. Code illisible n'exprimant pas l'intention mais seulement une mécanique (algorithme)
3. Transmission difficile ou impossible des connaissances, surtout dans un contexte de turn-over régulier
4. Absence de capitalisation sur la connaissance du modèle. Impossibilité d'élaborer un modèle riche et pérenne.
5. Perte de crédibilité et de confiance dans l'application, réputée "instable"



# Smart Client et ActiveRecord

- Les sujets de conception sont souvent résolu avec des anti-patterns comme:

## Smart Client

Inclut la logique métier dans la couche de présentation.

Mélange de préoccupations qui produit un code inextricable

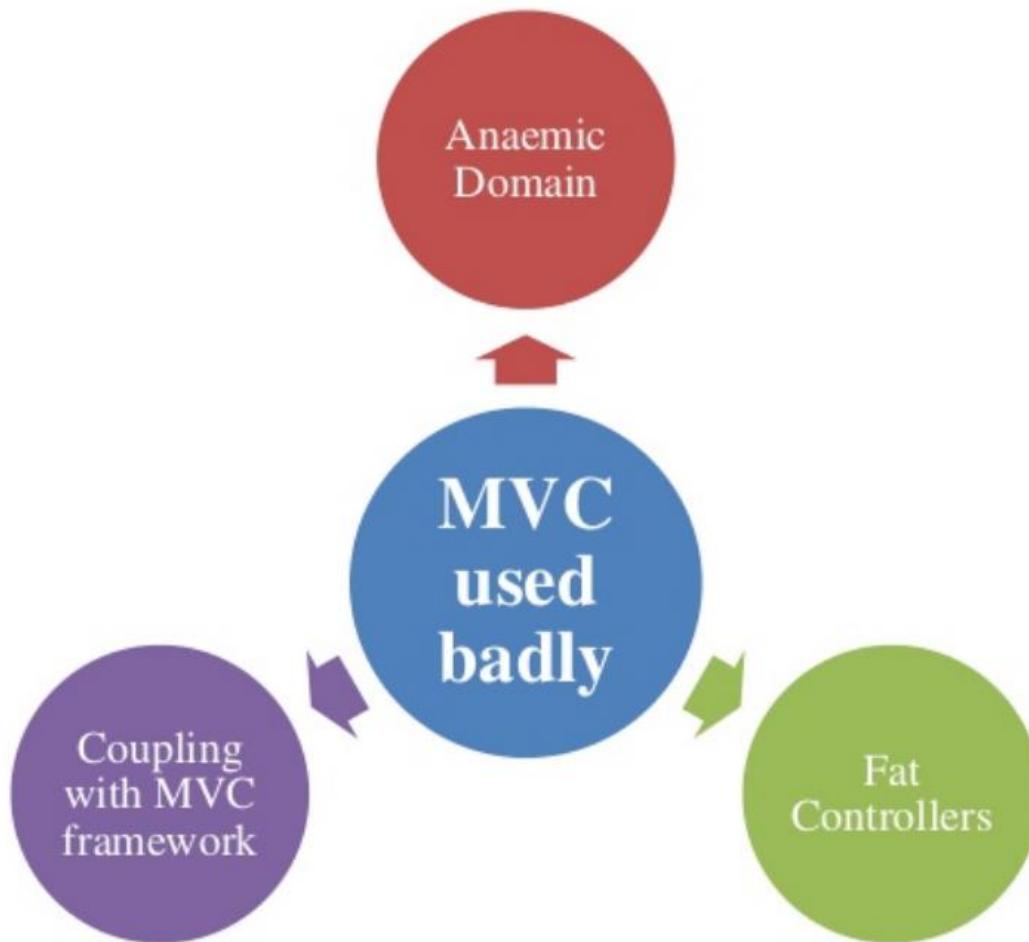
## Active Record

Modèle basé sur la donnée

Produit souvent une logique répétitive avec beaucoup de copier/coller



# Une mauvaise utilisation du MVC



*Crafted Design, Sandro Mancuso*



**BNP PARIBAS**

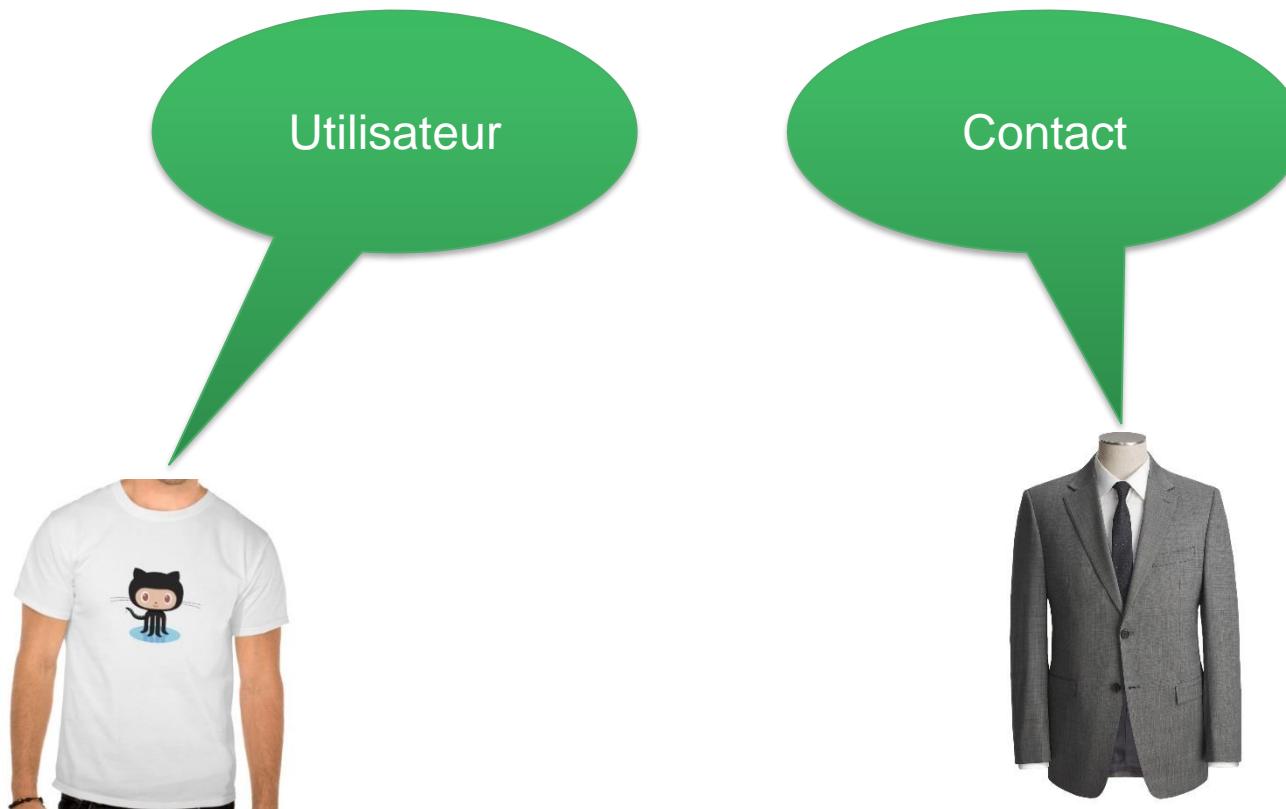
La banque d'un monde qui change

# Des applications CRUD

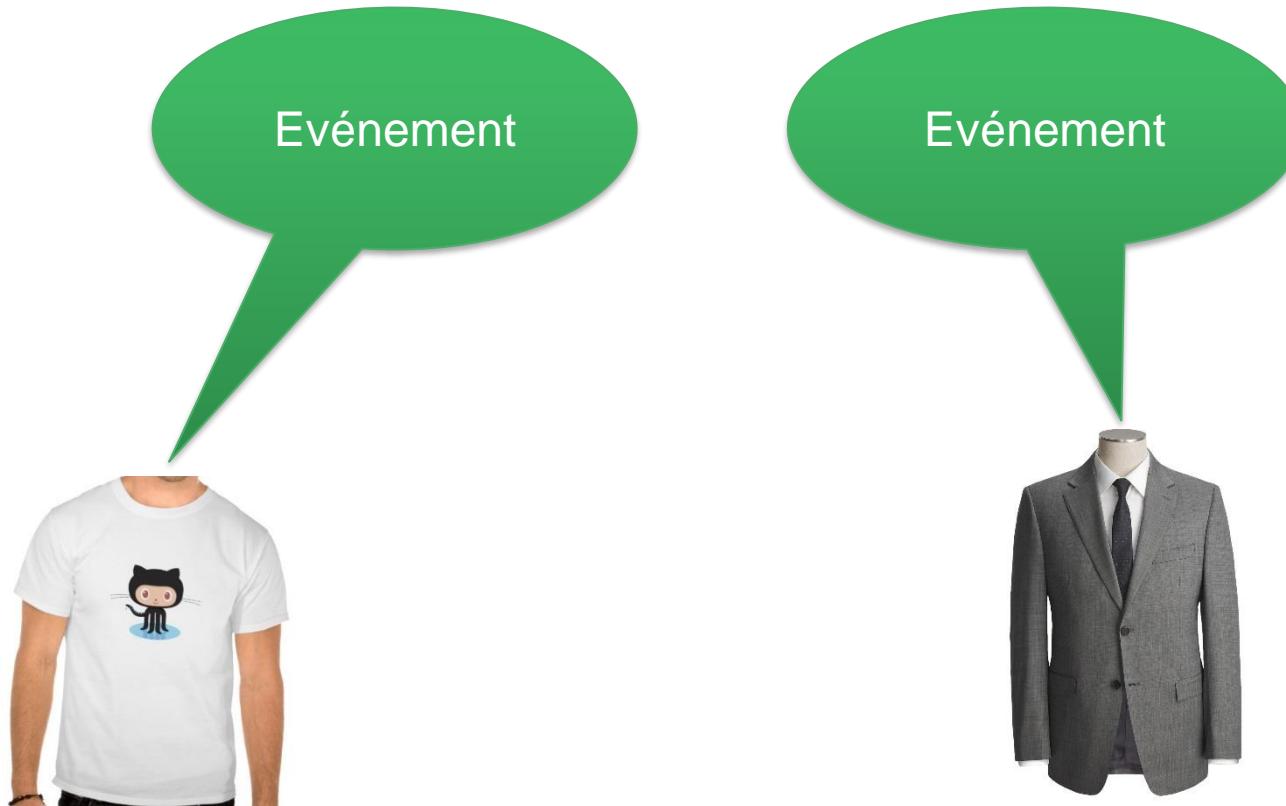
- Les applications CRUD sont des applications orientées sur l'orchestration des données, comportant peu ou pas de logique métier
  - Très similaire à un fichier CSV
  - On ne sait pas ce qui a été fait
  - On ne sait pas le pourquoi d'une application
  - Toute la logique métier **est déportée** au niveau de l'utilisateur
- L'existence des applications CRUD résultent parfois d'un manque de collaboration entre la MOE et la MOA, aboutissant à des applications qui cachent les intentions réelles



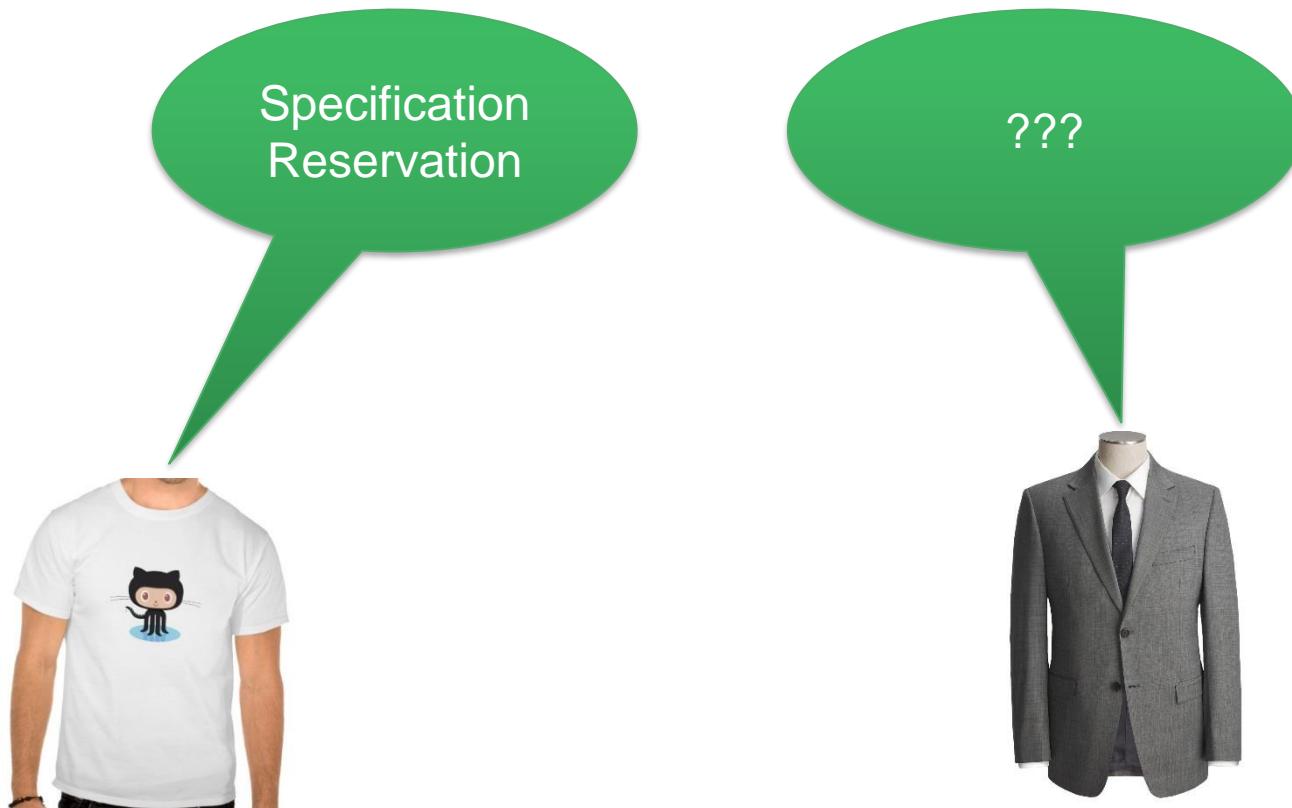
# Communication – Dictionnaire MOE-Métier - Terme différent



# Communication – Dictionnaire MOE-Métier – Même terme avec des significations différentes (des faux-amis)



# Communication – Dictionnaire MOE-Métier – Terme inconnu du métier



# Objectifs



- 1** Domaine métier et l'approche du DDD
- 2** Les différents modèles
- 3** Des problèmes de conception constatés
- 4** **Première approche de modélisation**
- 5** Le contenu de la méthode DDD
- 6** Processus de modélisation préconisé



# Domaine « Réservation de salles »

**Objectif:** Un participant souhaite réserver une salle sur une plage horaire

- La salle est identifiée par son nom
- L'utilisateur effectue une action de réservation

L'ensemble des salles existantes sont fournies par un référentiel immobilier



# Réservation salle - Processus de modélisation constaté

## (1/7)

- Identification des noms communs de l'application

Reservation

Salle



**BNP PARIBAS**

La banque d'un monde qui change

# Réservation salle - Processus de modélisation constaté (2/7)

- Ajout de propriétés à chaque concept

## Reservation

-date  
-heureDebut  
-heureFin

## Salle

-nom



# Réservation salle - Processus de modélisation constaté

## (3/7)

- Identification des actions (verbes)

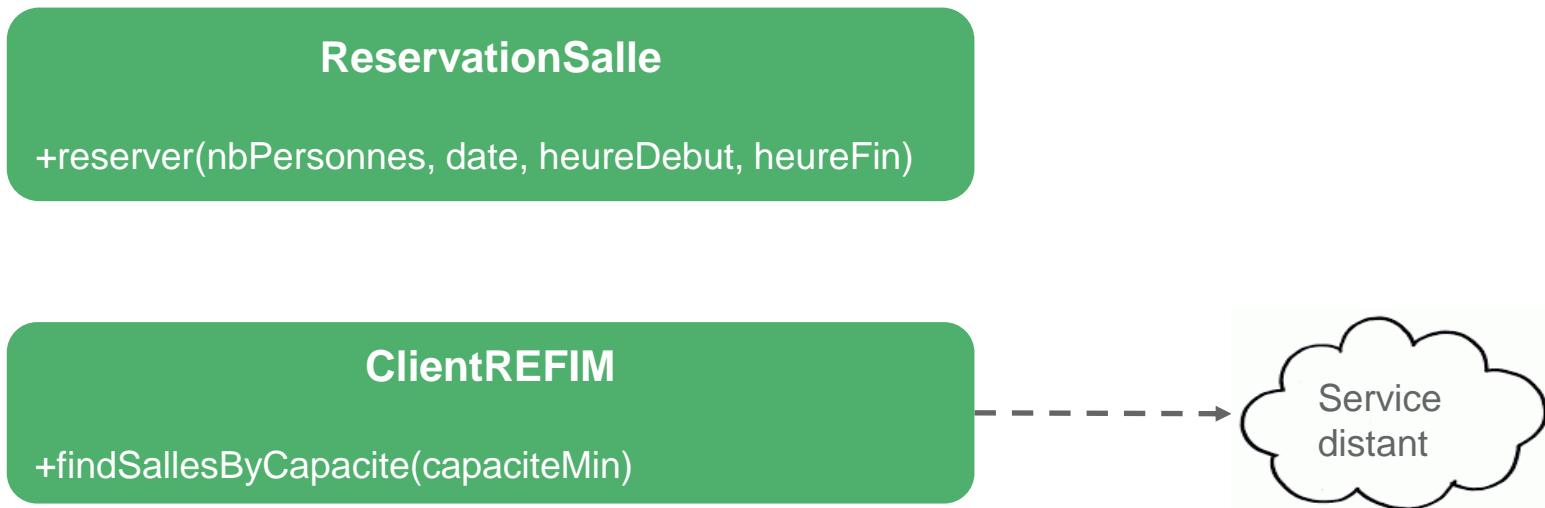
`reserverSalle()`

`findSalles()`



# Réservation salle - Processus de modélisation constaté (4/7)

- Identification du lieu où placer l'action (verbe): un "Service"
- Identification du "Service" technique nécessaire à l'action



# Réservation salle - Processus de modélisation constaté (5/7)

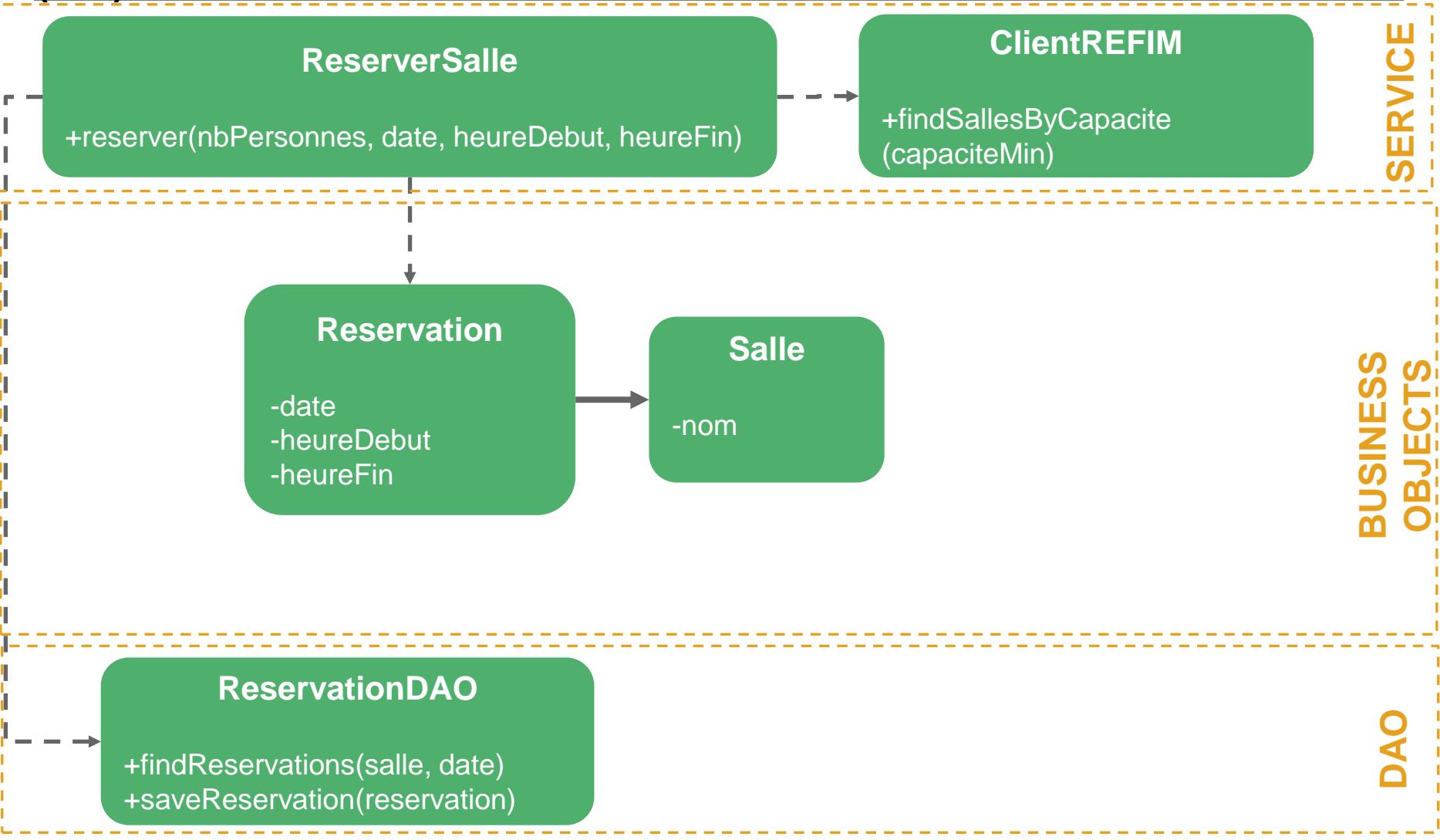
- Création des DAOs pour accéder aux tables

## ReservationDAO

```
+findReservations(salle, date)  
+saveReservation(reservation)
```



# Réservation salle - Processus de modélisation constaté (6/7)

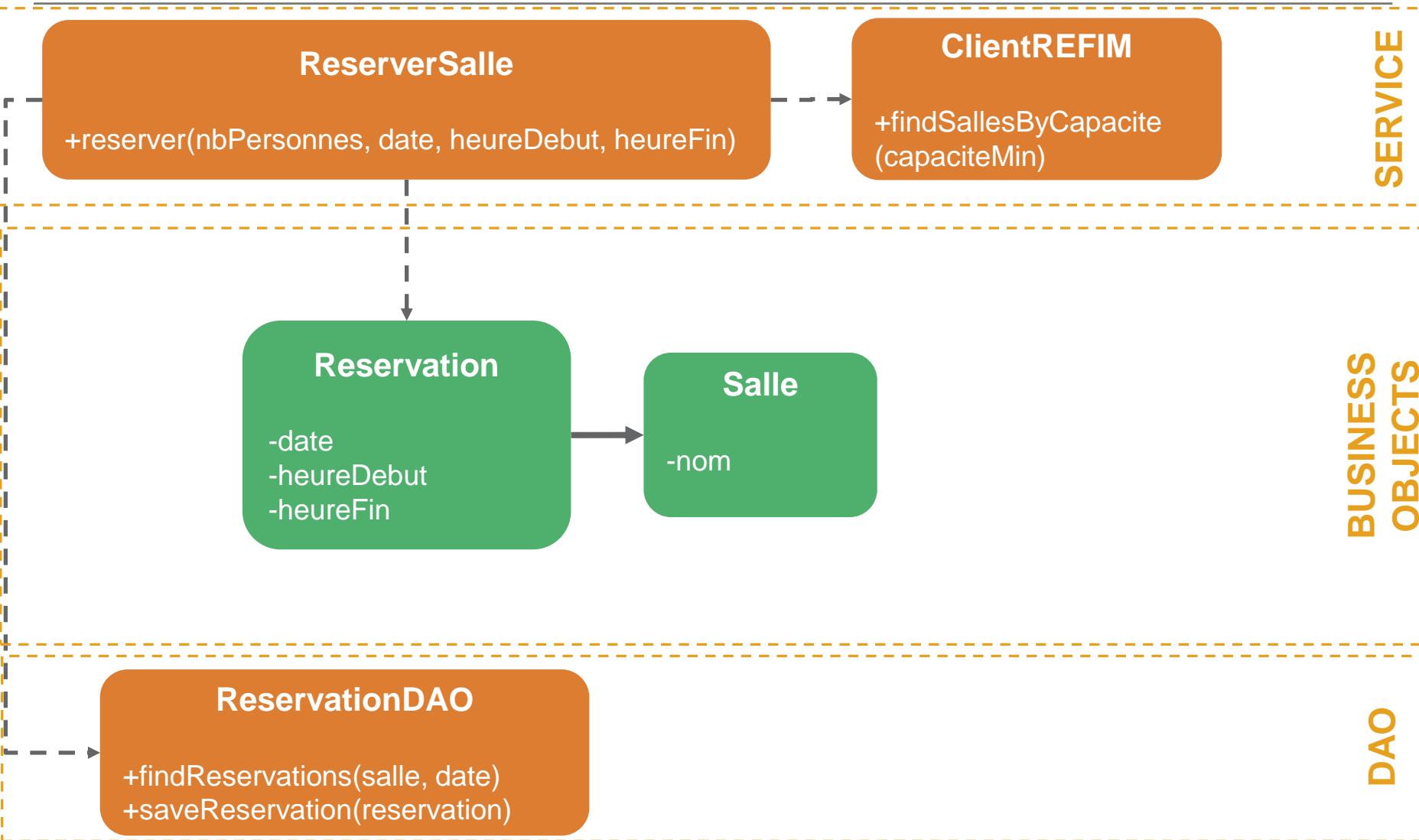


# Réservation salle – Application de la colorisation (7/7)

SERVICE

BUSINESS  
OBJECTS

DAO



**BNP PARIBAS**

La banque d'un monde qui change

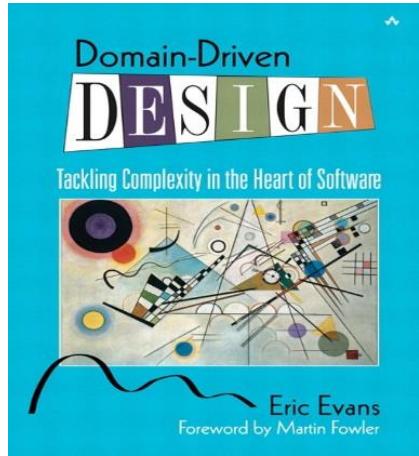
# Objectifs



- 1** Domaine métier et l'approche du DDD
- 2** Les différents modèles
- 3** Des problèmes de conception constatés
- 4** Première approche de modélisation
- 5** **Le contenu de la méthode DDD**
- 6** Processus de modélisation préconisé



# Complexité du domaine et goulot d'étranglement



*« Yet the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not handled in the design, it won't matter that the infrastructural technology is well conceived. A successful design must systematically deal with this central aspect of the software. »*



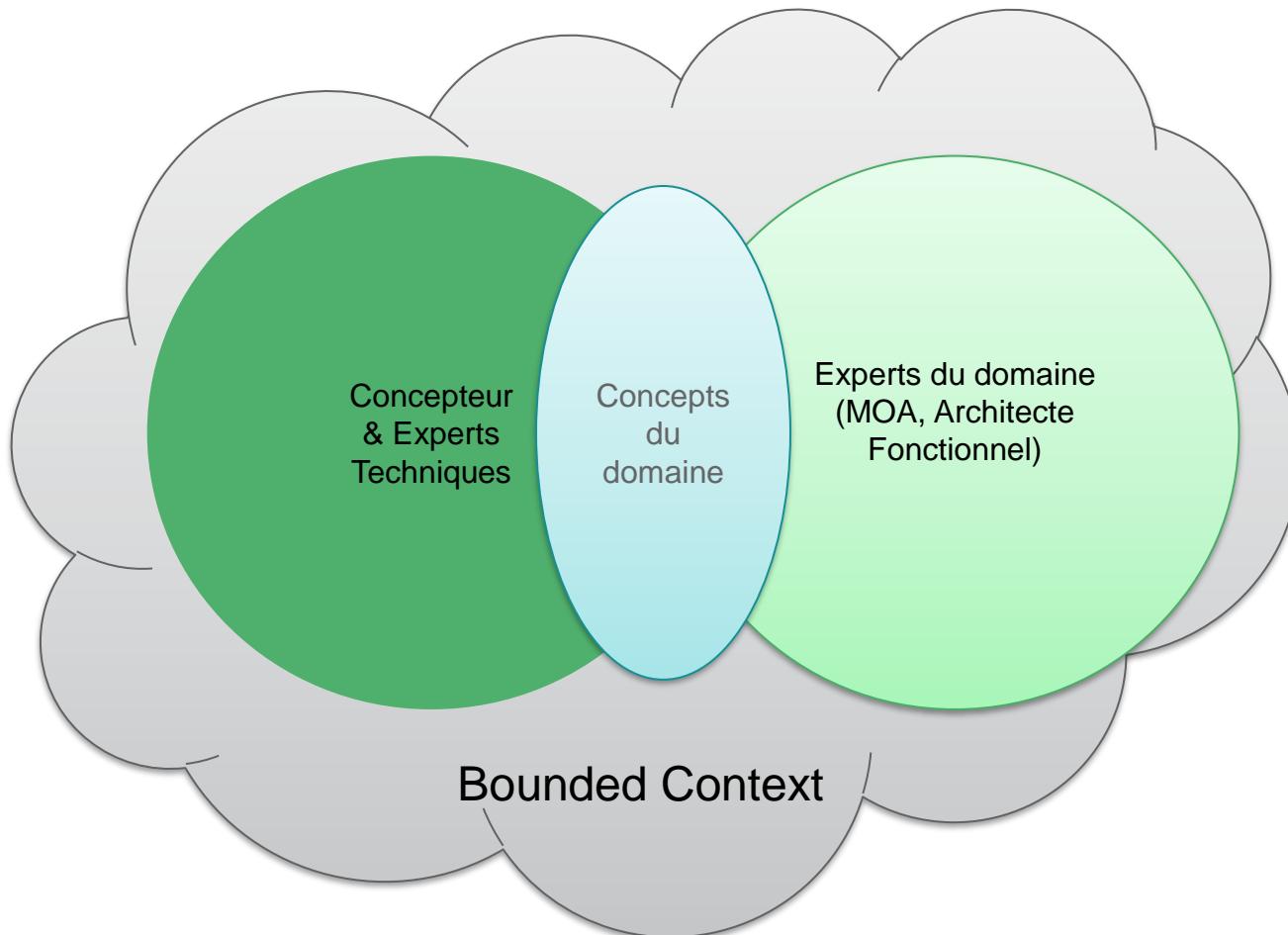
# 1. Le modèle du domaine (Domain Model) comme noyau du logiciel

- Lier le code à un modèle donne de la **signification** au code
  - Le modèle utilisé par le logiciel devient un **modèle pertinent**



- Placé au noyau de l'Architecture logicielle
  - Ne dépend pas des préoccupations techniques: lui évite de dépendre des détails
- Un modèle se construit par itération
  - Plusieurs cycles de refactoring sont à prévoir entre les développeurs et les experts métiers
  - Chaque cycle peut faire émerger de nouveaux concepts auparavant *implicites*

## 2. Ubiquitous Language



**Convergence de concepts métier entre deux catégories de population**



**BNP PARIBAS**

La banque d'un monde qui change

# Bounded Context (BC)

***Un BC est un contexte délimité par une frontière linguistique nette. A l'intérieur d'un BC, tous les concepts du modèle (noms et verbes) ont une signification spécifique, comprise et acceptée par tous les membres de l'équipe.***

On détermine donc les limites d'application d'un modèle

Les membres de l'équipe ont une vision précise des éléments qui doivent être consistants

Dans un BC, le modèle du domaine est exprimé par un ***Ubiquitous Language*** partagé par tous

Un domaine métier peut contenir plusieurs BC, chaque BC définissant son Ubiquitous Language



# Avantages des Bounded Context

Renforce la pureté et la puissance du modèle

Evite la confusion avec les autres contextes

Simplifie l'Architecture  
(Fait émerger des transitions explicites entre les différents contextes)

Favorise un partitionnement technique et organisationnel



# L'ubiquitous Language: l'aboutissement de la conception

- Désigne un langage commun centré autour du modèle du domaine métier, et partagé par les concepteurs/experts techniques avec les experts du domaine métier
- Si une idée ne peut pas être exprimée à travers l'ensemble des concepts métiers définis, on retourne en arrière et on étend le modèle
  - Peut être enrichi au fur et à mesure que des concepts implicites sont découverts
- Avantages
  - Diminution des risques de mauvaises communication, lève les ambiguïtés
  - Améliore la qualité des exigences quand celles-ci ne sont exprimées qu'avec ce vocabulaire
  - Un code source plus simple ET auto-documenté



### 3. Les patterns DDD Tactiques et Stratégiques

#### Tactique

Principes et briques de conception à l'intérieur d'un Bounded Context

#### Stratégique

Coordination et intégration de plusieurs Bounded Contexts



# Des cas d'utilisation de DDD

- Le DDD n'est pas toujours applicable mais de nombreuses situations s'y prêtent :

Cas d'utilisation	Commentaire
Un domaine métier complexe	La complexité d'une application est souvent sous-estimée
Une application avec une durée de vie importante	Pour des applications de 3, 5 ou même 10 ans, il est nécessaire d'avoir un niveau de maintenabilité important. Un des critères de la maintenabilité est un code exprimant des intentions
Un souhait de renforcer la communication entre les experts du domaine et les développeurs	Un manque de communication entraîne parfois des suppositions des développeurs et ainsi des applications ne répondant pas complètement aux besoins des utilisateurs



# Objectifs



- 1 Domaine métier et l'approche du DDD
- 2 Les différents modèles
- 3 Des problèmes de conception constatés
- 4 Première approche de modélisation
- 5 Le contenu de la méthode DDD
- 6 Processus de modélisation préconisé**



# Processus de modélisation - Préconisation

## Etape 1

On modélise le métier indépendamment de toutes contraintes.  
*« les différents éléments du langage »*

## Etape 2

On valide le modèle en appliquant les cas d'utilisations.  
Le modèle est considéré comme bon quand les cas d'utilisations sont simples  
(évident) à exprimer sur la base du modèle

## Etape 3 ou 4

On prend en compte les contraintes informatiques et on intègre l'application dans le  
système d'information  
(ex: *Conception de la base de données*)

## Etape 3 ou 4

On habille les services avec des écrans





# PRÉREQUIS DDD

## FONDAMENTAUX DE LA PROGRAMMATION OBJET



# Objectifs



## 1 Les fondamentaux de la programmation objet

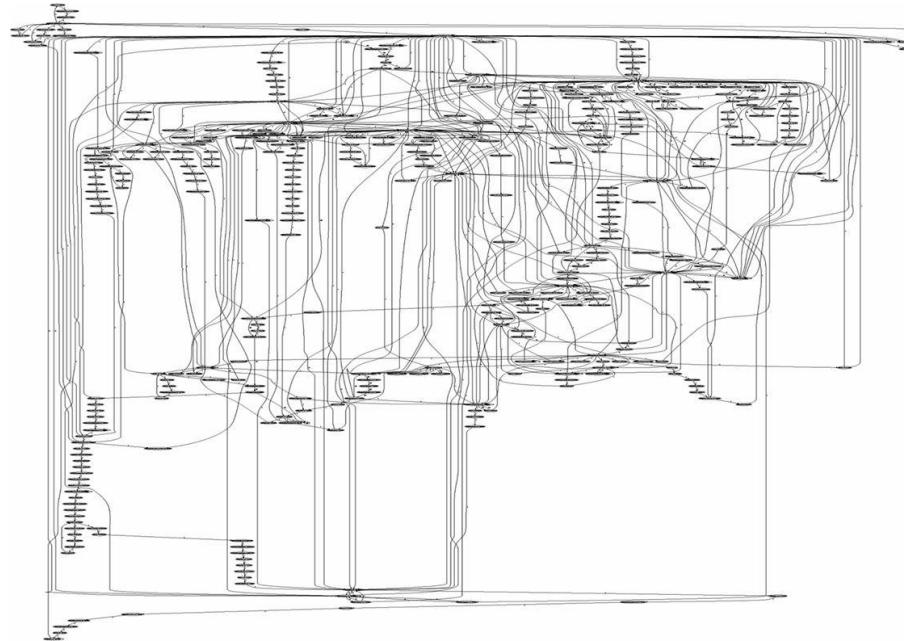
- ① Cohésion et couplage
- ② Principes SOLID
- ③ Focus sur DIP, IOC et DI
- ④ Design Patterns

## 2 Autres Principes Indispensables

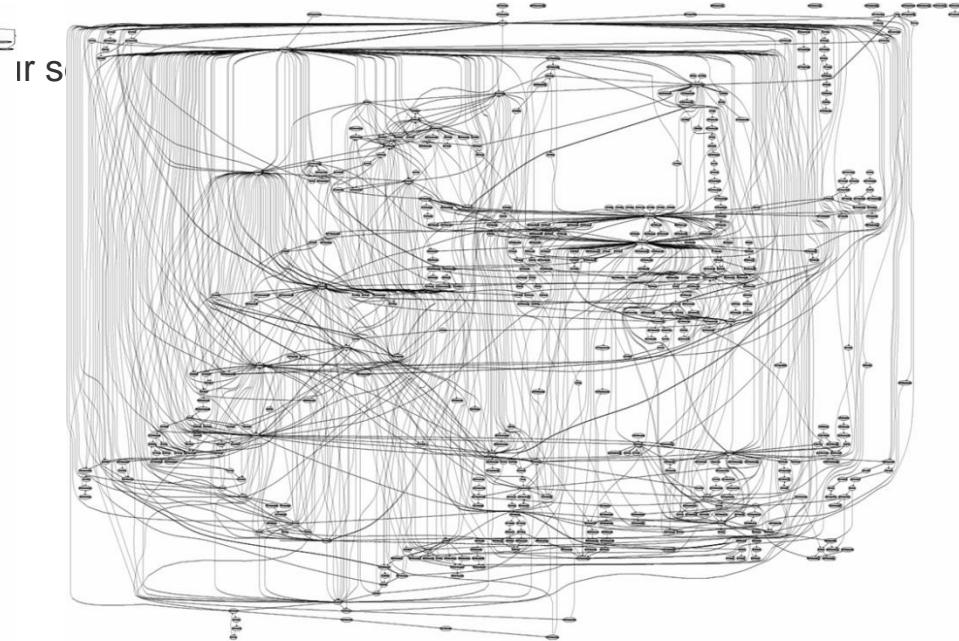
- ① Yagni
- ② Separated Interface
- ③ Dette Technique



# Cohésion et couplage



Apache



IIS

Quelle situation est la plus saine?

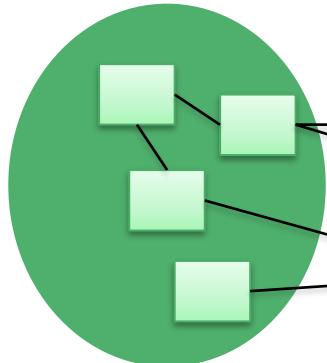


**BNP PARIBAS**

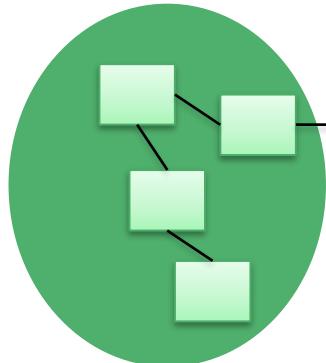
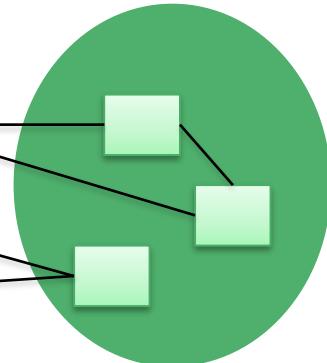
La banque d'un monde qui change

# Cohésion et couplage

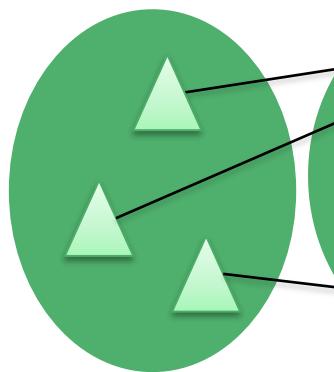
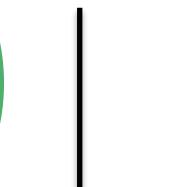
Une bonne conception est un système avec un faible couplage et une forte cohésion



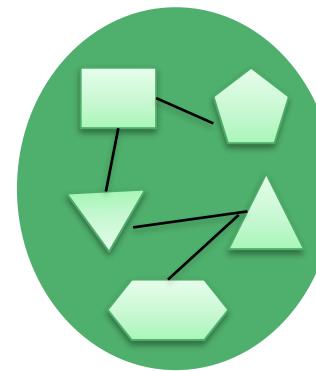
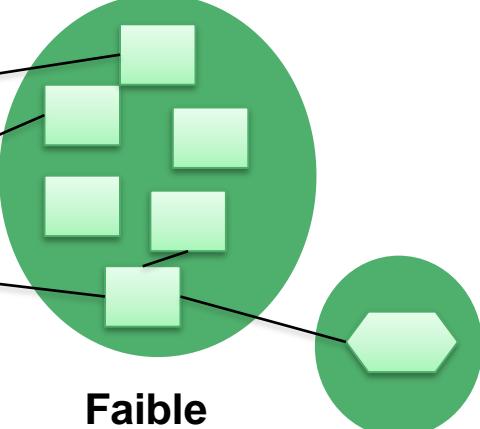
**Fort Couplage**



**Faible Couplage**



**Faible  
Cohésion**

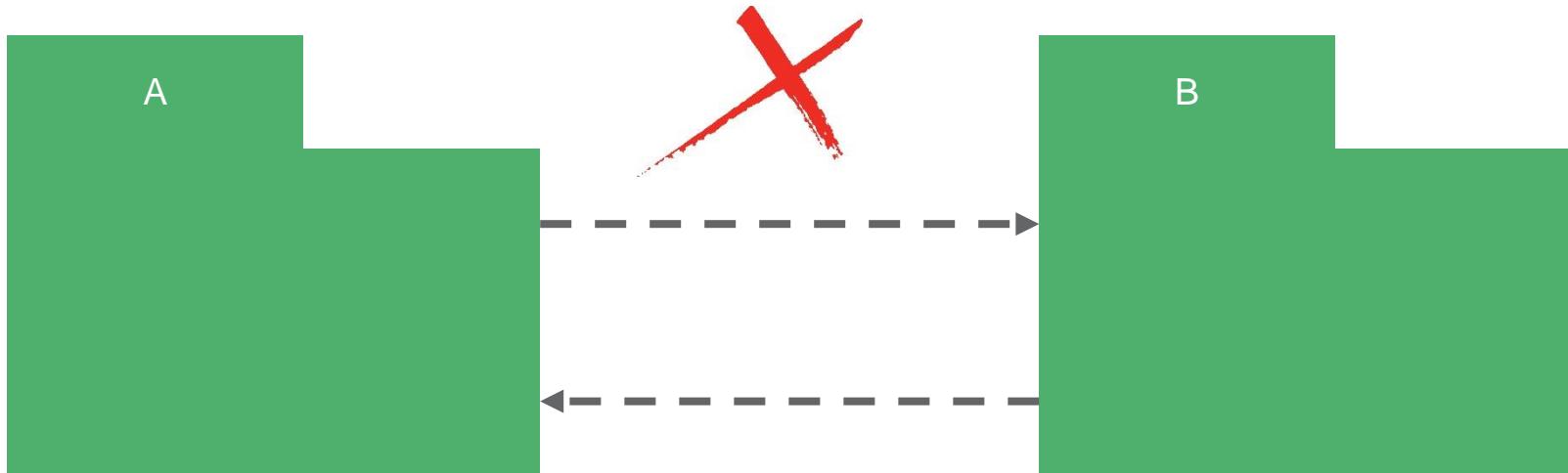


**Forte  
Cohésion**



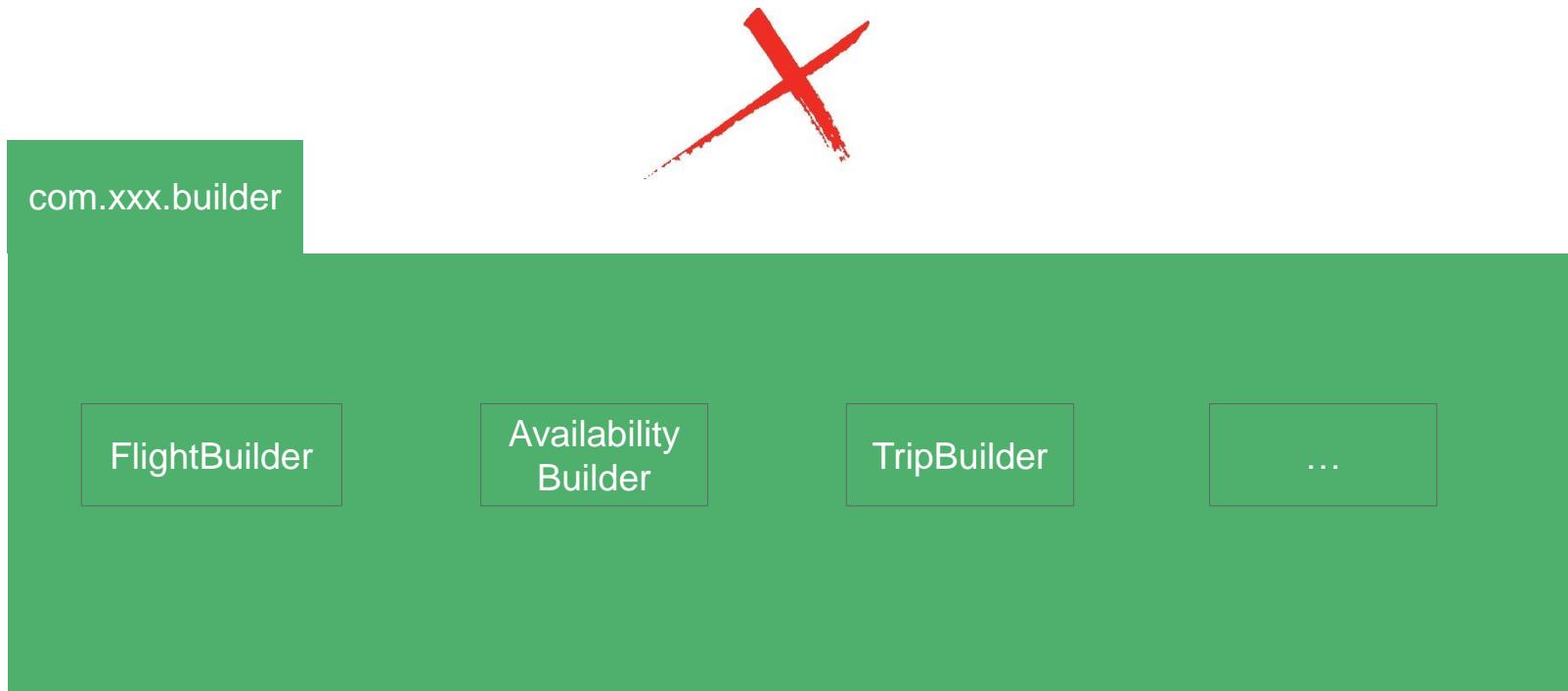
# Cohésion et couplage

- Couplage circulaire
  - Signifie que les deux modules ne sont qu'un
  - Augmente la fragilité et réduit l'agilité
  - Se dégrade rapidement en Big Ball Of Mud
  - A éliminer



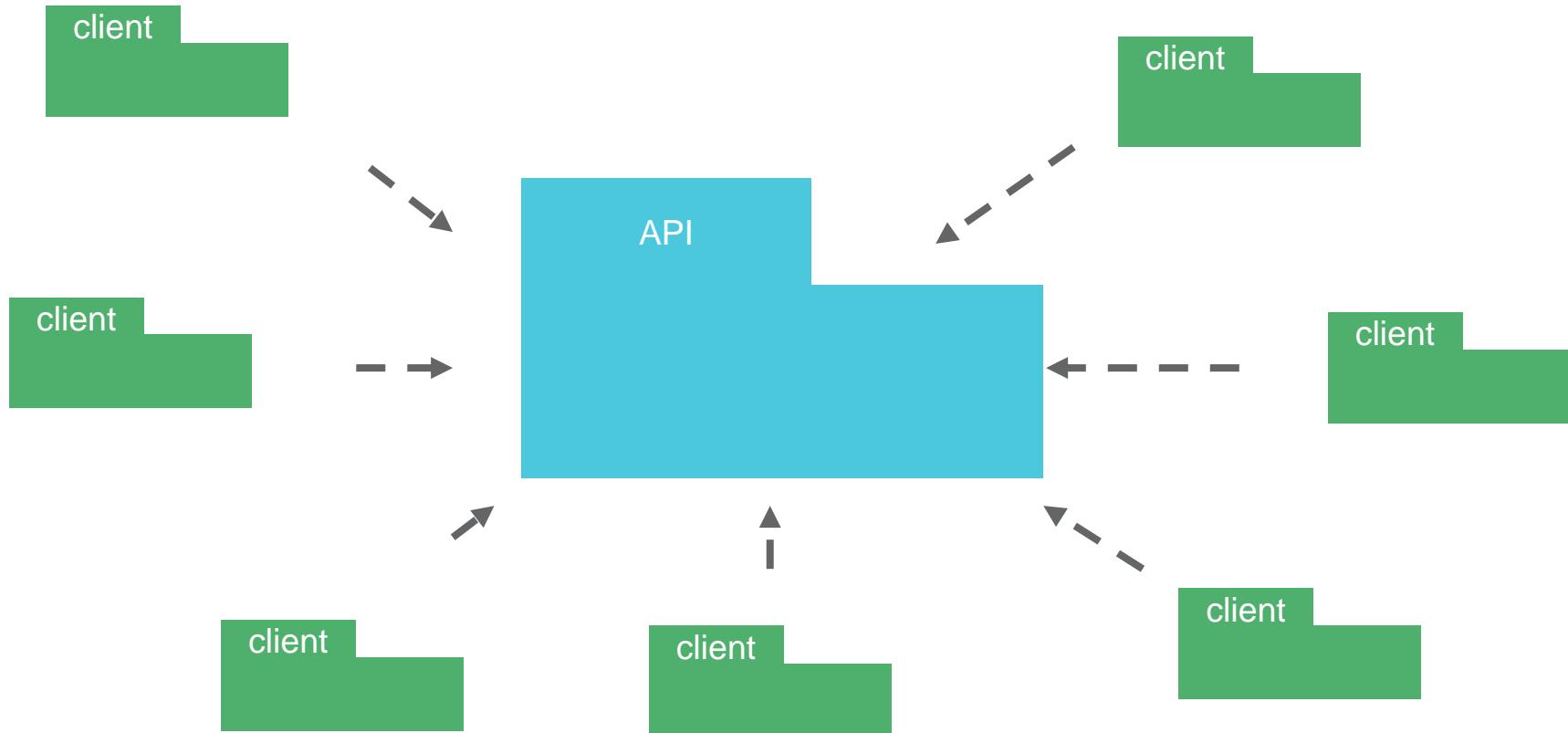
# Couplage et cohésion

- Faible cohésion
  - Très fréquent dans le code «vu dans la nature»
  - Conséquence naturelle des packages techniques au lieu de métier



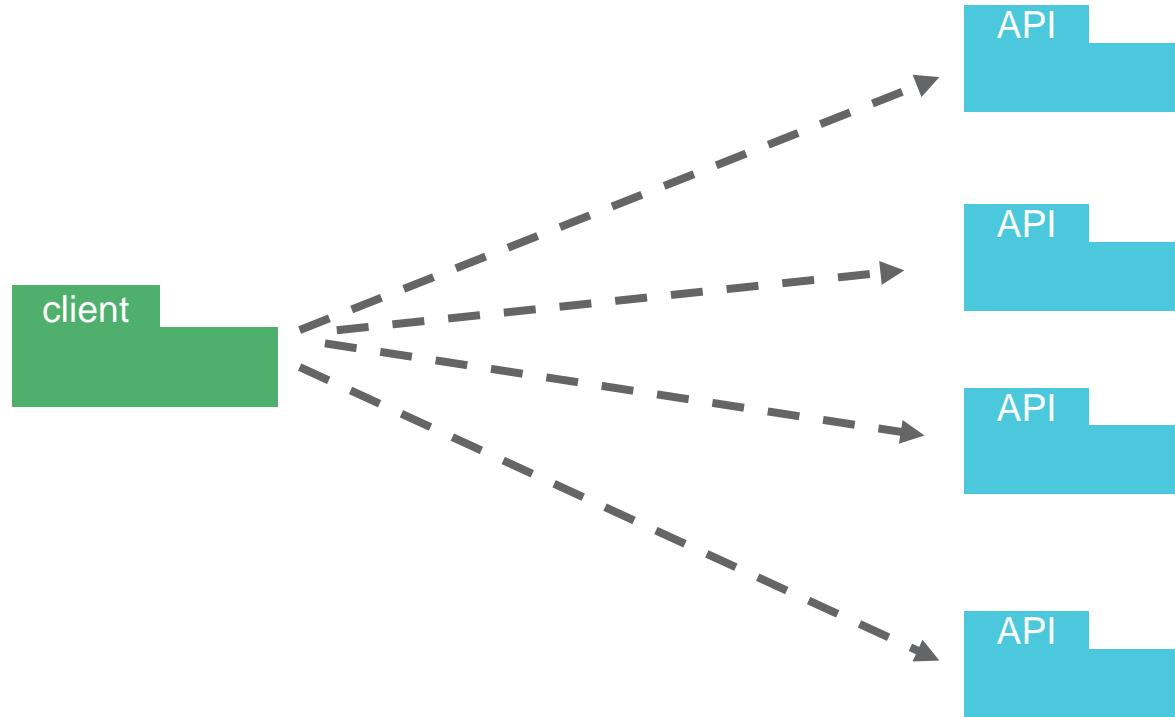
# Stabilité

Un module utilisé par beaucoup d'autres modules doit être stable: c'est la notion de responsabilité (s'il change, il risque d'impacter beaucoup d'autres modules)



# Instabilité

Un module qui dépend de beaucoup d'autres modules est instable: il peut être impacté par les changements de tous les modules dont il dépend

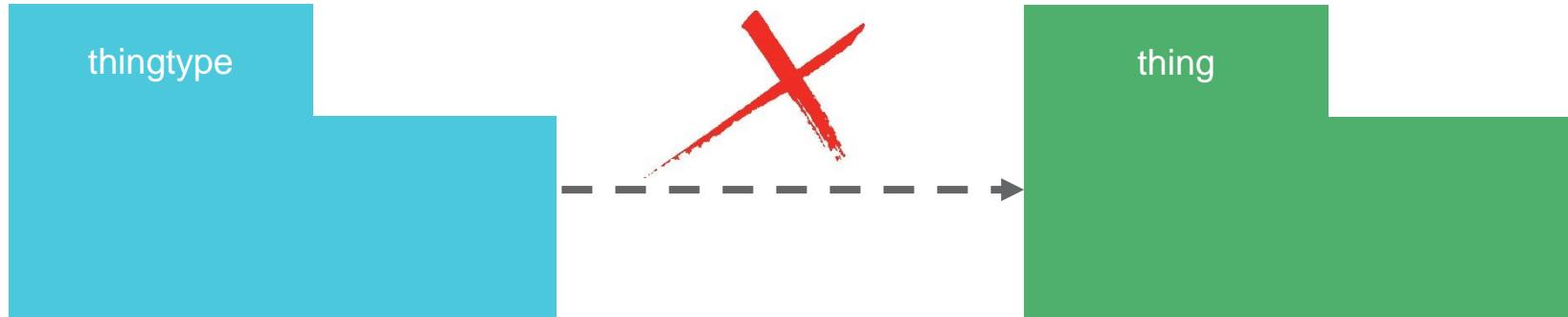


# Couplage et cohésion

Un module stable ne doit pas dépendre d'un module instable.

Autrement dit: les flèches de dépendances doivent aller de instable vers stable

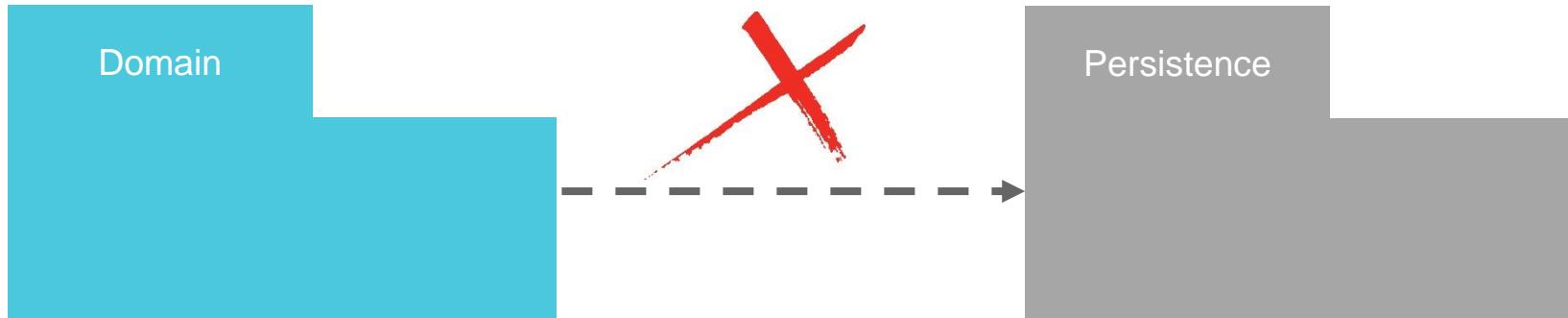
- Des outils tels que JDepend révèlent les erreurs de conception



# Couplage et cohésion

Abstraction et stabilité sont corrélées

- Donc un module abstrait ne doit pas dépendre d'un module concret
- Les généralités ne doivent pas dépendre des détails d'implémentation



# Objectifs



## 1 Les fondamentaux de la programmation objet

- ① Cohésion et couplage
- ② **Principes SOLID**
- ③ Focus sur DIP, IOC et DI
- ④ Design Patterns

## 2 Autres Principes Indispensables

- ① Yagni
- ② Separated Interface
- ③ Dette Technique



# Les principes SOLID

S

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

O

OPEN/CLOSED PRINCIPLE (OCP)

L

LISKOV SUBSTITUTION PRINCIPLE (LSP)

I

INTERFACE SEGRATION PRINCIPLE (ISP)

D

DEPENDENCY INVERSION PRINCIPLE (DIP)

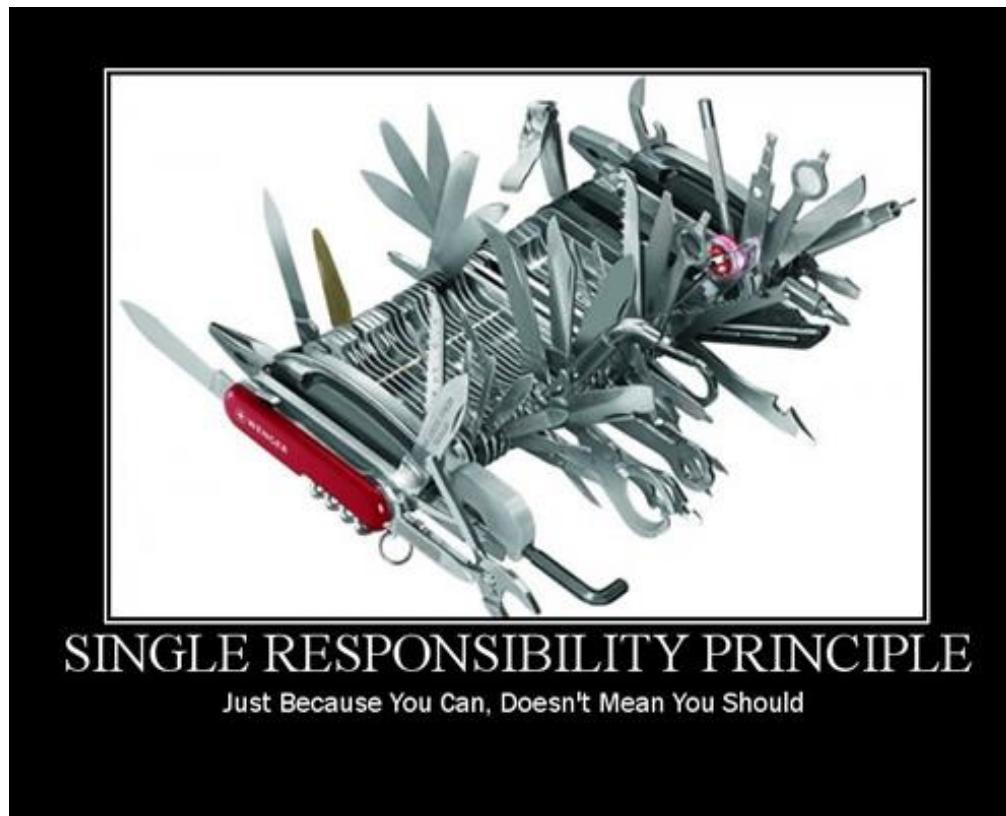


BNP PARIBAS

La banque d'un monde qui change

# Single Responsibility Principle (SRP)

*Une classe (ou module) ne doit avoir qu'une seule raison de changer, donc une seule responsabilité*



# Single Responsibility Principle (SRP)

Si ce principe n'est pas respecté:

1. Un changement d'une des responsabilités peut entraîner un changement sur les autres
  - A chaque responsabilité correspond un *axe de changement*
  - Ces axes doivent être orthogonaux
2. D'autre part, chaque responsabilité provoque des dépendances
  - Coupler les responsabilités induit donc un couplage des dépendances
  - La classe a besoin d'un « couple magique » de dépendances  
→ Elle peut cesser de fonctionner si l'une monte de version





# Single Responsibility Principle (SRP)

## Rectangle

- +double perimeter()
- +double area()
- +void draw()



**BNP PARIBAS**

La banque d'un monde qui change

Prérequis 68

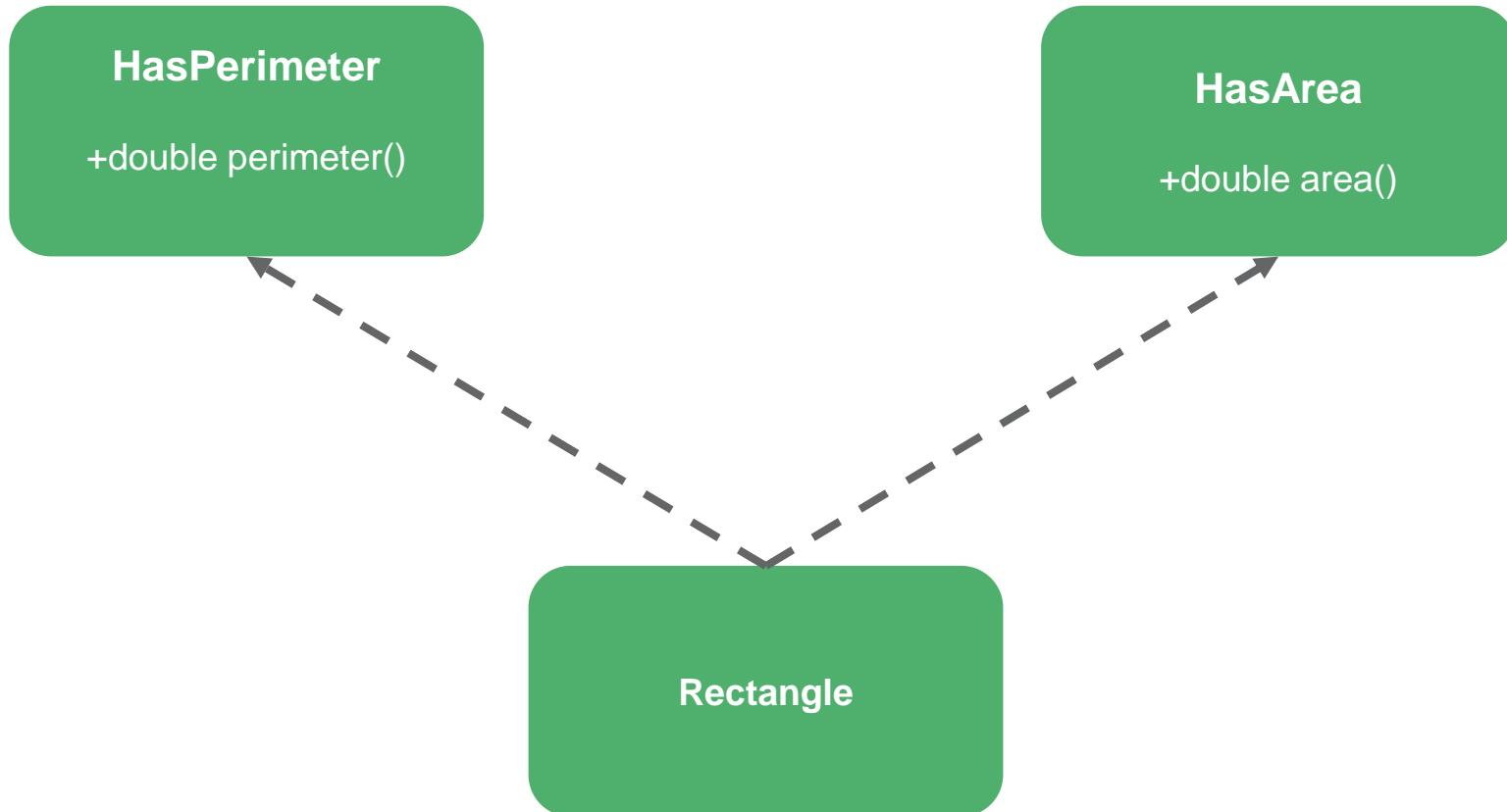


# Single Responsibility Principle (SRP)





# Single Responsibility Principle (SRP)



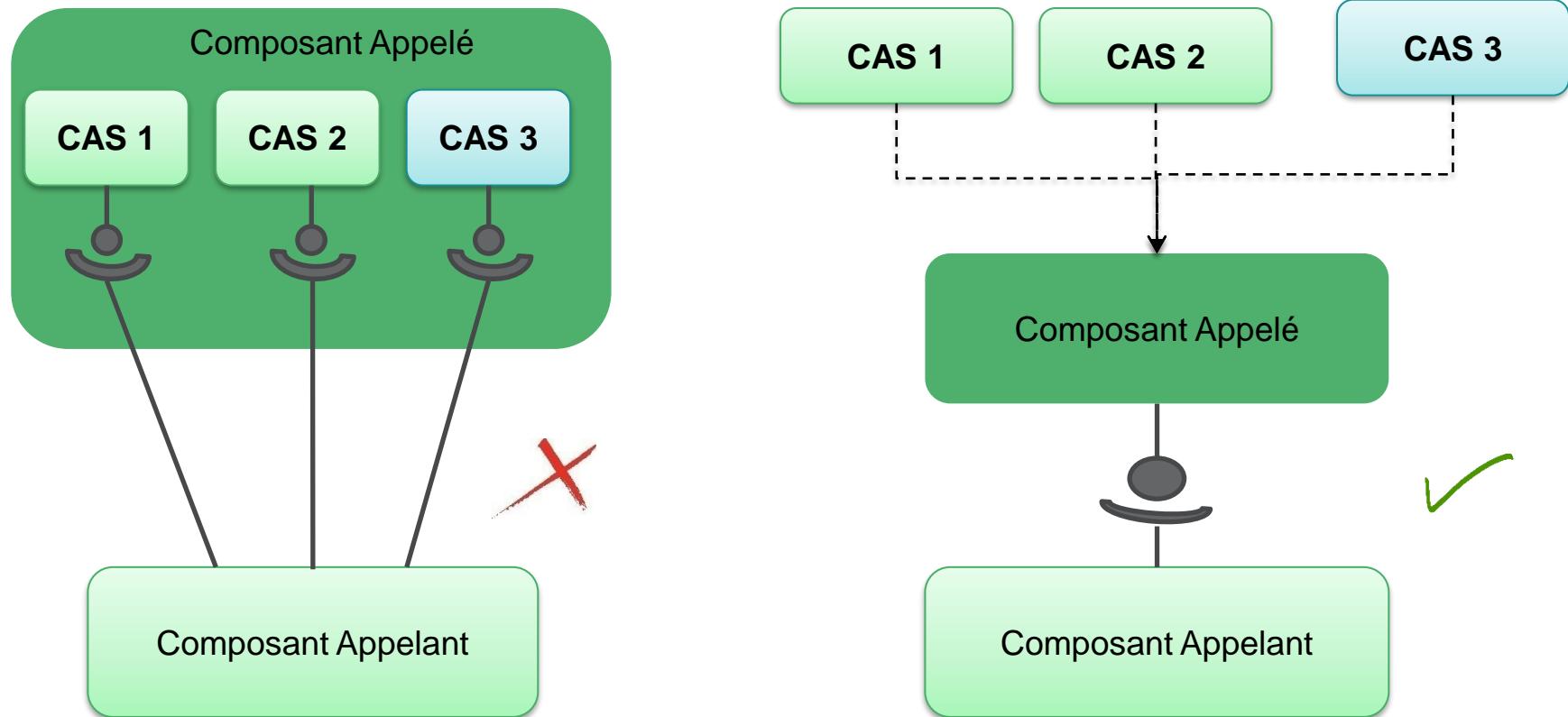
**BNP PARIBAS**

La banque d'un monde qui change

Prérequis 70

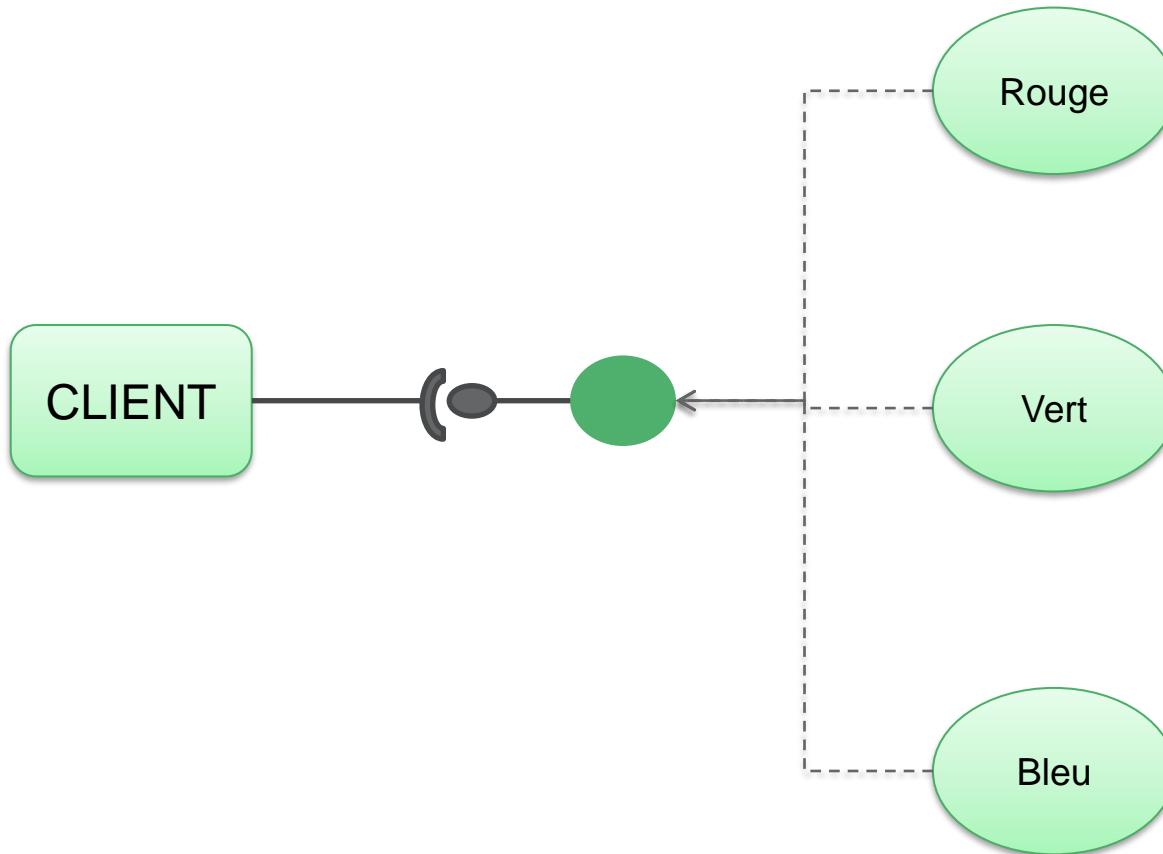
# Open/Closed Principle

***Les composants doivent être ouverts à l'extension,  
mais fermés à la modification***

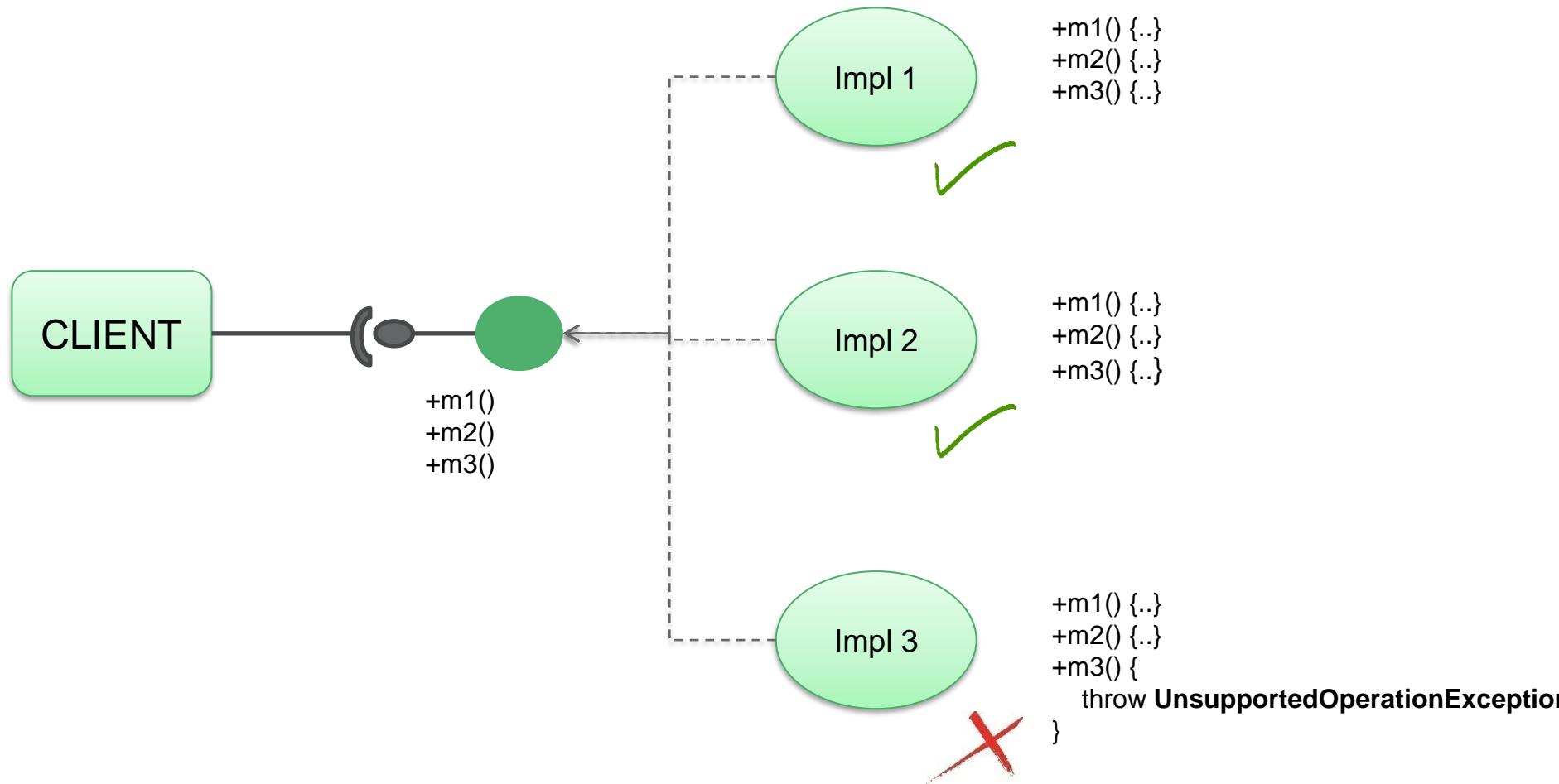


# Liskov substitution Principle (1/2)

Les sous-types doivent respecter le contrat du super-type



# Liskov substitution Principle (2/2)



# Interface Segregation Principle

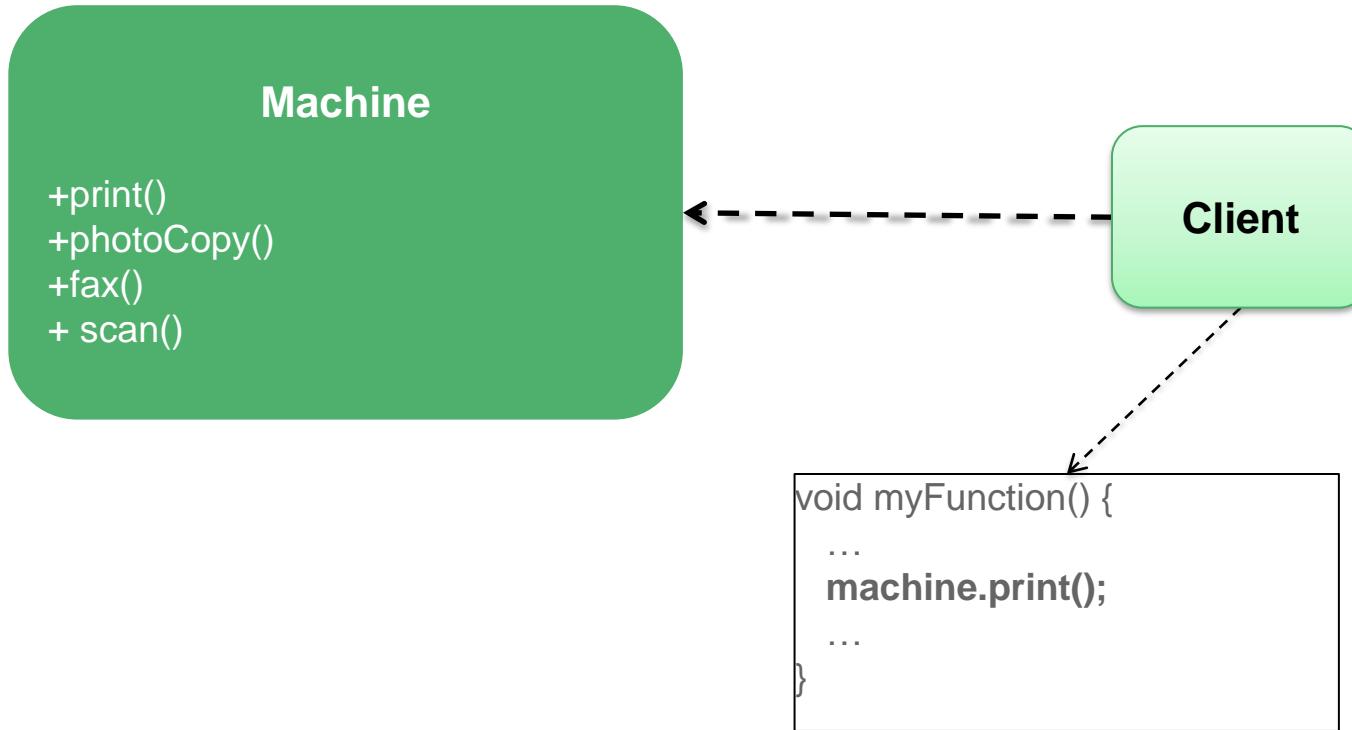
***Les clients d'une API ne doivent pas être forcés de dépendre de méthodes qu'ils n'utilisent pas***

Si ce principe n'est pas respecté:

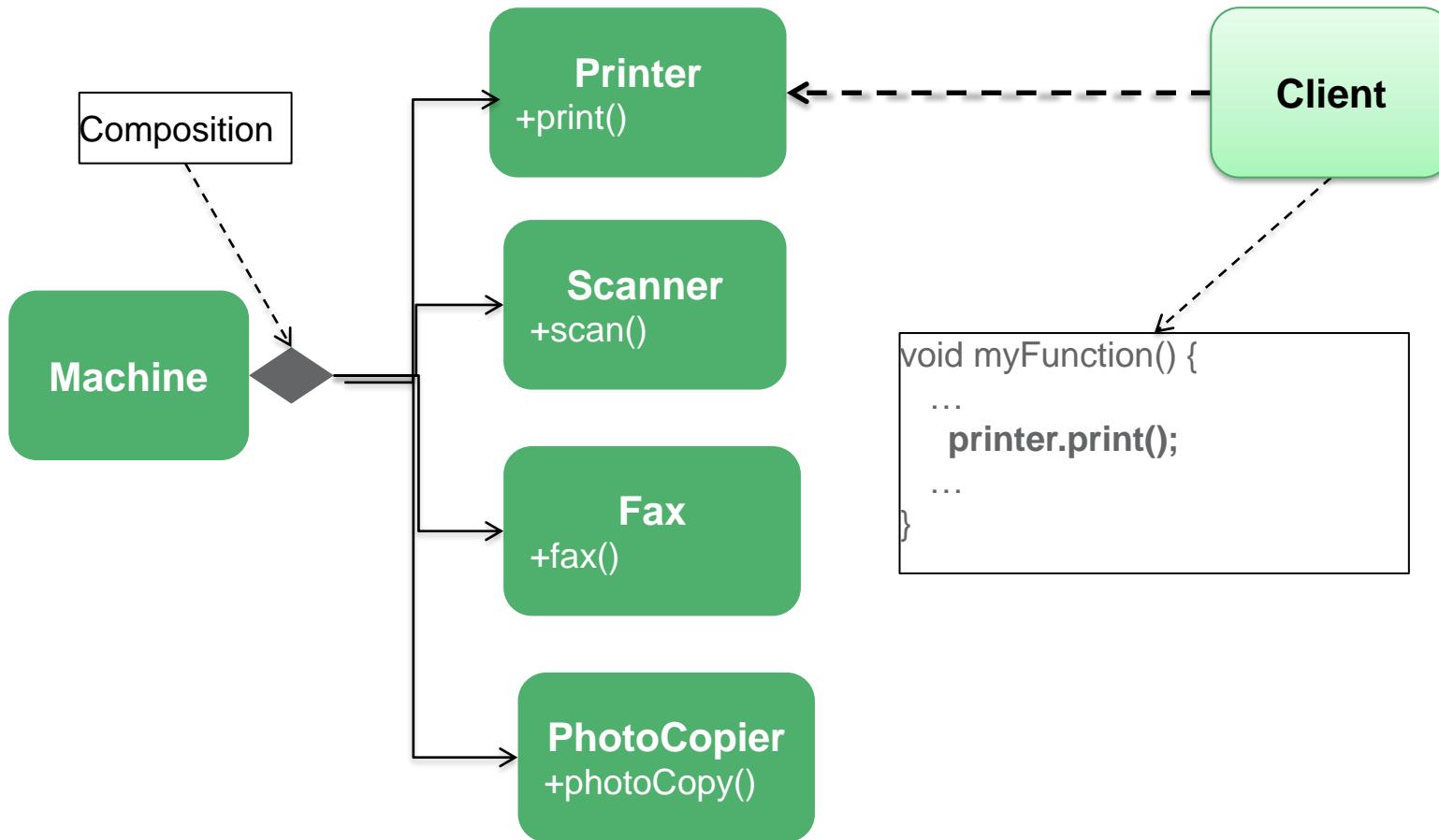
- Les clients d'un service fourni par l'interface pourront être impactés par un changement sur un autre service, alors qu'ils ne l'utilisent pas
- Le contrat de l'interface leur paraîtra confus



# Interface Segregation Principle (ISP)



# Interface Segregation Principle (ISP)



# Dependency inversion Principle

*Les abstractions ne doivent pas dépendre de détails d'implémentation.  
Les détails d'implémentation doivent au contraire dépendre  
d'abstractions.*

Si ce principe n'est pas respecté:

1. Si les abstractions dépendent de détails: les abstractions dépendent de modules instables, et sont donc instables. Tout le code est alors instable, donc fragile et rigide.
2. Si les détails ne dépendent pas d'abstractions, le modèle sera anémique.



# Objectifs



## 1 Les fondamentaux de la programmation objet

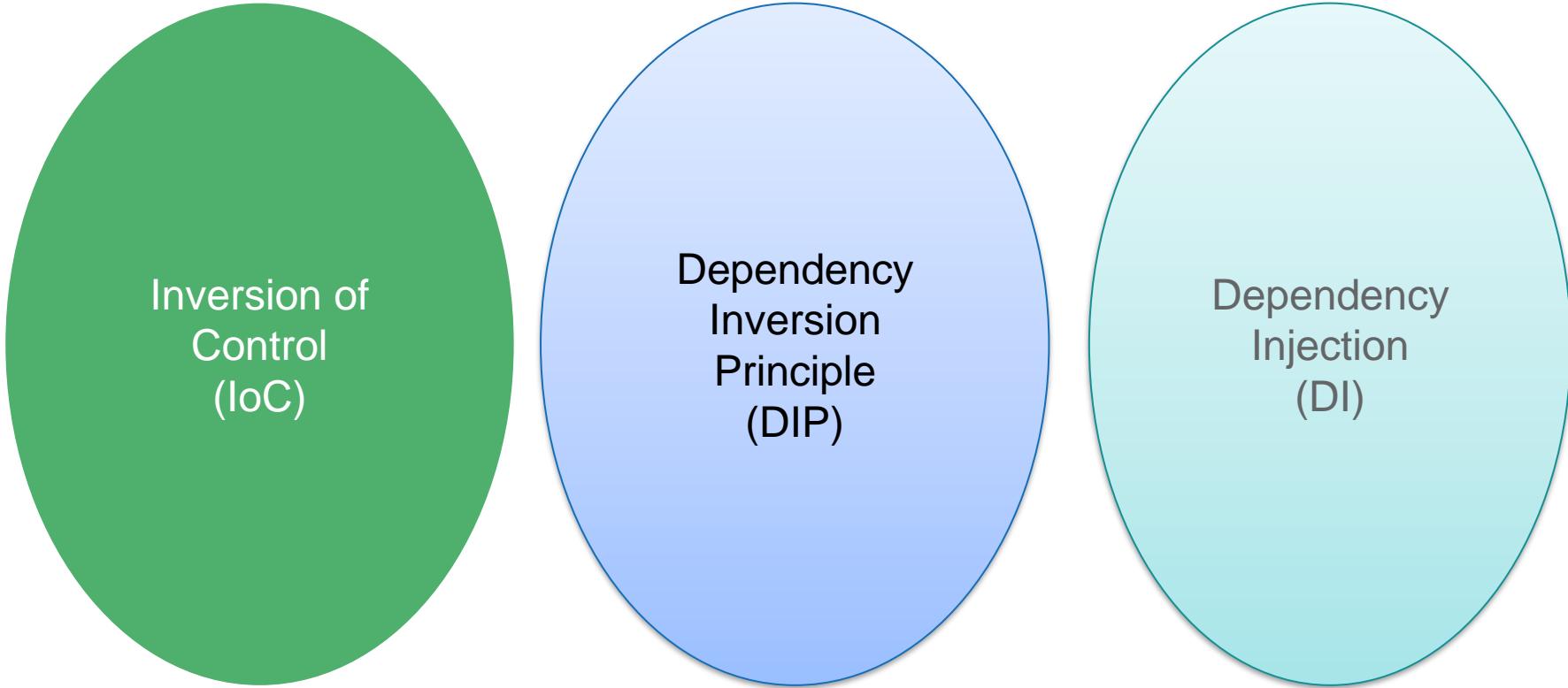
- ① Cohésion et couplage
- ② Principes SOLID
- ③ Focus sur DIP, IOC et DI**
- ④ Design Patterns

## 2 Autres Principes Indispensables

- ① Yagni
- ② Dette Technique



# Des notions souvent confuses



Inversion of  
Control  
(IoC)

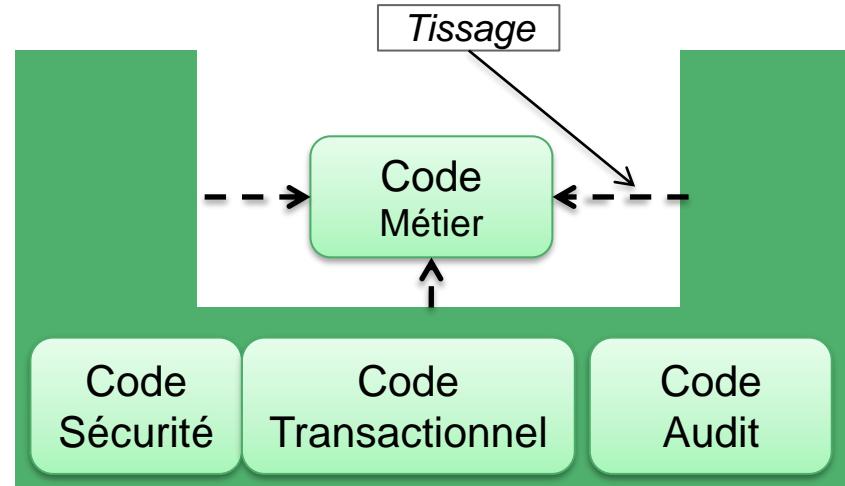
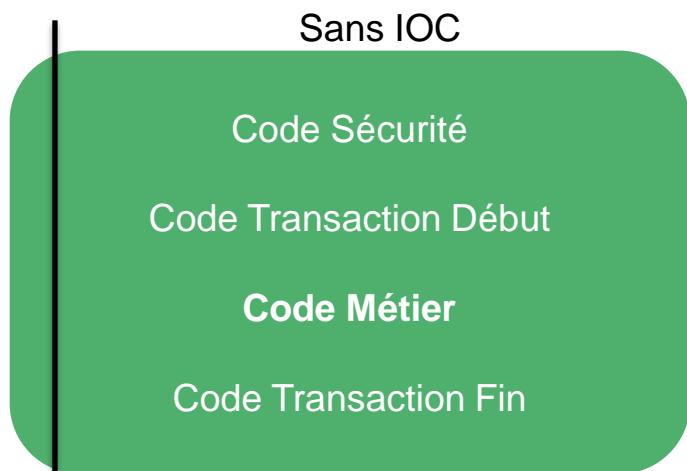
Dependency  
Inversion  
Principle  
(DIP)

Dependency  
Injection  
(DI)



# Inversion of Control (IoC)

IOC



- Appel *impératif* des éléments d'infrastructure (Sécurité, Transaction, etc)
- Mélange du code métier et des responsabilités techniques
- *Déclaration* des éléments d'infrastructure
  - Métadonnées XML ou Annotations
- Invocation de ces éléments par le container (ou framework)
  - Souvent implémenté par AOP



# Dependency Inversion Principle (DIP) – 1/6

DIP

Plusieurs formulations équivalentes:

L'abstrait ne doit pas dépendre du concret

Les généralités ne doivent pas dépendre de détails

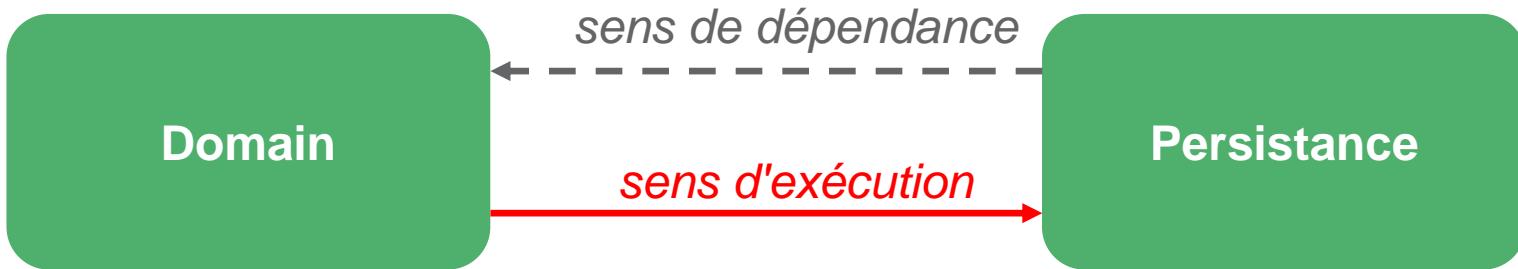
Les modules stables ne doivent pas dépendre des modules instables

- Permet à un module métier (abstrait, général, stable) d'invoyer un module d'infrastructure (concret, détail, instable) sans en dépendre
- Permet au module métier d'avoir un comportement riche sans introduire de dépendances vers l'infrastructure (BD, réseau, etc)



**L'ordre de dépendance est l'inverse de l'ordre d'exécution!**

**Cela fonctionne grâce au polymorphisme donc uniquement dans un langage OO.**

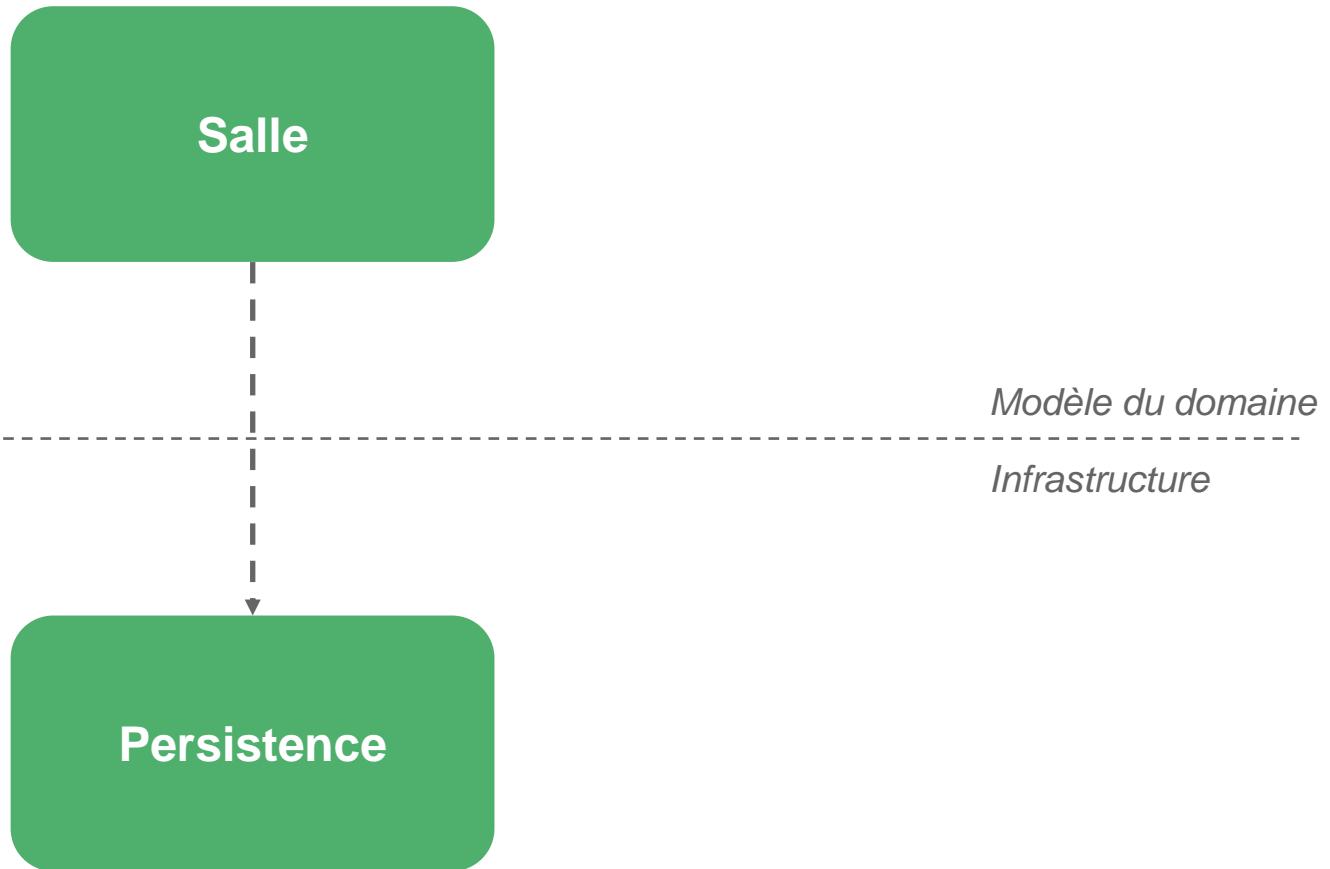


**Dans la suite on ne représente pas le sens d'exécution car il n'existe pas dans le code.**



# Dependency Inversion Principle (DIP) – 3/6

DIP



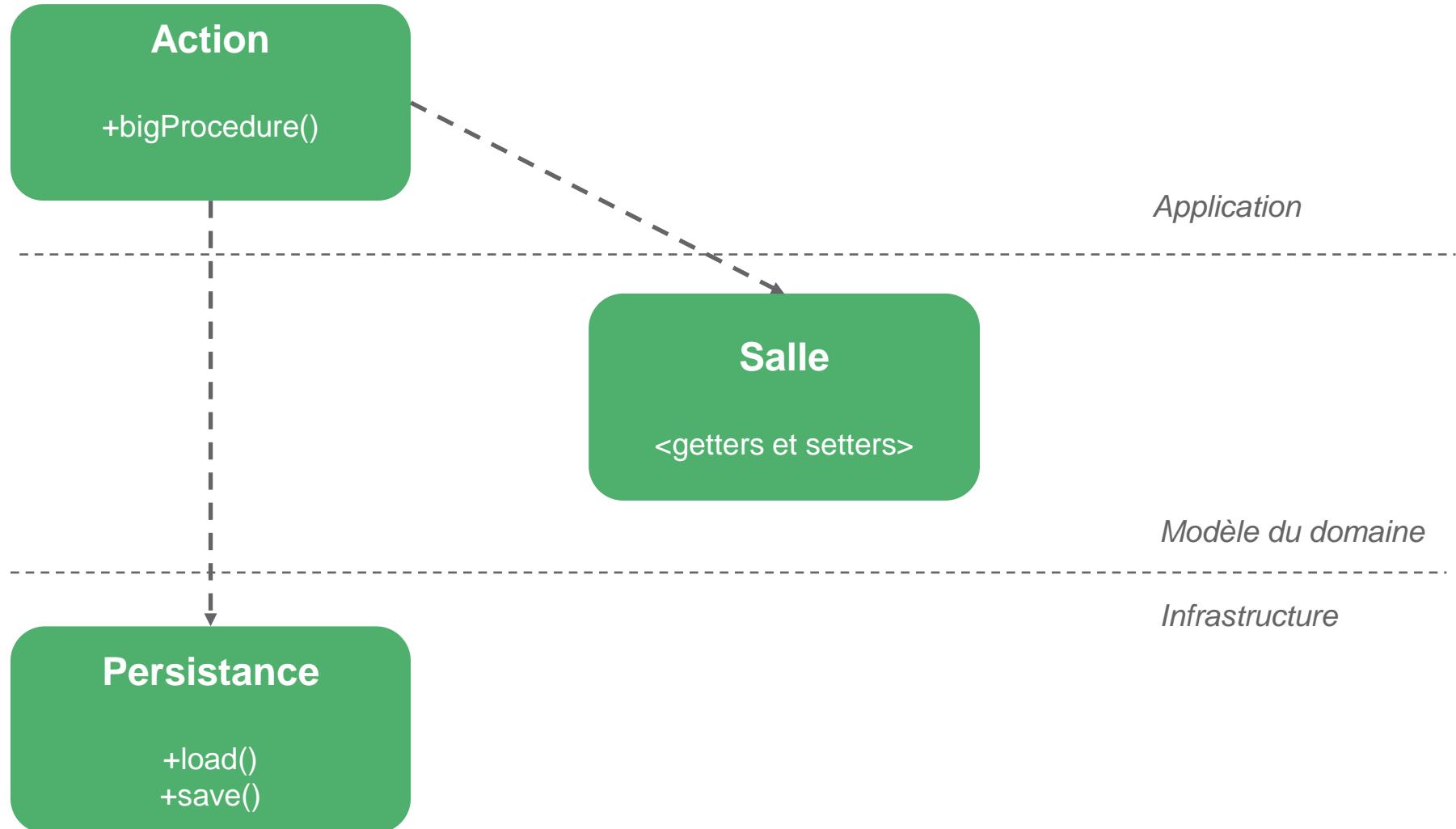
BNP PARIBAS

La banque d'un monde qui change

Prérequis 83

# Dependency Inversion Principle (DIP) – 4/6

DIP

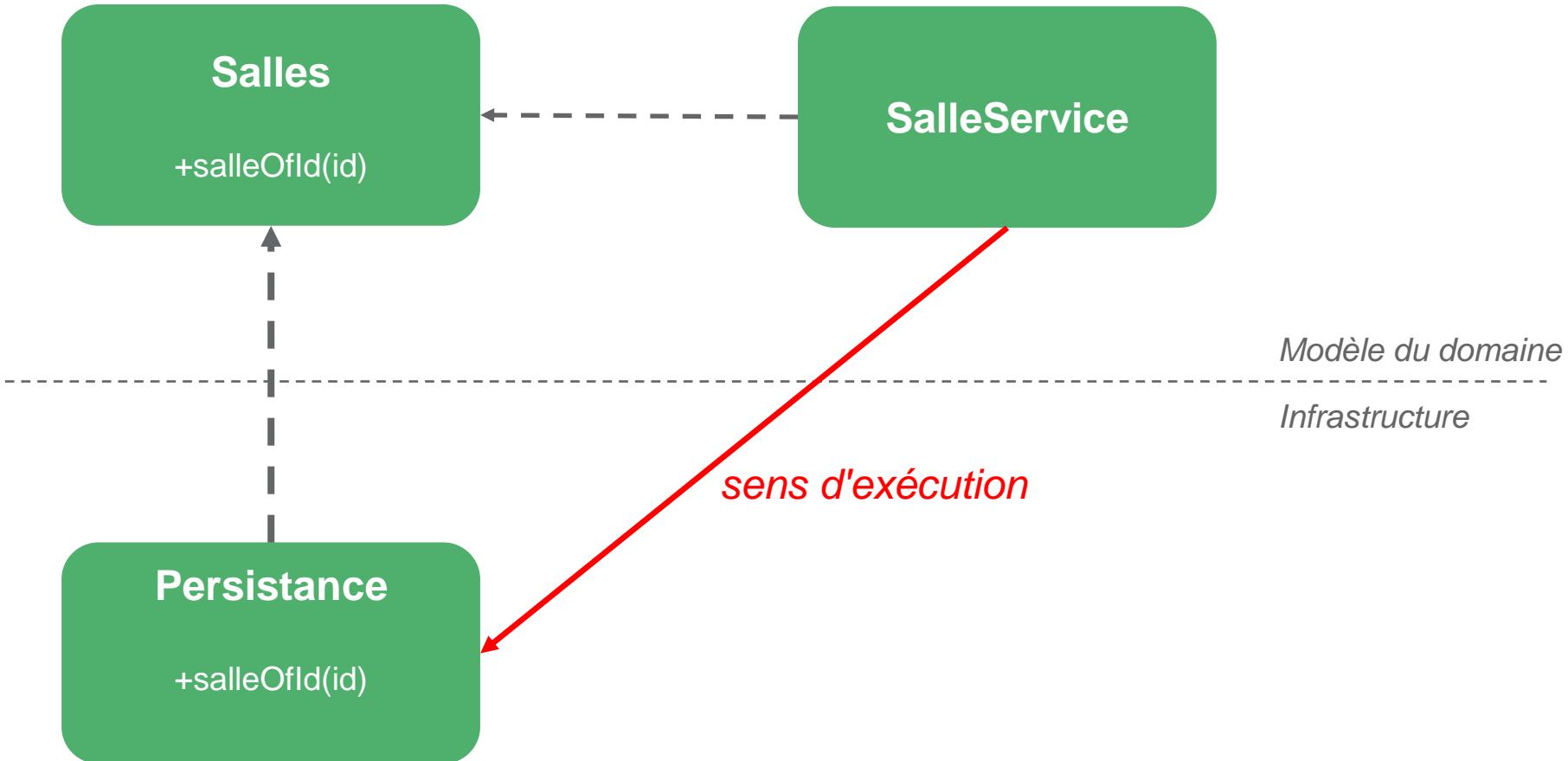


**BNP PARIBAS**

La banque d'un monde qui change

# Dependency Inversion Principle (DIP) – 5/6

DIP



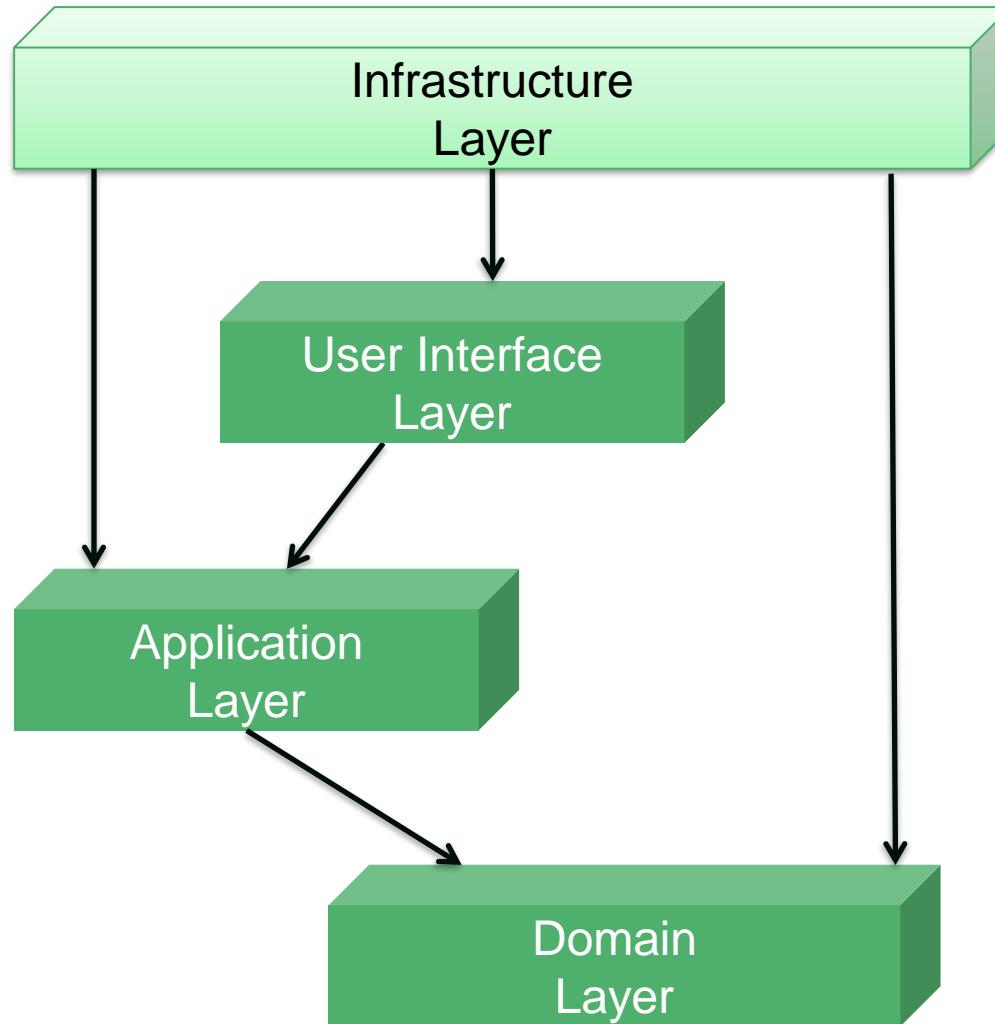
BNP PARIBAS

La banque d'un monde qui change

Prérequis 85

# Dependency Inversion Principle (DIP) – 6/6

DIP



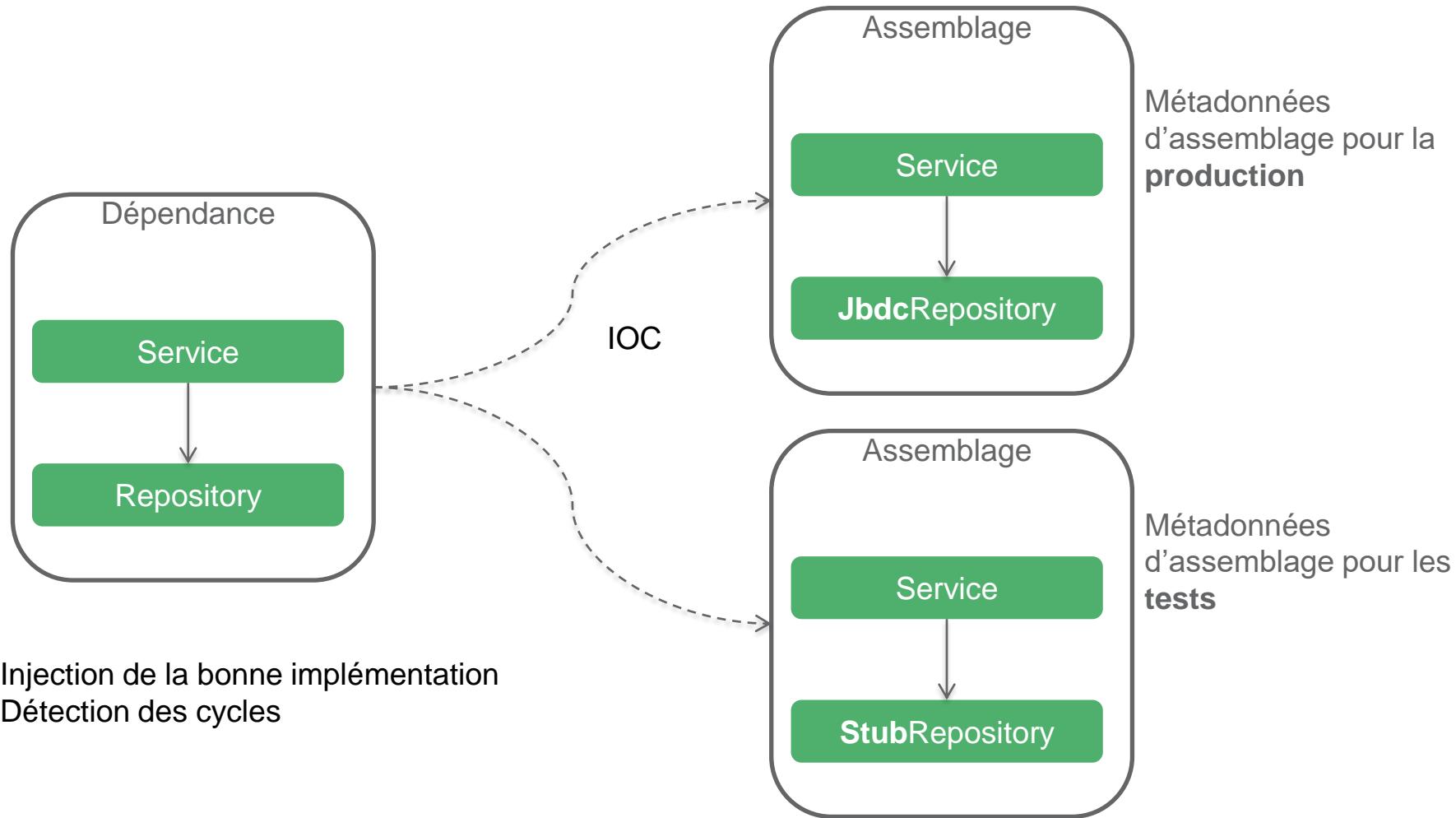
BNP PARIBAS

La banque d'un monde qui change

Prérequis 86

# Dependency Injection (DI)

DI



BNP PARIBAS

La banque d'un monde qui change

# Dependency Injection (DI)

La DI est un type particulier d'IoC où la préoccupation transverse est la construction d'un graphe de composants interdépendants

De multiples objectifs:

- Abstraire l'implémentation concrète injectée
  - Classe d'implémentation
  - Scope
  - Local ou Remote
- Déléguer au conteneur l'initialisation ordonnée d'un graphe de composants dépendants
  - Initialiser conformément à l'ordre de dépendance
  - Déetecter les cycles
  - Echouer tôt et clairement en cas de problème
  - Qualités de services: initialisation tardive, optionnelle, ..



# Objectifs



## 1 Les fondamentaux de la programmation objet

- ① Cohésion et couplage
- ② Principes SOLID
- ③ Focus sur DIP, IOC et DI
- ④ Design Patterns

## 2 Autres Principes Indispensables

- ① Yagni
- ② Separated Interface
- ③ Dette Technique

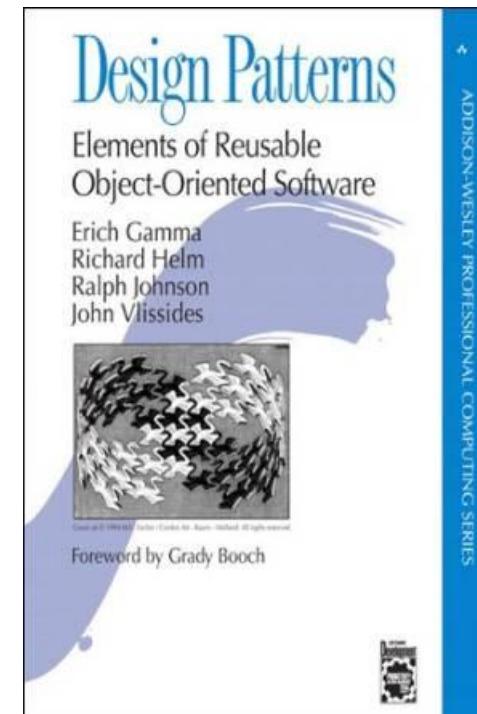


# Design patterns

Un pattern est une structure qui donne une solution classique à un problème fréquent

Le grand classique est *Design patterns: elements of reusable object-oriented software*

- communément appelé *GOF* (gang of four)



# Structure d'un Design pattern

Le GOF a standardisé la présentation des patterns en fiche comportant:

- un **nom**, utilisé pour la communication et comme moyen mnémotechnique
- un **problème** de conception
- une **solution**, formée d'une **structure** et de **participants**
- des **conséquences** de l'application du pattern
- des **variations** du pattern pour différentes situations



# Autres types de patterns

Les design patterns sont des patterns de conception (design), mais il existe aussi:

- Implementation patterns: propres à un langage
- Analysis patterns: patterns d'analyse propres à un domaine
- Architecture patterns
- Patterns spécialisés (patterns de concurrence)

→ Par abus de langage on parle souvent de design patterns



# Objectifs



## 1 Les fondamentaux de la programmation objet

- ① Cohésion et couplage
- ② Principes SOLID
- ③ Focus sur DIP, IOC et DI
- ④ Design Patterns

## 2 Autres Principes Indispensables

- ① Yagni
- ② Separated Interface
- ③ Dette Technique



# You Ain't Gonna Need It (Yagni)

*Ne pas ajouter de fonctionnalités non demandées "au cas où"*

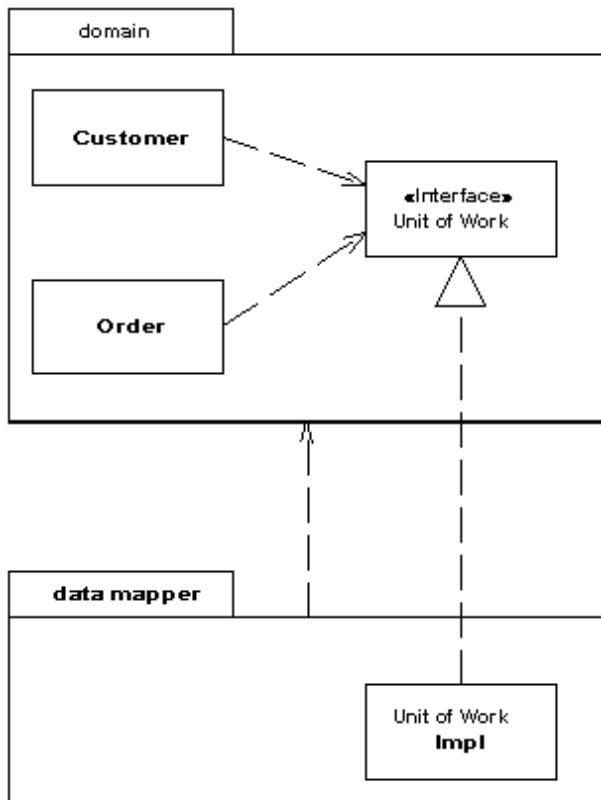
*Eviter la généralisation prématuée dans la conception, et la sur-conception (over-engineering).*

- Ajouter une fonctionnalité en anticipant sur le besoin des utilisateurs conduit souvent à une fonctionnalité inadaptée ou inutilisée (poids mort), et se fait au détriment des fonctionnalités indispensables.
- Le surcoût induit par la complexité d'une conception prématuérément généralisée n'est jamais amorti.



# Separated Interface (SI)

**Définir une interface dans un module différent de son implémentation**



- Maîtrise des dépendances entre les différentes parties du système
  - package en Java
- Un client ayant besoin de la dépendance vers une interface, n'a pas à se soucier du module d'implémentation

Source Patterns of Enterprise Application Architecture – Martin Fowler



**BNP PARIBAS**

La banque d'un monde qui change

# Dette IT - Enjeux

- L'investissement qu'il faudrait réaliser pour mettre notre patrimoine à l'état de l'art des besoins et des pratiques IT constitue une dette
- Les causes sont multiples :
  - Choix de délivrer certaines évolutions rapidement sans réflexion sur leur intégration dans l'existant
  - Impossibilité de s'aligner tout de suite sur une exigence du métier
  - Manque de modularité et d'évolutivité de la conception entraînant une dégradation de la qualité interne d'une application au fil des évolutions successives
  - Mauvaise gestion de l'obsolescence
  - ...
- Nous devons donc être en mesure d'évaluer et mesurer la dette des différentes composantes de notre patrimoine applicatif afin de mettre en œuvre un plan de remédiation adapté



# Les 6 axes de mesure

## Obsolescence technique et applicative

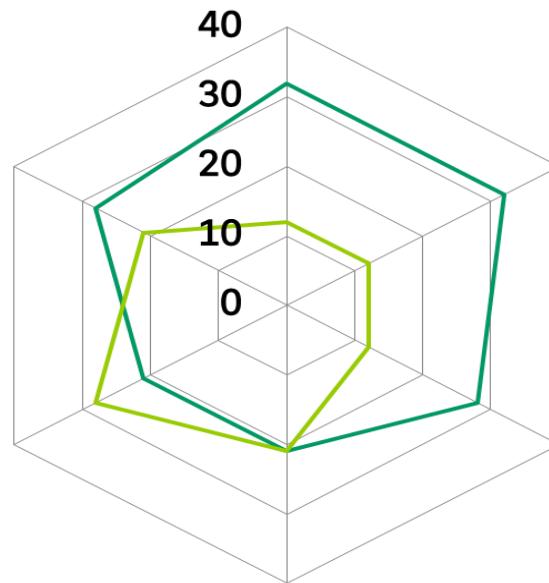
Les composants techniques et applicatifs suivent-ils un cycle d'évolution suffisant pour prendre en compte les nouvelles menaces (sécurité...) et opportunités (avancées technologiques...)

## Compétences

Dispose t'on du dispositif adapté pour maintenir durablement la connaissance du système et évaluer les impacts des demandes d'évolution

## Qualité du code, Fiabilité de la documentation

Les règles de programmation définies pour garantir la maintenabilité et la fiabilité sont elles correctement implémentées



## Disponibilité, Performance

Les niveaux de service sont-ils bien définis. Sont-ils respectés ?

## Conception et architecture

Les principes de modularité, de réutilisabilité, de standardisation sont ils respectés. L'application est-elle capable d'évoluer de façon souple, fluide et efficace

## Exploitabilité

La mise en œuvre de l'application est-elle bien construite sur des standards d'exploitation permettant l'industrialisation et limitant les interventions humaines



# Dette IT

- A l'image d'un « bon chef de famille », être responsable de son patrimoine c'est savoir gérer ses finances :
  - Connaître ses charges, c'est à dire les coûts fixes dictés par l'état actuel de son bien
  - Investir fonctionnellement pour enrichir son capital en conscience des charges supplémentaires engendrées
  - Investir techniquement pour entretenir et rénover son patrimoine afin de réduire - au moins stabiliser - ses charges.
- La finalité n'est pas forcement de rembourser l'intégralité de la dette, mais a minima de la mettre sous-contrôle
- Une dette importante nécessitera une refonte applicative importante. Il est donc préférable d'entretenir régulièrement notre patrimoine, pour éviter tout surendettement impactant trop fortement le coût, le délai et le risque des développements suivants, et in fine de la qualité de notre Système d'Information



# Objectifs



## ① De l'existant vers la cible DDD

### ② Briques DDD simples

- ① Value Object
- ② Entité

### ③ Composants DDD de type Service

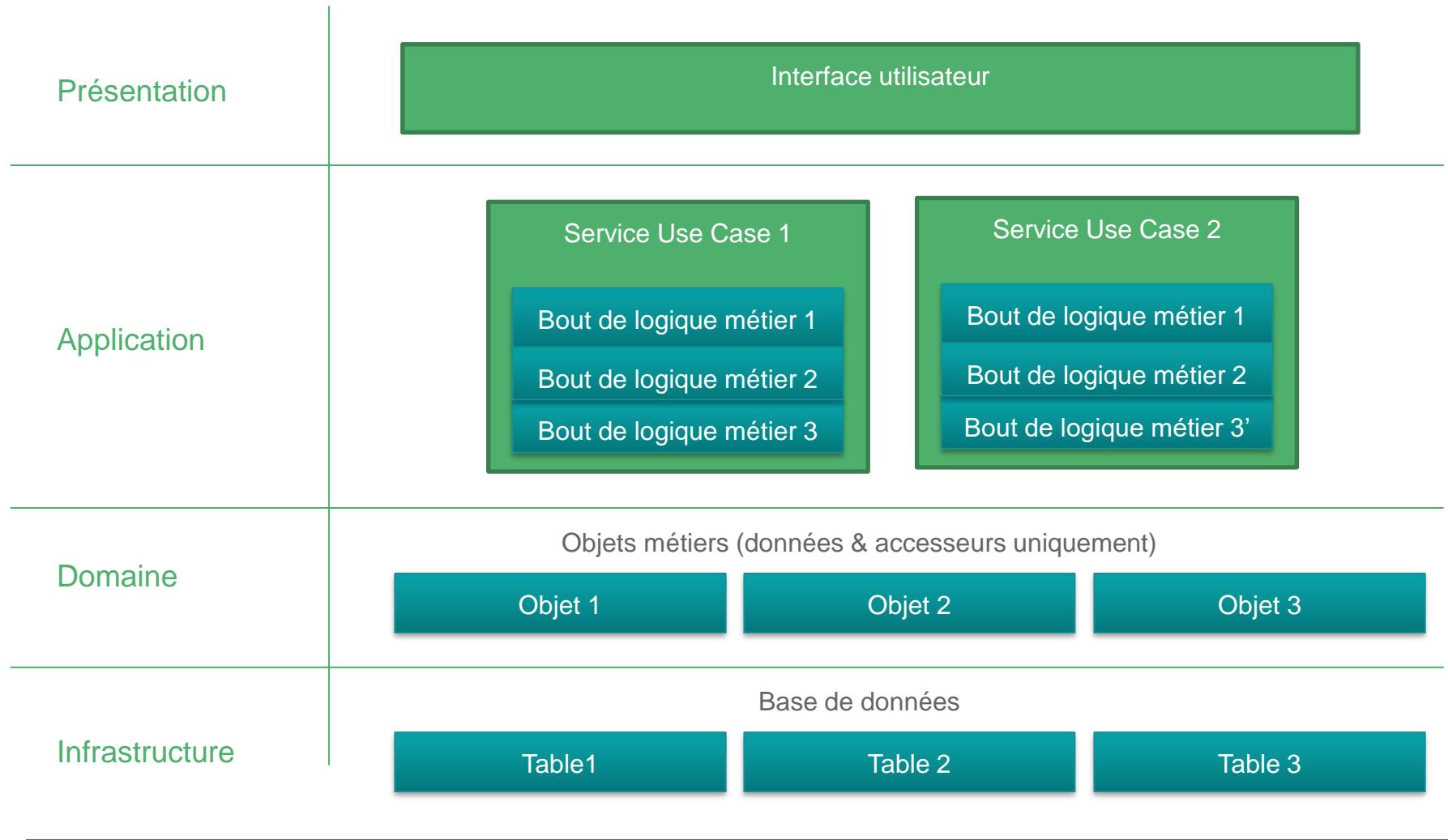
- ① Application Service
- ② Domain Service

### ④ Composants DDD de type Infrastructure

- ① Repository
- ② Infrastructure Service

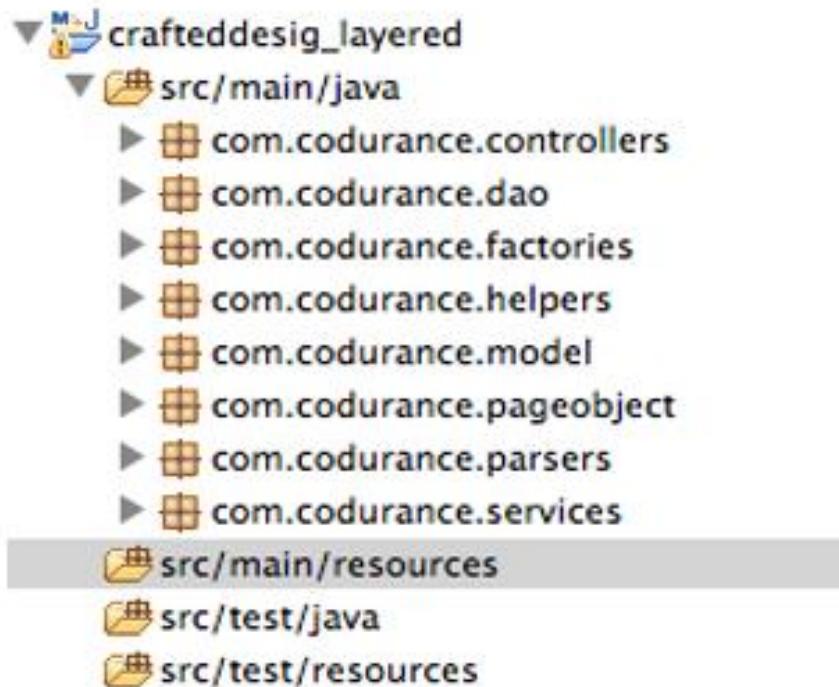


# L'existant avec un modèle squelettique



# Structure en packages existante avec un modèle squelettique

## Example: Layered structure

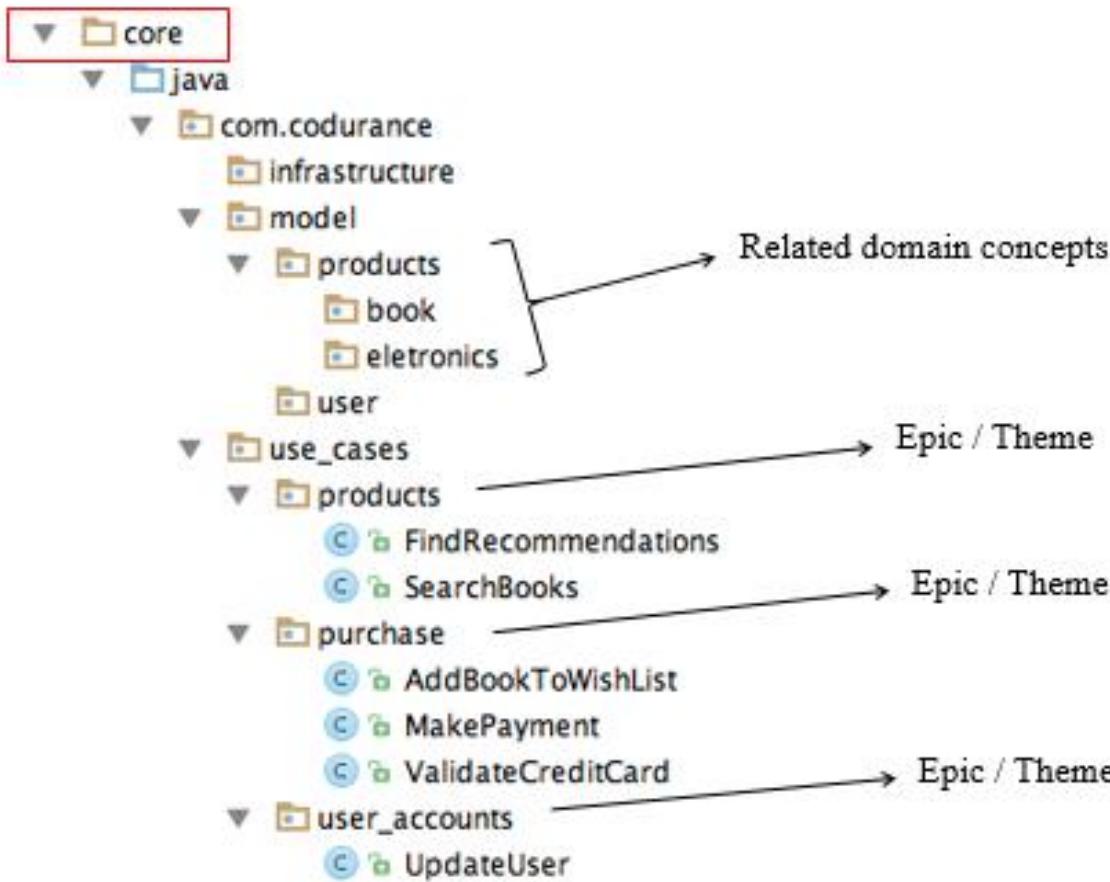


**What does this application do? What is it about?**



# Structure en packages cible DDD

## Core responsibility (*bigger project*)



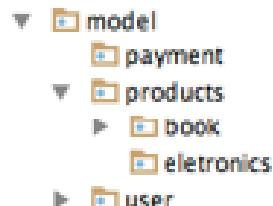
Crafted Design, Sandro Mancuso



# Structure en packages cible DDD

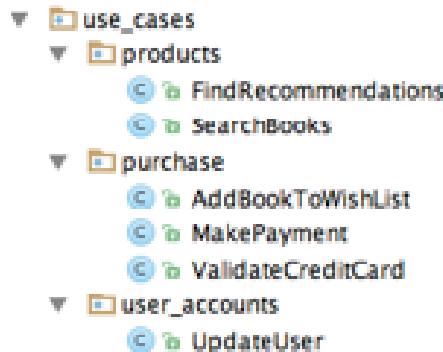
## Answering the two original questions

- What is the application about? (main concepts)



Expressed by nouns

- What does the application do? (main features)



Expressed by verbs  
(Actions)

Crafted Design, Sandro Mancuso



BNP PARIBAS

La banque d'un monde qui change

# Objectifs



① De l'existant vers la cible DDD

## ② Briques DDD simples

- ① Value Object
- ② Entité

③ Composants DDD de type Service

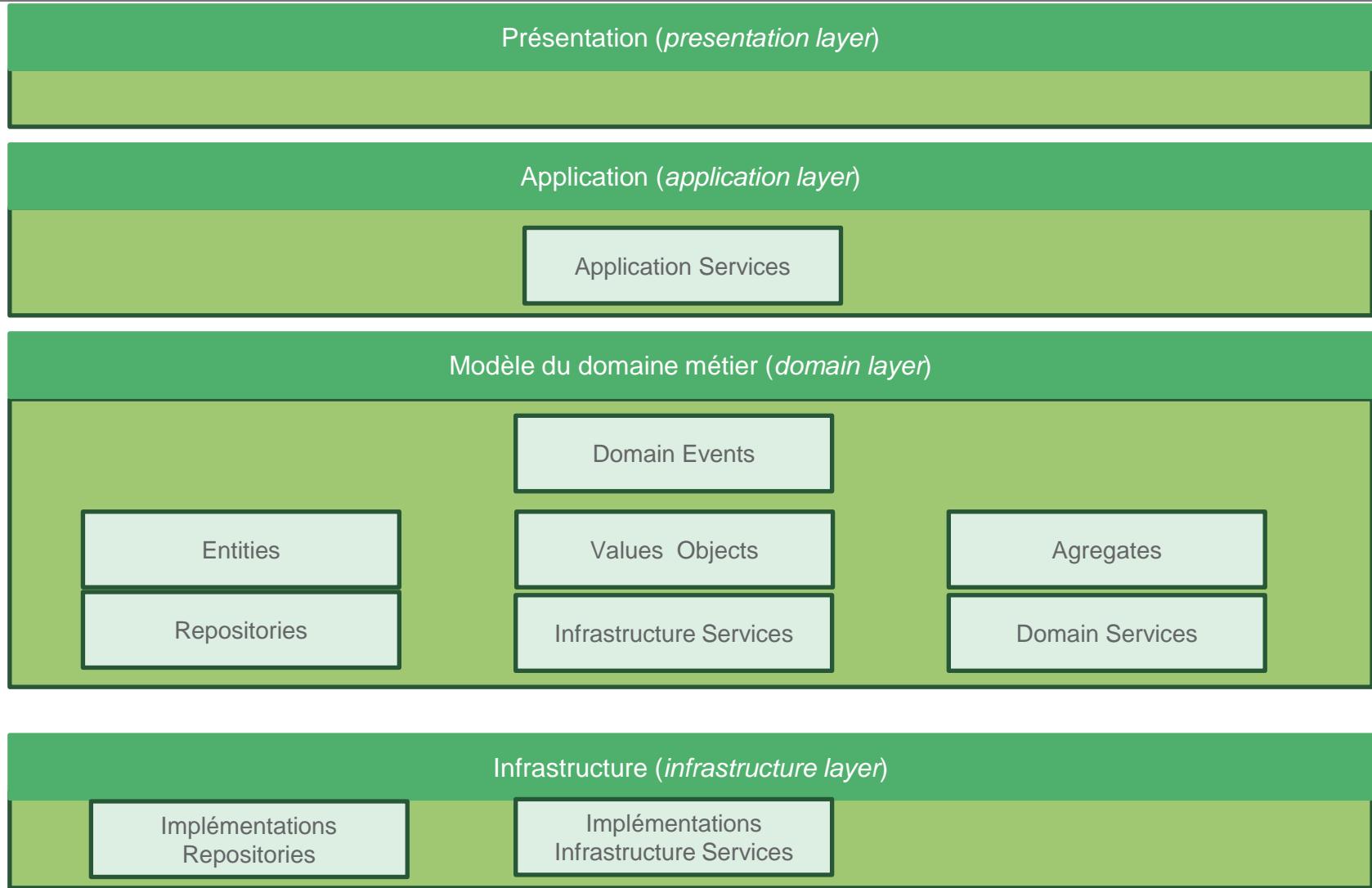
- ① Application Service
- ② Domain Service

④ Composants DDD de type Infrastructure

- ① Repository
- ② Infrastructure Service



# Vue en couches des briques DDD



# Value Object



① De l'existant vers la cible DDD

## ② Briques DDD simples

① Value Object

② Entité

③ Composants DDD de type Service

① Application Service

② Domain Service

④ Composants DDD de type Infrastructure

① Repository

② Infrastructure Service

⑤ Briques DDD plus avancées

① Agrégat

② Domain Event

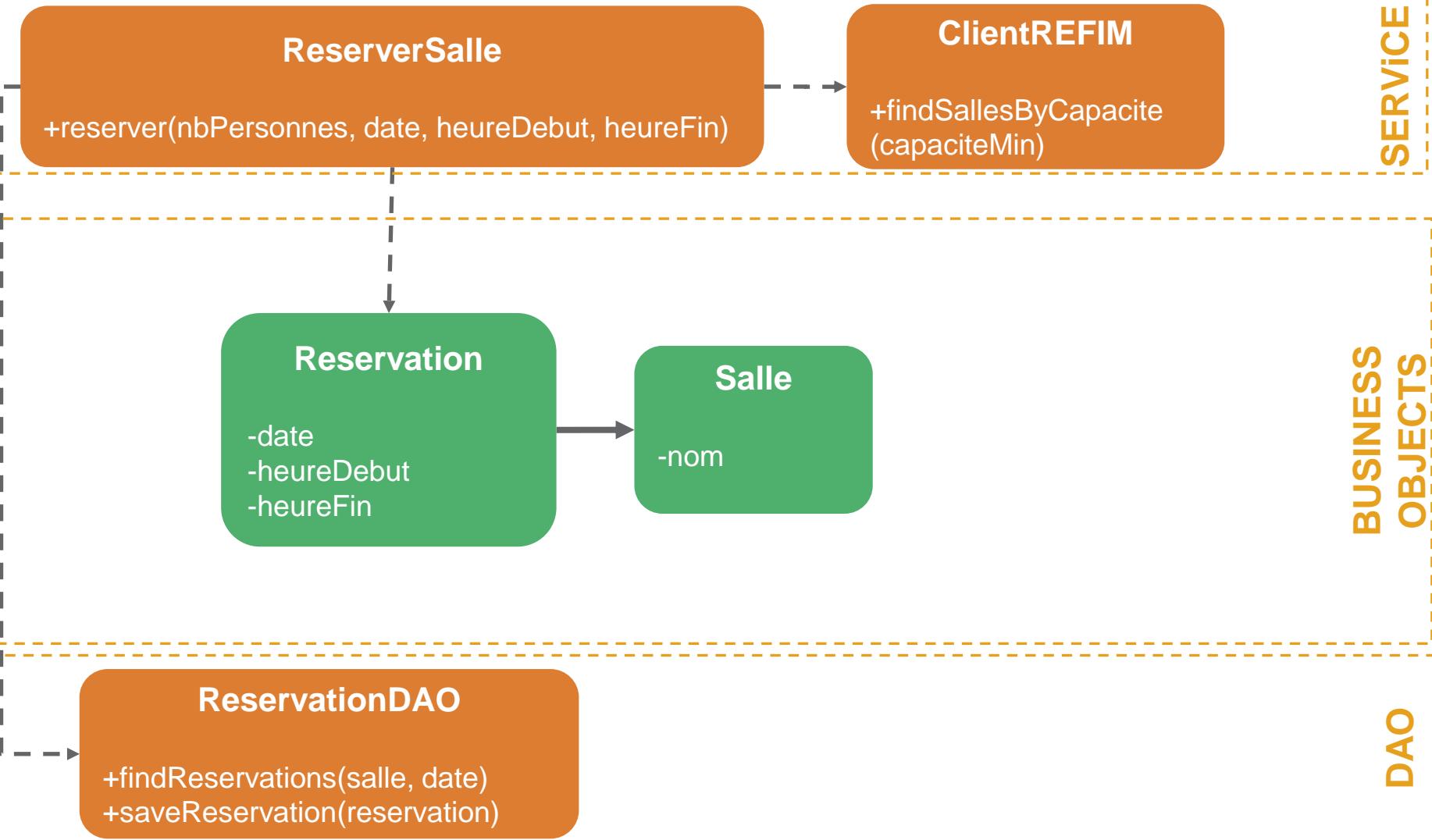


# Rappel: Modèle initial de la réservation de salle

SERVICE

BUSINESS  
OBJECTS

DAO



# ReserverSalle.reserver()

+ reserver(nbPersonnes, date, heureDebut, heureFin):

clientREFIM.findSalleByCapacite(nbPersonnes)

Salle

Salle

Salle

Salle

Salle

Salle

findFirst (

reservationPossible ( Salle , date, heureDebut, heureFin)

reservationDAO.saveReservation(

new Reservation( Salle selected , date, heureDebut, heureFin)

)



BNP PARIBAS

La banque d'un monde qui change

# ReserverSalle.reserver() en code



```
public class ReserverSalle {  
  
    private ClientREFIM clientREFIM;  
    private ReservationDAO reservationDAO;  
  
    public void reserver(int nbPersonnes, Date date, Time heureDebut, Time heureFin) {  
  
        clientREFIM.findSallesByCapacite(nbPersonnes) //Collection<Salle>  
            .findFirst(  
                | salle -> reservationPossible(salle, date, heureDebut, heureFin)  
            )  
            .ifPresent(  
                | salle -> reservationDAO.saveReservation(  
                    new Reservation(salle, date, heureDebut, heureFin)  
                )  
            )  
    }  
  
    private boolean reservationPossible(Salle salle, Date date, Time heureDebut, Time heureFin) {  
        ???  
    }  
}
```



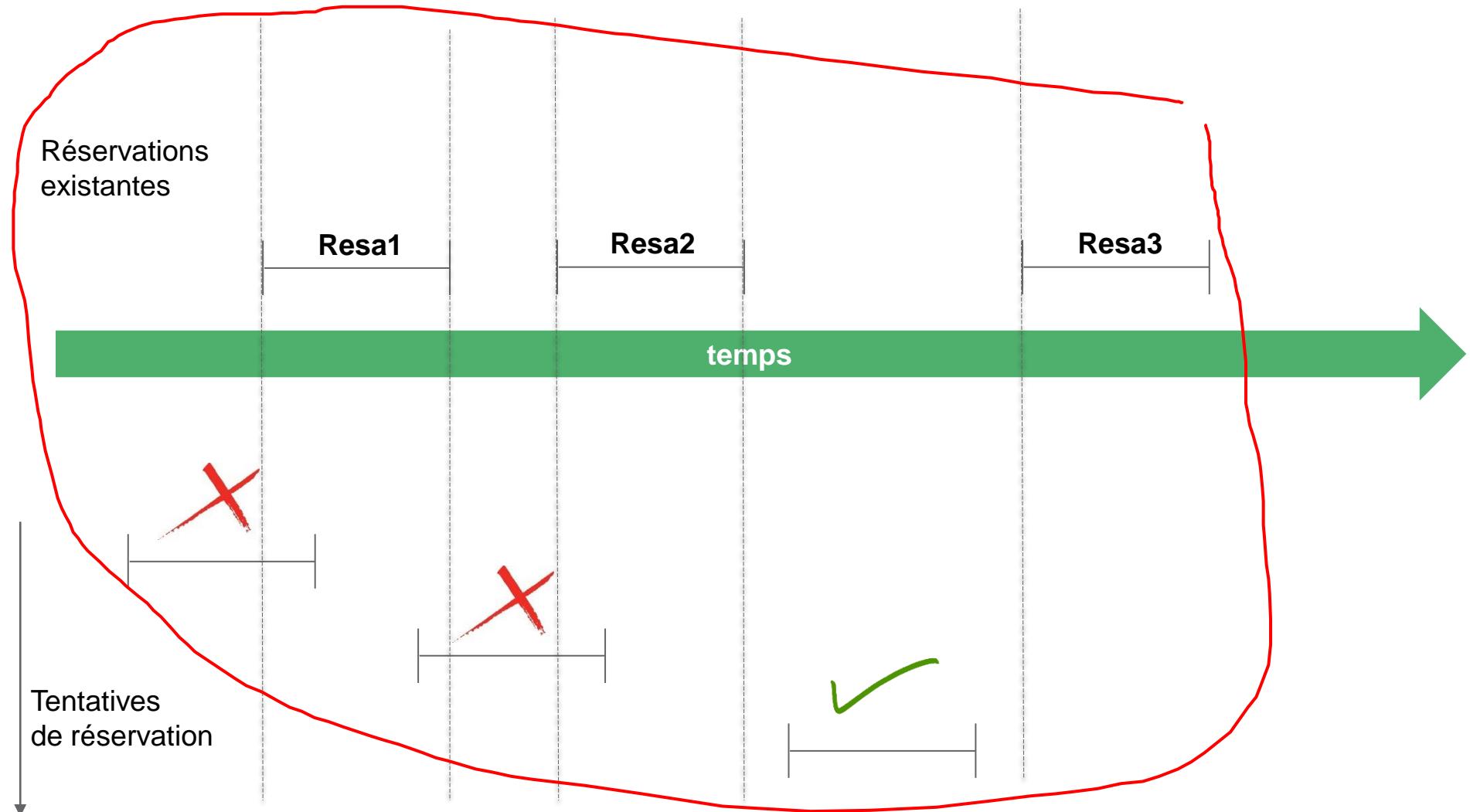


**Atelier:**  
***Implémenter `reservationPossible()`***

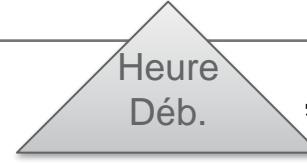
- Une implémentation papier ou un algorithme suffit
- Dans cette exercice on ne modifie pas le DAO `ReservationDAO`



# Problématique de disponibilité de créneaux



# [Schéma] - **reservationPossible()**

+ **reservationPossible(**  ,  ,  ,  **)**:

**reservationDAO.findReservations(salle,**  **)**

Reservation  
Prise

Reservation  
Prise

Reservation  
Prise

Reservation  
Prise

Reservation  
Prise

**map(**  
**)**

Reservation  
Prise

Heure  
Déb.  
Prise

Heure  
Fin  
Prise

**none(**  
**)**

Heure  
Fin  
Prise

>

Heure  
Déb.

&&

Heure  
Déb.  
Prise

<

Heure  
Fin

**VRAI / FAUX**

# Implémenter reservationPossible() - Première approche

```
public void reserver(int nbPersonnes, Date date, Time heureDebut, Time heureFin) {  
  
    clientREFIM.findSallesByCapacite(nbPersonnes) //Collection<Salle>  
        .findFirst(  
            salle -> reservationPossible(salle, date, heureDebut, heureFin)  
        )  
        .ifPresent(  
            salle -> reservationDAO.saveReservation(  
                new Reservation(salle, date, heureDebut, heureFin)  
            )  
        )  
    }  
  
private boolean reservationPossible(Salle salle, Date date, Time heureDebut, Time heureFin) {  
  
    return reservationDAO.findReservations(salle, date) //Collection<Reservation>  
        .none(  
            dejaPris ->  
                dejaPris.getHeureFin() > heureDebut  
                &&  
                dejaPris.getHeureDebut() < heureFin  
        )  
    }  
}
```



# Méthode `reservationPossible()`

---



**Atelier: Quels sont les *inconvénients de la solution précédente?***



**BNP PARIBAS**

La banque d'un monde qui change



# Méthode `reservationPossible()`

## Reprise Atelier

**Quels sont les *inconvénients de la solution précédente?***

### 1. Le code n'est pas assez explicite

- Code peu compréhensible: purement mécanique, n'exprimant aucune intention
- Très forte probabilité de bugs
- Aucune expression du domaine

### 2. Compréhension locale du code impossible

- On est obligé de regarder le code appelant pour vérifier si les heures de début/fin sont passées dans le bon ordre

### 3. Pas de logique de validation

### 4. Algorithme qui tend à être facilement dupliqué dans d'autres sous-systèmes

- On pourra facilement violer le principe DRY ("Don't repeat yourself" – Eviter la duplication)



# Notion de Value Object (VO)

- Une valeur est **intangible** et **immutable**
  - Existence indépendante du temps et de l'espace
  - Qui ne change pas
    - l'application ne change pas le cycle de vie de l'objet

Distance

Longueur

ISBN

Couleur

Durée

...



BNP PARIBAS

La banque d'un monde qui change

# Value Object (VO) – Définition formelle

*Quantité ou description immuable, regroupant en un tout cohérent un ensemble de valeurs primitives, avec un nom explicitant le sens de cet ensemble, et un comportement propre au nouveau concept ainsi introduit.*



# Méthode `reservationPossible()`

---



## Atelier:

**Pourrait-on améliorer le code la méthode  
reservationPossible() avec le concept de  
ValueObject?**





# Rappel ReserverSalle.reserver()

```
public class ReserverSalle {  
  
    private ClientREFIM clientREFIM;  
    private ReservationDAO reservationDAO;  
  
    public void reserver(int nbPersonnes, Date date, Time heureDebut, Time heureFin) {  
  
        clientREFIM.findSallesByCapacite(nbPersonnes) //Collection<Salle>  
            .findFirst(  
                | salle -> reservationPossible(salle, date, heureDebut, heureFin)  
            )  
            .ifPresent(  
                | salle -> reservationDAO.saveReservation(  
                    new Reservation(salle, date, heureDebut, heureFin)  
                )  
            )  
    }  
  
    private boolean reservationPossible(Salle salle, Date date, Time heureDebut, Time heureFin) {  
        ???  
    }  
}
```



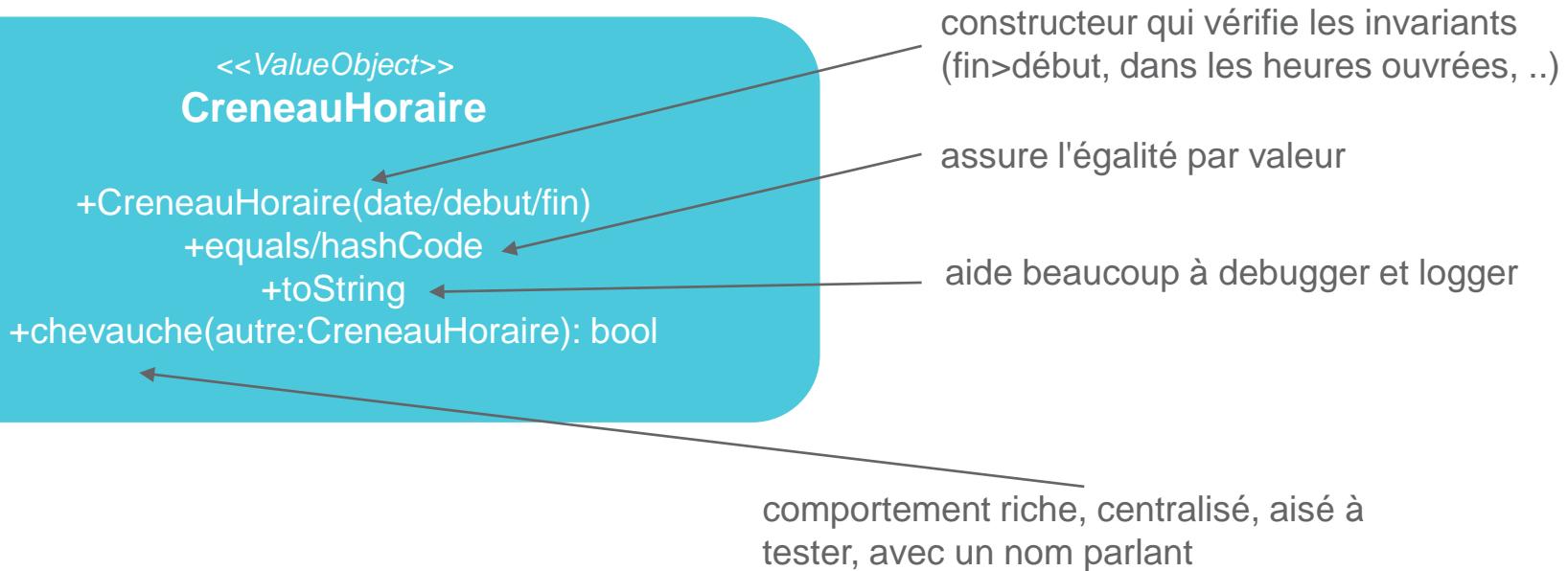


# Value Object: reserverSalle avec VO

## Reprise Atelier:

**Pourrait-on améliorer le code la méthode** reservationPossible() **avec le concept de ValueObject?**

A1: Oui, on peut représenter les plages horaires et les durées temporelles par des ValueObjects:



# [Schéma] - **reservationPossible()** avec CreneauHoraire

+ **reservationPossible( Salle , Creneau Souhaite ):**

**reservationDAO.findReservations(salle, creneauSouhaite.getDate())**

Reservation

Reservation

Reservation

Reservation

Reservation

**map(**

Reservation

Creneau

**)**

**none(**

**)**

Chevauchement?

Creneau  
Souhaite

Creneau

VRAI / FAUX



BNP PARIBAS

La banque d'un monde qui change

# [Schéma] - ReserverSalle.reserver() avec CreneauHoraire

+ reserver(nbPersonnes,  ) :

clientREFIM.findSalleByCapacite(nbPersonnes)

Salle

Salle

Salle

Salle

Salle

Salle

findFirst (

reservationPossible (

Salle

 )

reservationDAO.saveReservation(

new Reservation(

Salle selected

 )

)



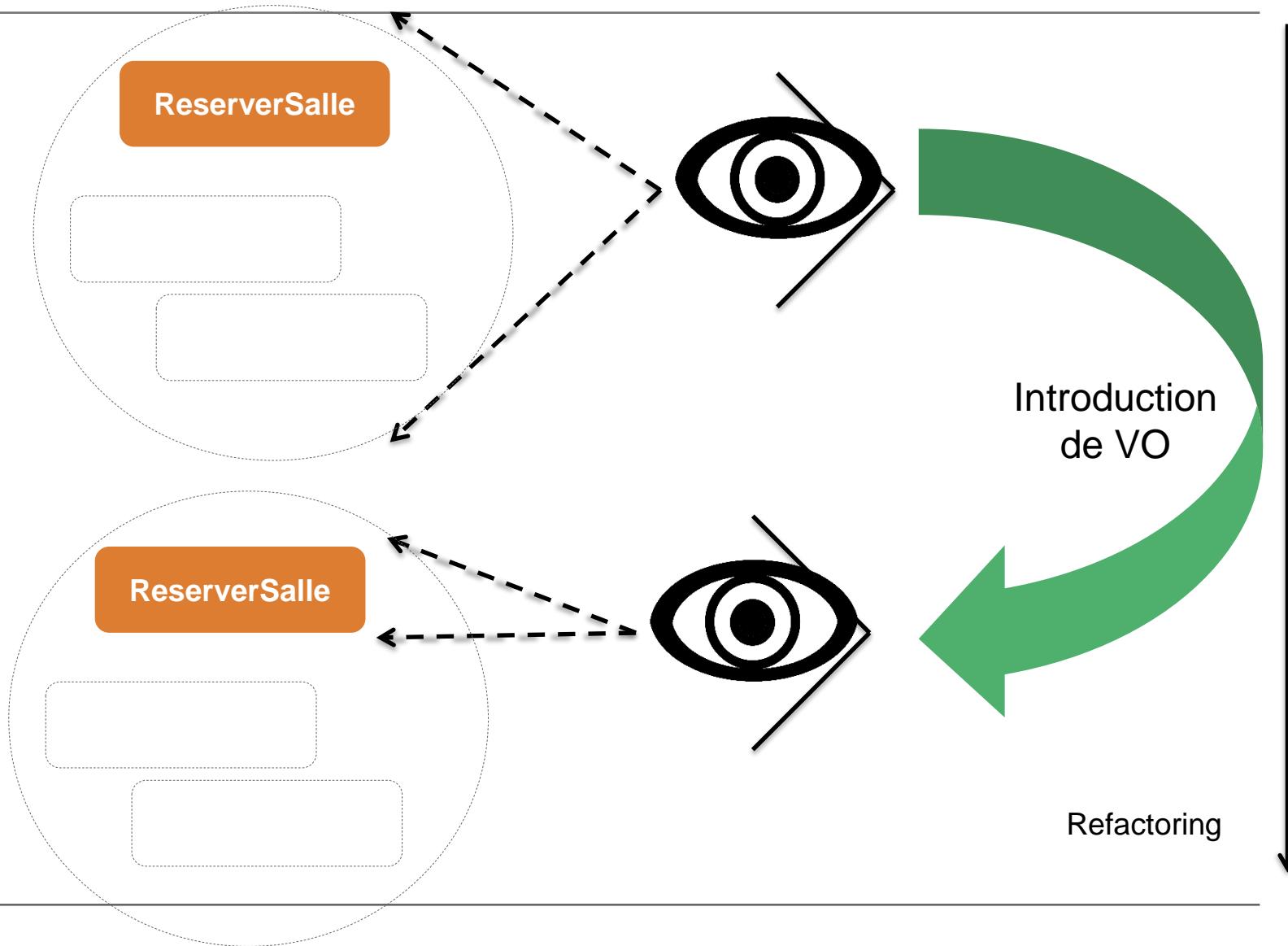
# [Code] – ReserverSalle avec VO



```
public class ReserverSalle {  
    ...  
  
    public void reserver(int nbPersonnes, CreneauHoraire creneau) {  
  
        clientREFIM.findSallesByCapacite(nbPersonnes) //Collection<Salle>  
            .findFirst(salle -> reservationPossible(salle, creneau))  
            .ifPresent(salle ->  
                reservationDAO.saveReservation(new Reservation(salle, creneau))  
            )  
  
    }  
  
    private boolean reservationPossible(Salle salle, CreneauHoraire creneauSouhaite) {  
        return reservationDAO  
            .findReservations(salle, creneauSouhaite.getDate())//Collection<Reservation>  
            .map(Reservation::getCreneau)  
            .none(creneauReserve -> creneauSouhaite.chevauche(creneauReserve))  
    }  
}
```



# Apport de Value Object (VO)

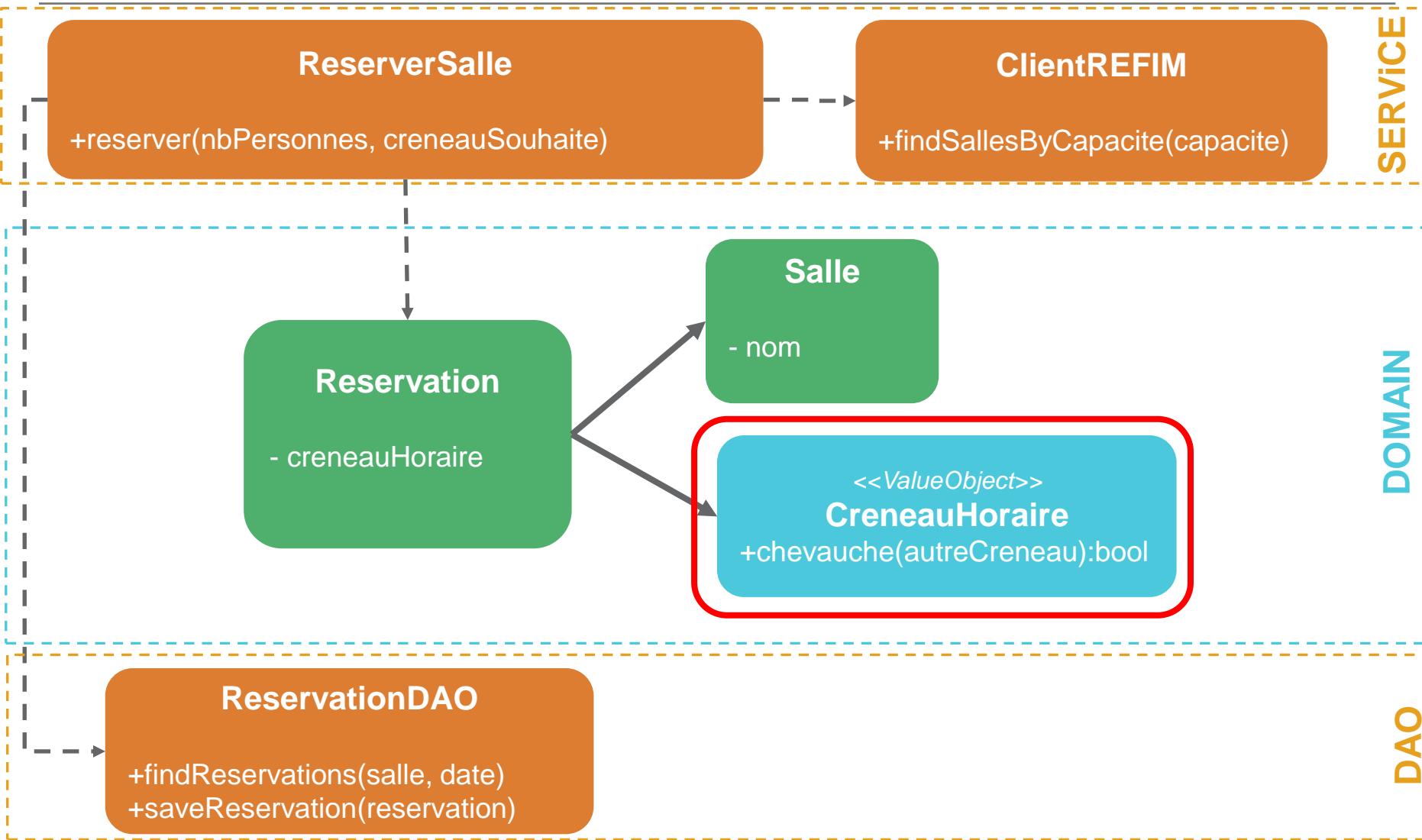


# Reprise du modèle avec le VO CreneauHoraire

SERVICE

DOMAIN

DAO



# Contenu d'un Value Object

MAGNITUDE et UNITE

Conversion vers une autre unité

CONVERSION depuis/vers représentation textuelle

REGLES DE VALIDATION

Appartenance à un intervalle, conformité à un format, ..

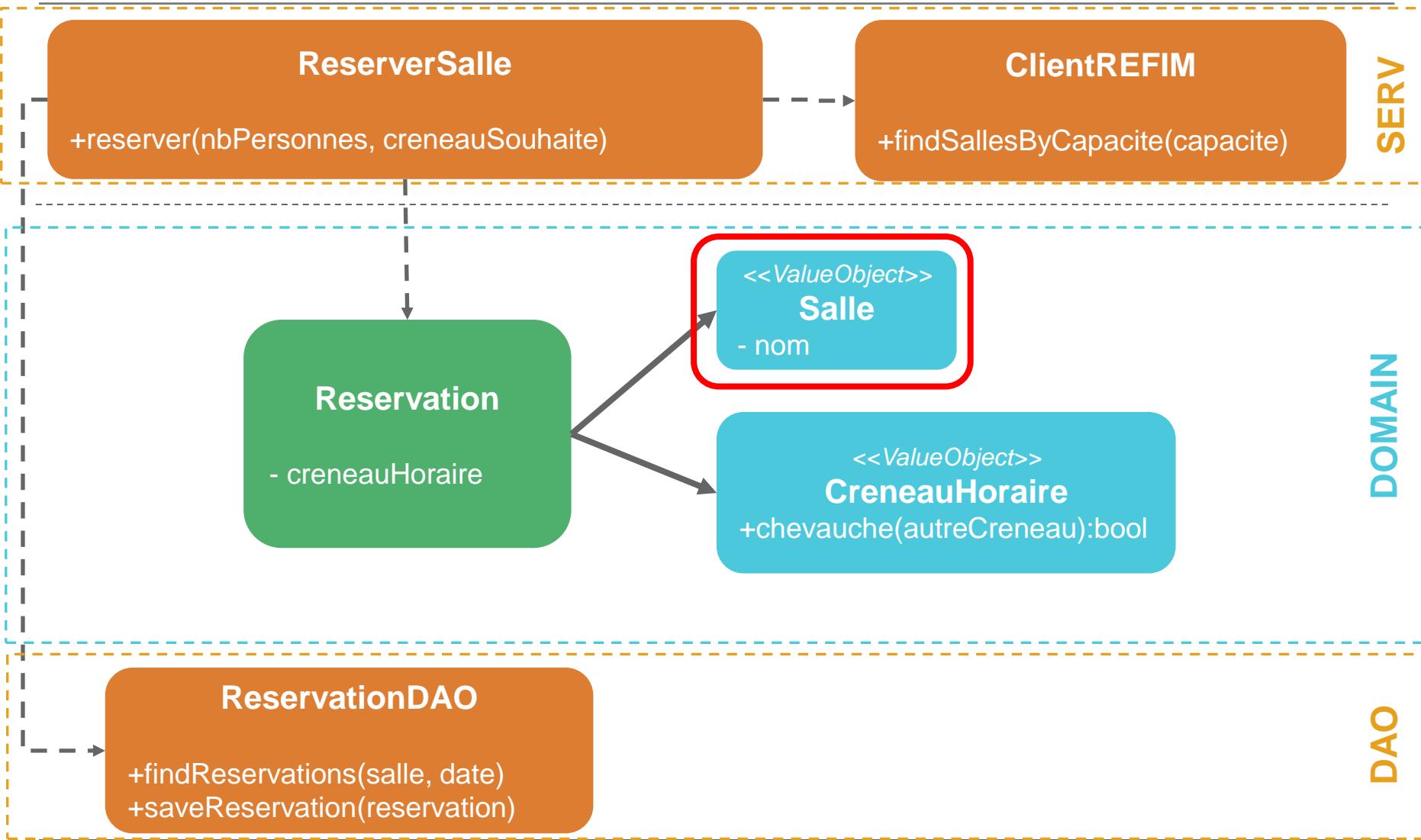
EGALITE: toujours

Comparaison: parfois (attention à ne pas imposer un ordre naturel de tri  
là où il faudrait des ordres contextuels)

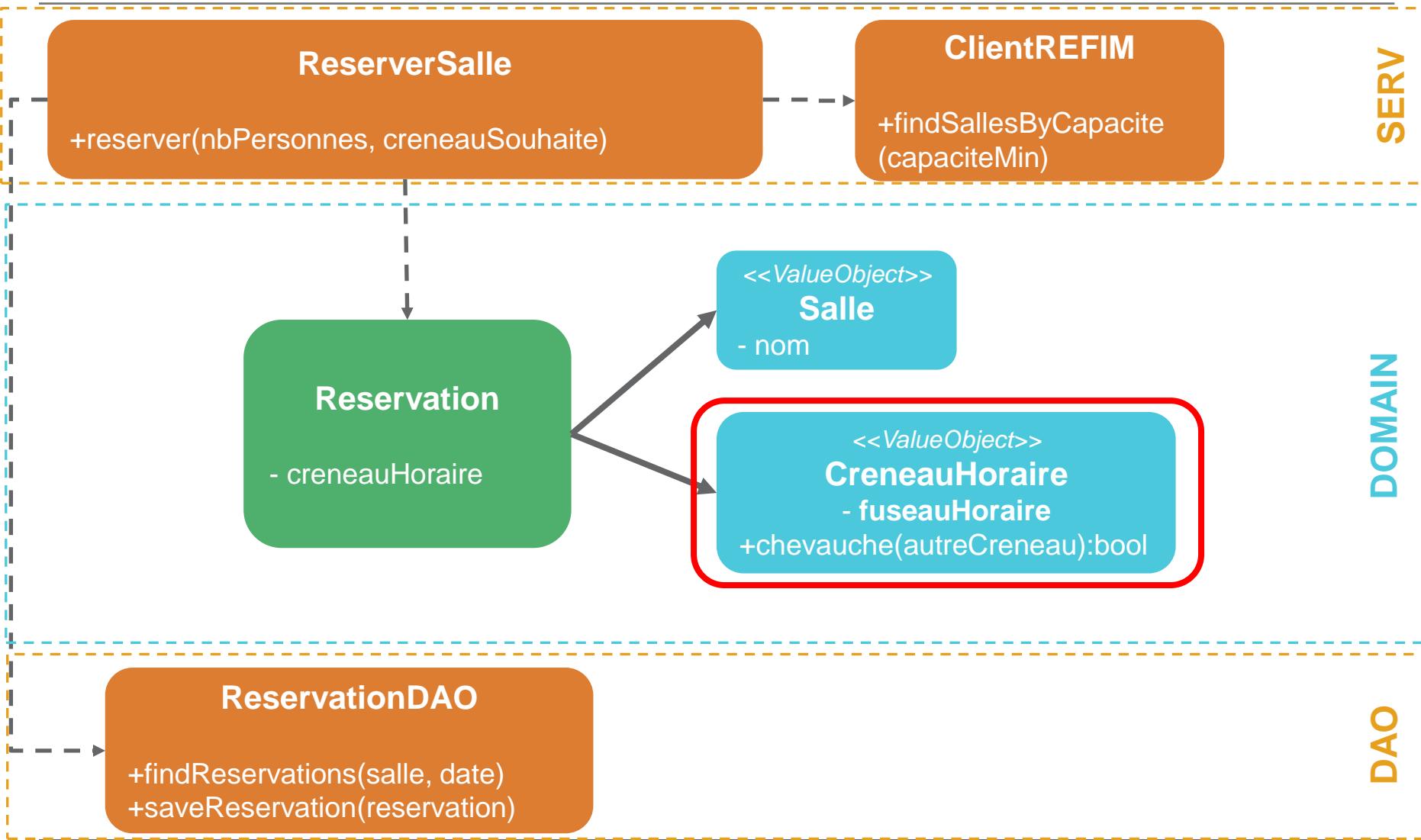
COMPORTEMENT propre au domaine



# Reprise du modèle avec le VO Salle



# Ajout d'un champs FuseauHoraire



# Caractéristiques des Value Object

- Mesure, quantifie, ou décrit une caractéristique d'un autre objet
- Quantité, ou description, ou type code/lookup
  - Enums java excellentes pour ces derniers
- Sans identité / pas une instance / pas d'historique
  - Simplification du modèle mental
  - Simplification technique
    - ORM: simplification du mapping et du cascade
    - BD: évite de créer un graphe de tables complexe
- Ne peut être distingué d'un VO de même valeur / Est remplaçable
  - Java/C#: Doivent toujours surcharger equals/hashCode
- Immutable
  - Doit être bien formé dès sa construction



# VO autopортант

- N'a pas besoin de dépendance vers d'autres Value Object
- Possède un comportement riche
- Des exemples:

**CreneauHoraire**

+chevaucher(creneauHoraire)

**Pigment**

+melanger(pigment)  
+diluer(dilution)



**BNP PARIBAS**

La banque d'un monde qui change

# Utilisation correcte et incorrecte des VOs

## Incorrecte

Constructeur retournant un VO non-validé

Immutabilité pas complètement assurée (non trivial)

**VO anémique**  
getters/setters

**Role Creep**  
Trop de responsabilités et de dépendances

**Types trop nombreux**  
Un VO pour FirstName, un pour LastName, ...

## Correcte

Evite l'anti-pattern *primitive obsession*

Nommage métier

**Conceptual Whole**

Forme un tout conceptuel insécable: comme 50000 \$

**Side-effect-free behavior**

Absence d'effets collatéraux

Pas de modification

Les commandes retournent un nouveau VO

**Closure of operations**

Opérations internes au type (ex: plus, minus)

**Composable**

Distance + Duration = Speed



# Les VO ne sont pas des DTOs

	DTO	VO
Intention	Transfert de données	Représentation d'un concept du Domain Model
Contenu	Ensemble des données à faire transiter (réseau, passage entre couches, ..). Peut agréger des données hétérogènes pour un cas d'utilisation	Des données fortement cohérentes
Comportement	Pas de comportement	Comportement riche



# Les Entités



**①** De l'existant vers la cible DDD

## **② Briques DDD simples**

**①** Value Object

**②** Entité

**③** Composants DDD de type Service

**①** Application Service

**②** Domain Service

**④** Composants DDD de type Infrastructure

**①** Repository

**②** Infrastructure Service

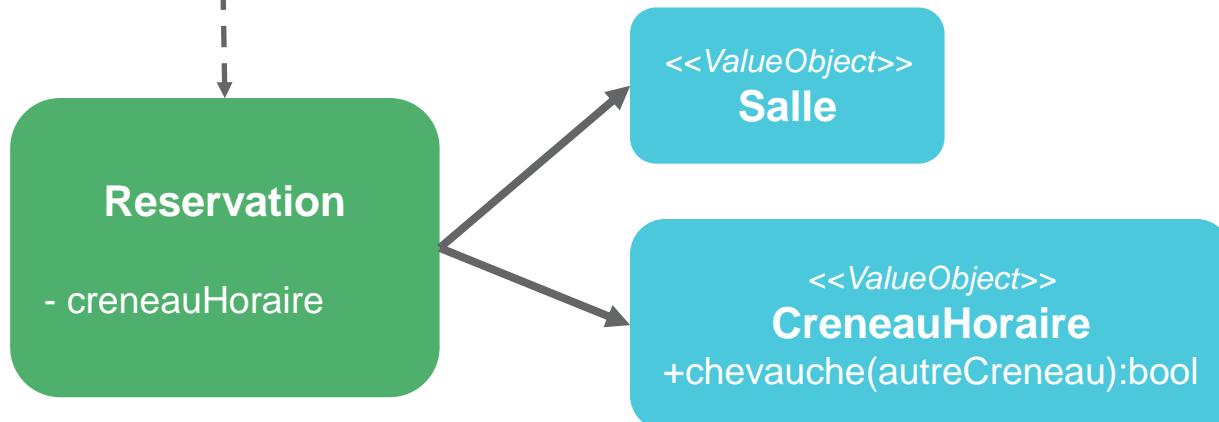
**⑤** Briques DDD plus avancées

**①** Agrégat

**②** Domain Event



# Rappel du modèle actuel après l'introduction des VO





# Ajout d'un état à une réservation

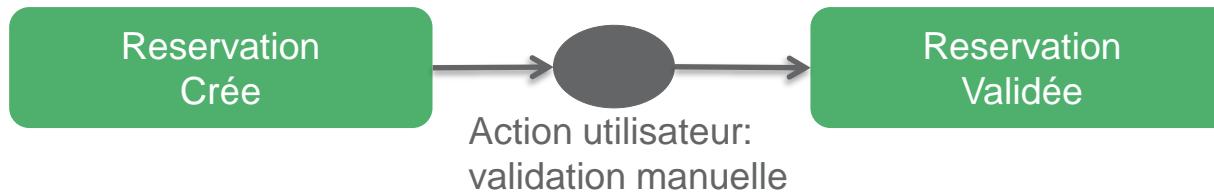
## Atelier:

*On veut rajouter un cycle de vie à la réservation:*

*Initialement, elle est temporaire*

*Si elle validée, elle devient définitive*

*Si elle est refusée, elle sera supprimée*



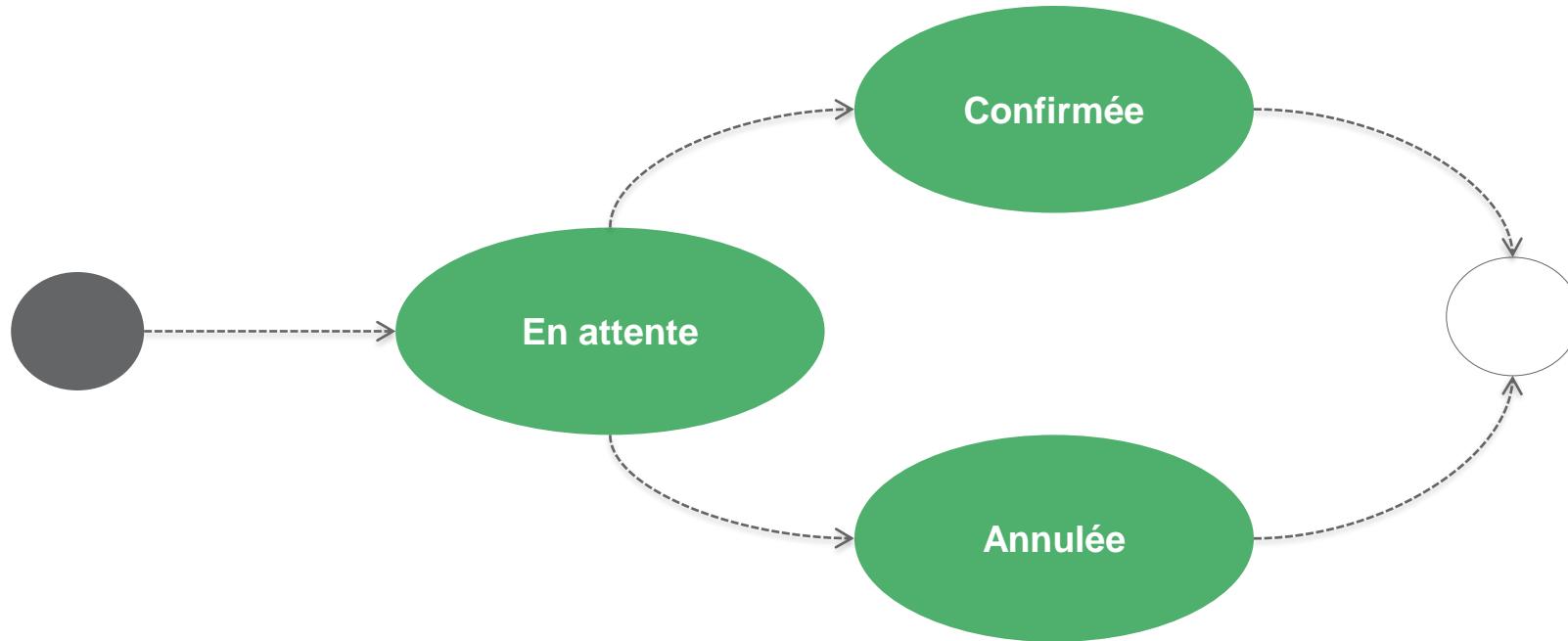
**Q1: Quelles méthodes ajouteriez-vous à *Reservation* pour implémenter les fonctionnalités ci-dessus?**

→ *Indication: donner de l'importance au nommage aux méthodes*





# Ajout d'un état à une réservation



3 états possible à la Réservation



**BNP PARIBAS**

La banque d'un monde qui change

# Les Entités

- A l'instar d'une personne, certains objets sont intrinsèquement définis par la continuité de leur identité et non par leurs attributs (qui eux peuvent varier dans le temps).
- **Identité et état variable** sont indissociables:
- L'identité est le point d'ancrage qui permet un suivi continu de l'objet dans le temps.
- Cette identité est incarnée par **un ou plusieurs champs métier ou techniques**, dont l'ensemble a la propriété d'**unicité**.



# Point commun entre les Value Object et les Entités

Types nommés suivant les concepts du Domain Model

Encapsulation de l'état

Les méthodes accédant à l'état ont une sémantique





# Entity vs VO : les distinguer n'est pas toujours trivial

## Atelier de distinction

« Entité ou VO ?»  
**Billet Banque**

« Entité ou VO ?»  
**Adresse**

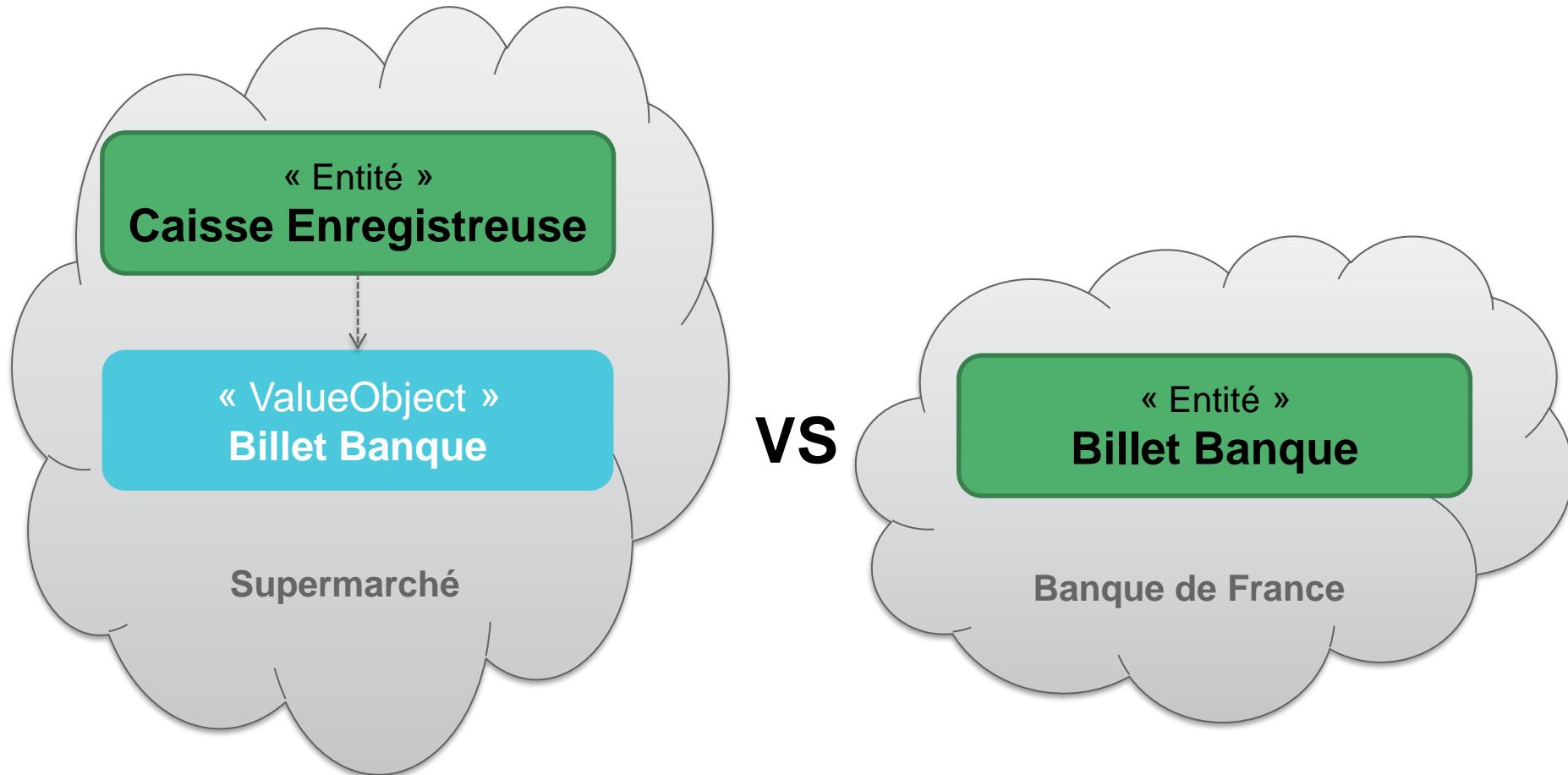


**BNP PARIBAS**

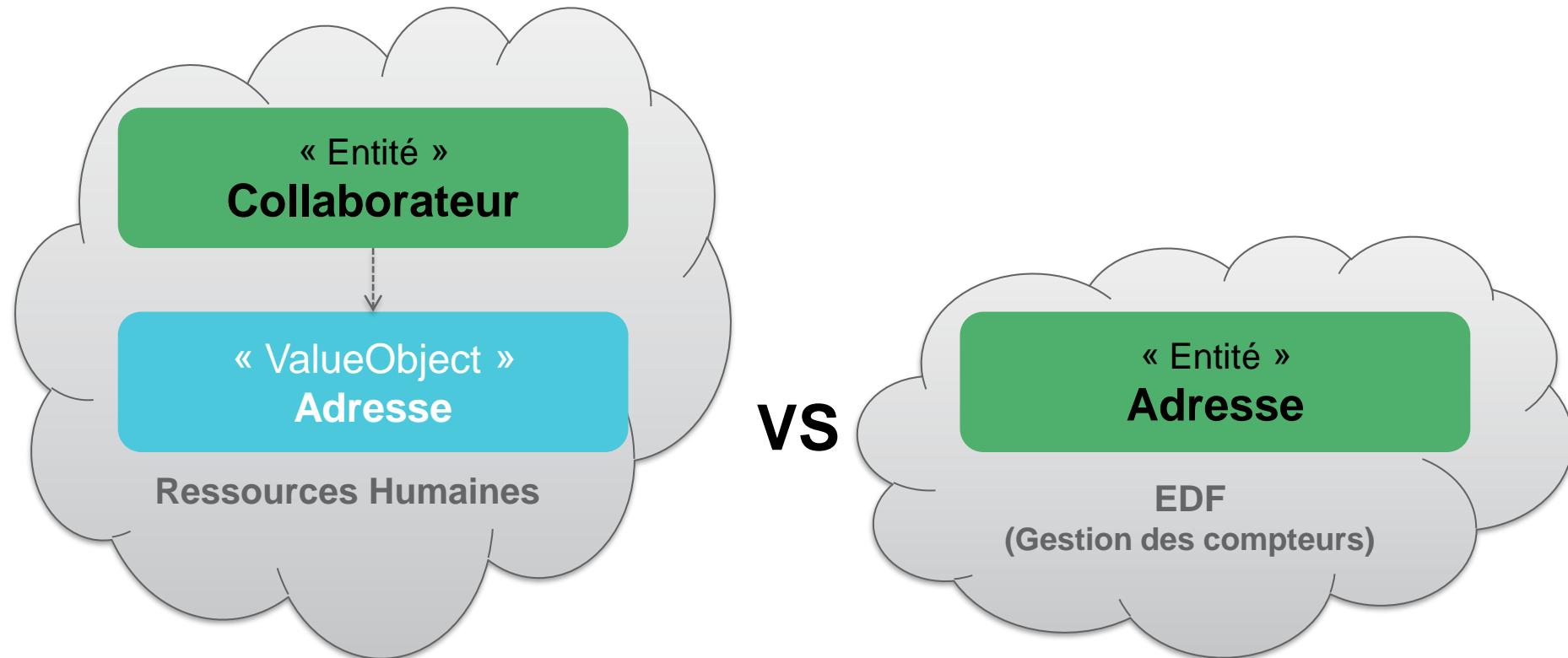
La banque d'un monde qui change



# Entity vs VO : Cas du Billet de Banque



# Entity vs VO : Cas de l'Adresse



# Entités sans comportement

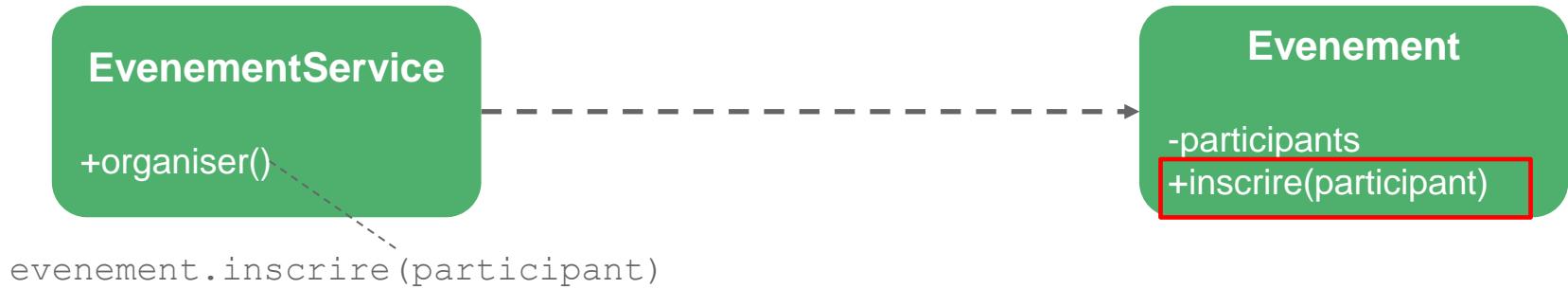
- **Conception à base de "structure de données" sans comportement:**
- l'état n'est pas encapsulé
- compréhension locale impossible sans examiner ses clients



# Entités avec comportement

## Conception à base d'Entités ayant un comportement riche:

- l'état est encapsulé
- compréhension locale possible grâce à une sémantique métier



# Contenu d'une Entité

## EGALITE: fréquemment

Cas typique: sérialisation, sauvegarde en BD, ...  
Dans ce cas le critère doit être un VO de type ID

## ETAT

(varie dans le temps)

## RELATIONS

avec d'autres Entités (associations)

## COMPORTEMENT METIER

utilisant l'état encapsulé



# Entité vs Value Object (VO)

	Value Object	Entité
Caractéristique principale	Ses attributs	Son identité
Comportement lors d'un changement d'état	Production d'un nouveau VO	Mutation de la même Entité
Critère d'égalité	Suivant sélection métier d'un ensemble d'attributs	Suivant référence (en mémoire) ou VO ID
Comportement temporel	Intemporel	Variation de l'état dans le temps
Autonomie (vis à vis d'un cas d'utilisation)	Ne peut pas être utilisé seul, caractérise une Entité.	Peut être utilisée seule
Composition	Ne peut agréger que d'autres VO	Peut agréger d'autres Entités et VO





# Ajout d'un état à une réservation

---

## Reprise Atelier:

*On veut rajouter un cycle de vie à la réservation:*

*Initialement, elle est temporaire*

*Si elle validée, elle devient définitive*

*Si elle est refusée, elle sera supprimée*

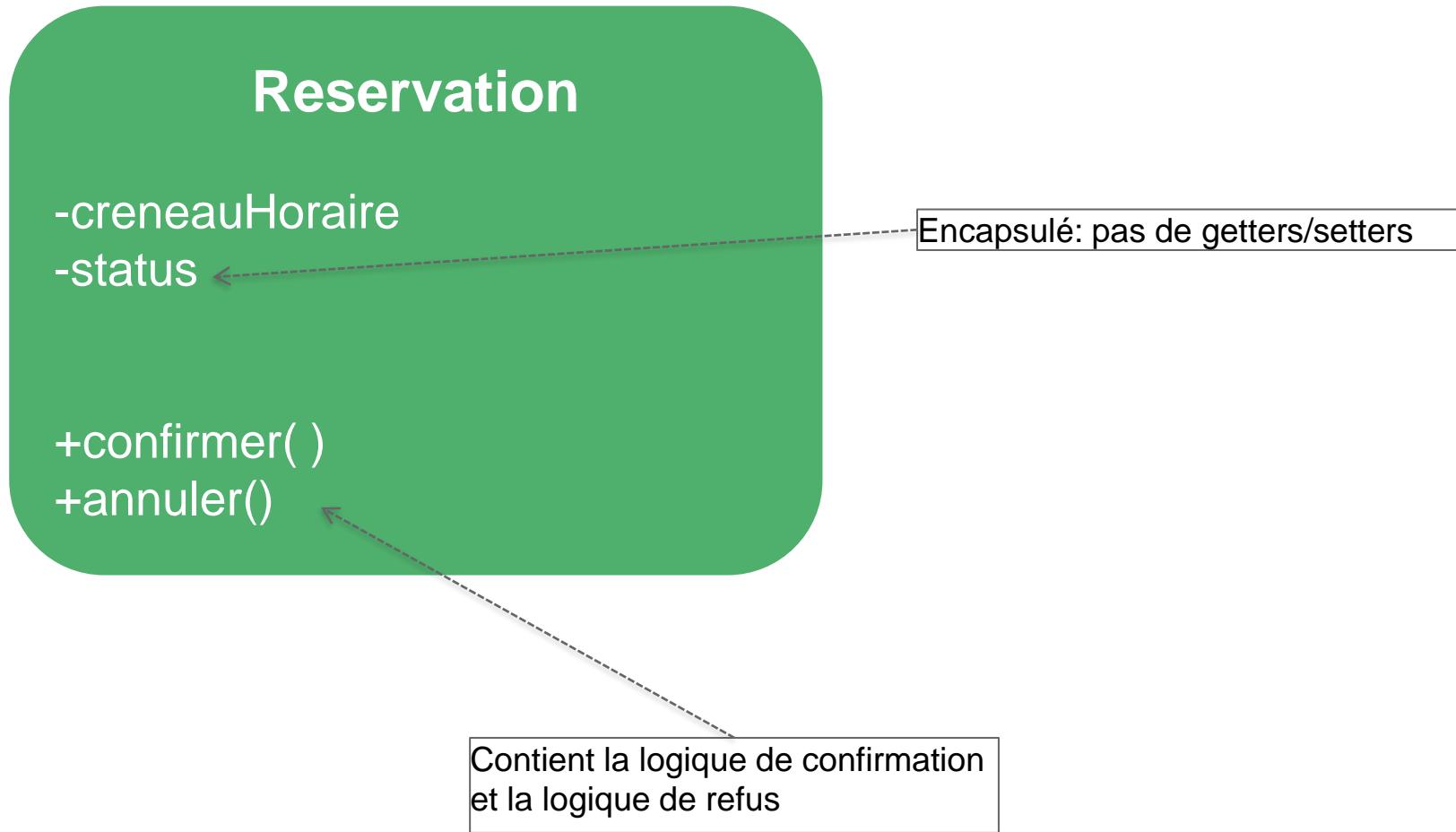
*Rappel Q1: Quelles méthodes ajouteriez-vous à Reservation pour implémenter les fonctionnalités ci-dessus?*

- →*Indication: donner de l'importance au nommage aux méthodes*





# Ajout d'un état à une réservation

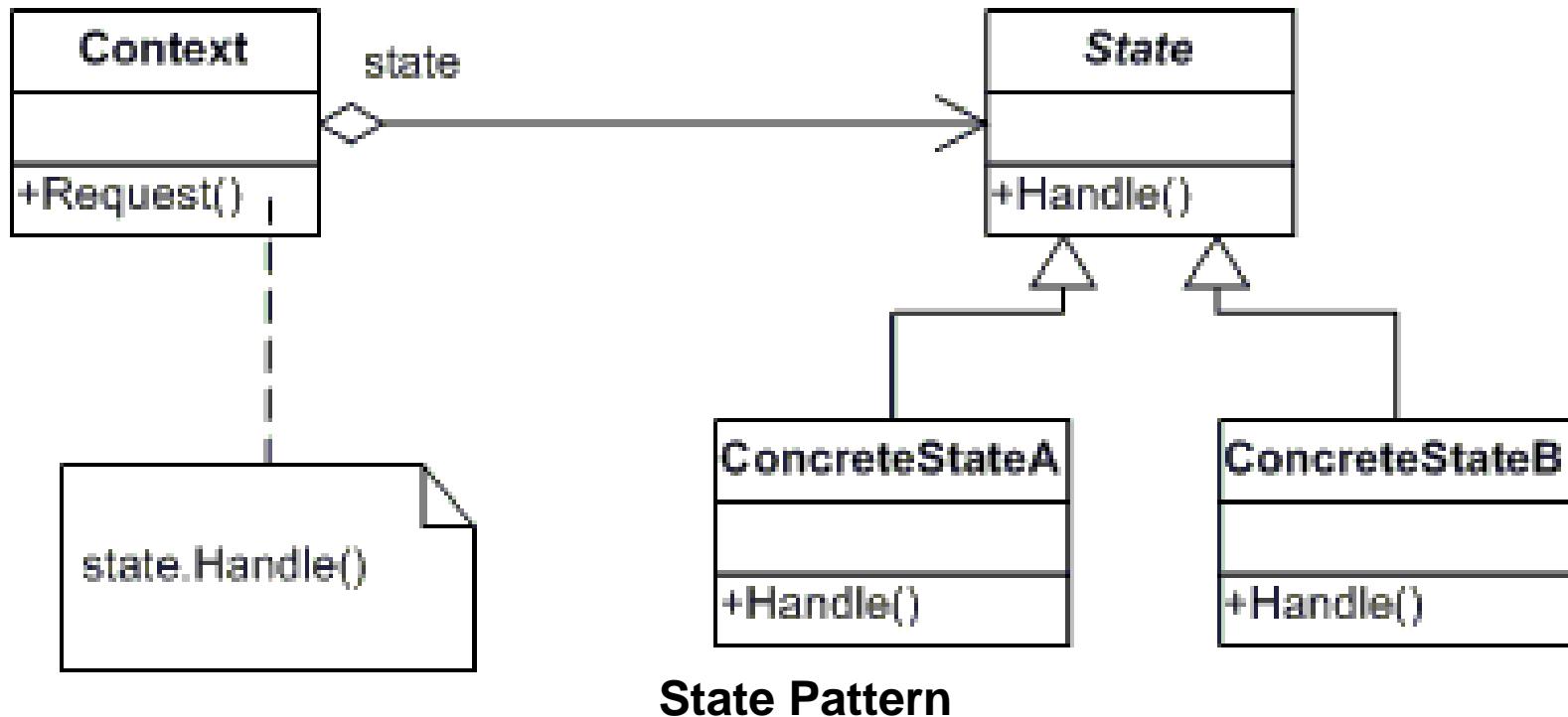


**BNP PARIBAS**

La banque d'un monde qui change

# Cas du Design Pattern State

- Pour implémenter le comportement précédent, le pattern State est souvent utilisé



*Design Patterns, GOF*



**BNP PARIBAS**

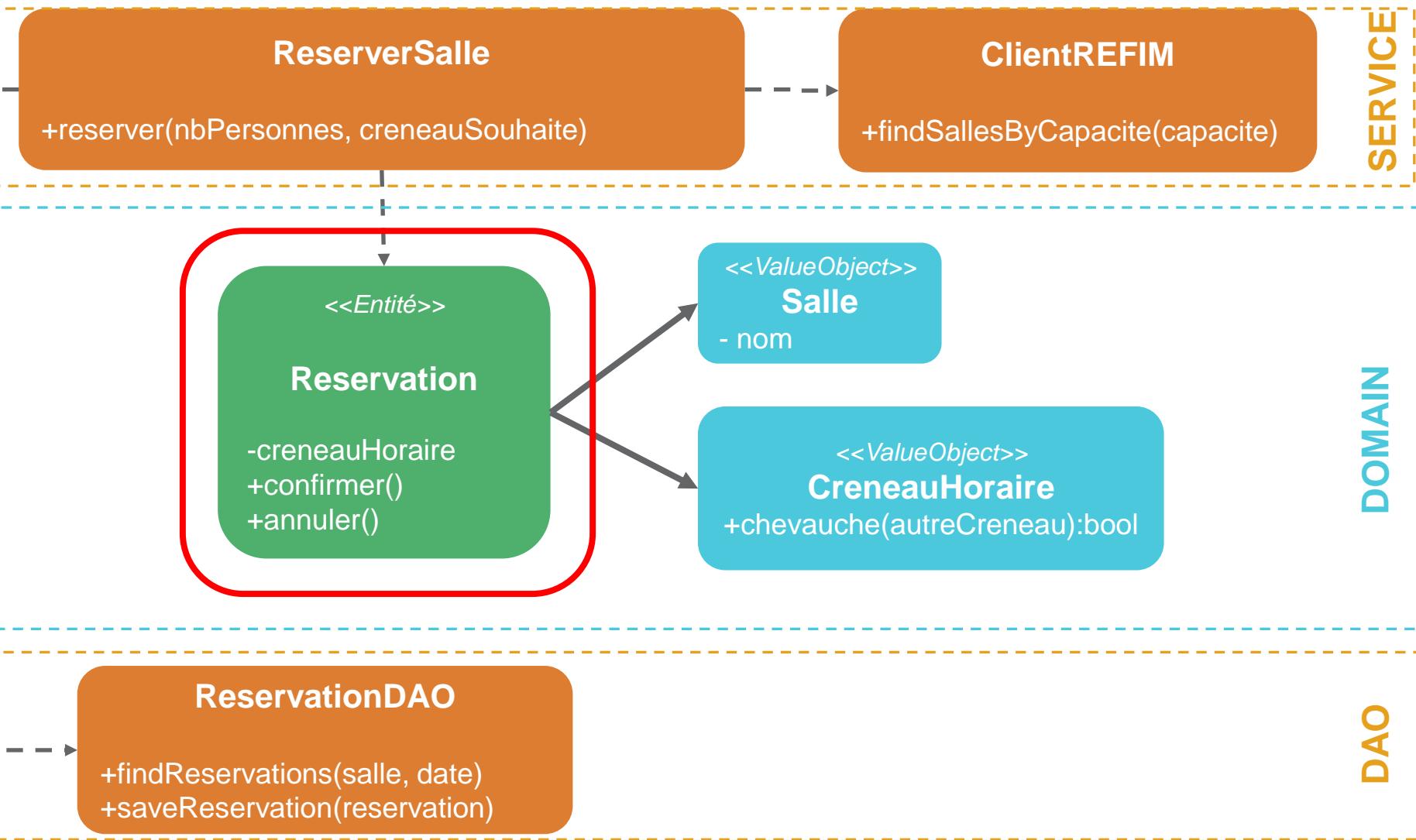
La banque d'un monde qui change

# Transformation de Reservation en Entité

SERVICE

DOMAIN

DAO



# Utilisation des Entités

## Incorrecte

### Confusion avec un VO

Avoir une identité ou pas peut dépendre du contexte (ex: billets de banque)

### Entité anémique

- Setters génériques violant l'encapsulation
- Export de l'état pour une utilisation externe

### Forme un graphe d'associations complexe avec beaucoup d'autres Entités

- Peut engendrer des difficultés techniques importantes (cf. Agrégats)

## Correcte

### Utilisent l'ubiquitous language

- Entités nommées suivant les noms communs du métier
- Méthodes nommées suivant les verbes du métier

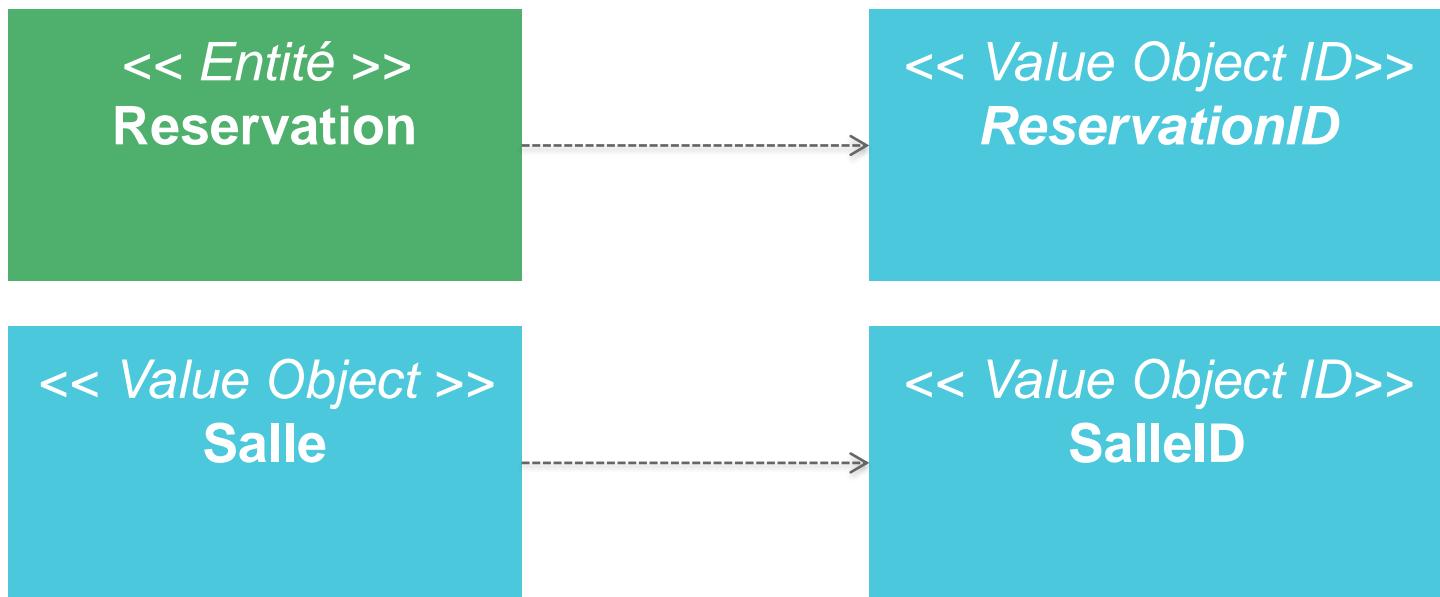
### Entité riche

- Etat modifié par des méthodes ayant une sémantique métier, et non par des mutateurs
- Etat "consommé sur place", à l'intérieur de l'entité



# L'identité d'une Entité

- Chaque entité doit avoir une manière précise d'établir son identité afin de se distinguer des autres identités ayant les mêmes attributs
- L'unicité de l'attribut d'identité doit être garantie dans tout le système
- **Incorporation** dans le domaine métier:



# Des exemples de VO ID

*<< Value Object ID>>*  
**ConferencelD**

*<< Value Object ID>>*  
**ContratID**

*<< Value Object ID>>*  
**PersonnelID**

*<< Value Object ID>>*  
**CommandeID**



# Stabilité de l'identité

- Afin de permettre de retrouver l'Entité par son identité, il est indispensable que cette identité soit affectée une fois pour toutes et ne change plus jamais: elle doit être **immutable**.
- Si cette identité est composée d'un seul type immutable (int, String, ...), il suffit de l'affecter à la création de l'Entité
- Si cette identité est composée de plusieurs types immutables, il est nécessaire de créer un objet qui les encapsule

# Les VO ID

- Un type particulier de VO peut être utilisé pour représenter l'identité d'une Entité, tout en masquant les détails d'implémentation
- 1. Pour assurer la stabilité de l'identité**
    - Le VO ID encapsule la préoccupation d'unicité de l'Entité
  - 2. Pour encapsuler la règle de création de l'identité**
    - Il existe plusieurs politiques possibles, potentiellement complexes, et il donc souhaitable d'encapsuler cette complexité
  - 3. Pour expliciter le type d'Entité identifié**
    - Utiliser directement les types int/string/... ne permet pas de distinguer cette identité de celle des autres types d'Entités et peut donc causer des erreurs.
    - Le VO ID est fortement typé et évite donc cette confusion



# Caractéristiques des VO ID

Critère unique d'égalité des entités

Unique dans un Bounded Context

Immutable

Fortement typé

Peut être sérialisé sous forme de champs int/String/... (persistance, réseau)

Typiquement moins de comportement que un VO générique mais peut inclure de la validation métier sur la valeur d'identité (ex: ISBN)



# Typologie des VO ID en fonction des usages

## Par type de contenu

- 1 ou plusieurs champs métier
- UUID/GUID
- Numéro de séquence

## Par provenance

- Fourniture par l'utilisateur
- Généré par l'application
  - Fournit une identité concise
  - Pas lisible (excepté suivi de commande)
- Généré au moment de la persistance  
Implicite en étant créé tard lors de la persistance  
Explicite mais complexe
- Depuis un autre BC  
Le user choisit une entité suivant un critère

## Par instant de construction

- Précoce  
Dès la création de l'objet
- Tardive  
Au moment de la persistance: la stabilité n'est pas assuré tout le temps. Ce cas est à éviter sauf bonne raison à expliciter dans les commentaires à minima.



# Objectifs



- ① De l'existant vers la cible DDD
- ② Briques DDD simples
  - ① Value Object
  - ② Entité
- ③ Composants DDD de type Service**
  - ① Application Service**
  - ② Domain Service
- ④ Composants DDD de type Infrastructure**
  - ① Repository
  - ② Infrastructure Service

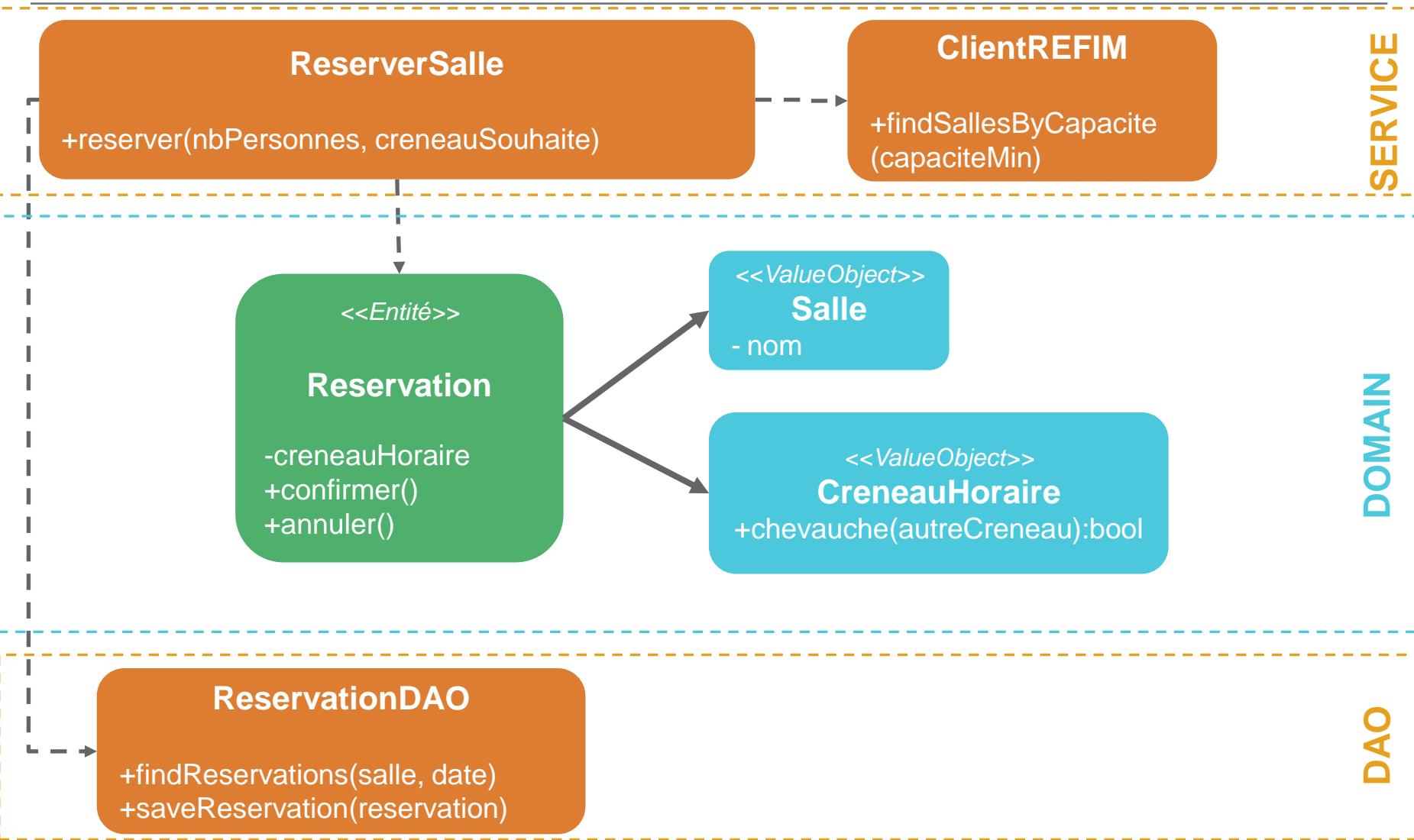


# Rappel: Modèle actuel

SERVICE

DOMAIN

DAO



# Refactor: antipattern fat controller

## Atelier:

*Actuellement, ReserverSalle est un contrôleur MVC, qui présente les problèmes suivants:*

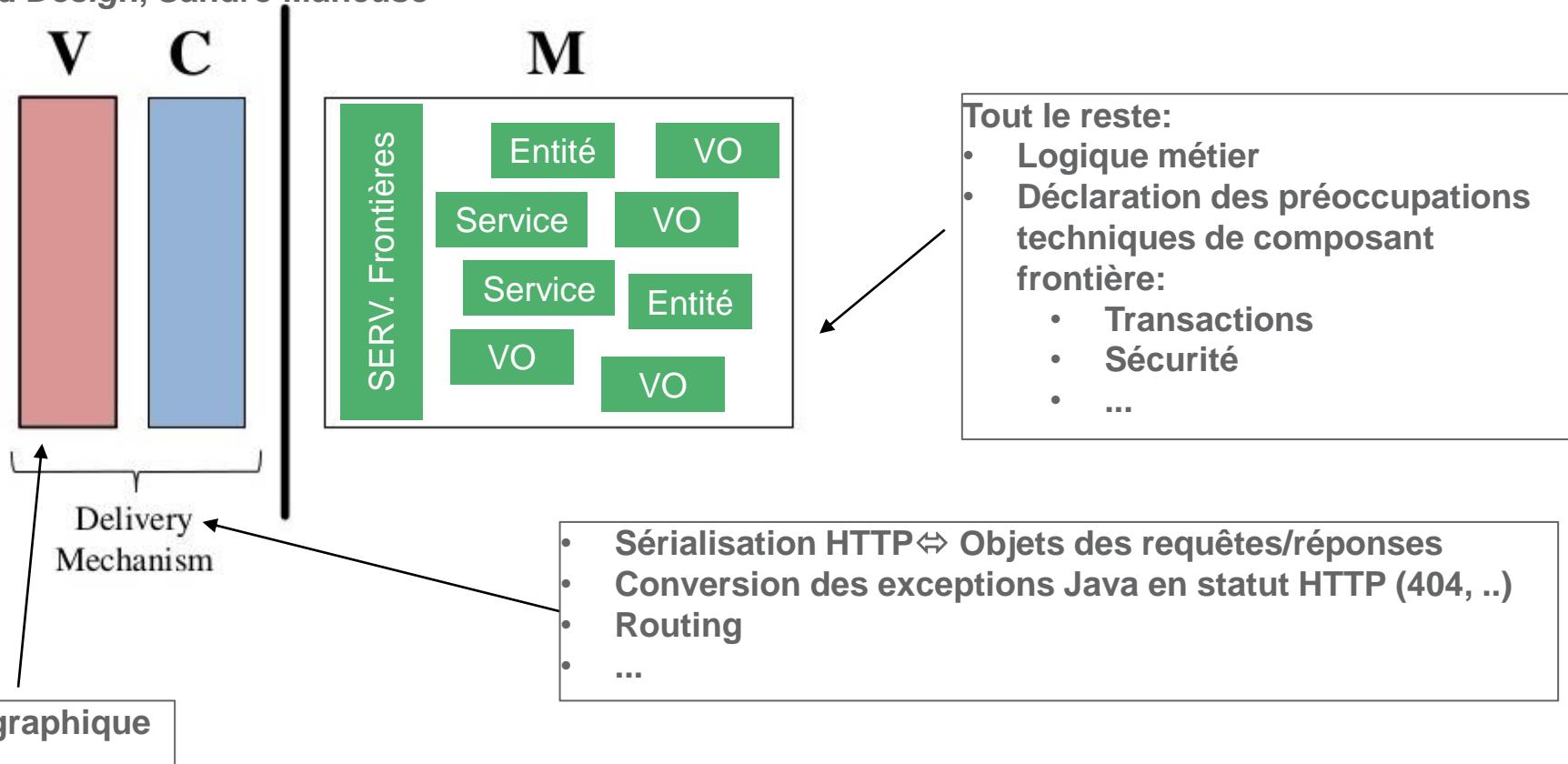
- *La logique métier est fortement couplée au framework MVC*
- *La logique métier est difficile à tester*
- *Le code est peu lisible*
- *Les montées de version du frameworks présentent un risque de régression*

**Q1: Proposez un refactor**



# MV\* dans un contexte DDD

*Crafted Design, Sandro Mancuso*



**Le modèle n'est pas le modèle de persistance**

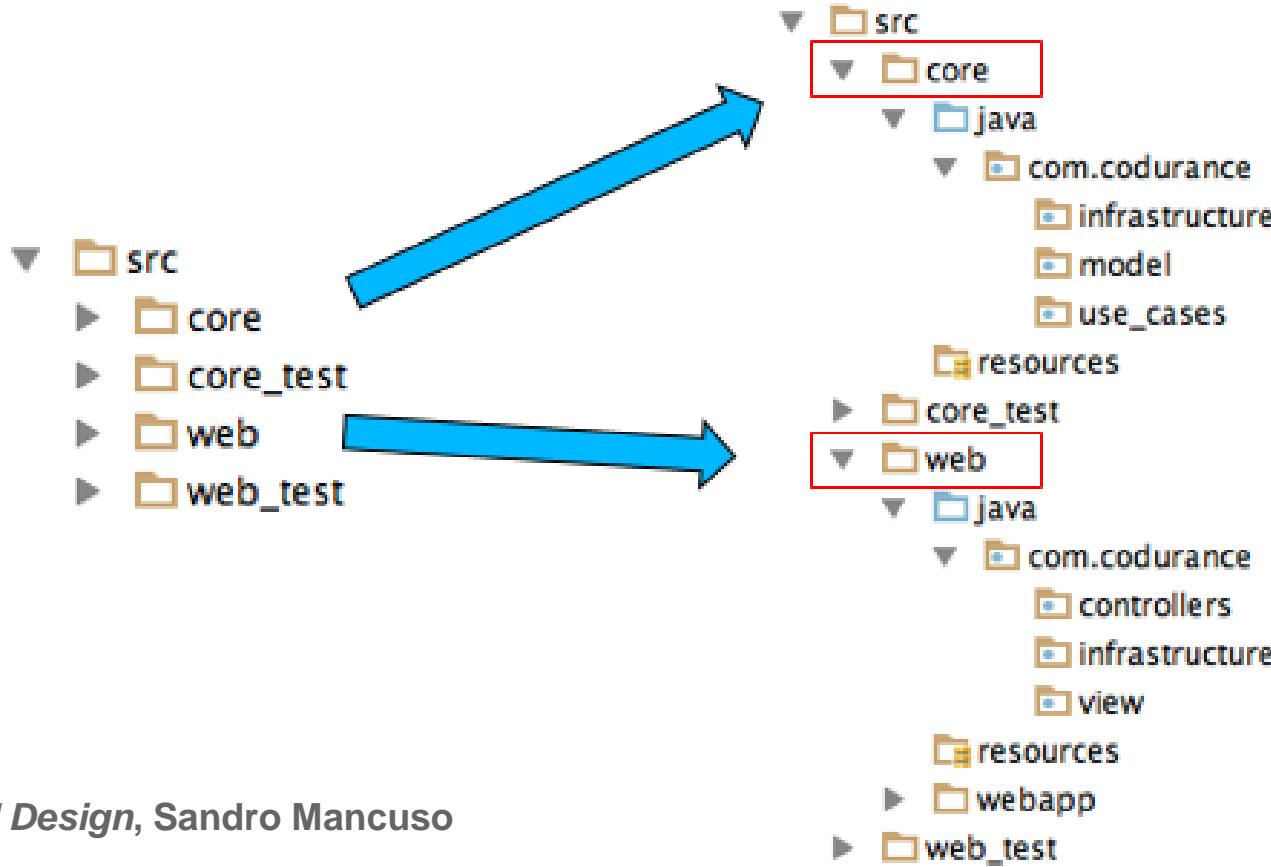


**BNP PARIBAS**

La banque d'un monde qui change

# Séparation des Application Services (AS) du framework

So, how does the app structure look like?



*Crafted Design, Sandro Mancuso*



BNP PARIBAS

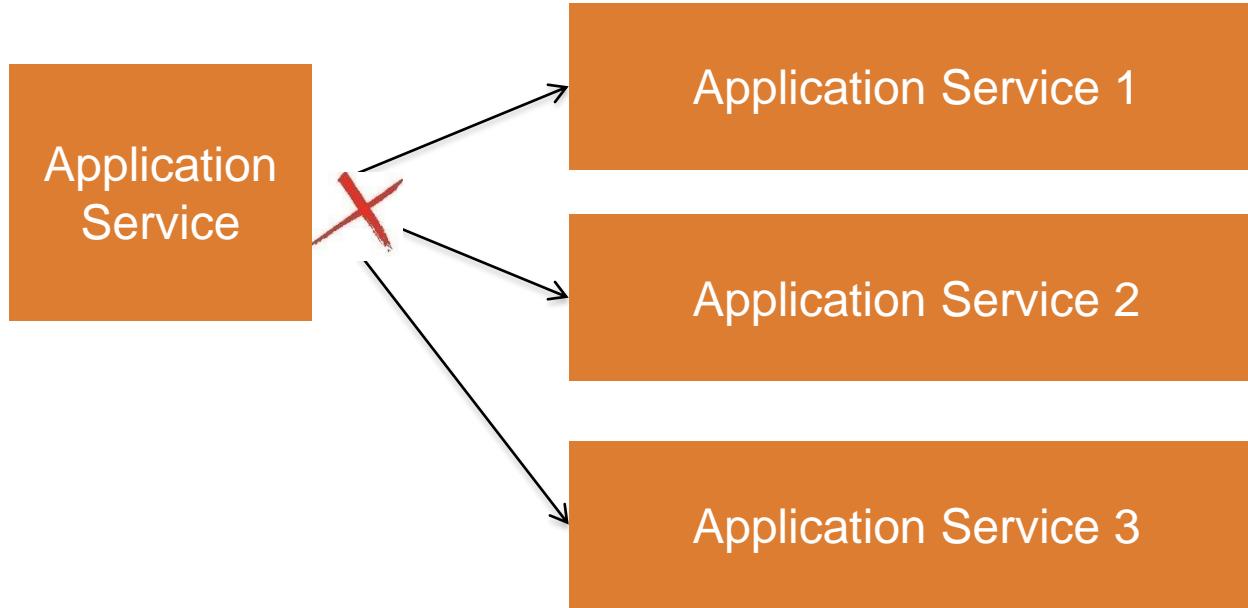
La banque d'un monde qui change

# Application Service & Uses Case

- Chaque use case va s'incarner par un Application Service
  - Granularité plus épaisse
- Utilisation d'une forme verbale (Verbe + Complément d'objet)
  - Notion d'Action
- La transaction métier est au niveau du use case



# Un application Service (AS) n'est pas composé d'autres AS



On ne compose pas les Application Service entre eux (*pas auto-composable*)  
Chaque Application Service représente une granularité grosse: le use case



# Contenu d'un Application Service

Des méthodes correspondant à des **use cases**  
(avec la même **granularité**)

La déclaration des préoccupations d'un **composant frontière**

Sécurité, démarcation transactionnelle, ..

La coordination (spécifique au use case) de ces composants

Pas d'état



# Refactor: antipattern fat controller

## Reprise Atelier:

*Actuellement, ReserverSalle est un contrôleur MVC, qui présente les problèmes suivants:*

- *La logique métier est fortement couplée au framework MVC*
- *La logique métier est difficile à tester*
- *Le code est peu lisible*
- *Les montées de version du frameworks présentent un risque de régression*

**Q1: Proposez un refactor**

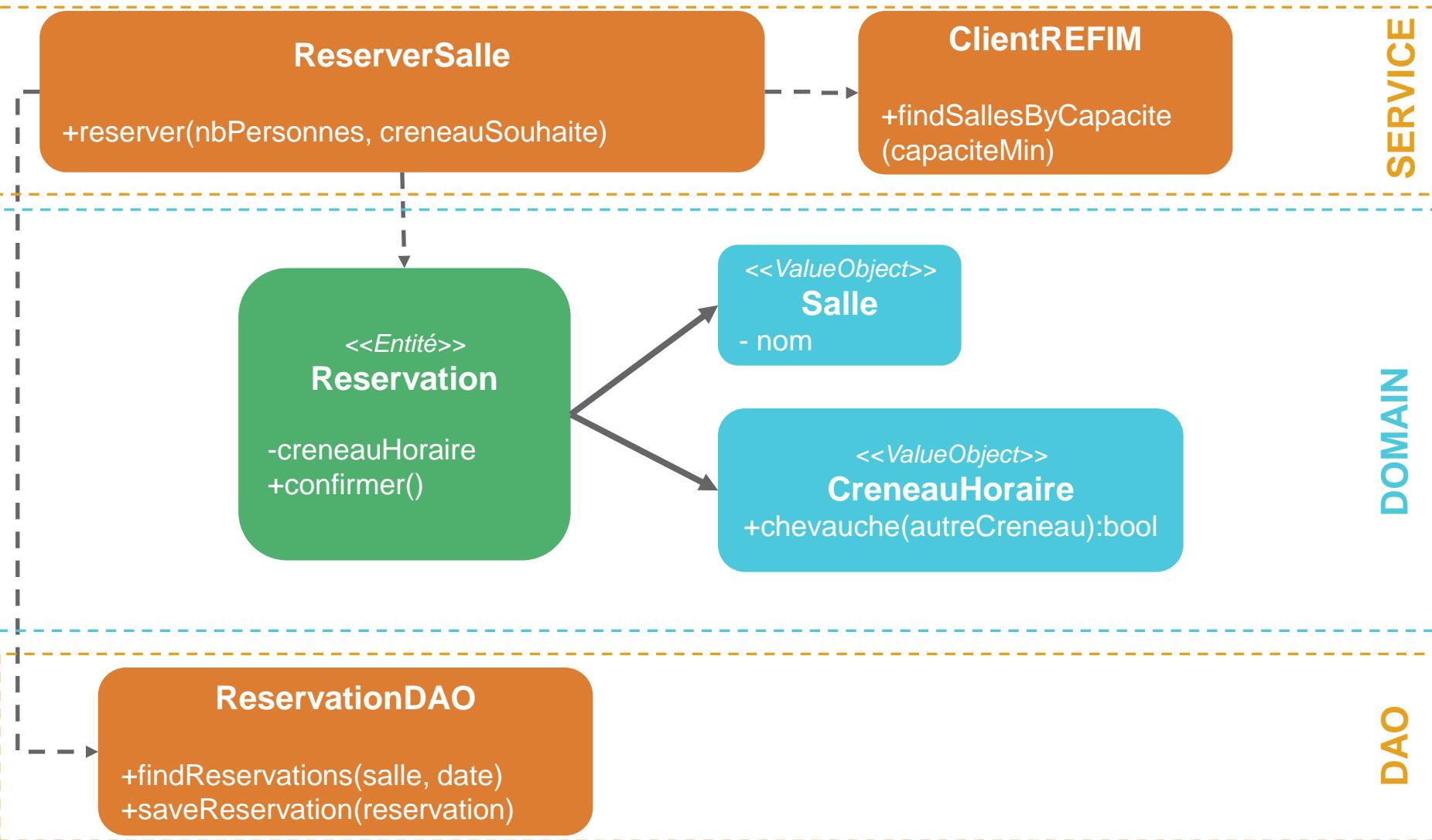


# Rappel du Modèle actuel

SERVICE

DOMAIN

DAO



# Modèle après extraction d'un Application Service



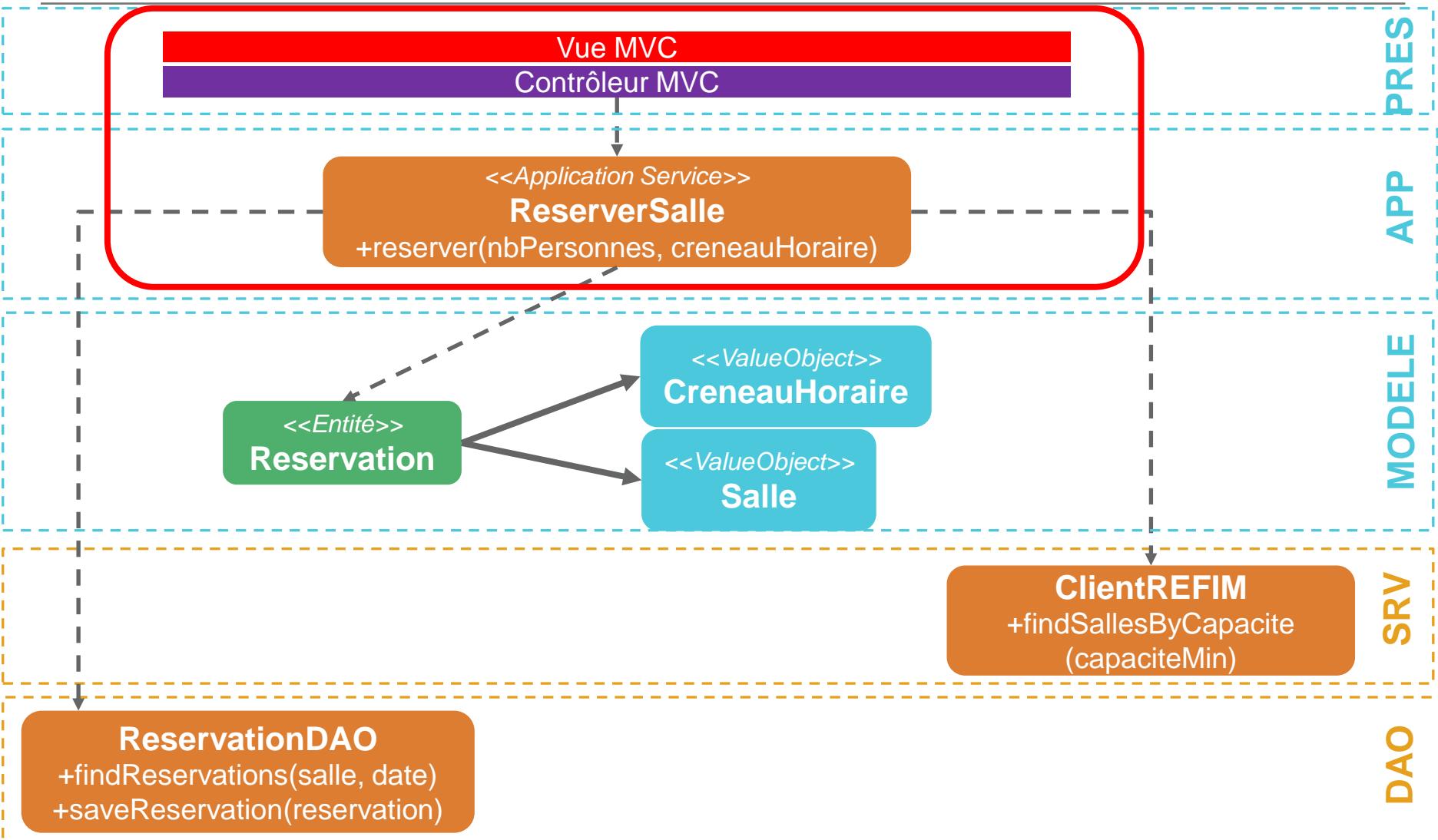
PRES

APP

MODELE

SRV

DAO



BNP PARIBAS

La banque d'un monde qui change

DDD Tactique (2/3) 167

# Utilisation des Application Services

## Incorrecte

### Mauvais nommage des méthodes

- Ne reprend pas le vocabulaire du use case
- Emploie un nom commun au lieu d'une forme verbale

### Oublie d'être un composant frontière

Les autres composants ont une granularité trop fine (ex: scope transaction)

### Granularité trop fine

Oblige le contrôleur MVC à appeler aussi un autre service

### Dépend d'un autre Application Service

Doit être autonome

### Faible cohésion entre les méthodes

Application Service fourre-tout

## Correcte

### Méthodes correspondant bien à un use case et nommés conformément à l'Ubiquitous Language

### Testé par des tests comportementaux (BDD)

Encore mieux: écrire ces tests d'abord

### Peu épais

Seulement de la coordination et déclaration des préoccupations de frontière

### Sans duplication avec un autre Application Service



# Objectifs



- ① De l'existant vers la cible DDD
- ② Briques DDD simples
  - ① Value Object
  - ② Entité
- ③ Composants DDD de type Service**
  - ① Application Service
  - ② Domain Service
- ④ Composants DDD de type Infrastructure
  - ① Repository
  - ② Infrastructure Service
- ⑤ Briques DDD plus avancées
  - ① Agrégat
  - ② Domain Event



*Sometimes, it just isn't a thing.*

—Eric Evans





# Rappel du modèle actuel

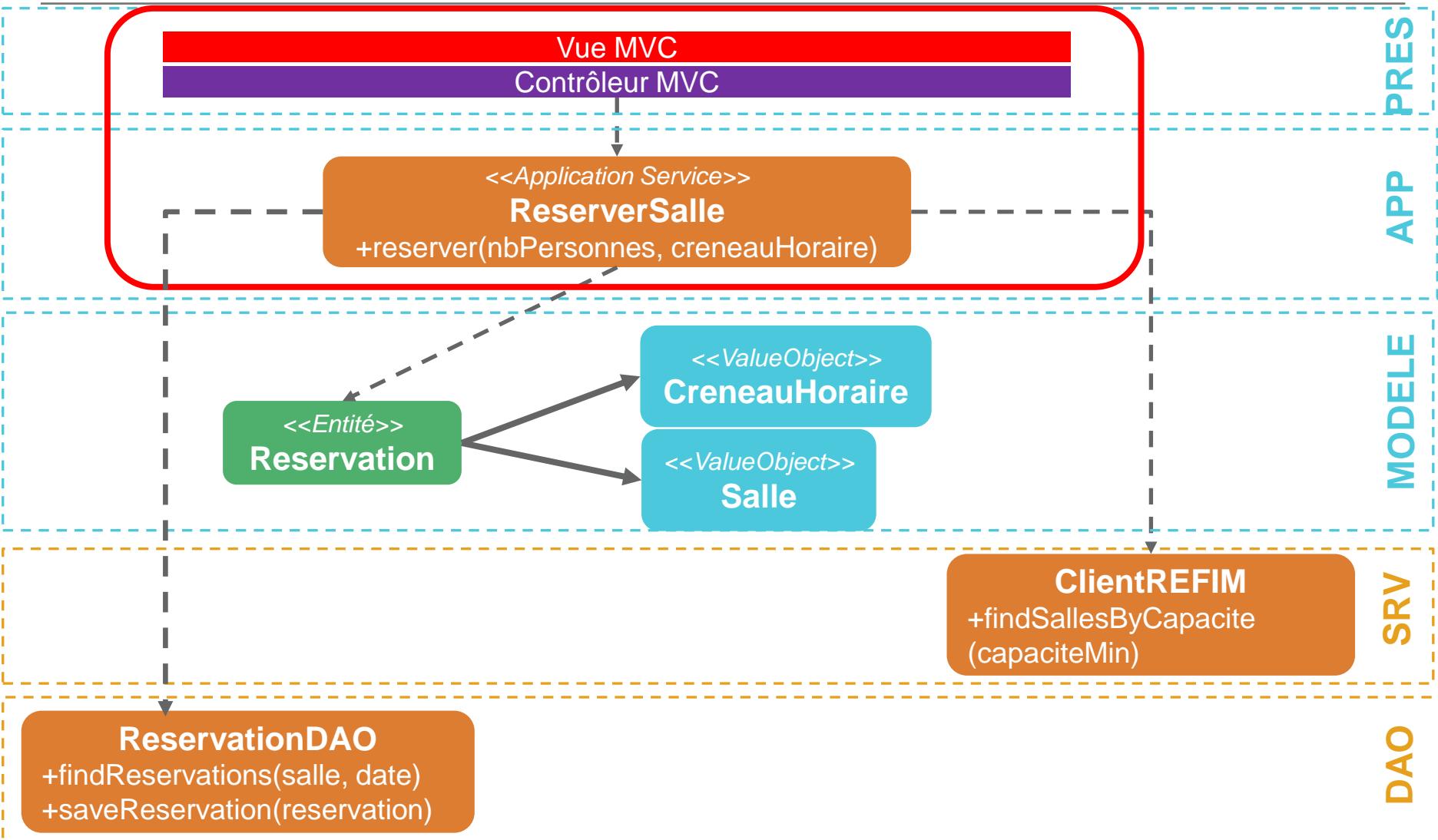
PRES

APP

MODELE

SRV

DAO



BNP PARIBAS

La banque d'un monde qui change



# Code de l'Application Service ReserverSalle

```
@ApplicationService
public class ReserverSalle {
    ...

    @Transactional
    @RolesAllowed("employe")
    public void reserver(int nbPersonnes, CreneauHoraire creneau) {

        clientREFIM.findSallesByCapacite(nbPersonnes) //Collection<Salle>
            .findFirst(salle -> reservationPossible(salle, creneau))
            .ifPresent(salle ->
                reservationDAO.saveReservation(new Reservation(salle, creneau))
            )
    }

    private boolean reservationPossible(Salle salle, CreneauHoraire creneauSouhaite ) {
        return reservationDAO
            .findReservations(salle, creneauSouhaite.getDate())//Collection<Reservation>
            .map(Reservation::getCreneau)
            .none(creneauReserve -> creneauSouhaite.chevauche(creneauReserve))
    }
}
```



# Focus sur le nommage des types «Application Service»

Pour ce concept DDD Application, plusieurs noms sont possibles  
2 typologies de noms sont généralement utilisées

<<Application Service>>  
**ReserverSalle**

<<Application Service>>  
**ReserverSalleActions**

1

Verbe

2

Suffixe « Actions »

Le plus proche du domaine  
Notre choix dans le reste de la formation!



BNP PARIBAS

La banque d'un monde qui change



# Refactor: Antipattern Fat Application Service

## Atelier:

*Nous avons désormais séparé les responsabilités MVC de l'Application Service.*

*Q1: Est-ce suffisant? Quelles pourraient être les problèmes qui persistent?*





# Refactor: Antipattern Fat Application Service

## Reprise Atelier:

**Q1: Est-ce suffisant? Quelles pourraient être les problèmes qui persistent?**

L'application Service (AS) implémente la quasi-totalité de la logique métier

- La séparation des préoccupations n'est pas respectée
- C'est l'antipattern ***abus de Transaction Script***

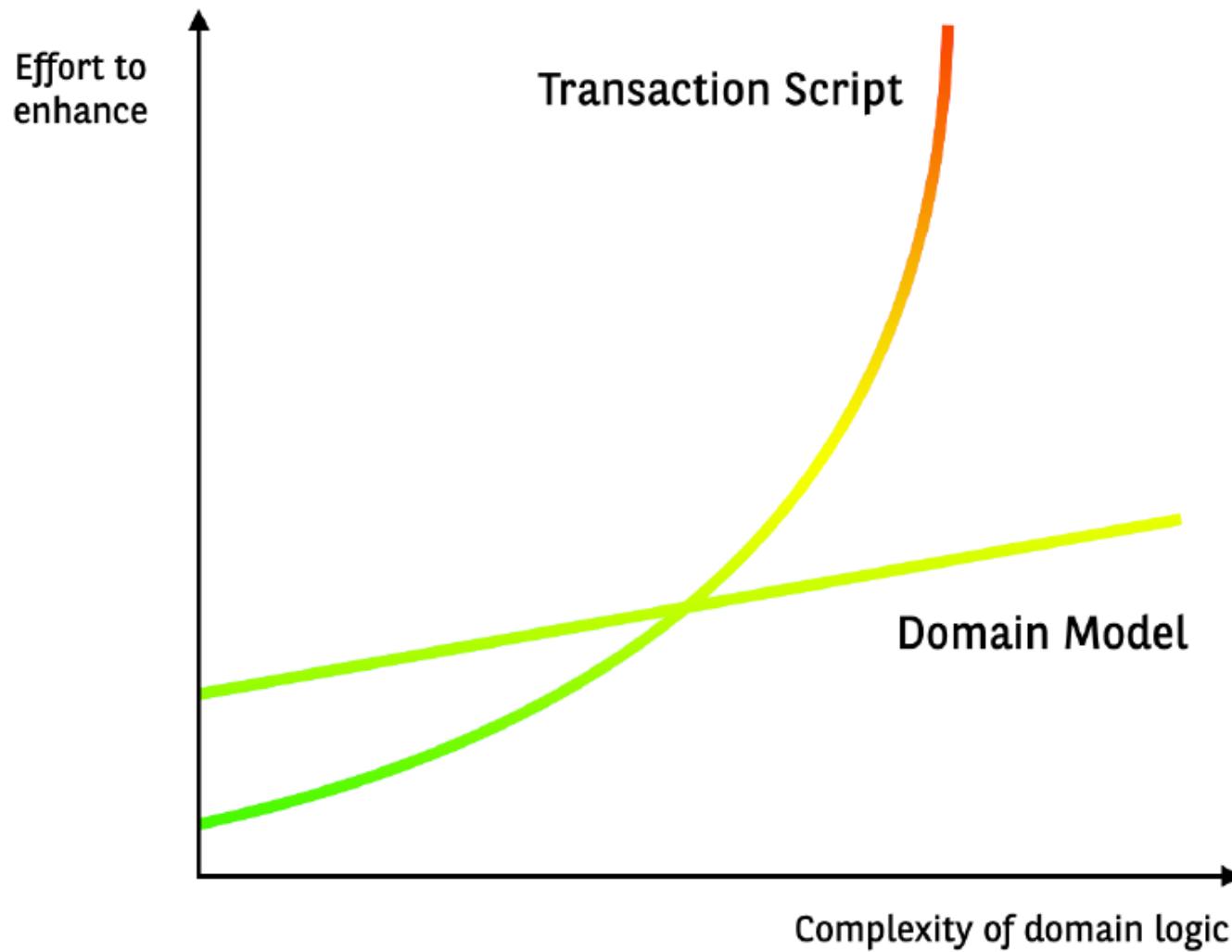


# Antipattern abus de Transaction Script

- Les Transaction Script sont souvent utilisés en raison de leur très grande simplicité de mise en œuvre
  - Naturel pour des applications avec très peu de logique métier
  - On a généralement un procédure Transaction Script par transaction de base de données
- Cependant ils deviennent problématique quand cette logique devient plus complexe
  - Le traitement de chaque requête est implémenté par une fonction longue et complexe
  - Cette logique est en partie dupliquée en la modifiant



# Domain Model vs Transaction Script





# Refactor: Antipattern Fat Application Service

## Atelier:

*Nous savons maintenant que l'Application Service ne respecte pas complètement la séparation des responsabilités*

*Q1: Quel concept pourrait être introduit?*



BNP PARIBAS

La banque d'un monde qui change



# Refactor: Antipattern Fat Application Service

## Reprise Atelier:

*Nous savons maintenant que l'Application Service ne respecte pas complètement la séparation des responsabilités*

*Q1: Quel concept pourrait être introduit?*

→ DDD introduit le concept de Domain Service



BNP PARIBAS

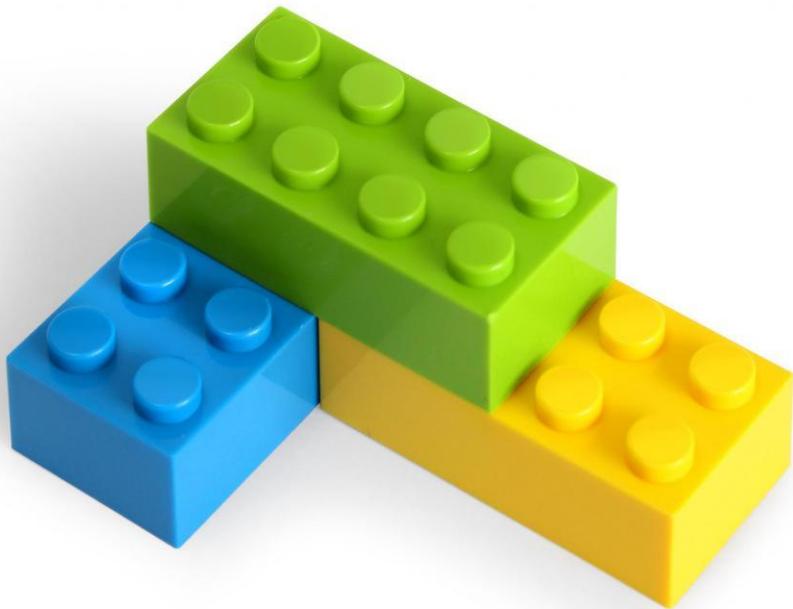
La banque d'un monde qui change

# Domain Service (DS)

- Un DS comporte des règles métier exprimées par la MOA
  - Processus métier significatif
  - Ubiquitous language évoquant un service et non une Entité
- Un DS implémente un concept du domaine qu'il n'est pas naturel de mettre dans une Entité ou un Value Object
  - Comportement à cheval sur plusieurs Entités
    - ex: transfert entre 2 comptes
  - Transformation d'un graphe d'Entités en un autre graphe d'Entités
  - ...



# Composition vs Orchestration de Domain Services



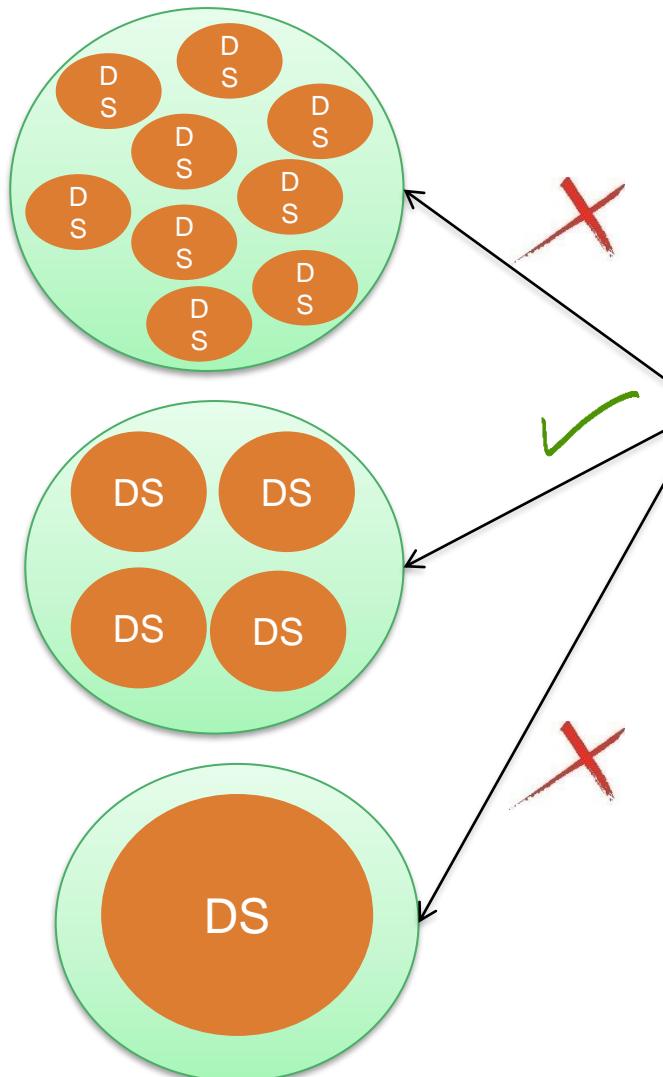
**Composition** d'un **Domain Service (DS)**  
**réutilisable** de plus gros grain, à partir de DS  
de plus petit grain



Un **Application Service (AS)** **orchestre** des DS.  
Il n'est pas réutilisable mais au contraire  
**spécifique à un use case**  
(Un AS utiliserait en fait beaucoup moins de  
"pièces")



# Granularité des Domain Service (DS)



**L'enjeu est de trouver la bonne granularité des composants Domain Service (DS)**

Quelques conseils pour trouver la granularité des DS:

- Partir de la connaissance du domaine
- Le DS doit survivre à plusieurs refactors (ex: > 3)
- **Mais rien ne remplace l'expérience du concepteur!!**



# Application Service (AS) vs Domain Service (DS)

	AS	DS
Intention	Coordonner les objets du domaine	Implémenter la logique métier
Granularité	Use case	Règle métier exprimée par la MOA, ou plus fine (algorithme)
Composant frontière	Oui	Non
Rôle dans la relation de dépendance	Utilisateur	Fournisseur
Auto-composabilité	Ne peut pas appeler d'autres AS	Peut appeler d'autres DS
Prend en paramètres de méthodes	<ol style="list-style-type: none"> <li>1. Des Entités et VO du Domain model</li> <li>2. Ou des types primitifs</li> <li>3. Ou un objet commande</li> </ol>	Des Entités et VO du Domain model

# Domain Service et état

- Contrairement à Entité/Value Object, un Domain Service n'a pas d'état interne propre: il est **Stateless**
  - *Cela va permettre de savoir comment on les compose*
- **Attention:** ce n'est pas une fonction pure!
  - Il n'est pas nécessairement "read-only"  
→ L'invoquer peut changer l'état de l'application
  - Il n'est même pas idempotent  
→ 2 invocations ne sont pas nécessairement équivalentes à 1 invocation





# Extraction du Domain Service

---

## Atelier:

*Q1: Quelles responsabilités peut-on laisser dans l'Application Service?  
Quelles responsabilités doit-on extraire dans un Domain Service?*





# Quel Domain Service extraire?

```
@ApplicationService
public class ReserverSalle {
    ...
    @Transactional
    @RolesAllowed("employe")
    public void reserver(int nbPersonnes, CreneauHoraire creneau) {
        clientREFIM.findSallesByCapacite(nbPersonnes) //Collection<Salle>
            .findFirst(salle -> reservationPossible(salle, creneau))
            .ifPresent(salle ->
                reservationDAO.saveReservation(new Reservation(salle, creneau)))
    }
}

private boolean reservationPossible(Salle salle, CreneauHoraire creneauSouhaite ) {
    return reservationDAO
        .findReservations(salle, creneauSouhaite.getDate())//Collection<Reservation>
        .map(Reservation::getCreneau)
        .none(creneauReserve -> creneauSouhaite.chevauche(creneauReserve))
}
}
```





# Extraction d'un Domain Service

```
@DomainService
public class ReservationService {

    private ClientREFIM clientREFIM;
    private ReservationDAO reservationDAO;

    public Optional<Salle> findSalleReservable(int nbPersonnes, CreneauHoraire creneau) {
        return clientREFIM
            .findSallesByCapacite(nbPersonnes)
            .findFirst(salle -> reservationPossible(salle, creneau))
    }

    private boolean reservationPossible(Salle salle, CreneauHoraire creneauSouhaite ) {
        return reservationDAO
            .findReservations(salle, creneauSouhaite.getDate())
            .map(Reservation::getCreneau)
            .none(creneauReserve -> creneauSouhaite.chevauche(creneauReserve))
    }
}
```





# Simplification de l'Application Service

```
@ApplicationService
public class ReserverSalle {

    private ReservationService reservationService;
    ...

    @Transactional
    @RolesAllowed("employe")
    public void reserver(int nbPersonnes, CreneauHoraire creneau) {

        reservationService
            .findSalleReservable(nbPersonnes, creneau)
            .ifPresent(salleReservable ->
                reservationDAO.saveReservation(new Reservation(salleReservable, creneau))
            )
    }
}
```





# Modèle après extraction du Domain Service

PRES

APP

MODELE

SRV

DAO



<<Application Service>>

**ReserverSalle**

+reserver(nbPersonnes, creneau)

<<Domain Service>>

**ReservationService**

+findSalleReservable  
(nbPersonnes,creneauHoraire)

<<Entité>>

**Reservation**

<<ValueObject>>  
**CreneauHoraire**

<<ValueObject>>  
**Salle**

**ClientREFIM**

+ findSallesByCapacite(capaciteMin)

**ReservationDAO**

+findReservations(salle, date)  
+saveReservation(reservation)



BNP PARIBAS

La banque d'un monde qui change

# Utilisation des Domain Service

## Incorrecte

Stateful

**Le comportement pourrait être mis dans une Entité ou un Value Object**  
Entraîne des Entités anémiques

**Interface superflue**

```
interface ReservationService  
class ReservationServiceImpl
```

## Correcte

Nommage du type et des méthodes suivant l'Ubiquitous language

**Testé isolément par des tests unitaires**  
Les dépendances sont remplacées par des bouchons dans ces tests



# Focus sur le nommage des types «Domain Service»

Pour ce concept DDD Application, on suffixe de « Service »



1

Suffixe « Service »

2

Autre nom

Souvent le choix par défaut

Notre choix dans le reste de la formation!



BNP PARIBAS

La banque d'un monde qui change

# Plan



- ① De l'existant vers la cible DDD
- ② Briques DDD simples
  - ① Value Object
  - ② Entité
- ③ Composants DDD de type Service
  - ① Application Service
  - ② Domain Service
- ④ Composants DDD de type Infrastructure**
  - ① Repository
  - ② Infrastructure Service



# Rappel du modèle actuel (version épurée)

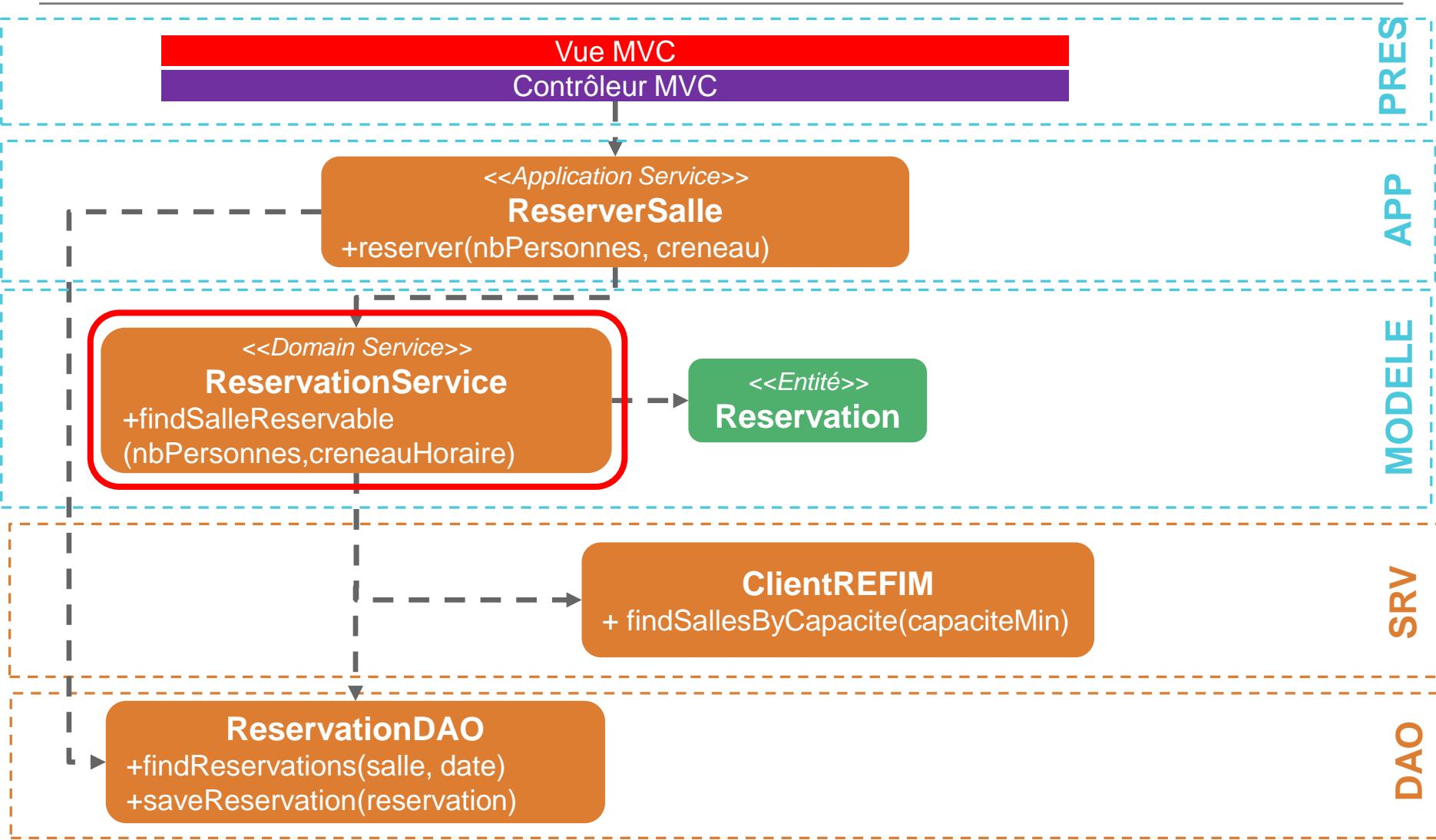
PRES

APP

MODELE

SRV

DAO





## Atelier:

- *Actuellement les Application Service / Domain Service dépendent directement de ReservationDAO*
- *Or ReservationDAO fait partie du package de persistance.*

**Q1: Pourquoi cette dépendance n'est pas acceptable?**



# Le problème de la dépendance envers la persistance

En dehors de la force de l'habitude, la difficulté essentielle est la suivante:

- Un comportement non-trivial nécessite souvent un accès à la persistance
- Or le modèle du domaine ne doit pas dépendre de la persistance:

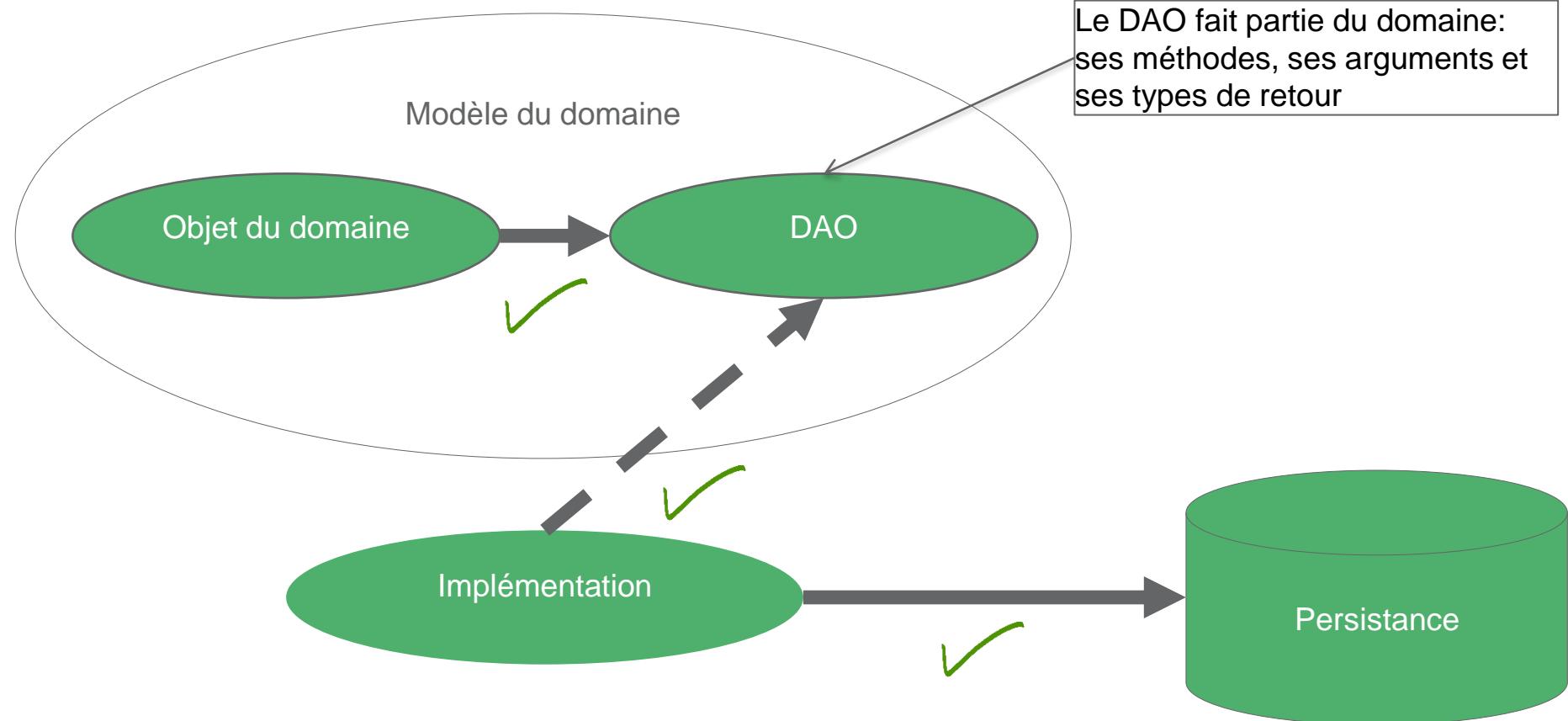
*Les modules stables , abstraits, et généraux ne doivent pas dépendre des modules contenant les détails d'implémentations, instables*



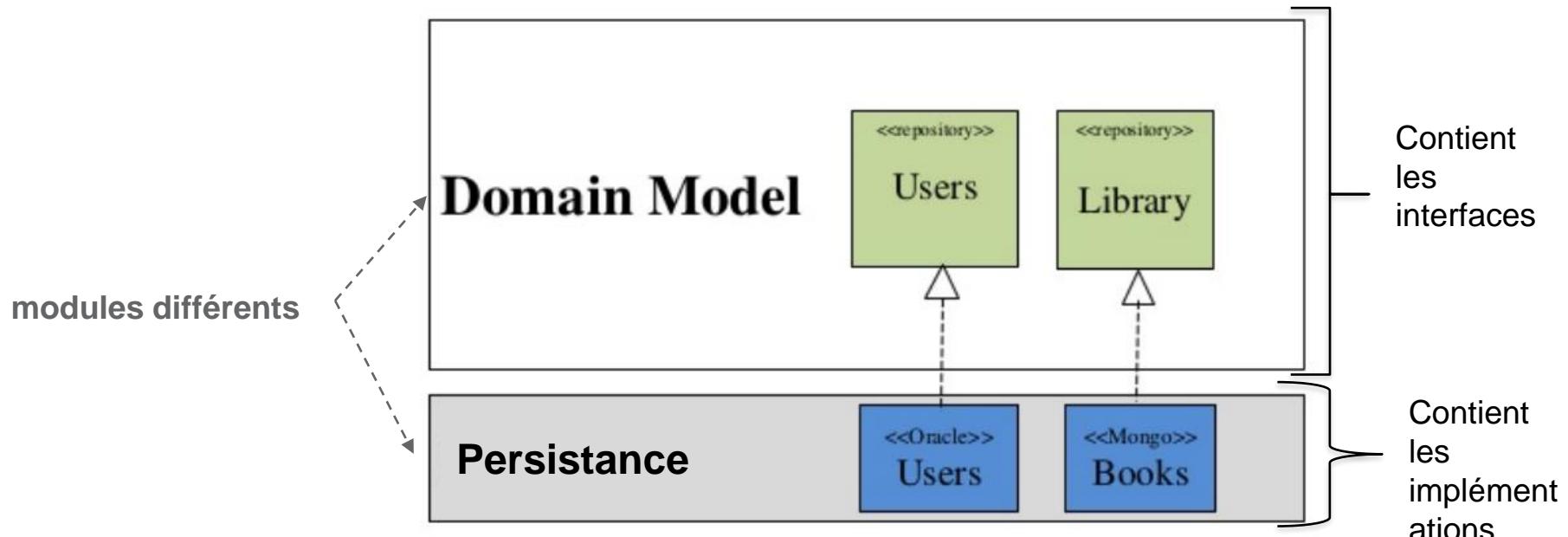
# Les Repositories: une solution pour casser la dépendance envers la persistance

Comment le modèle peut-il accéder à la persistance, sans induire de dépendance de la couche *modèle* vers la couche *persistance*?

- Avec le DIP



# Notion de *Separated Interface*



Crafted Design, Sandro Mancuso



BNP PARIBAS

La banque d'un monde qui change

## Reprise Atelier:

- *Actuellement les Application Service / Domain Service dépendent directement de ReservationDAO*
- *Or ReservationDAO fait partie du package de persistance.*

*Q1: Pourquoi cette dépendance n'est pas acceptable?*





# Application du DIP sur les DAO

MODELE

INFRA



**Introduction d'une interface dans le Domain Model  
et d'une implémentation spécifique dans la couche Infrastructure**



**BNP PARIBAS**

La banque d'un monde qui change



# Incorporation du DIP DAO dans le fil rouge

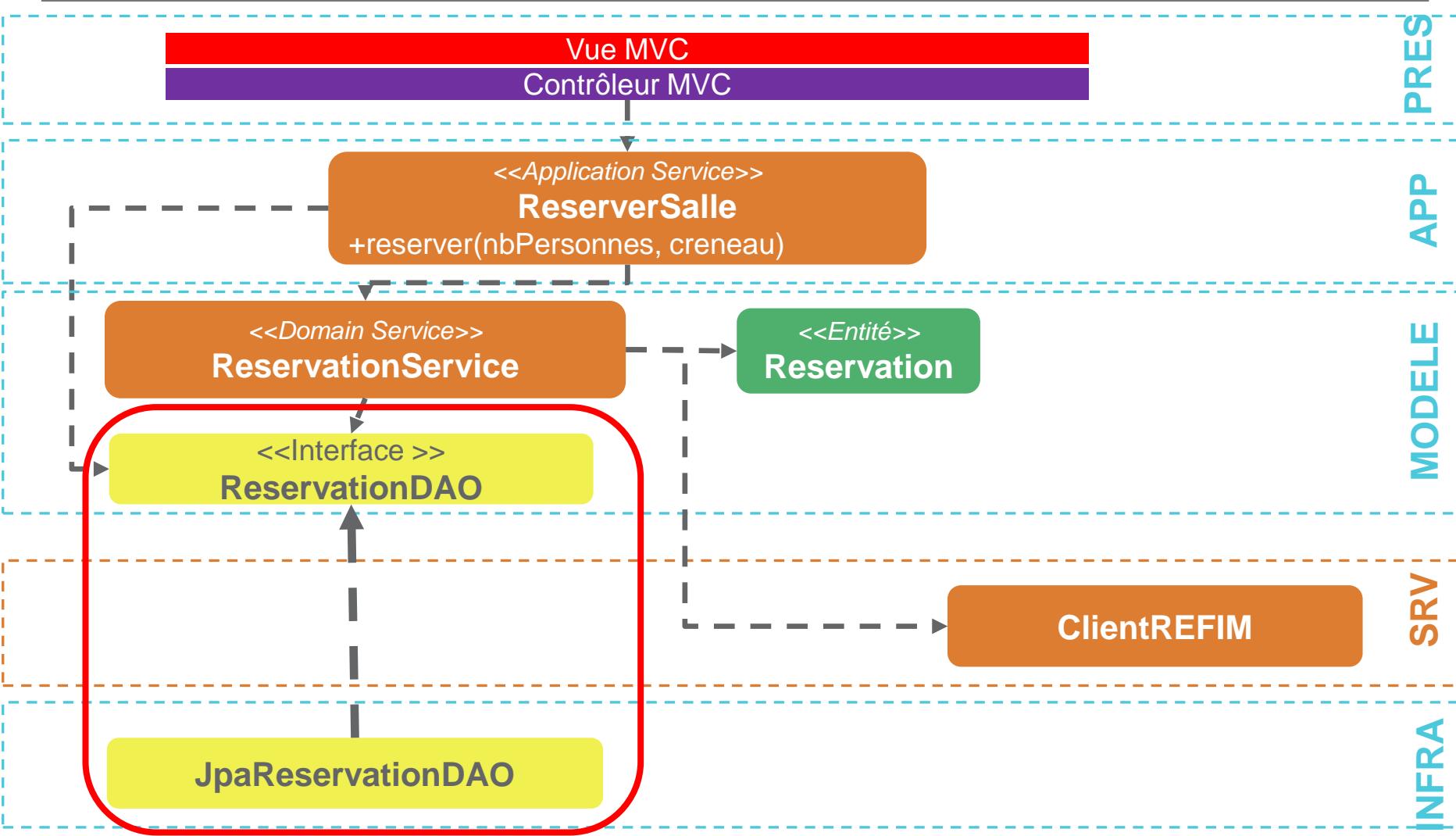
PRES

APP

MODELE

SRV

INFRA



BNP PARIBAS

La banque d'un monde qui change

DDD Tactique (3/3) 200



# Problématiques de sémantique du DAO

## Atelier:

*On a placé l'interface `ReservationDAO` dans le Domain Model pour casser la dépendance vers la couche persistance.*

*On s'interroge maintenant sur la sémantique de `ReservationDAO` :*

- 1. Que pensez-vous du nommage de `ReservationDAO` ?**
  
- 2. Même question pour les méthodes de `ReservationDAO`:**
  - `saveReservation(reservation)`
  - `findReservations(salle, date)`
  
- 3. Que remarque-t-on en regardant le contenu du module `domain.reservation`, qui comprend `Reservation` et `ReservationDAO`?**





# Problématiques de sémantique du DAO

## Solution Atelier:

**1.** *Actuellement, le nommage de la classe ReservationDAO est trop technique et n'est pas un concept du Domain Model*

**2.** *De même, les méthodes de ReservationDAO sont trop éloignées conceptuellement du Domain Model:*

- *saveReservation(reservation) introduit une dépendance conceptuelle vers la persistance*
- *findReservations(salle, date) est trop générique*

**3.** *Le module domain.reservation est hétérogène, car il comprend Reservation et une interface nommée ReservationDAO*





# Remplacement du DAO par une sémantique du Domain model

---

## Atelier:

*Malgré le placement de l'interface DAO dans la couche Domain Model (pour casser la dépendance technique vers la couche persistance), il subsiste plusieurs problèmes sémantiques.*

**Q1: Pouvez-vous améliorer la conception existante afin d'adhérer à l'esprit du Domain Model?**



# Les Repositories: des ensembles requêtables

*"A Repository represents all objects of a certain type as a conceptual set.  
It acts like a collection, except with more elaborate querying capability."*

~ Eric Evans

- Les Repositories représentent un moyen d'accès à un ensemble d'objets
  - BD
  - Fichier
  - Cache distribué
  - ...
- Ils donnent l'illusion d'une simple collection en mémoire
  - Indépendant de la persistance
  - Suit la sémantique des collections (ex: même comportement que add/remove)



# Contenu d'un Repository

Des QUERIES génériques  
de type findByXXX, ...

Des USE CASE OPTIMAL QUERIES  
retournent une agrégation d'informations (typiquement complexe)  
spécifique à un cas d'utilisation

Des commandes ADD et REMOVE toujours explicites

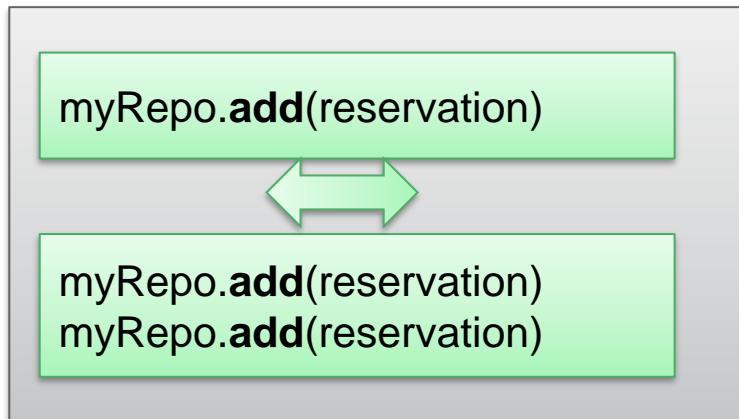
Une commande UPDATE parfois explicite

- Explicite avec les persistence-oriented Repositories
- Implicite avec les collection-oriented Repositories



# Collection & Idempotence des opérations

**Une opération idempotente est une opération ayant le même résultat qu'elle soit invoqué une fois ou plusieurs fois**



- Les contraintes de l'idempotence: on attend le même résultat si l'élément est déjà présent ou si l'élément est déjà supprimé ou en cours de suppression
- Souvent difficile à implémenter!
- L'idempotence est un choix mais indispensable dans le cadre d'une architecture distribuée résiliente



# DAO vs Repository: des sémantiques différentes

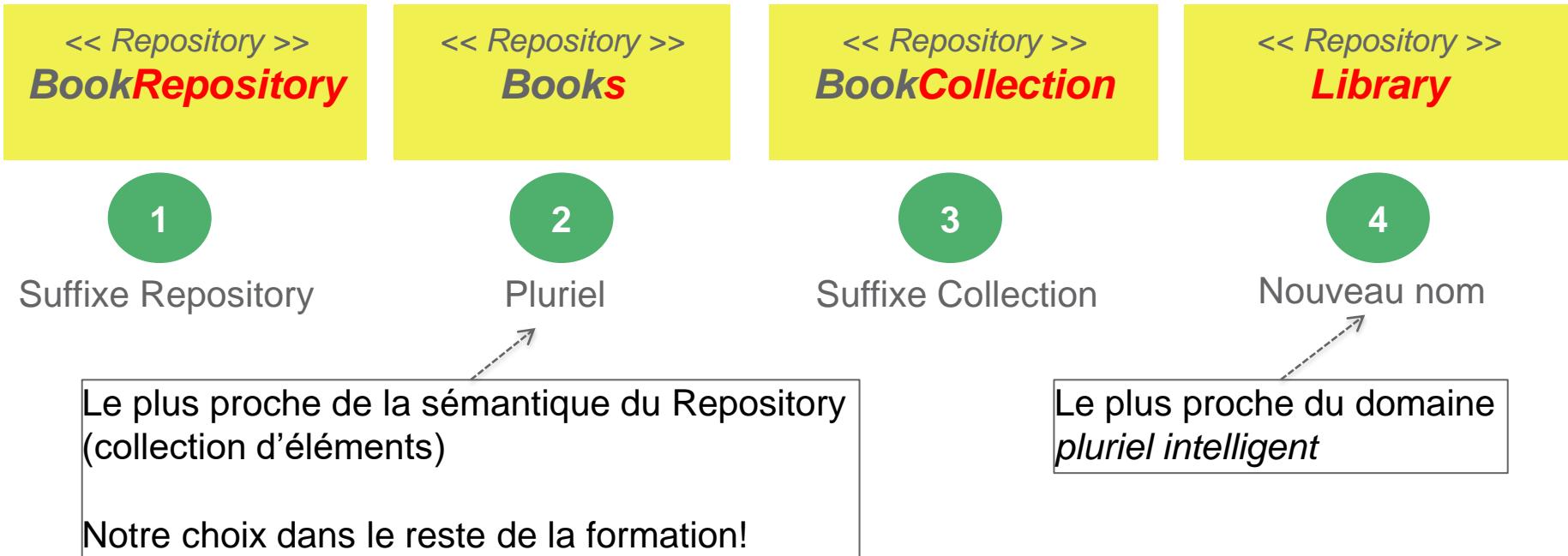
	DAO	Repository
Fonctionnalités	CRUD	Collection augmentée ( recherches + fonctionnalités spécifiques)
Nommage	Orienté technique	Suivant l'Ubiquitous Language. Typiquement, la forme plurielle d'une Entité
Granularité	Souvent mono Table	Agrégats *
Idempotence	A la sémantique d'une BD: add n'est pas idempotent	A la sémantique d'un <b>Set</b> : add/remove sont idempotents
Comportement transactionnel	Participe à (mais n'initie pas) la transaction courante La démarcation transactionnelle est faite par l'Application Service	

# Focus sur le nommage des types « Repository »

Pour ce concept DDD Repository, plusieurs noms sont possibles

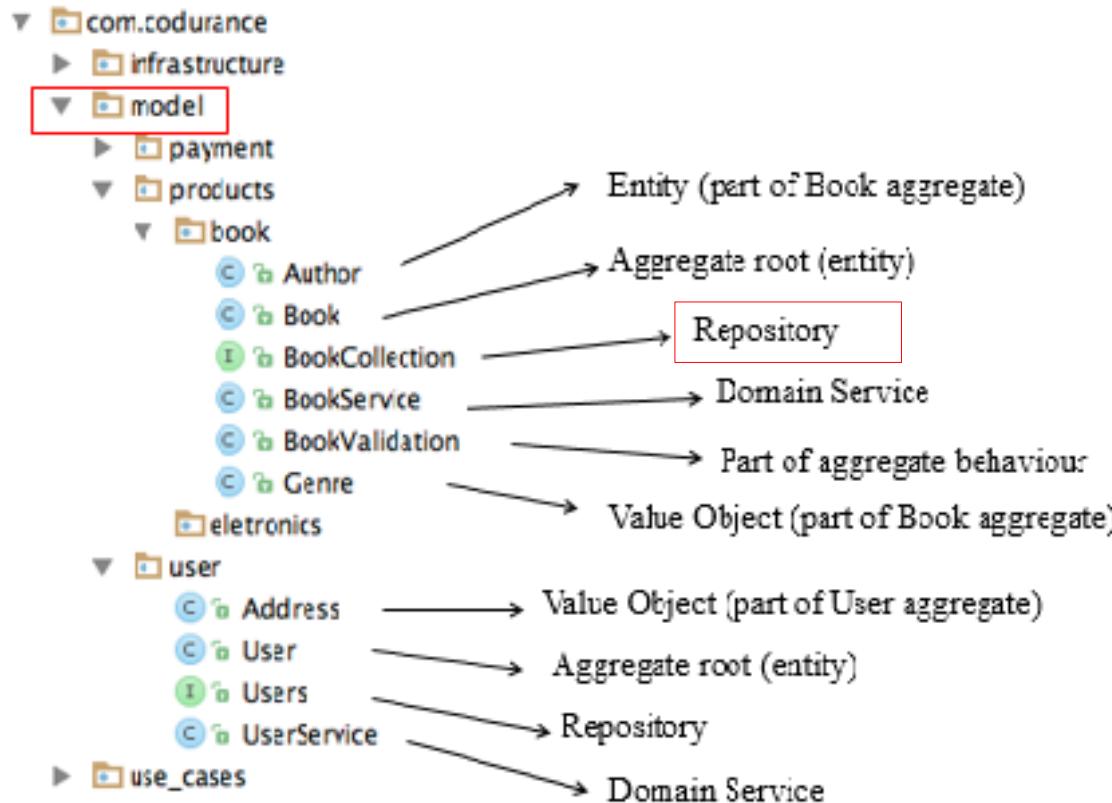
4 typologies de nommages sont généralement utilisées

Le choix dépend de votre contexte



# Inclusion des Repositories dans le Domain model

## What is inside model packages?



*Crafted Design, Sandro Mancuso*



BNP PARIBAS

La banque d'un monde qui change

# Remplacement du DAO par une sémantique du Domain model



## Reprise Atelier:

*Malgré le placement de l'interface DAO dans la couche Domain Model (pour casser la dépendance technique vers la couche persistance), il subsiste plusieurs problèmes sémantiques.*

*Q1: Pouvez-vous améliorer la conception existante afin d'adhérer à l'esprit du Domain Model?*

→ *Utilisation d'un « Repository »*





# Modèle après l'introduction du Repository

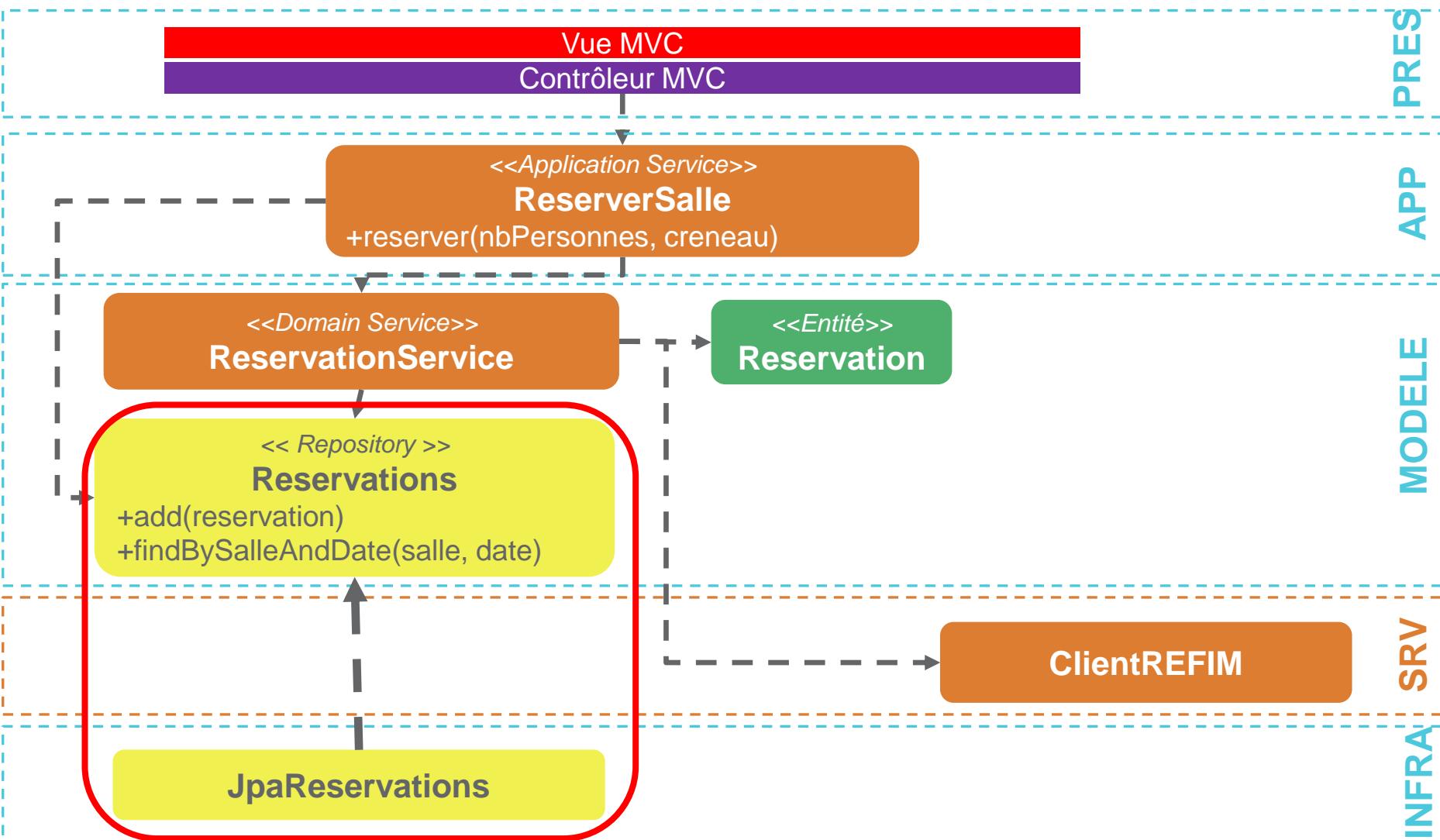
PRES

APP

MODELE

SRV

INFRA



BNP PARIBAS

La banque d'un monde qui change

DDD Tactique (3/3) 211

# Plan



- ① De l'existant vers la cible DDD
- ② Briques DDD simples
  - ① Value Object
  - ② Entité
- ③ Composants DDD de type Service
  - ① Application Service
  - ② Domain Service
- ④ Composants DDD de type Infrastructure
  - ① Repository
  - ② Infrastructure Service



# Généralisation du référentiel des Salles

## Atelier:

- *Actuellement, le Domain Service ReservationService référence ClientREFIM qui est une classe technique dans la couche service.*
- *Il s'agit d'une contingence informatique qui ne fait pas partie du Domain Model.*

**Q1: Comment pourrait-on remplacer le référentiel immobilier REFIM par un concept DDD?**



# Infrastructure Service (IS)

- Un infrastructure Service est une **contingence informatique**
- Il contient des éléments techniques, n'ayant pas pour objectif de représenter un concept métier (pas nécessairement compréhensible par la MOA)
  - Intégration
  - Technique de mapping
  - Communication extérieure
- Il respecte des parties du DIP
  - L'interface est dans la couche Domain: le besoin est exprimé par le métier → **L'intention est exprimée dans la couche domaine.**
  - L'implémentation est dans la couche Infrastructure: comment on répond au besoin ne fait partie du métier → **La résolution de l'intention est expliquée dans l'infrastructure.**
- Il s'agit d'une généralisation d'un Repository
  - Les IS ont une sémantique moins précise

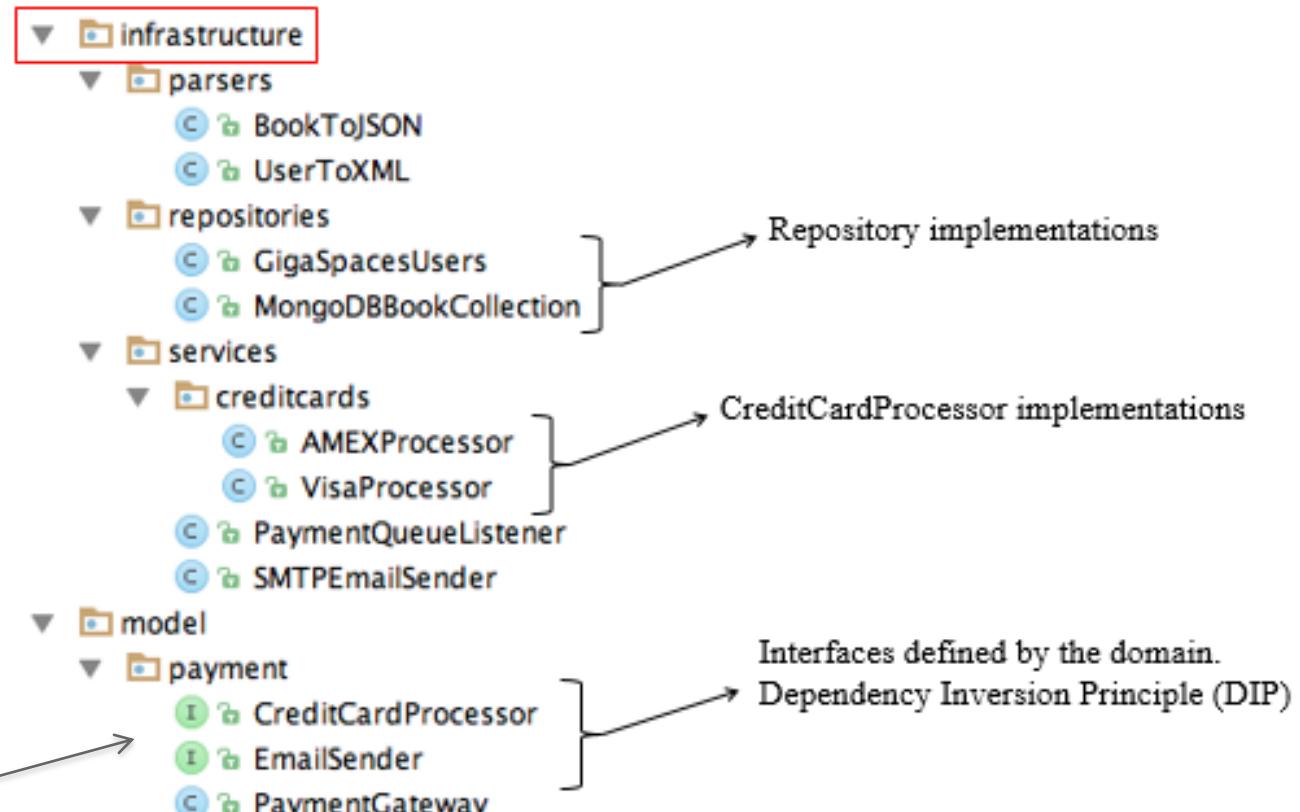


# Repository vs Infrastructure Service (IS)

	Repository	Infrastructure Service
Représente typiquement	Un data store, dans le même SI	Un service technique (intégration, mapping, processing, etc)
Sémantique	Collection augmentée (ajout, suppression)	Pas forcément de notion de collection d'élément
Participe aux transactions?	Oui	Pas forcément
Implémentation	Respect de l'inversion de dépendance (DIP)	

# Contenu du module infrastructure

## What is inside infrastructure?



*Crafted Design, Sandro Mancuso*



BNP PARIBAS

La banque d'un monde qui change



# Généralisation du référentiel des Salles

---

## Reprise Atelier:

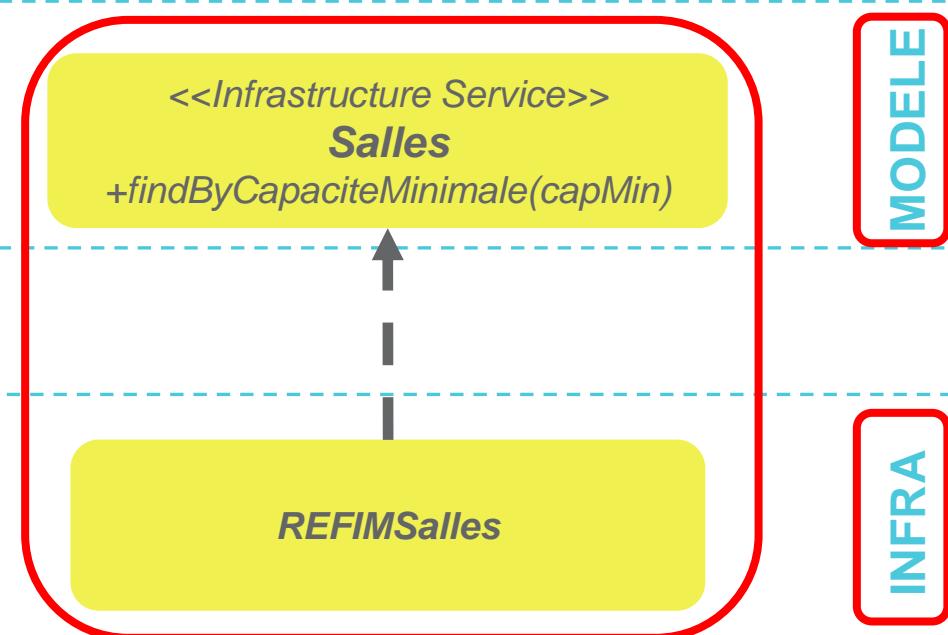
*Q1: Comment pourrait-on remplacer le référentiel immobilier REFIM par un concept DDD?*

→ *Utilisation d'un « Infrastructure Service »*





# L'infrastructure Service Salles



**L'infrastructure Service Salles est une interface dans le Domain Model et l'implémentation spécifique REFIMSalles est dans la couche Infrastructure**



# Focus sur le nommage des types « Infrastructure Service »

- Pour ce concept DDD Infrastructure, plusieurs noms sont possibles
- 3 typologies de noms sont généralement utilisées
- Le choix dépend de la contingence informatique et du contexte

<<Infrastructure Service>>

**SalleIntegration**

*ou*

**SalleMapping**

*ou*

**SalleXX**

<<Infrastructure Service>>

**Salles**

<<Infrastructure Service>>

**ReferentielDeSalle**

1

Suffixe Intégration  
*ou*  
Suffixe Mapping  
*ou*  
SufficeXX...

2

Pluriel

Le plus proche du domaine  
Pas toujours possible!  
(pas nécessairement  
une collection d'éléments)

3

Préfixe spécifique  
à la contingence  
(ex: Referentiel)



# Modèle final du fil rouge après l'introduction du Infrastructure Service



PRES

APP

MODELE

INFRA



<<Application Service>>  
**ReserverSalle**

+reserver(nbPersonnes, creneau)

<<Domain Service>>  
**ReservationService**

<<Entité>>  
**Reservation**

<<Repository>>  
**Reservations**

+add(reservation)  
+findBySalleAndDate(salle, date)

<<Infrastructure Service>>  
**Salles**

+findByCapaciteMinimale(capMin)

**JpaReservations**

**REFIMSalles**



**BNP PARIBAS**

La banque d'un monde qui change

DDD Tactique (3/3) 220

# Prérequis Cohérence transactionnelle et cohérence éventuelle



## Cohérence transactionnelle et cohérence éventuelle



# Agrégats et invariants transactionnels

Comprendre les invariants en contexte concurrent est un préalable au découpage du modèle du domaine en agrégats.

On va donc suivre le plan suivant:

1. Qu'est ce qu'un invariant
2. Protections nécessaires en un contexte de concurrent
3. Trois exemples de protections
4. Cas d'un graphe étendu d'Entités
5. La cohérence éventuelle pour découper le graphe

**Cohérence transactionnelle est la traduction de *transactional consistency*.**  
**Cohérence éventuelle est la traduction de *eventual consistency*.**

**En Anglais *eventual* n'a pas la connotation "incertain" d'*éventuel* en Français.**  
**Eventual indique une survenance retardée mais certaine.**



# Notion d'invariant

Un invariant d'un type d'élément est une propriété toujours vraie pour ce type d'élément. Ex:

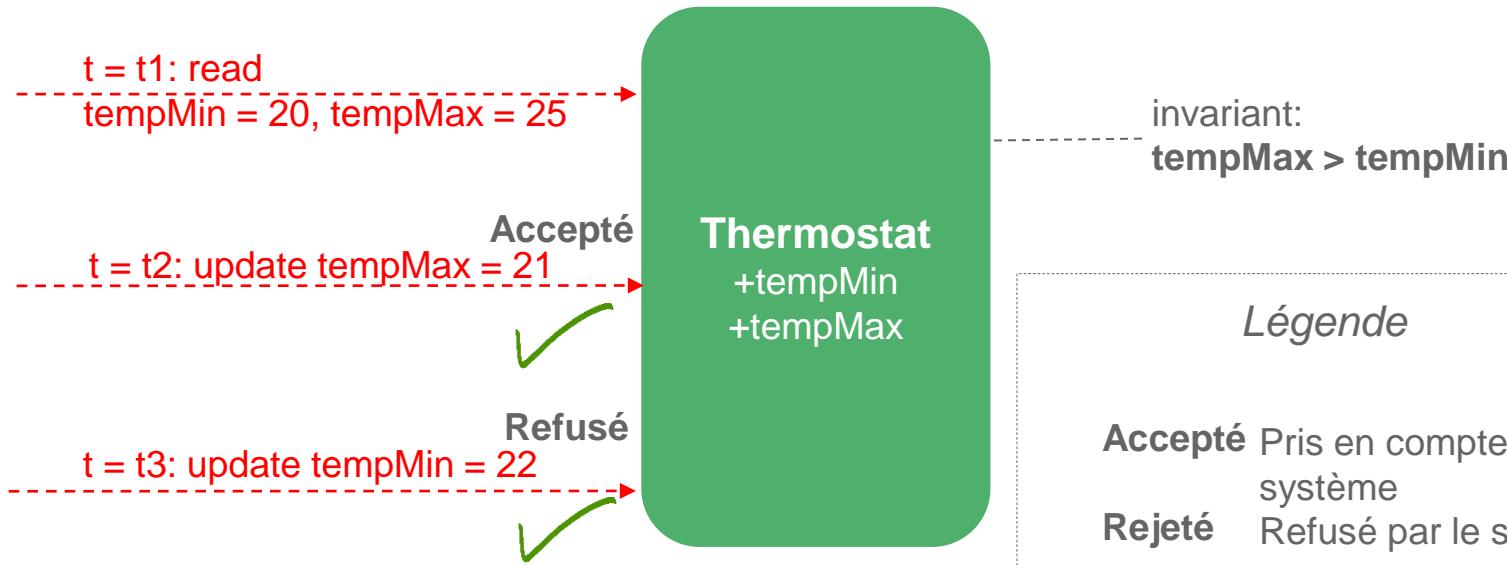
- **Pour un voyage**, la date de retour est toujours postérieure à la date de départ
- **Pour la plage de température d'un thermostat**, la température minimum est toujours inférieure à la température maximum
- **Pour un ordre d'achat**, la somme des montants de toutes les lignes ne doit pas excéder le total maximum autorisé

*Dans la suite, on suppose que l'implémentation étudiée est déjà cohérente dans un contexte non-concurrent, c'est-à-dire qu'elle respecte les invariants.*

*Cela implique que toute les fonctionnalités offertes placent le système dans un nouvel état cohérent, qu'elles réussissent ou qu'elles échouent (propriété d'atomicité).*



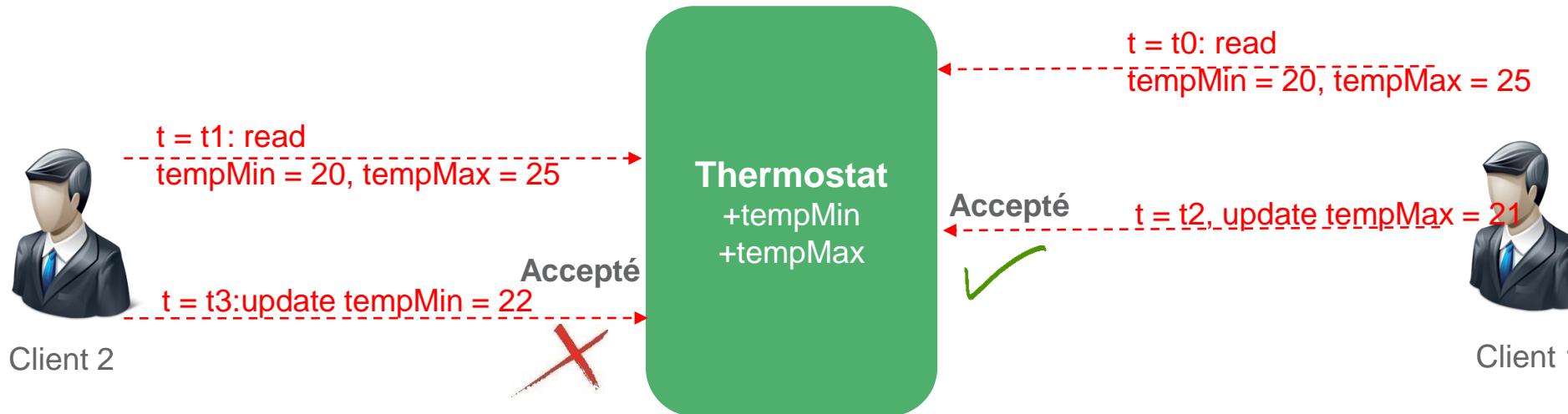
# Un exemple d'invariant



*Une implémentation cohérente préserve les invariants en rejetant les changements qui les violent*



# L'invariant mis en péril par les modifications concurrentes



*Sans précautions, les modifications concurrentes ne préservent pas les invariants*



# Les protections traditionnelles, 1: « la transaction longue »

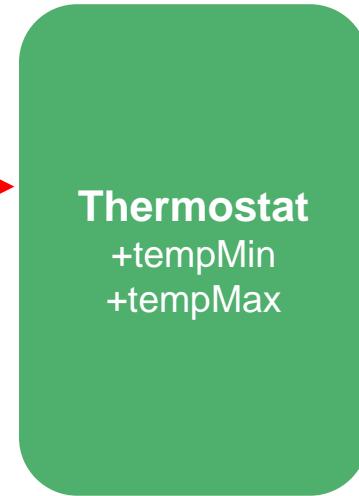


Client 2

Bloqué



$t = t1: \text{read}$



Accepté

$t = t0: \text{read}$   
 $\text{tempMin} = 20, \text{tempMax} = 25$

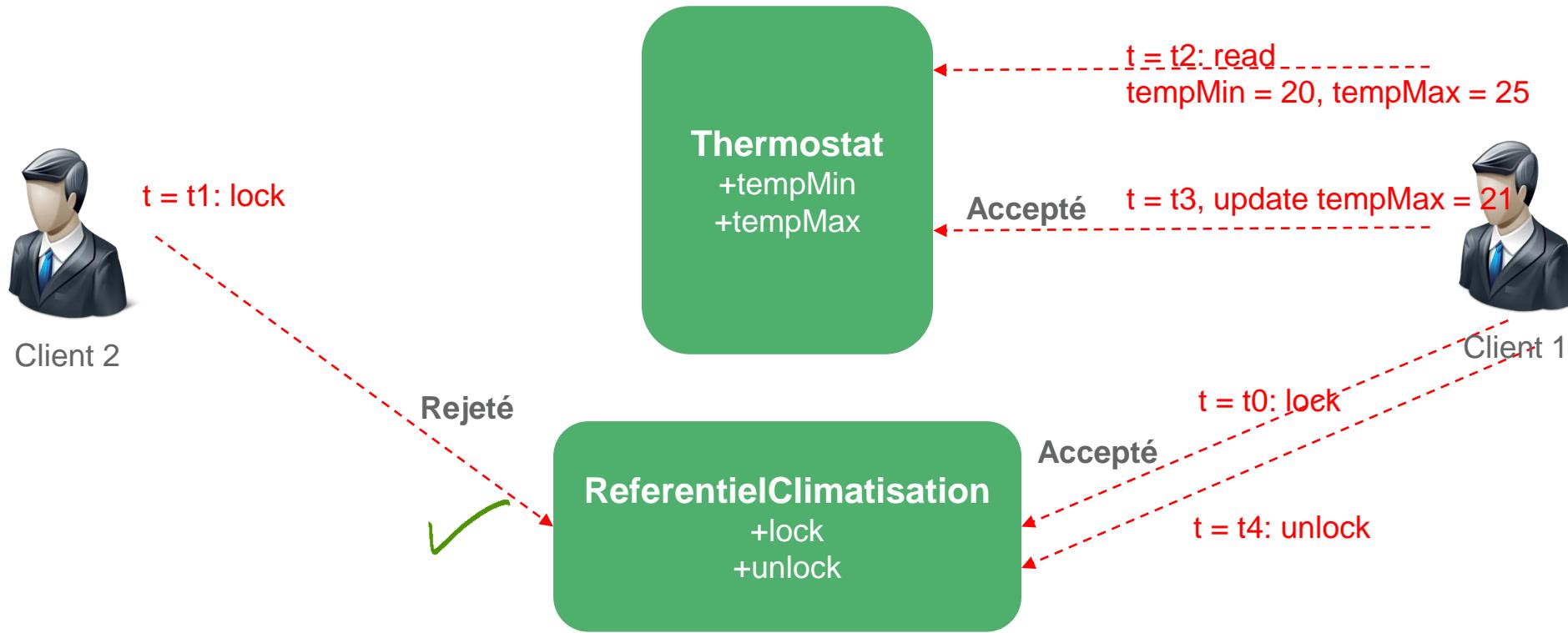


Client 1

*Aline la transaction métier sur une transaction technique.  
L'objet portant l'invariant est bloqué pendant toute la durée de la transaction métier.*



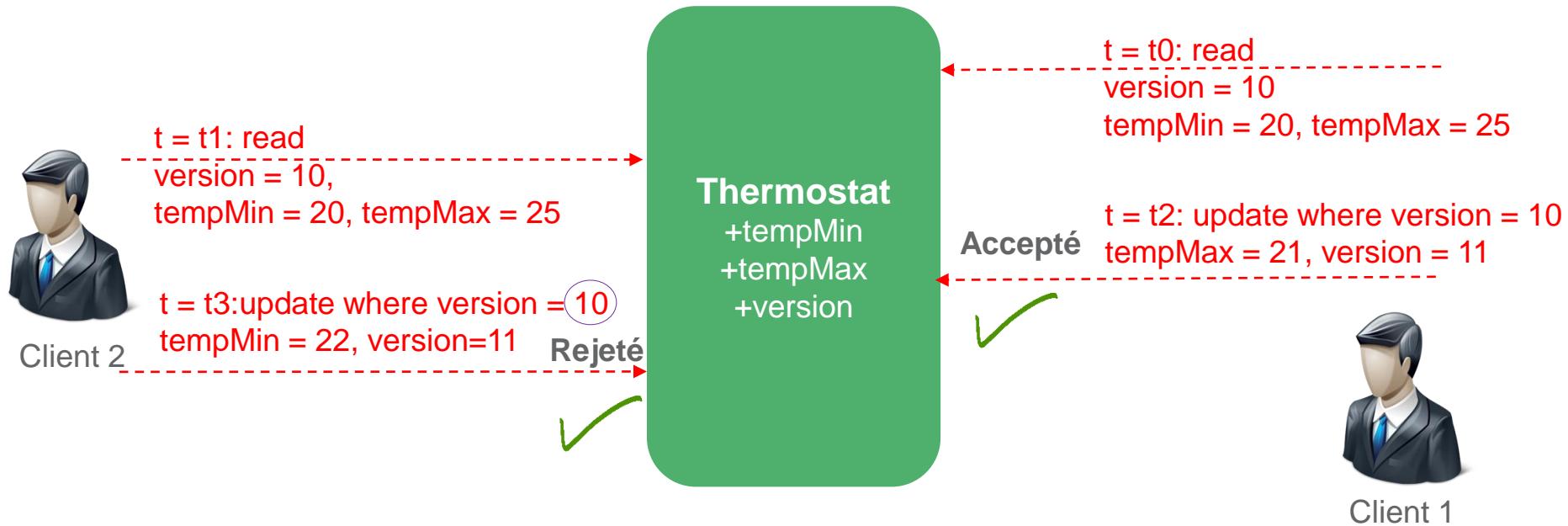
# Les protections traditionnelles, 2: « le pattern Pessimistic Offline Lock »



*Un objet externe à l'objet portant l'invariant (le Référentiel) est verrouillé pendant toute la durée de la transaction métier.*



# Les protections traditionnelles, 3: « le pattern Optimistic Offline Lock »

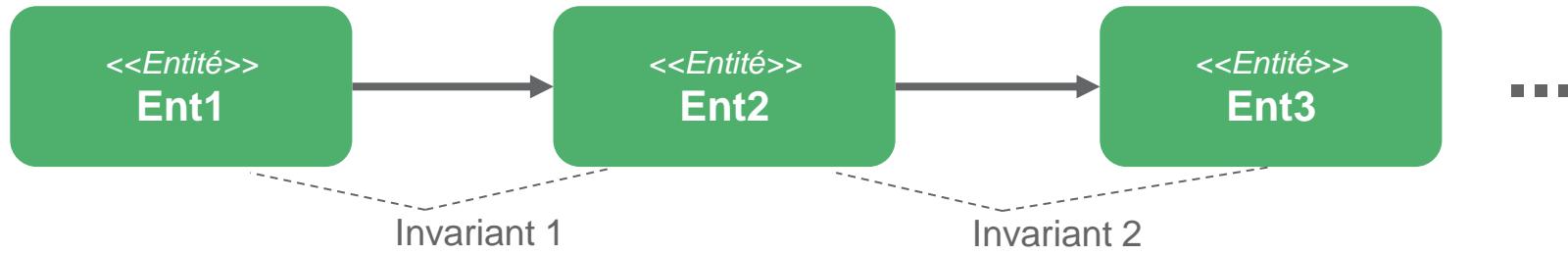


*Lors de chaque lecture, on lit le numéro de version courant.*

*Lors d'une modification, on vérifier que la version n'a pas changé. Si la modification est possible on incrémente la version, sinon on la rejette.*



# Invariants transactionnels dans un graphe d'Entités

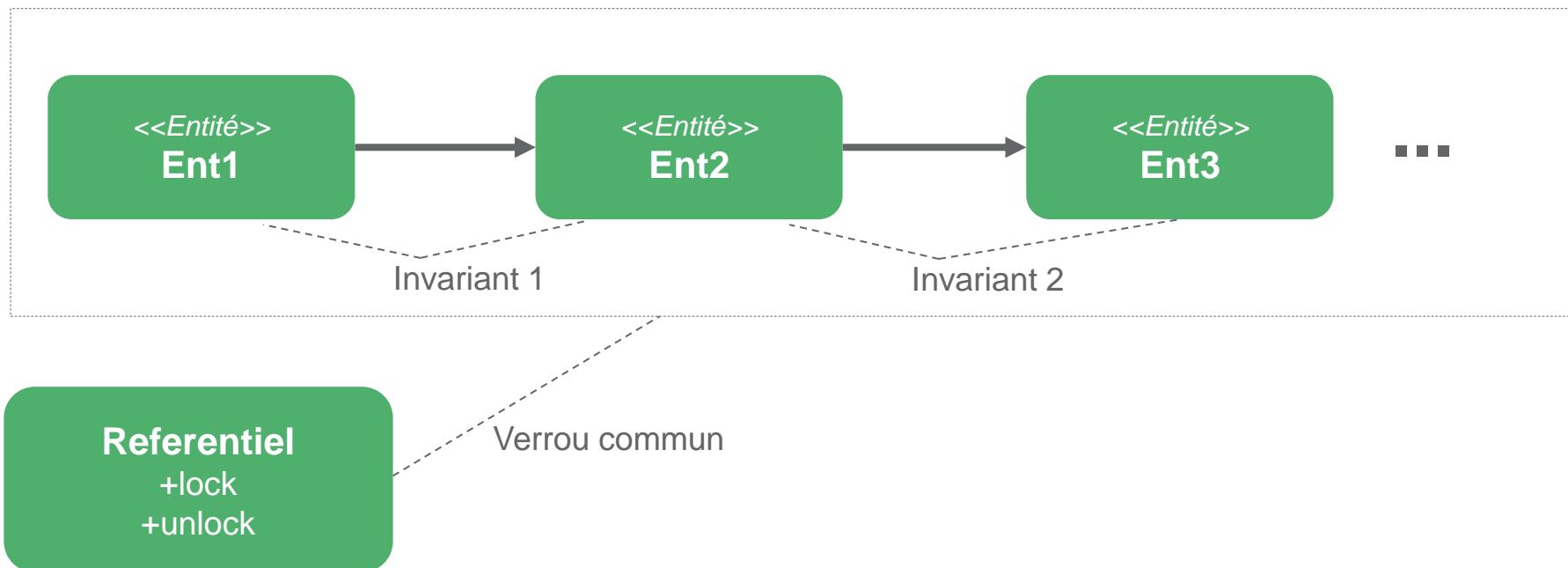


*Dans un graphe d'Entités, des invariants multiples rendent l'analyse des invariants transactionnels difficile.*

*La seule solution est alors d'utiliser une protection commune à l'ensemble du graphe.*



# Impact d'un graphe étendu



*Dans le cas d'un graphe étendu, un verrou commun a un impact plus grand sur les utilisateurs*



# Les invariants transactionnels: pas toujours nécessaires

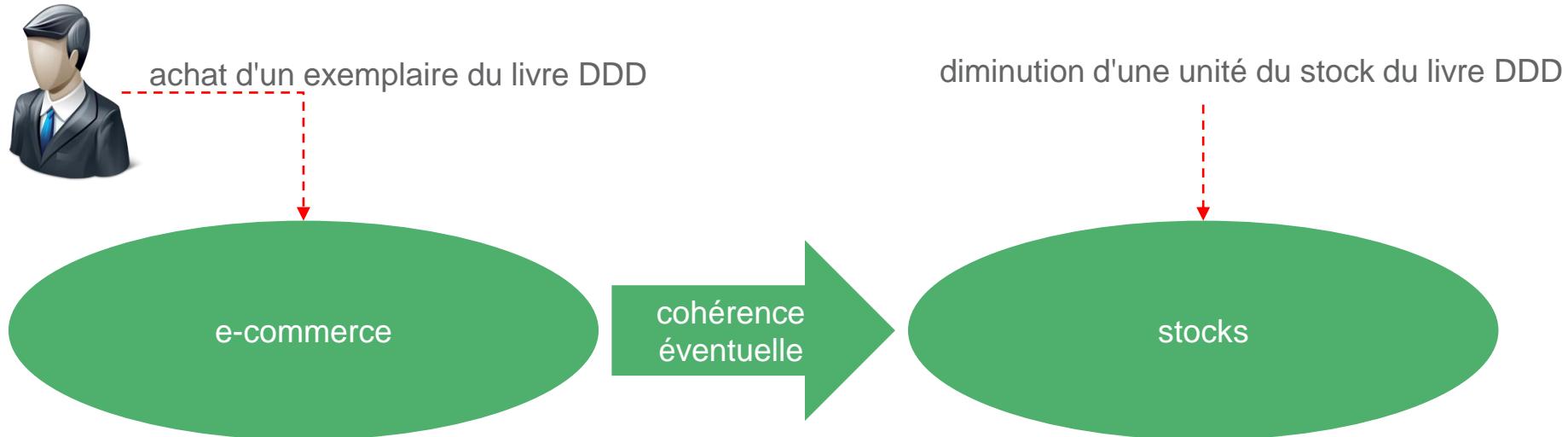
*La formulation d'un besoin oriente parfois le concepteur vers un invariant transactionnel*

***La cohérence éventuelle est en fait souvent suffisante***

*Interroger l'émetteur du besoin en terme de **fenêtre d'incohérence acceptable** peut être un moyen d'éclaircir ce point.*



# Cohérence éventuelle: exemple 1



# Cohérence éventuelle: exemple 2

t = t0: solde = 0



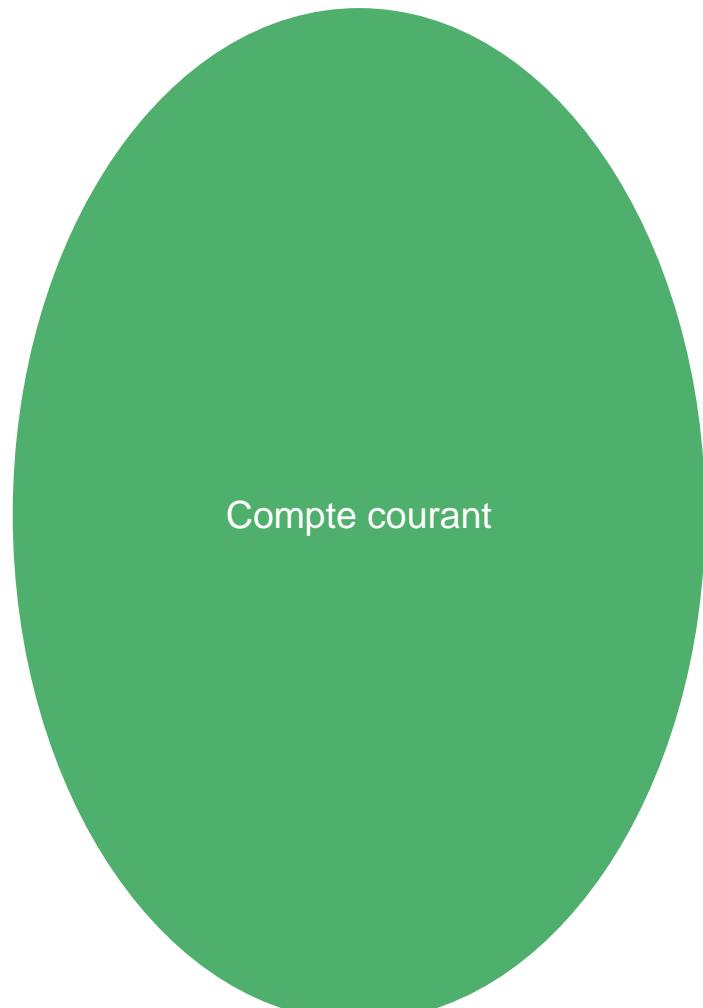
t = t1: salaire crédité (+1000)



t = t2: débit (-100)  
REFUSE



t = t3: débit (-100)  
ACCEPTE



BNP PARIBAS

La banque d'un monde qui change

# Cohérence transactionnelle VS éventuelle

	Cohérence transactionnelle		Cohérence éventuelle
Applicabilité	Invariants stricts		Invariants souples
Fenêtre d'incohérence	Inexistante		Du centième de seconde à plusieurs jours (exemple: réconciliation par batch)
Scalabilité	Faible		Elevée
	Pessimiste	Optimiste	
Inconvénients	Scalabilité faible	Perte du travail non sauvegardé	Pas acceptable pour les invariants stricts Complexité de mise en oeuvre

# Agrégat



## ① Briques DDD plus avancées

- ① Agrégat
- ② Domain Event
- ③ CQRS



BNP PARIBAS

La banque d'un monde qui change

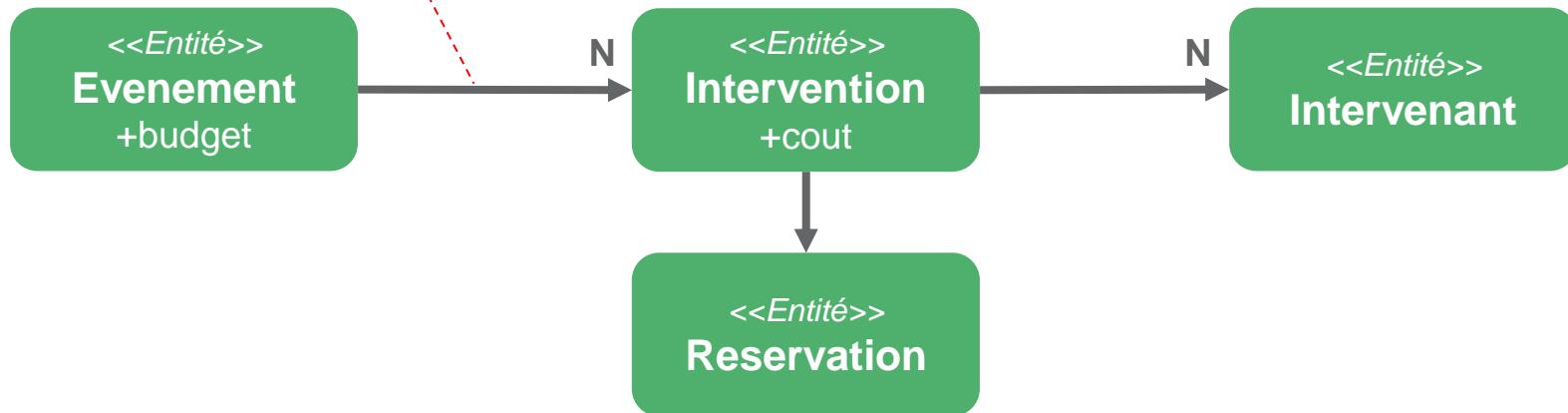
# Organisation d'un événement

## Atelier:

- **On fait évoluer notre application pour permettre d'organiser des événements:**
  - **Un événement est un enchaînement d'interventions.**
  - **Un ou plusieurs intervenants animent chaque intervention**
- **La MOA spécifie un invariant strict:**

*Invariant:*

*coût total des interventions <= budget de l'événement*



Q1: Comment protéger cet invariant?

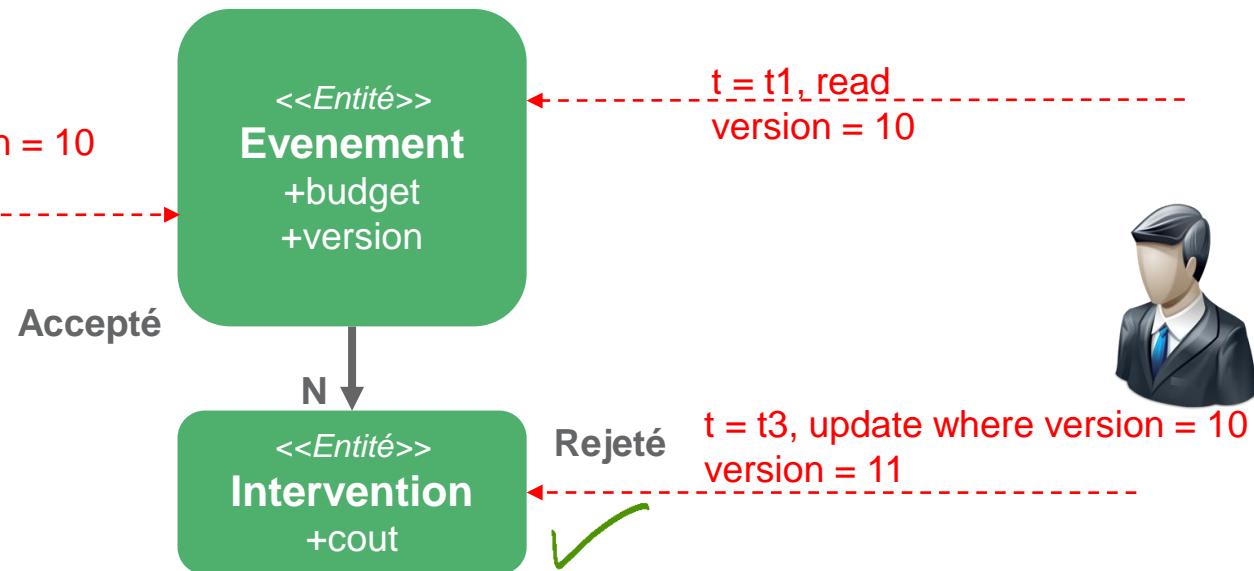
# Protection transactionnelle de l'invariant

## Solution Atelier:

- Pour préserver cet *invariant strict* lors d'éditions concurrentes, on peut (par exemple) utiliser un verrou optimiste (champ *VERSION*) sur *Evenement*.
- La version d'*Evenement* est vérifiée, puis incrémentée, lors des modifications d'un *Evenement* ou d'une de ses *Interventions*:



$t = t2$ , update where version = 10  
version = 11

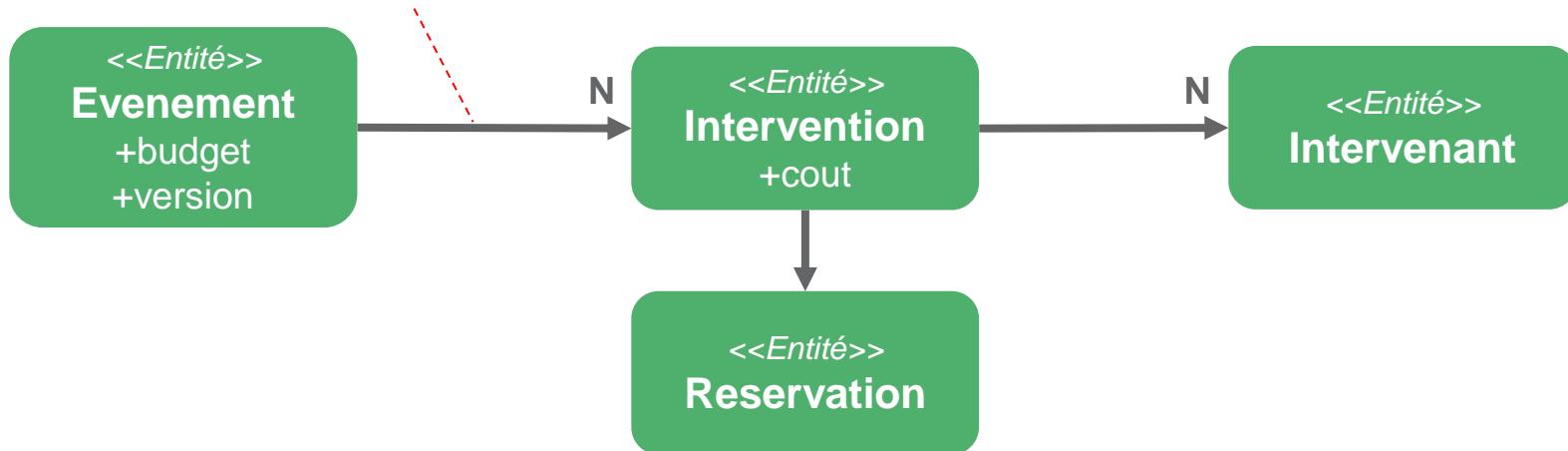


# Organisation d'un événement

## Atelier:

*Invariant:*

*coût total des interventions <= budget de l'événement*



Q2: Quelles difficultés cette conception est-elle susceptible de causer?



# Trouver la bonne granularité?

- C'est l'enjeu central en DDD
- Il s'agit d'un processus itératif
- Constat
  - Le découpage en agrégats est difficile à faire d'emblée (en amont de la conception)
    - C'est pour cela qu'aussi l'agilité devient un prérequis du DDD afin de modéliser le domaine
  - On itère au fur et à mesure
    - Mais dans la plupart des cas, on évite de découper trop les graphes

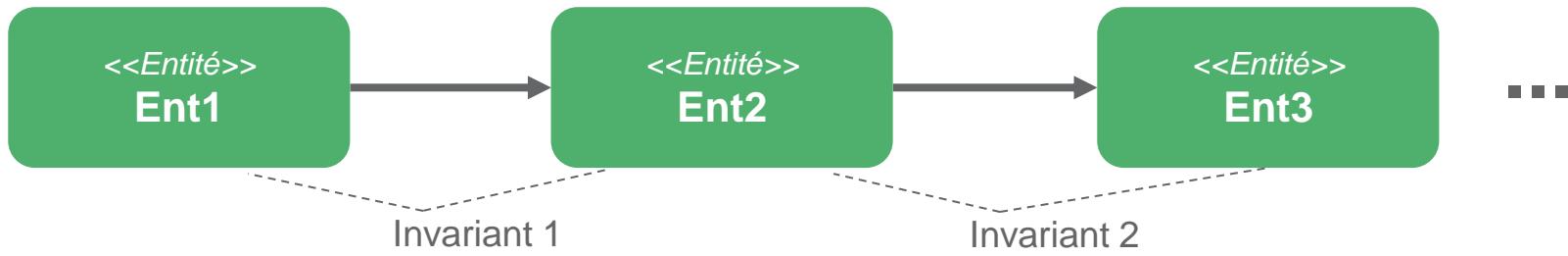


# Problématiques associées aux associations complexes



- *Les conceptions utilisant un graphe illimité de relations rendent difficile la détermination de la frontière d'un objet composé d'autres objets.*
- *De plus, cette frontière dépend souvent du contexte.*
- ***Le problème général est la difficulté à saisir le modèle***

# Associations complexes et invariants



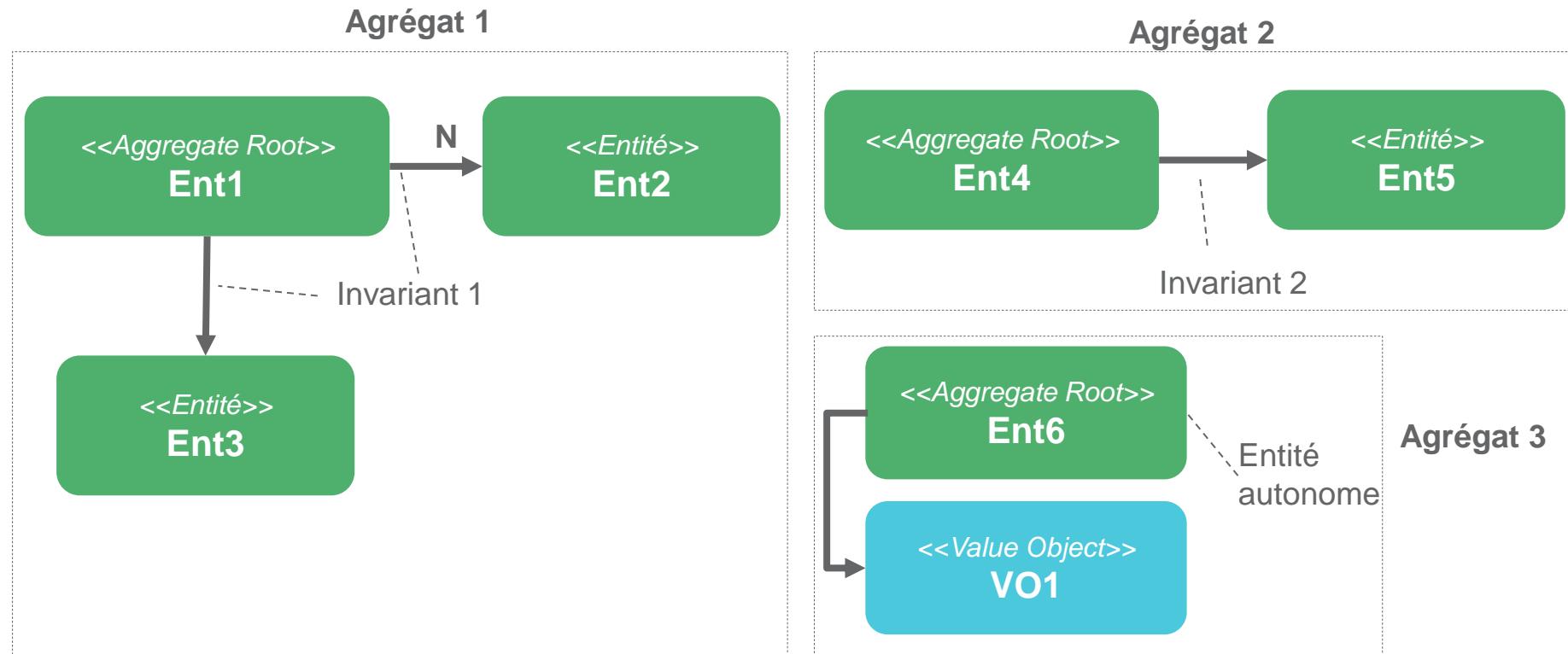
*Il est difficile de garantir la cohérence des changements apportés aux éléments du modèle, car on ne peut pas les considérer isolément.*

*Les invariants s'appliquent à leur ensemble et non à chacun d'eux.*

*Un invariant fréquent est l'intégrité référentielle: un objet ne peut exister sans l'objet auquel il est associé (FK)*



# Les Agrégats, frontières de la cohérence transactionnelle

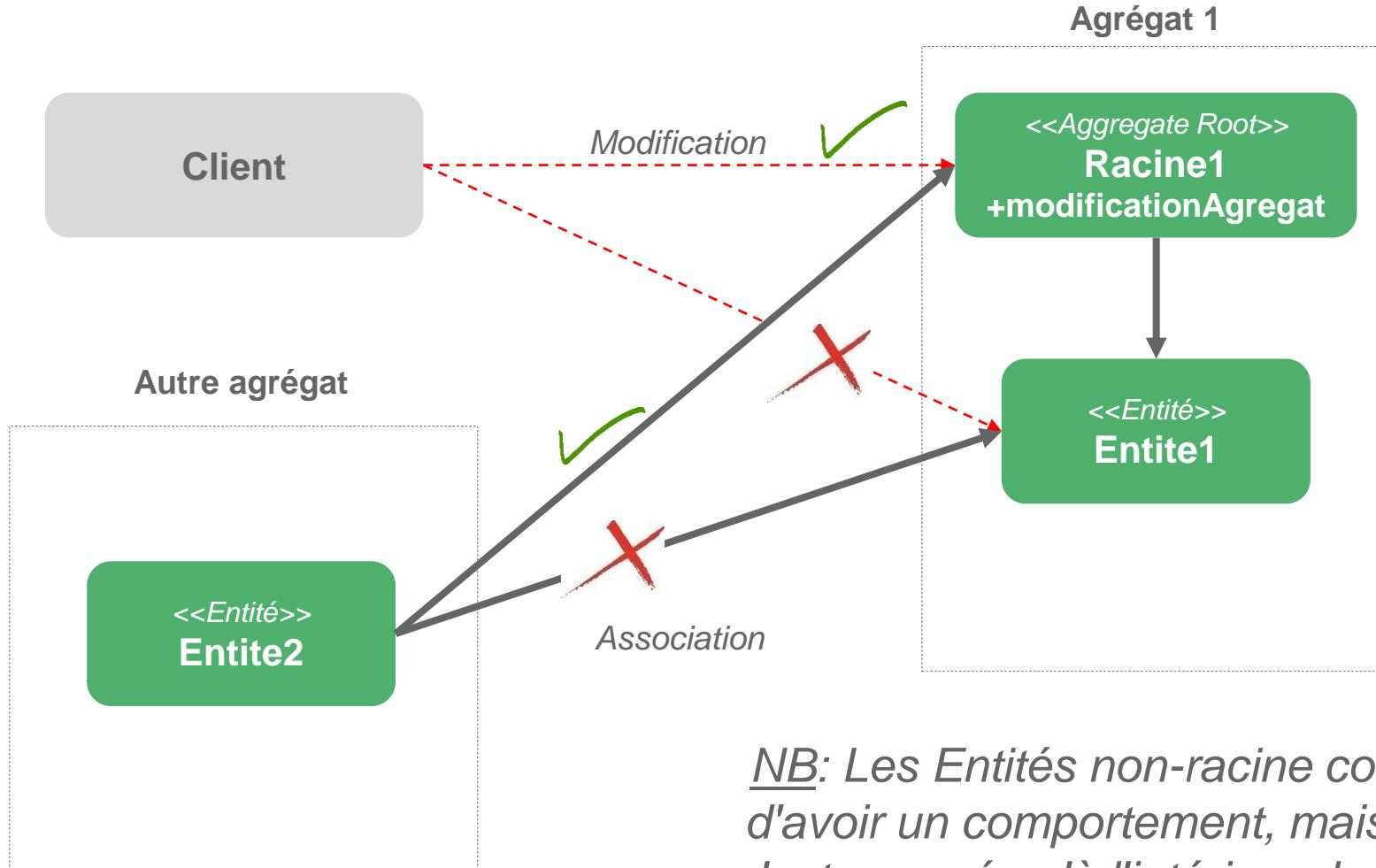


*Les frontières des Agrégats sont choisies de façon à permettre la cohérence transactionnelle des invariants:*

- Elles englobent les Entités participant à chaque invariant.
- Ne jamais modifier deux agrégats dans une même transaction



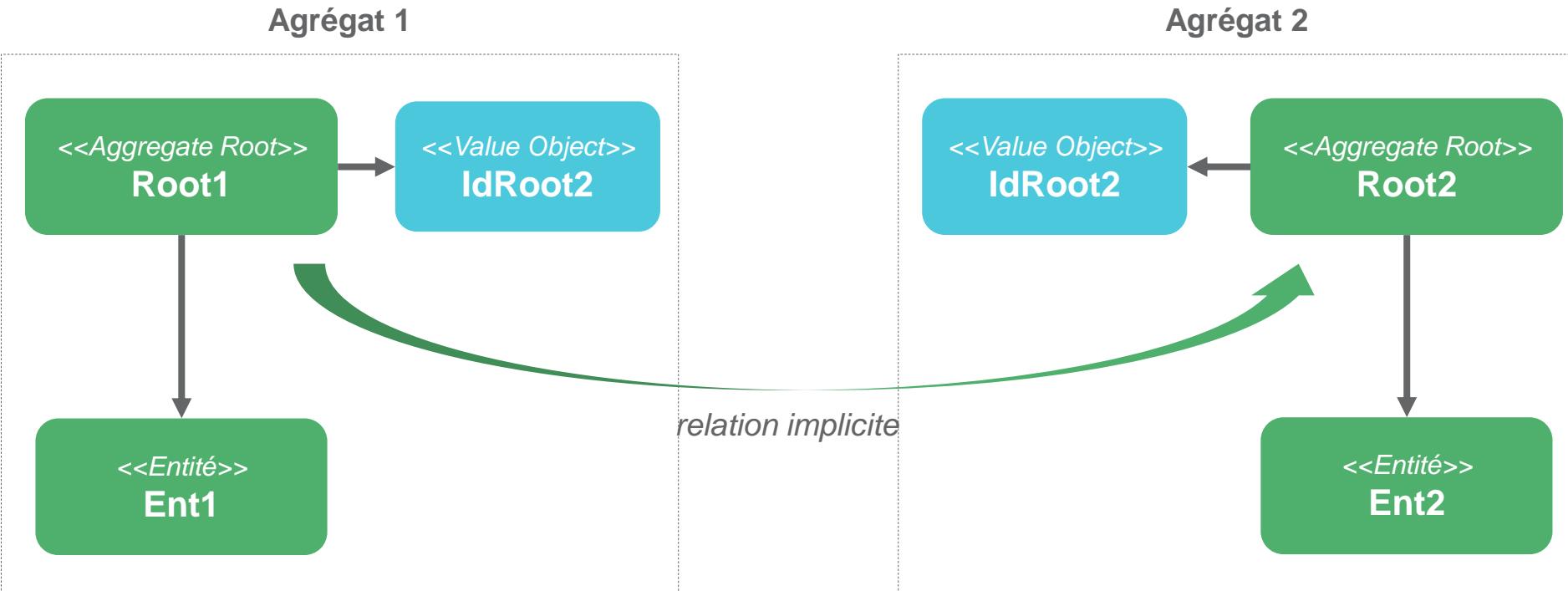
# L'Aggregate Root, point d'entrée unique de l'Agrégat



NB: Les Entités non-racine continuent d'avoir un comportement, mais celui-ci n'est exposé qu'à l'intérieur de l'Agrégat



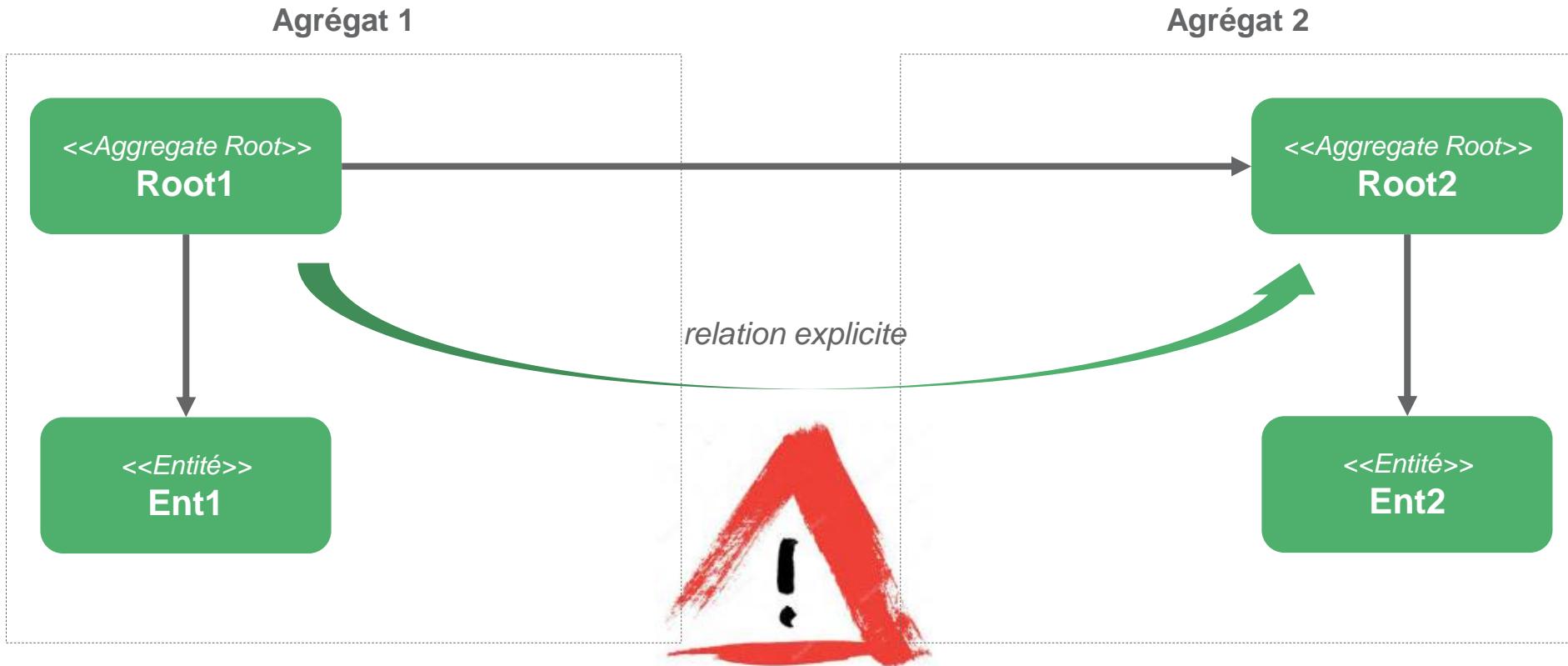
# Relation entre Agrégats par identité



*Remplacer une relation directe par un VO identité partagé permet:  
de limiter le risque de modification de deux agrégats dans une même  
transaction*



# Relation entre Agrégats par référence directe



*Les relations directes entre agrégats ne sont pas interdites à condition de ne jamais modifier les deux Agrégats dans une même transaction*



# Contenu d'un Aggregate

Un et un seul AGGREGATE ROOT  
point d'entrée unique de l'agrégat

Des ENTITES  
Quelques unes au maximum

Des VALUE OBJECTS  
sans limitation de nombre

Un (ou plusieurs) invariant



# Contenu d'un Aggregate Root

## Un NUMERO DE VERSION

garantissant la cohérence transactionnelle des invariants (verrouillage optimiste)

## Des VALUE OBJECTS

caractérisant son état (sans limitation de nombre)

## Des VALUE OBJECTS Identités

- Sa propre identité, constante
- Les identités d'Aggregate Roots d'autres Agrégats

## Les méthodes d'accès centralisé à l'état de l'Agrégat:

- Les COMMANDES de modification de l'Agrégat
- Les QUERIES de calcul d'un état agrégé

## Des références vers des ENTITES

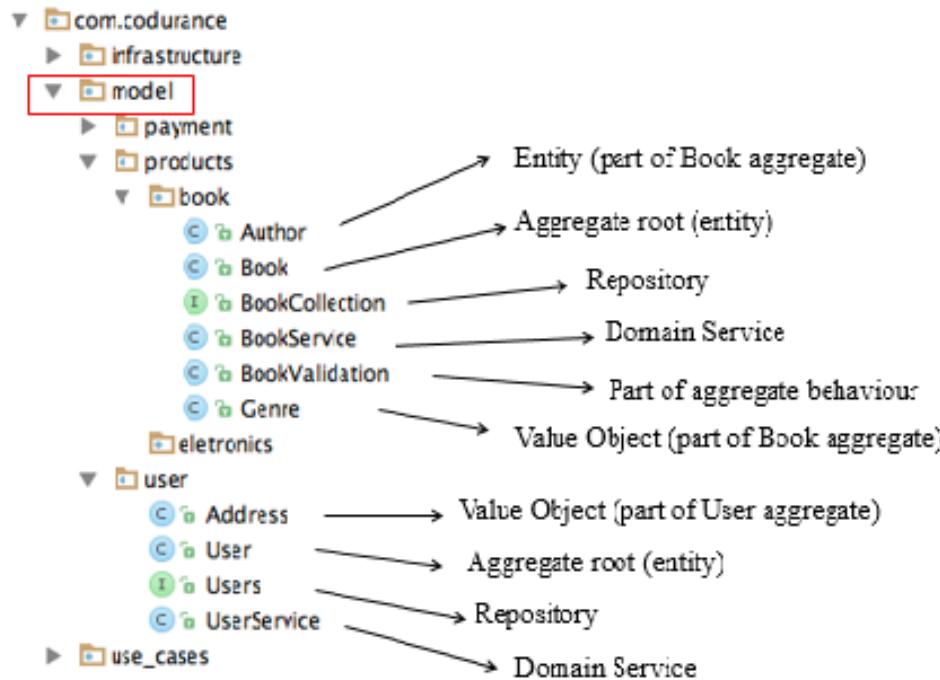
- Du même Agrégat: aussi peu que possible mais en préservant les invariants transactionnels
- D'autres Agrégats: aussi peu que possible, et en lecture seule



# Agrégat et Modules

Crafted Design, Sandro Mancuso

## What is inside model packages?



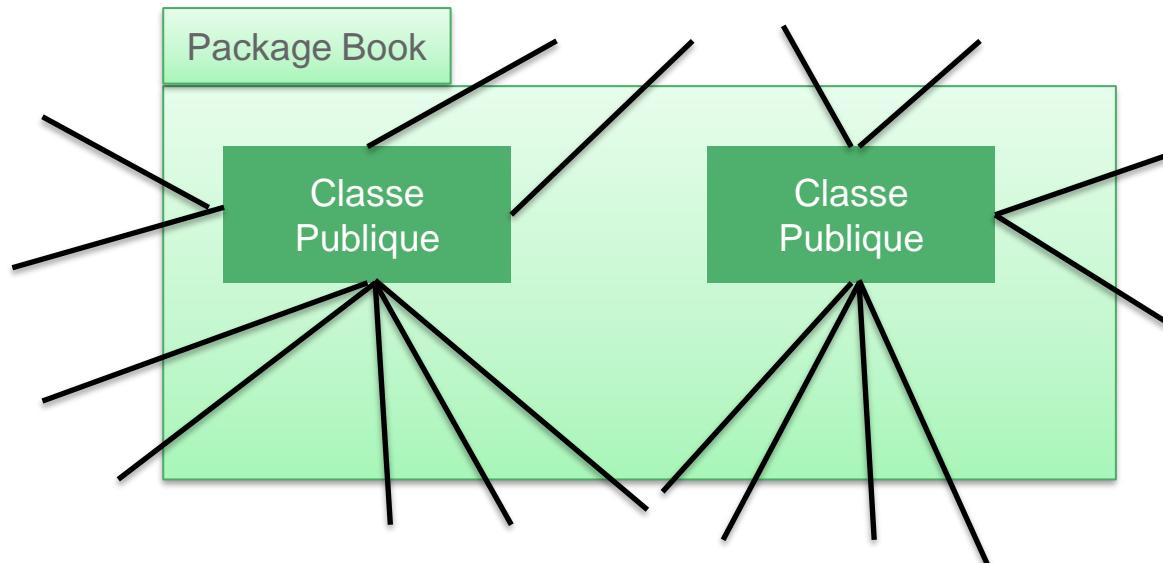
*Les agrégats déterminent les sous-modules*



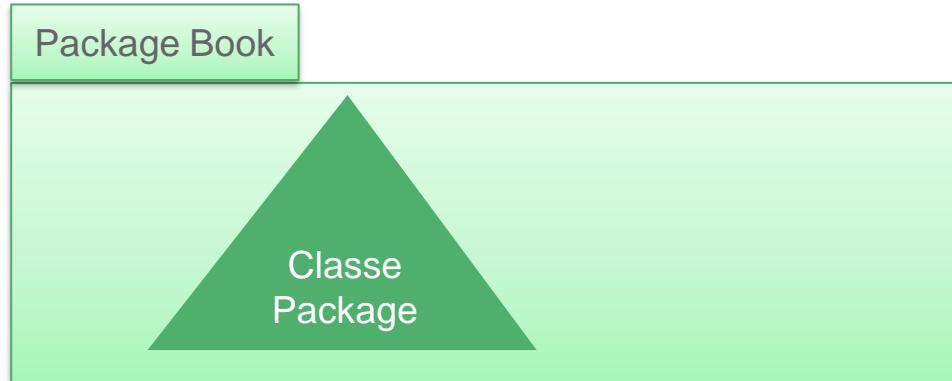
BNP PARIBAS

La banque d'un monde qui change

# Problématique de visibilité



Trop de classes de visibilité **publique**  
entraîne un design touffu



Des classes de visibilité **package**  
restreignent l'accès au package



# Problématique de test des agrégats & Packaging

- Afin que les classes de test des agrégats puissent avoir accès aux classes applicative, on met les classes de test dans le même package que les classes des agrégats
- On utilise généralement un outil de build (ex: Maven) qui facilite le découpage entre les classes de production et les classes de test
  - « src/main/java »
  - « src/test/java »
- Au moment de l'étape de packaging (ex: construction d'un JAR depuis Maven), seule les classes de production sont incluses



# Agrégat et Repository

*Créer un Repository par Agrégat.*

*En lecture, le Repository permet de charger l'Aggregate Root, les autres entités de l'Agrégat ne peuvent être atteintes qu'en suivant les liens d'association.*

*En écriture, le Repository encapsule la création, la suppression et la modification d'un Agrégat sans exposer les Entités non-racine.*



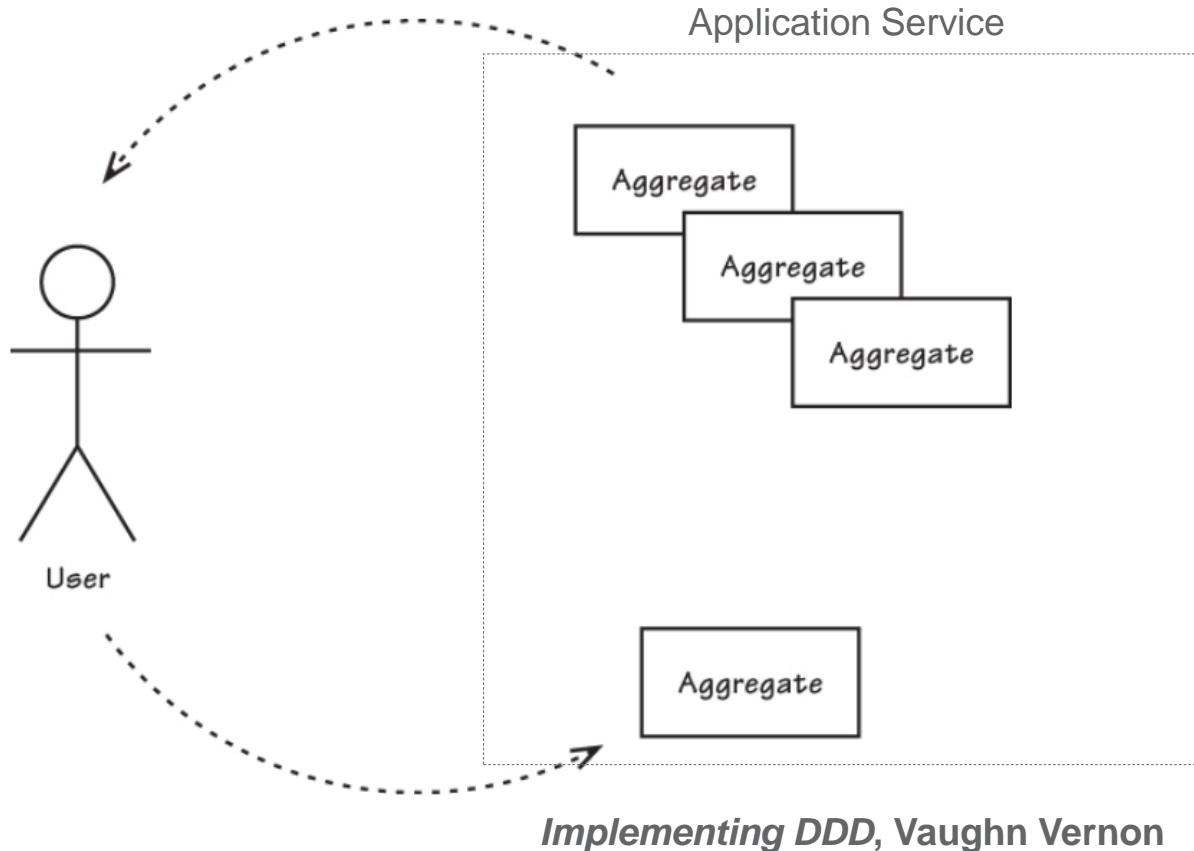
# Agrégat et Factory

*Si la création d'un Agrégat est complexe, encapsuler cette création dans une Factory*

*Le contrat de la Factory est de produire un Agrégat complètement construit, satisfaisant tous ses invariants.*



# Agrégat et Application Service



## En lecture:

l'Application Service assemble un « Application View » portant les informations de plusieurs Agrégats

## En écriture:

l'Application Service accepte une commande portant les paramètres permettant de modifier un Agrégat

*Un Application-service permet de modifier un seul Agrégat, mais d'agréger les informations de plusieurs Agrégats*

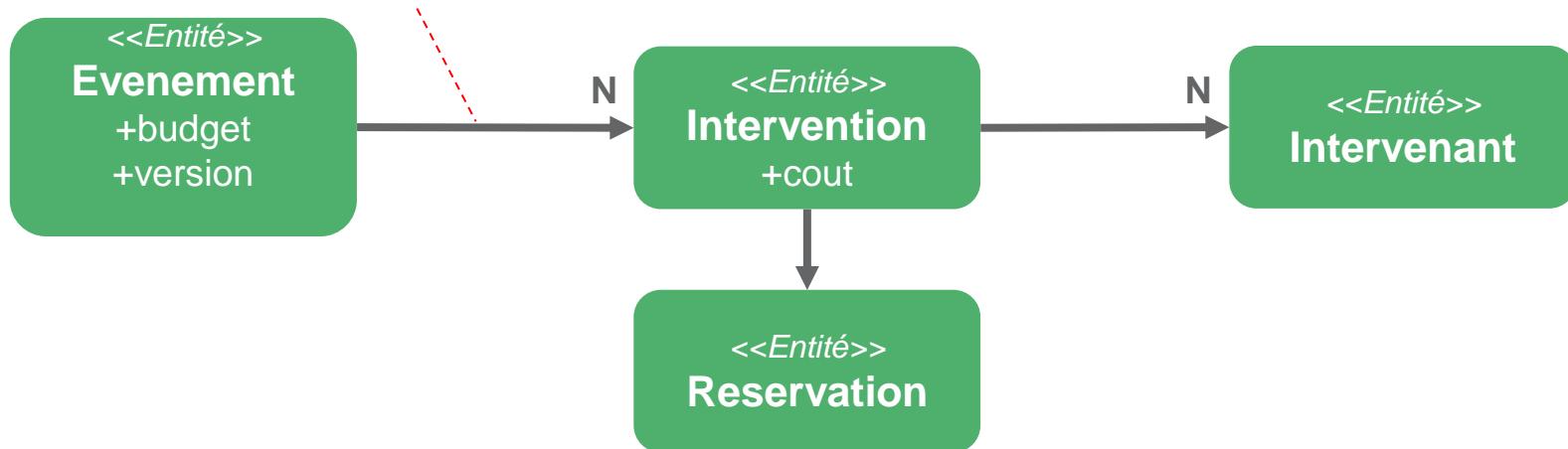


# Organisation d'un événement

## Reprise Atelier:

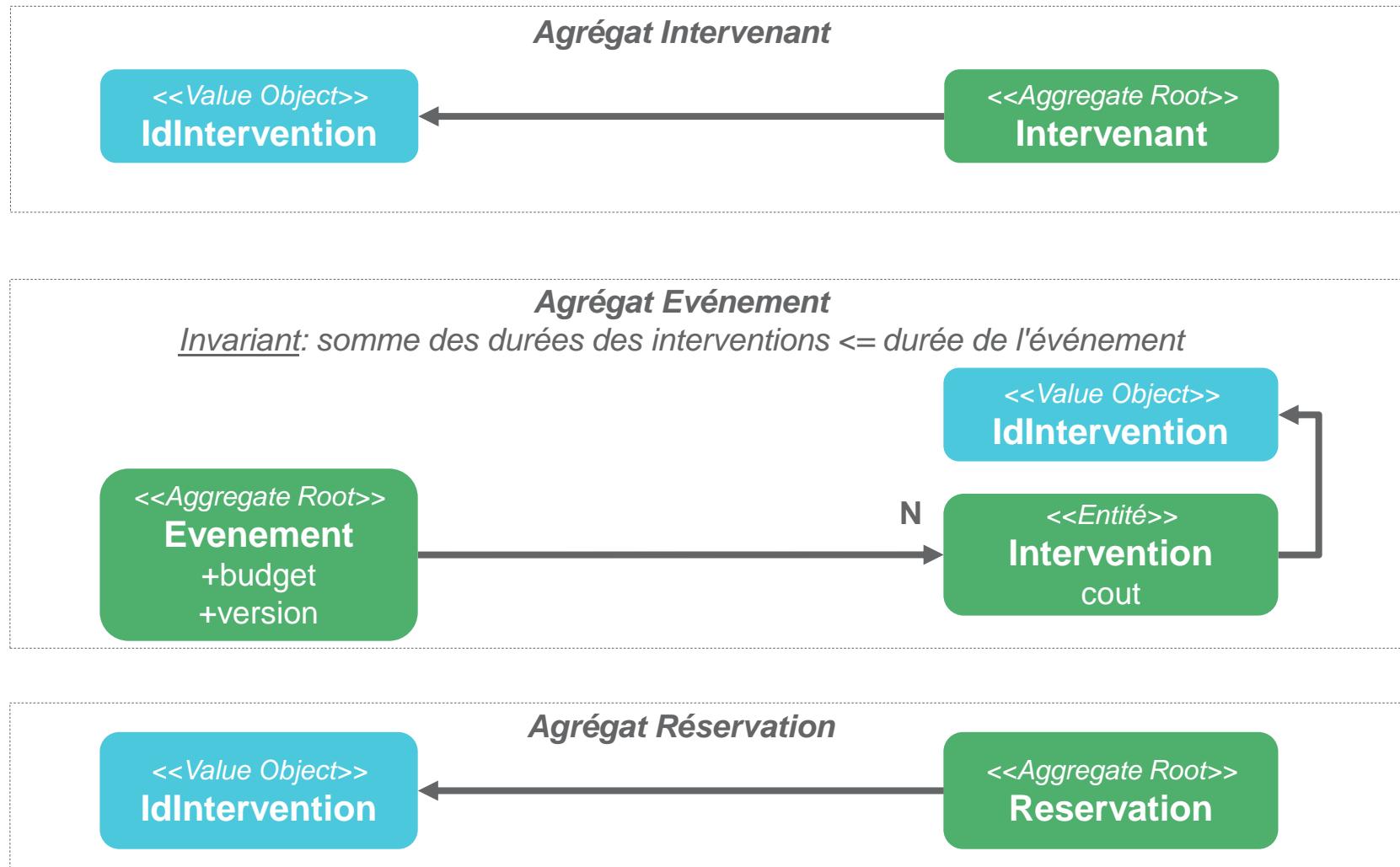
*Invariant:*

*coût total des interventions <= budget de l'événement*



Q2: Quel découpage en Agrégats serait plus pertinent?

# Organisation d'un événement: solution



# Utilisation des Aggregate Root

## Incorrecte

### Agrégats de trop gros grain

- Découpés selon de faux invariants
- Echecs de commits nombreux, ou limitation drastique de la scalabilité avec un verrou pessimiste

### Modifier deux agrégats dans une même transaction

- Indique un mauvais découpage des Agrégats
- Il est dangereux d'injecter un Repository dans une Entité

### Partitionnement/sharding ne respectant l'intégrité des Agrégats

Les entités d'un Agrégat doivent être sur le même noeud

## Correcte

### Agrégats de petites tailles

- Très peu d'Entités, état représenté par des VO
- Les conflits provoquant l'échec d'un commit sont rares

### Politiques de consistance éventuelle clairement définies

- fenêtre d'incohérence acceptable
- batchs de mise à jour si nécessaire

### Encapsulation de la complexité de l'Agrégat

- Aggregate Root point exposant toutes les commandes/queries nécessaires
- Complexité de création de l'Agrégat masquée par une Factory
- 1 Repository par Agrégat



# Domain Event



## ① Briques DDD plus avancées

- ① Agrégat
- ② **Domain Event**
- ③ CQRS



BNP PARIBAS

La banque d'un monde qui change

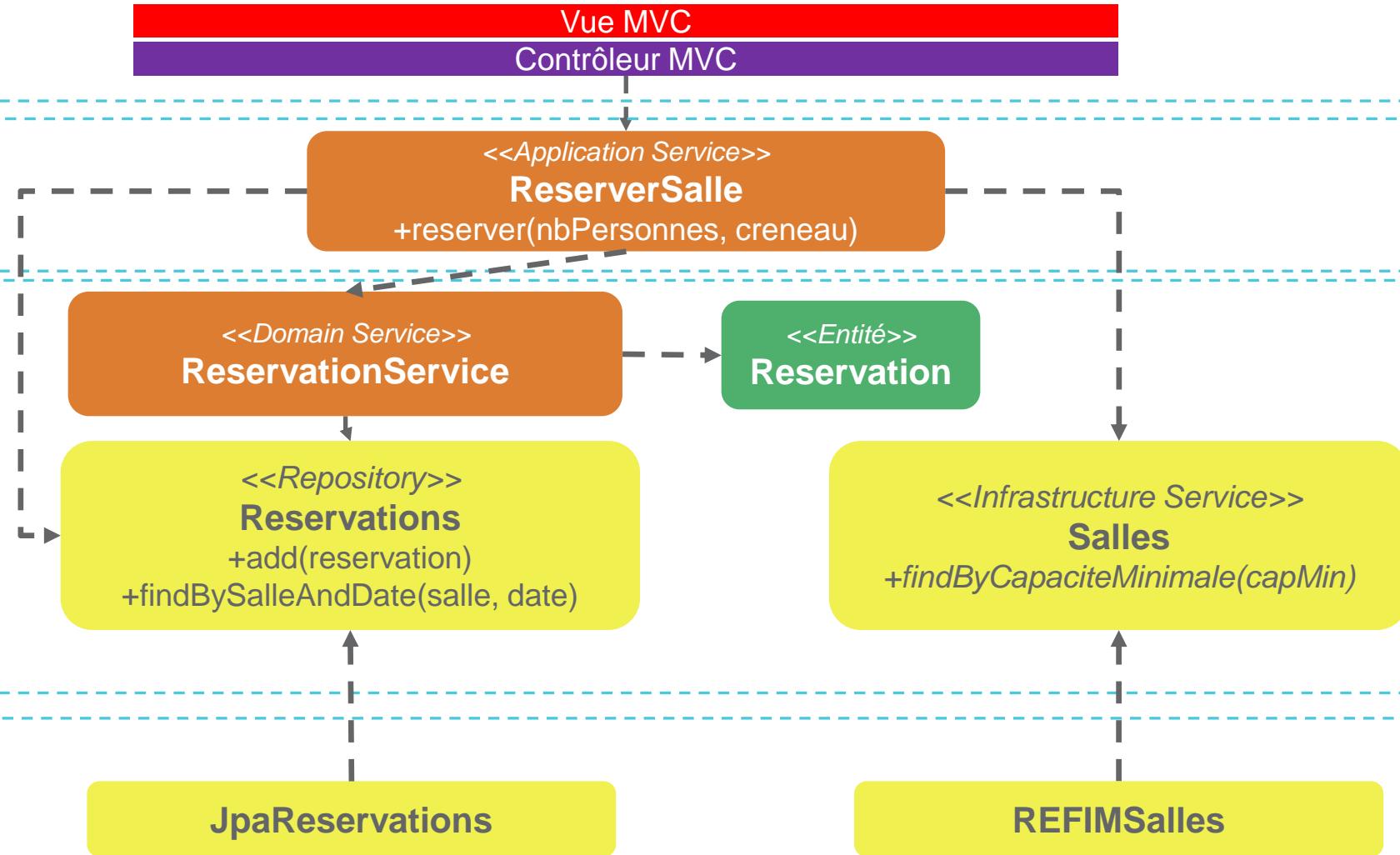
# Rappel: Modèle actuel

PRES

APP

MODELE

INFRA





# Ajout d'une fonctionnalité d'audit

## Atelier:

*Lorsqu'une réservation est confirmée, on souhaite l'ajouter à un audit trail*

- Chaque réservation réussie doit (par exemple) être horodatée et enregistrée pour audit ultérieur*

*Q1: Comment ajouter ces fonctionnalités de la façon la moins intrusive possible?*



# Domain Event: Définition

Les Domain Event modélisent l'activité dans le domaine par une série d'événements discrets:

- S'est produit dans le passé
- Intéresse les experts métier
  - Un événement technique n'est pas un Domain Event
- Fait partie de l'Ubiquitous language
- Est instantané (sans durée) et atomique (insécable)

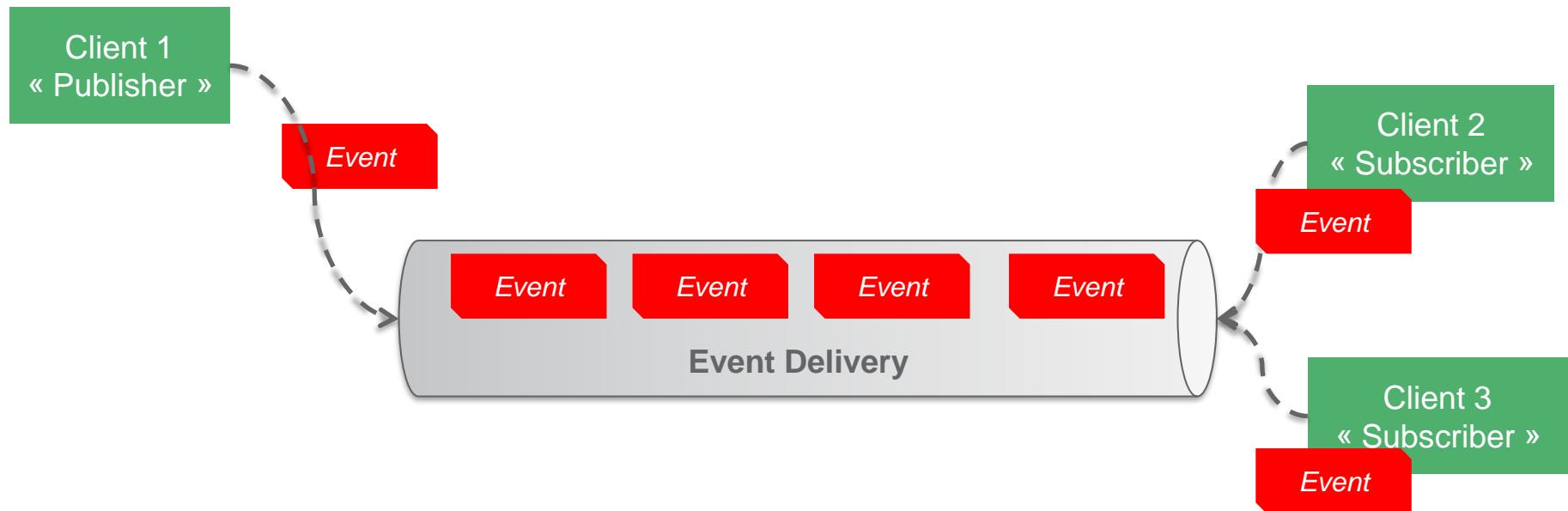
→ Exemples:

- ContratSigné
- PaquetEnvoyé
- PerteInventaireSignalée



# Le pattern Observer

*Découpe un producteur d'événement et un ou plusieurs consommateurs du même type d'événements*



# Les Domain Events sont dans le passé

	Commande	Domain Event
Nommage	SigneContrat	ContratSigne
Exprime	Une intention	Un résultat, un historique
Peut échouer / être rejeté?	Oui	Non, on est sûr que l'événement est déjà passé



# Applications des Domain Events

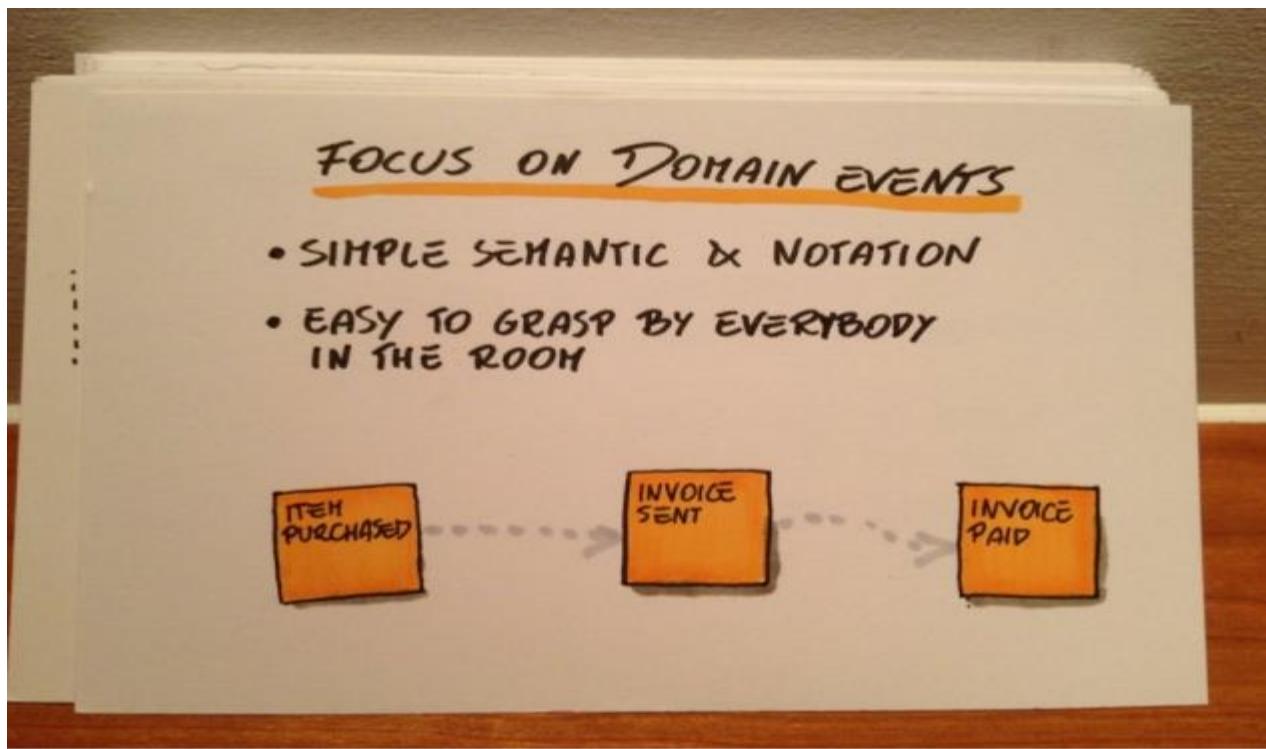
1. Formulation spontanée de la MOA
  - Quand <UN EVENEMENT> se produit, effectuer <UNE ACTION>
2. Audit Trail (traçabilité, conformité réglementaire/compliance)
3. Tracking (suivi opérationnel/fonctionnel)
  - Pour analyse ultérieure (mesure du fonctionnement)
  - Rétroaction éventuelle sur l'opérationnel (amélioration du process)
4. Représenter l'histoire de l'application par une suite de Domain Events
  - Event Sourcing
5. S'intégrer avec des systèmes externes, par notification
  - Définir la politique d'*eventual consistency* !



# Modélisation à l'aide des Domain Events

→ Facilitent la conception en:

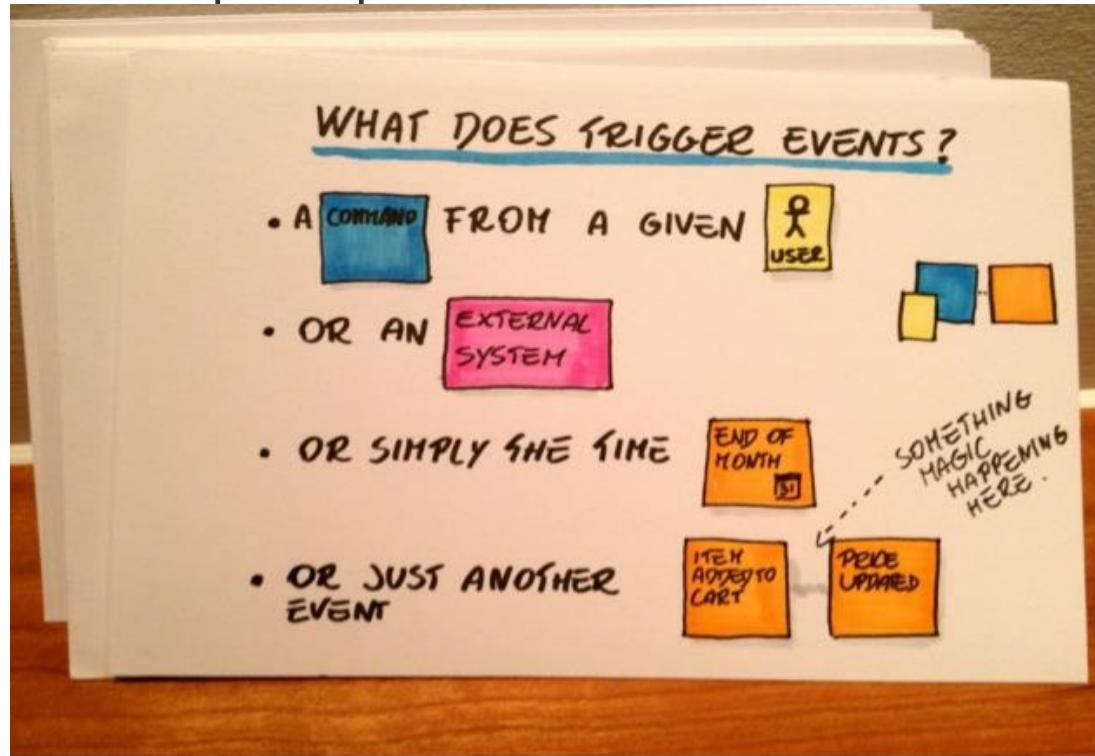
- plaçant des jalons sur un axe temporel
- ancrant des étapes certaines, visibles de l'extérieur d'un processus complexe



# Modélisation à l'aide des Domain Events

→ Facilitent l'analyse en remontant dans le temps à partir d'un événement:

- Quelles sont les causes d'un événement?
- Quels sont ses prérequis?



# Contenu d'un Domain Event

IDENTIFIANTS des entités impactées

DATE/HEURE de survenance

Le CONTEXTE avant et après l'événement  
Les CIRCONSTANCES de l'événement

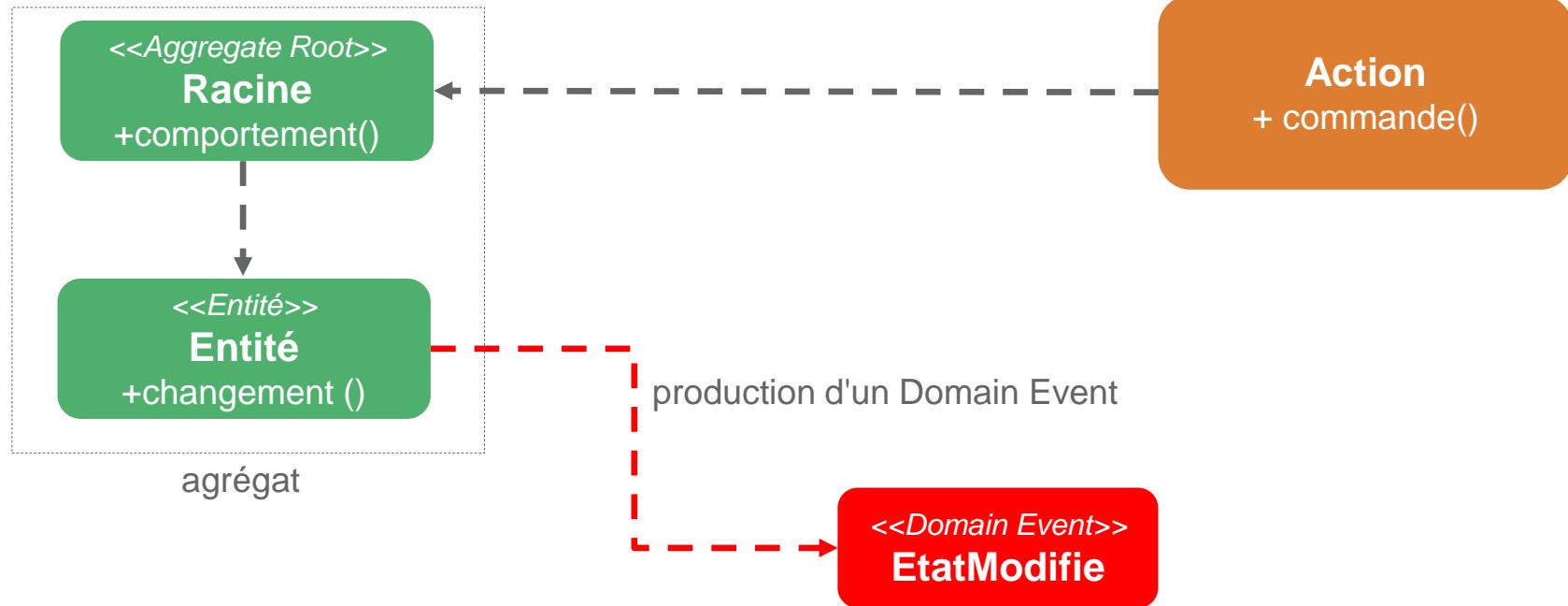
NUMERO DE SERIE de l'événement  
(pour permettre à sa consommation d'être *idempotente*)

PAS DE COMPORTEMENT

Les consommateurs décident comment tenir compte de l'événement



# Production et consommation d'un Domain Event





# Ajout d'une fonctionnalité d'audit

## Reprise Atelier:

- *Lorsqu'une réservation est validée, on souhaite l'ajouter à un audit trail*
  - *Chaque réservation réussie doit être horodatée et enregistrée pour audit ultérieur*

*Q1: Comment ajouter ces fonctionnalités de la façon la moins intrusive possible?*





# Modèle de la notification

APPLICATION

MODEL

INFRA

<<Application Service>>  
**ReserverSalle**  
+confirmer(reservationId)

<<Application Service>>  
**EvenementsReservation**  
+confirmee(event)

<<Entité>>  
**Reservation**  
+confirmer()

<<Infrastructure Service>>  
**Notifications**  
+notifieLogistique(event)

produit

<<Domain Event>>  
**ReservationConfirmee**  
+reservationId

**NotificationsParMail**

envoie sous forme de mail

Publish  
Subscribe



**BNP PARIBAS**

La banque d'un monde qui change

DDD Tactique avancé 269

# Agrégat



## ① Briques DDD plus avancées

- ① Agrégat
- ② Domain Event
- ③ CQRS



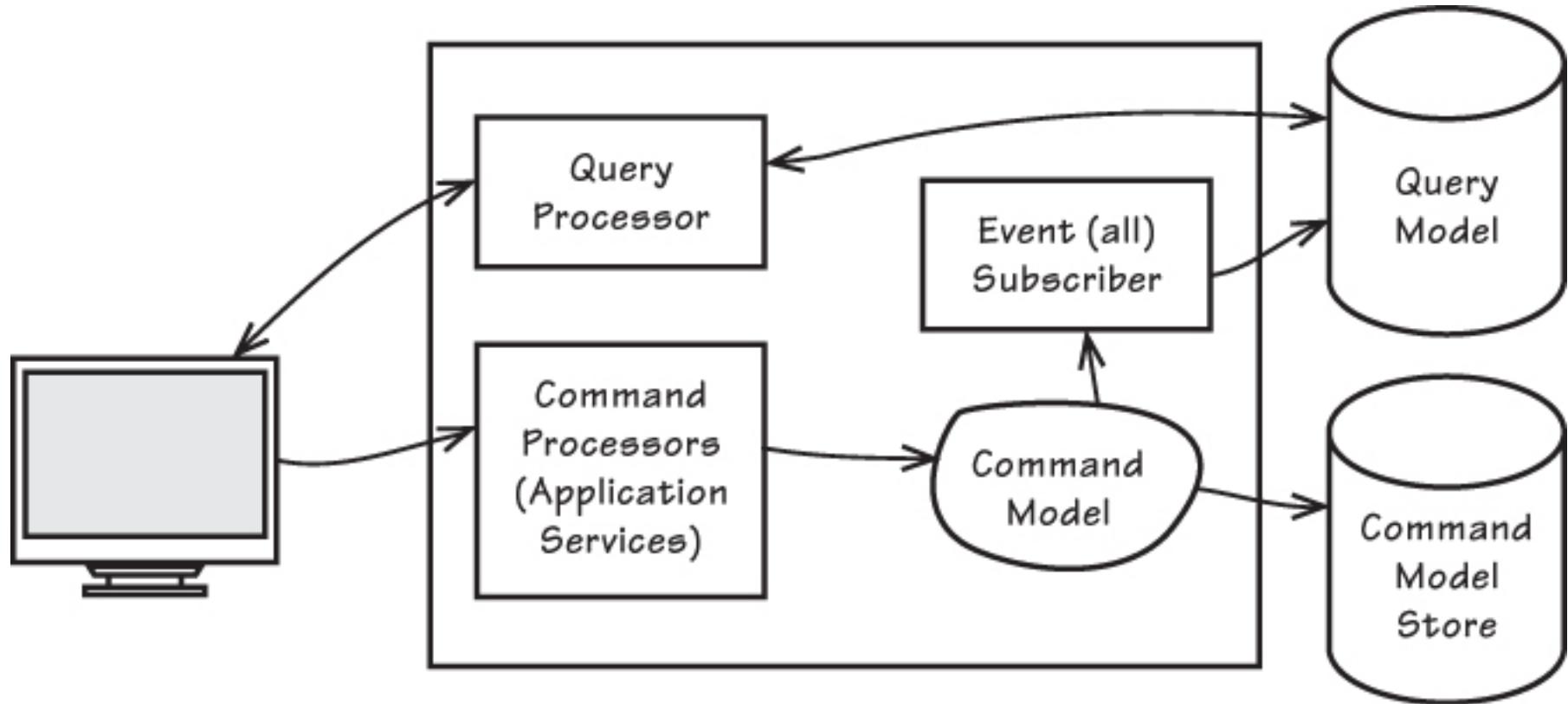
BNP PARIBAS

La banque d'un monde qui change

# CQRS – Une architecture orienté Evénement

- Acronyme de **Command Query Responsabilty Segragation**
- Principe: Chaque méthode, de manière exclusive, est soit:
  - Une Commande (**Command**) qui exécute une modification
  - Une Requête (**Query**) qui retourne une donnée à l'appelant
- Au niveau objet
  - Si une méthode modifie l'état de l'objet, il s'agit d'une commande,
    - La méthode ne doit rien retourner (ex: *void*)
  - Si une méthode retourne une valeur, il s'agit d'une requête
    - Pas de modification (directe ou indirecte) de l'état de l'objet





*Implementing DDD, Vaughn Vernon*



# Atelier sur les agrégats (1 / 5)

## Atelier:

Besoins exprimés par le métier :

- *Products have backlog items, releases, and sprints.*
- *New product backlog items are planned.*
- *New product releases are scheduled.*
- *New product sprints are scheduled.*
- *A planned backlog item may be scheduled for release.*
- *A scheduled backlog item may be committed to a sprint.*

Quelles règles implicites pouvez-vous proposer au métier ?

Quelle modélisation supporte l'ensemble des règles explicites et implicites ?



# Atelier sur les agrégats (2 / 5)

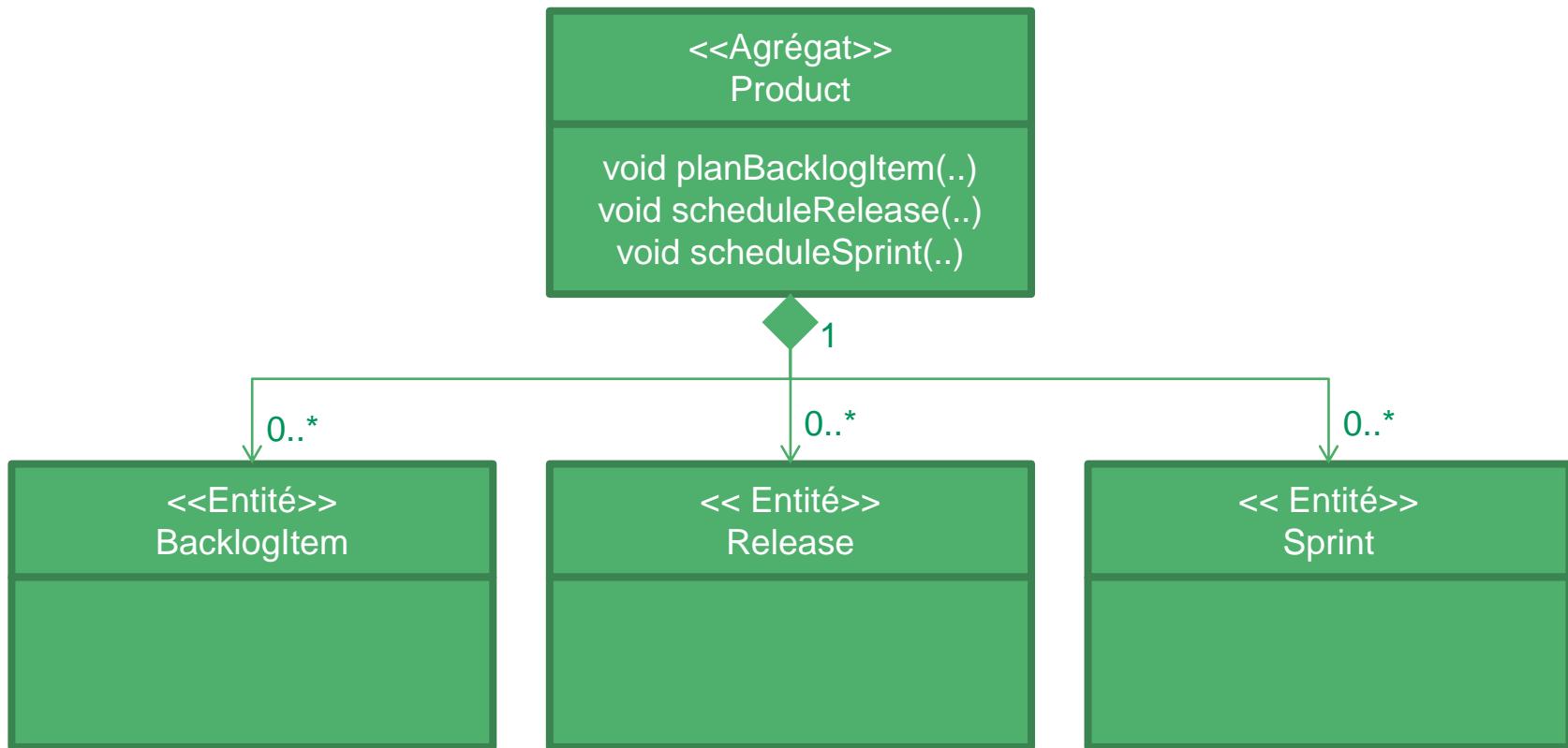
Règles implicites rendues explicites après une session de modélisation :

- *If a backlog item is committed to a sprint, we must not allow it to be removed from the system.*
- *If a sprint has committed backlog items, we must not allow it to be removed from the system.*
- *If a release has scheduled backlog items, we must not allow it to be removed from the system.*
- *If a backlog item is scheduled for release, we must not allow it to be removed from the system.*

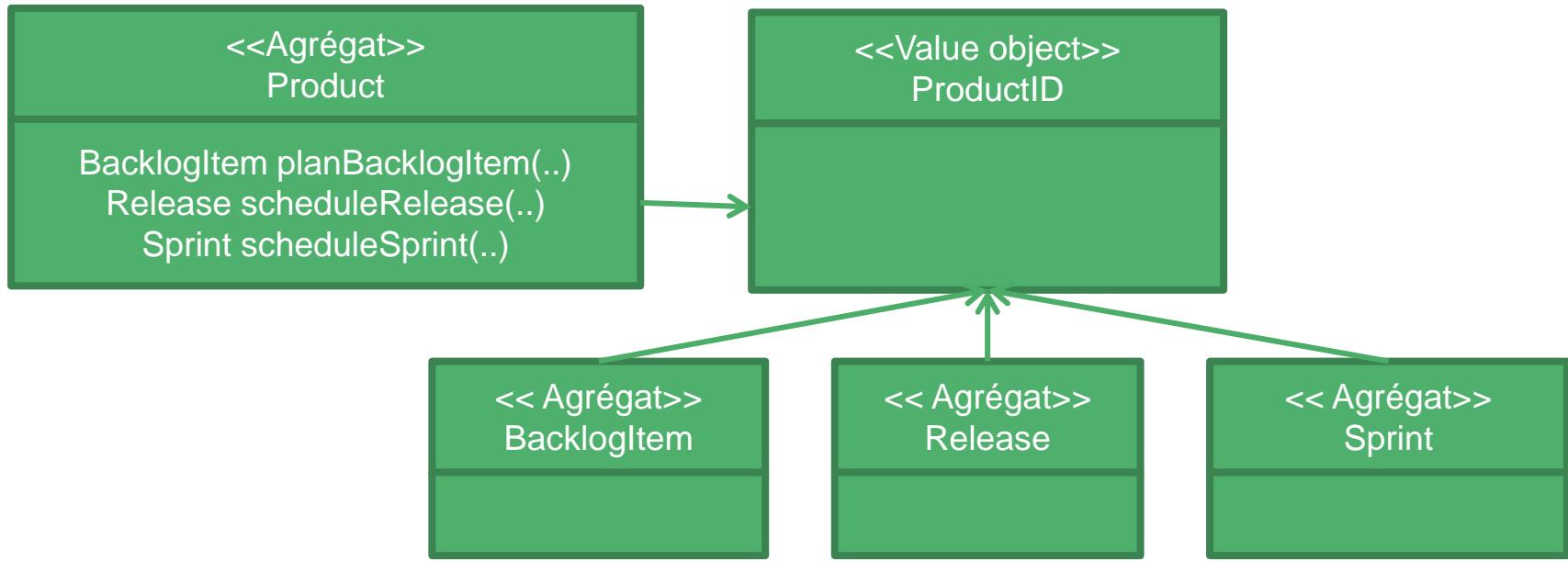


# Atelier sur les agrégats (3 / 5)

## Solution la plus évidente :

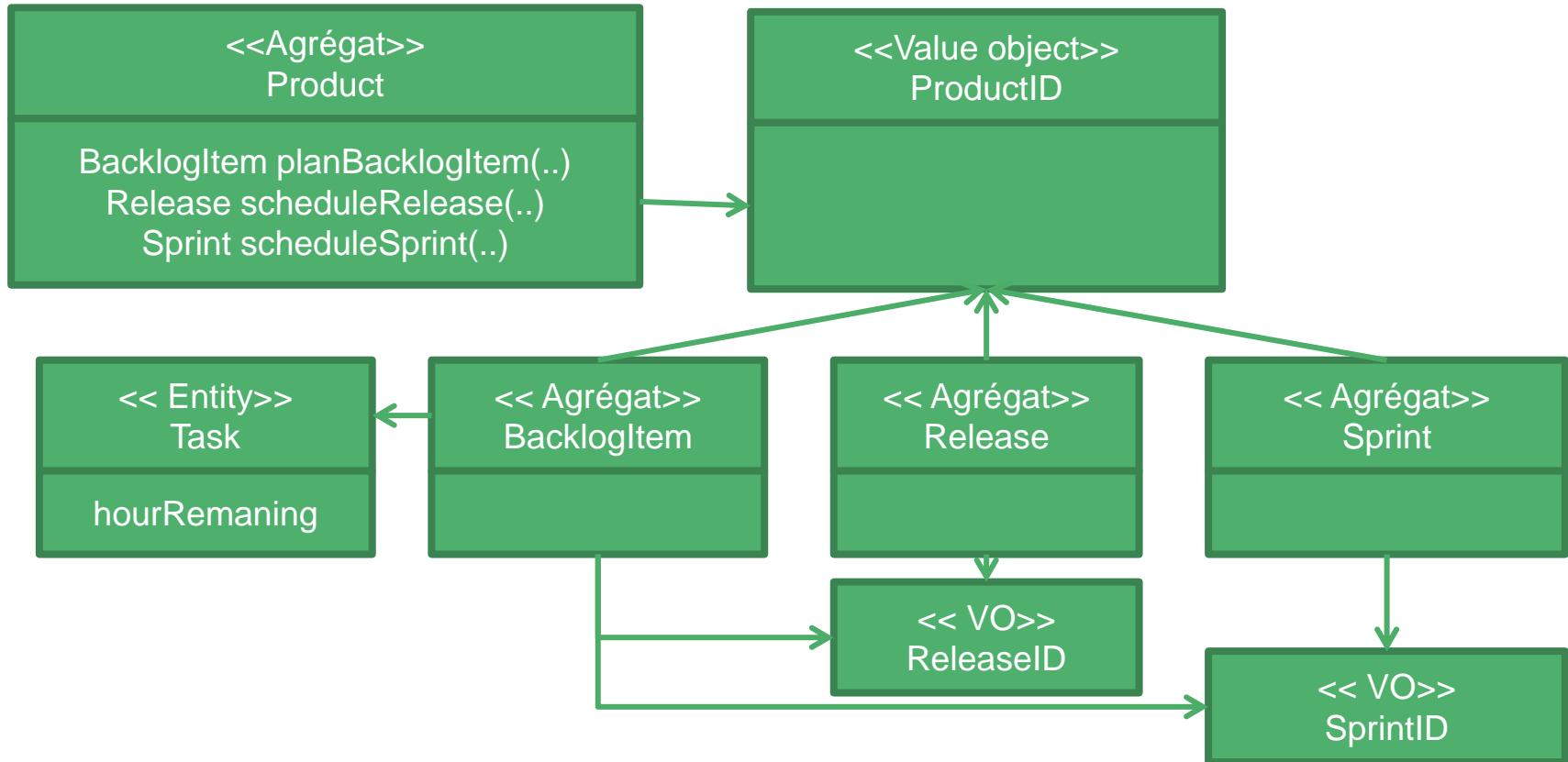


# Atelier sur les agrégats (4 / 5)



# Atelier sur les agrégats (4 / 5)

Nouvelle règle : Si la somme des estimations restantes est à zéro, la Release doit passer au statut « READY ». Comment faire ?



# Atelier sur les agrégats (5 / 5)

3 réponses potentielles:

- *Tout piloter depuis un service de la couche application : Cette solution fonctionne sur le papier mais elle est mauvaise car elle nécessite de modifier deux agrégats depuis un même service applicatif. L'exploitation ne sera pas satisfaisante.*
- *Utiliser des évènements du domaine : Vérifier la règle de gestion sur la somme des temps restants. Si la somme est nulle, on génère un événement. Un service abonné à cet évènement le consomme et met à jour la Release.*
- *Solution « eventual consistency » : est-ce bien le rôle d'un développeur de clôturer la version ? Vérifier si l'application immédiate de cette règle est une bonne chose : on ne peut plus revenir sur une autre tâche en cas de problème. La règle de gestion pourrait être appliquée lorsque le scrum master consulte le projet pour lui proposer de clôturer la release...*

