Domain Driven Design (DDD)

A quoi ca sert ? Est-ce utile ?

Objectifs

La conception de logiciels portant sur un domaine **métier complexe** se heurte très souvent aux mêmes problèmes récurrents :

- Code fragile et **rigide**, qui **vieillit mal**
- Transmission difficile des **connaissances** (turn-over régulier)
- **Perte** de crédibilité et de confiance dans l'application

Le **modèle** du domaine devient le **noyau** du logiciel, que ce soit du point de vue de l'architecture, du nommage des composants, ou de l'effort apporté.

- ~~Méthode~~ ~~Technologie~~ Façon de concevoir

Vocabulaire technique

Modèle

- Objets ayant une identité propre (entités)
- Les spécifications ou règles métier
- Les événements du domaine (métier)

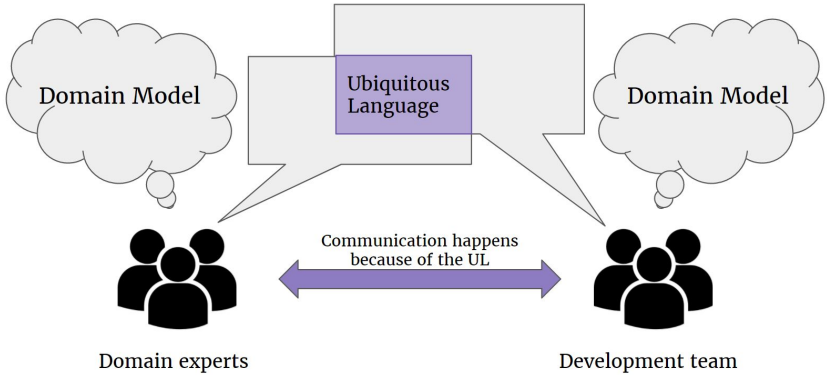
Vocabulaire technique

Contexte borné (bounded context)

- bornes naturelles des équipes métiers (structure équipes orientée produit)
- quand on utilise un langage différent (un produit à la vente s'exprime en prix, un produit en livraison s'exprime en poids et dimensions, etc...)

Ubiquitous Language (Language commun)

Définition du langage commun à tous les acteurs du projet



Vocabulaire technique

Entité

Une entité doit avoir une identité métier unique (ID).

Ex application bancaire :

virement = l'émetteur + destinataire + montant ➜ ~~entité~~ (domaine est une transaction bancaire et non seulement un virement)

⇒ rajout identifiant de transaction pour faire une entité

Vocabulaire technique

Objets-valeur (Value Objects)

Les objets valeurs sont complémentaires pour une entité :

- Ils n’ont pas d’identité
- Ils sont partagés entre plusieurs objets métiers
- Créés mais pas modifiés

Vocabulaire technique

Service

- Groupe les méthodes d’un même domaine

- Ex service de transaction :
  - les transactions d’argent (virement)
  - les transactions d’action (titre et valeur en bourse)

- Pas de logique métier

Vocabulaire technique

Module

- Domaine métier complexe: 100x de Services, Entités et Objets-valeurs.
- Structurer les fonctionnalités en modules logiques: simplifier
- Couplage faible (architecture flexible)

Vocabulaire technique

Agrégat

- A une racine (root) de type Entité

- Contrôle l'accès à une entité: un service doit passer par un agrégat pour accéder à des valeurs dans une entité.

- Ex de virement bancaire, le service transaction fait appel à l'agrégat correspondant et, par exemple, vérifie si le solde de l'émetteur est suffisant pour faire le virement.

- Contient les règles métier.

Vocabulaire technique

Fabrique (Factory)

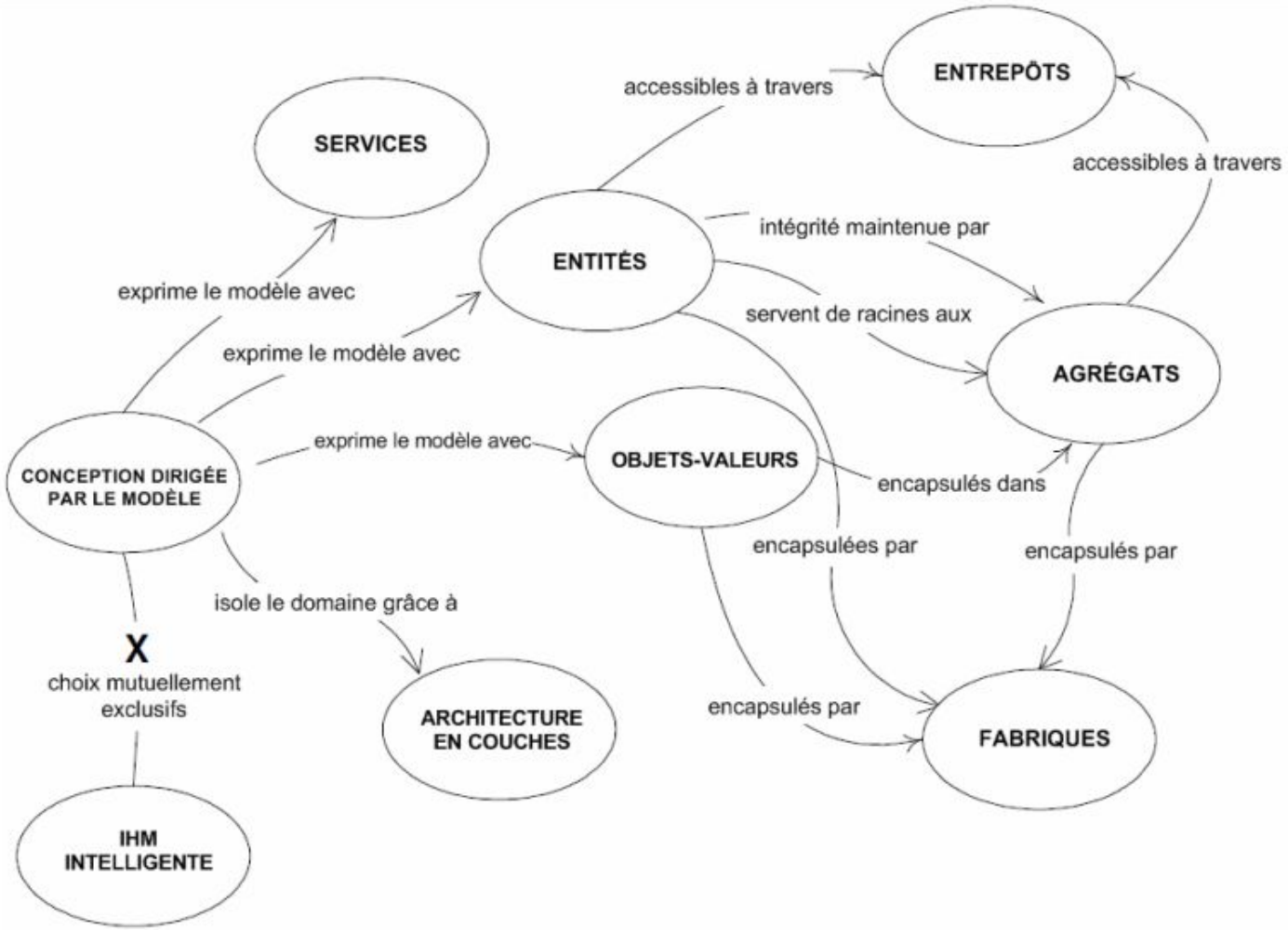
- Agrégat complexe : création simplifiée et propre

- Contient la logique et règles métier

Vocabulaire technique

Entrepôt (Repository)

- Accès à la BDD
- Interface : découple le code et le métier de la BDD
- Utilise les fabriques pour restituer des données.


ENTREPÔTS
accessibles à travers
SERVICES
accessibles à travers
intégrité maintenue par
ENTITÉS
exprime le modèle avec
servent de racines aux
AGRÉGATS
exprime le modèle avec
exprime le modèle avec
OBJETS-VALEURS
CONCEPTION DIRIGÉE PAR LE MODÈLE
encapsulés dans
encapsulées par
encapsulés par
isole le domaine grâce à
X
choix mutuellement exclusifs
ARCHITECTURE EN COUCHES
encapsulés par
FABRIQUES
IHM INTELLIGENTE

Evolution

- Le domaine métier change souvent
- Adapté aux méthodologie agiles
- Désignation d’un responsable de domaine
- Toute l’équipe de développement doit participer à la conception

Avantages

- Approche utilisateur : **expressif** et naturel pour un métier complexe
- **Dialogue** entre acteurs du projet obligatoire
- Meilleure **appréhension** du fonctionnel pour la maintenance et les tests: les règles métier sont explicitées, concentrées dans une couche bien définie de l'application
- La **robustesse** vis-à-vis des changements dans le SI grâce à la couche d'infrastructure

Inconvénients

- **Recherche** ou la **collecte** de données agrégées beaucoup de relations entre entités → grand nombre de jointures et problèmes de performances

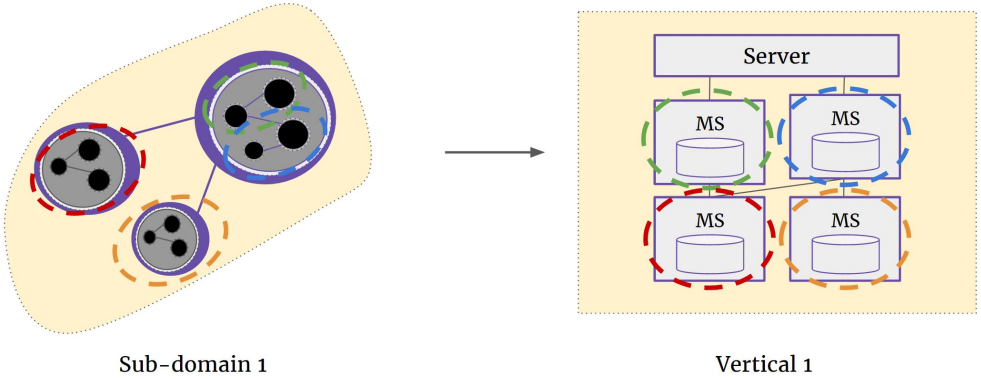
Dans le contexte microservices

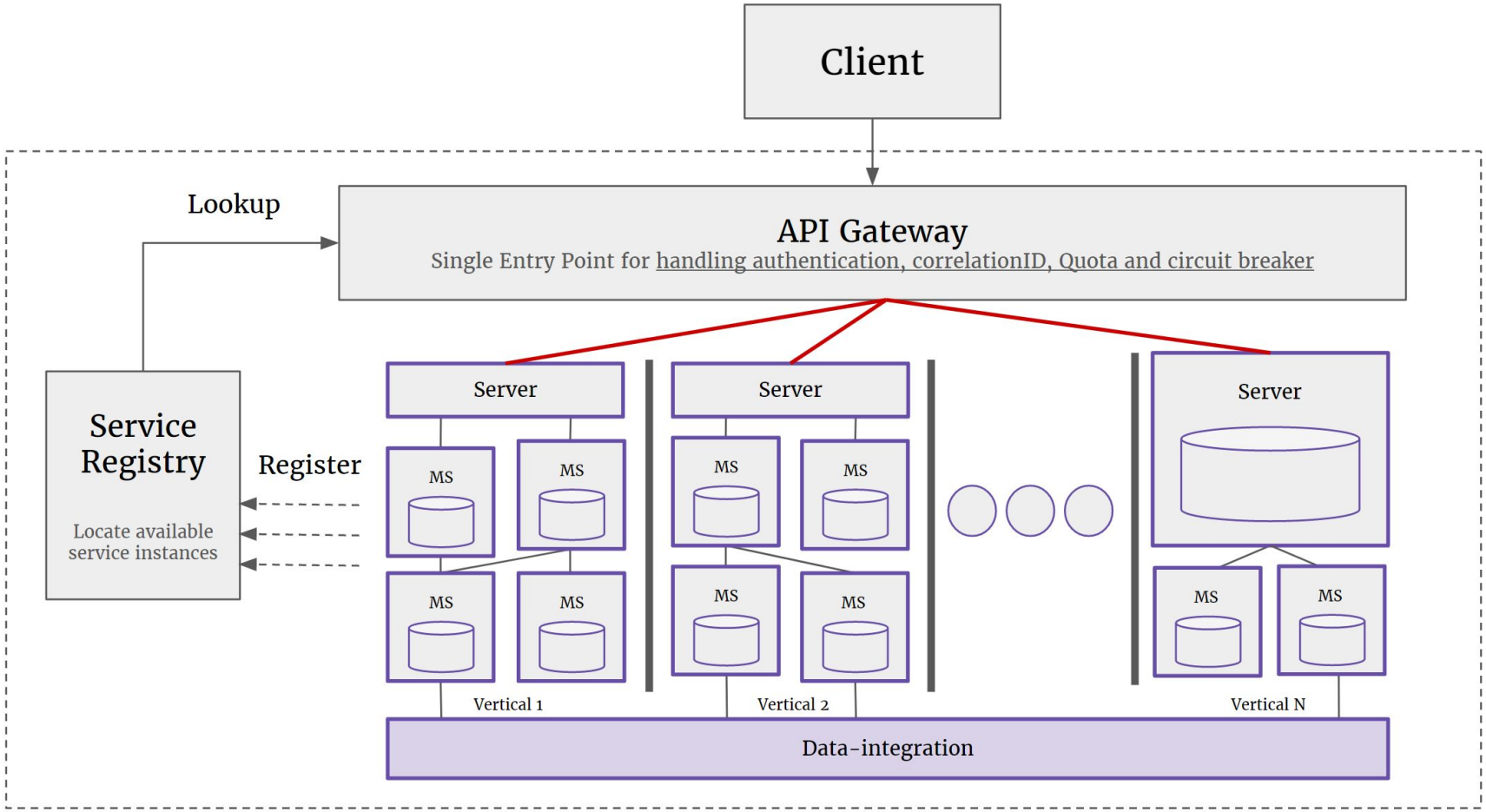
Architecture microservices

- assemblage d’unités indépendantes
- faiblement couplées
- fonctionnent de concert
- se répartissent les rôles pour atteindre un but fonctionnel.

Décomposition verticale

- "un microservice devrait faire une
  - une responsabilité unique (Single Res
  - contexte borné unique dans le domair
- Facteur “feature team” à prendre e




Client
Lookup
API Gateway
Single Entry Point for handling authentication, correlationID, Quota and circuit breaker
Service Registry
Locate available service instances
Register
Server
Server
Server
MS
Vertical 1
Vertical 2
Vertical N
Data-integration

Séparation en couches (N-Layer)

- Applicative :
  - API
  - minimaliste : coordination et délégation
- Modèle-domaine :
  - informations métier
  - régles métier
  - POCO
  - ignore la persistance
- Infrastructure :
  - persistence
  - ne “contamine” pas la couche modèle (indépendant)

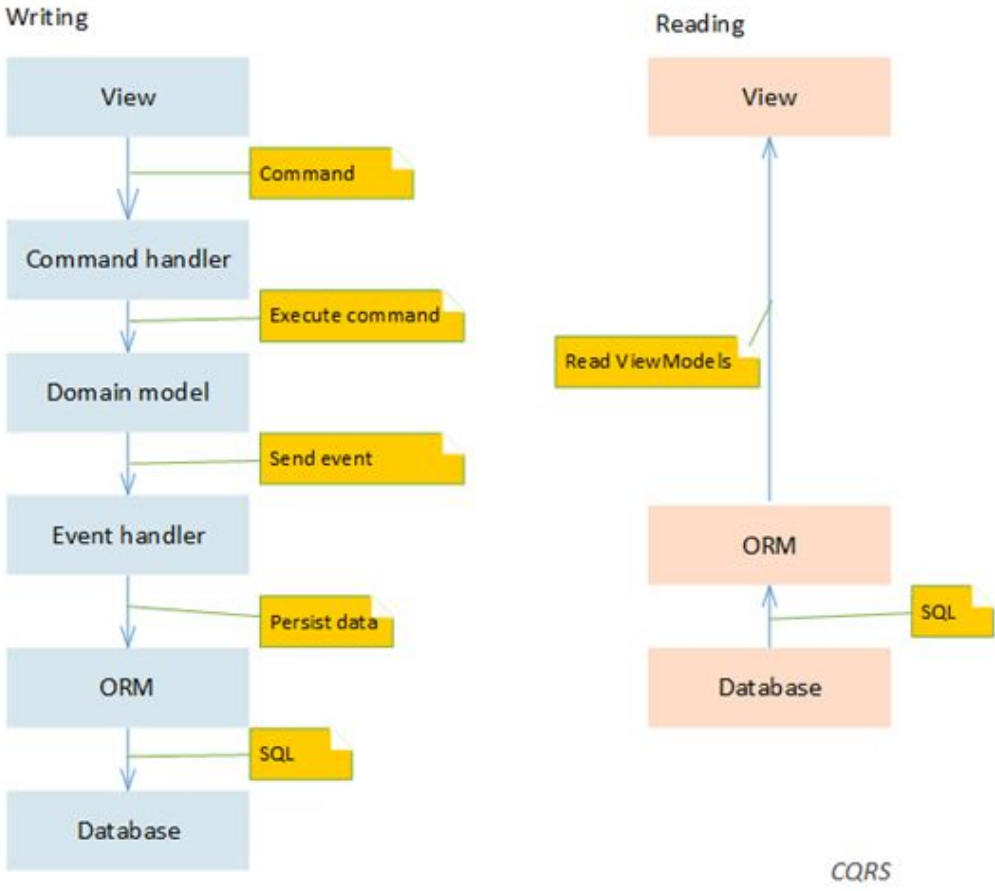
Architecture CQRS

Command

- évènements domaine-métier
- utilise agrégats

Query

- projection à partir des événements
- modèles-métiers dédiés lecture



Erreurs communes (1/2)

- Même architecture pour tous les contextes bornés (complexité variable)
- Réutiliser un modèle existant (“Don’t Repeat Yourself”).
- Résoudre une problématique sans chercher à en comprendre l’origine
  mauvaise compréhension ⇒ mauvaise définition (language, contexte borné)
- Négliger la carte de contexte (indispensable à la compréhension entre contexte bornés).
- Principes de DDD > problèmes de code ou technique !!!
- Ne pas garder d'ambiguïtés dans l’ubiquitous language ou limites de contexte (fort impact dans la conception et développement)

Erreurs communes (2/2)

- Trop d’abstraction : DDD n’a pas pour but de rajouter des couches d’abstraction inutile mais d’isoler la couche métier.

- Appliquer DDD lorsque le domaine est simple ou lorsque les acteurs du métier ne perçoivent pas l’intérêt du DDD ⇒ implication obligatoire et utile seulement si métier complexe.

- Sous-estimer le coût pour appliquer une démarche DDD, en effet DDD est coûteux en ressources et en temps car il faut impliquer les acteurs du métier et les développeurs dans l’élaboration de l’ubiquitous languageet dans la connaissance du domaine.

Exemple

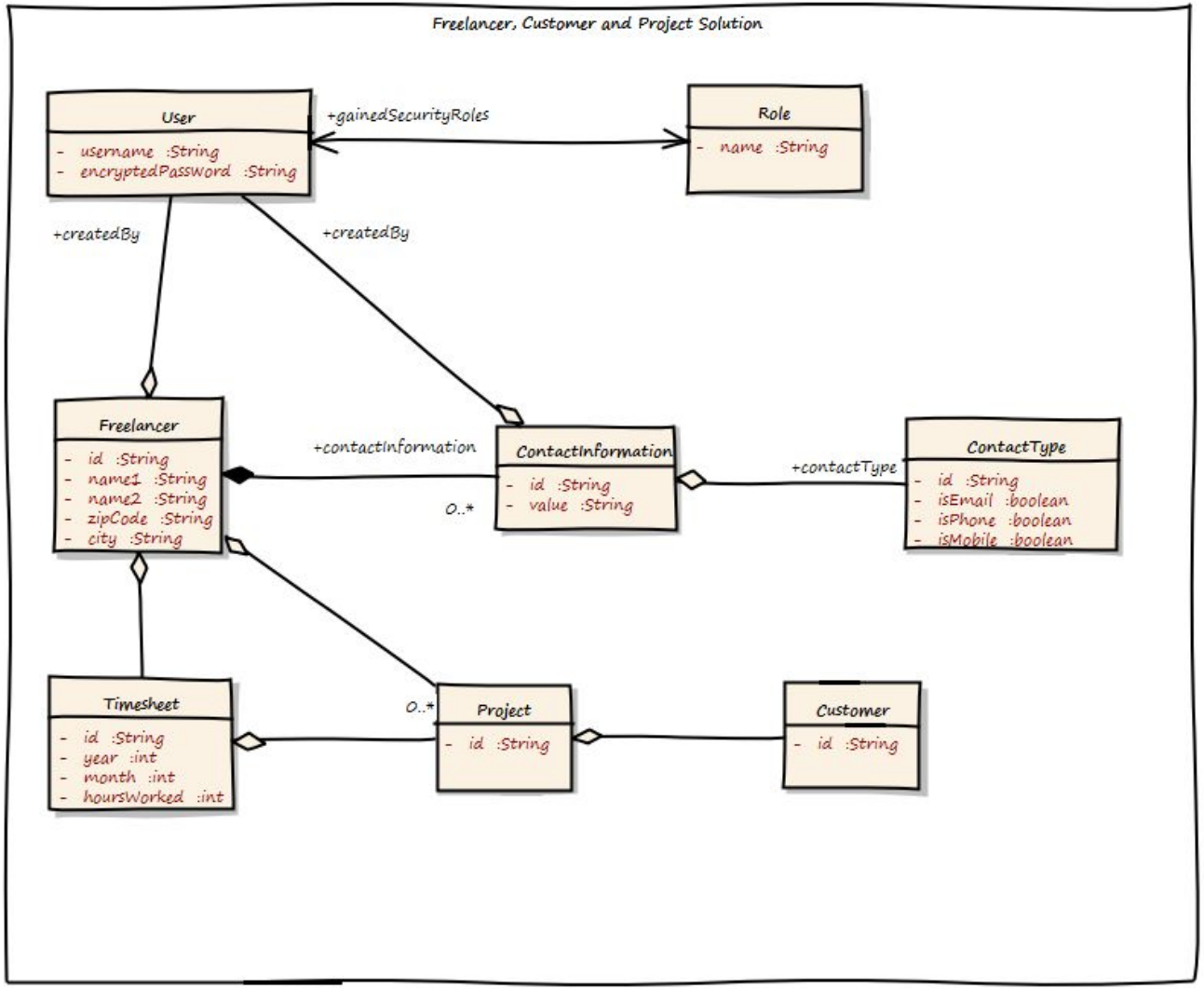
Inconvénients

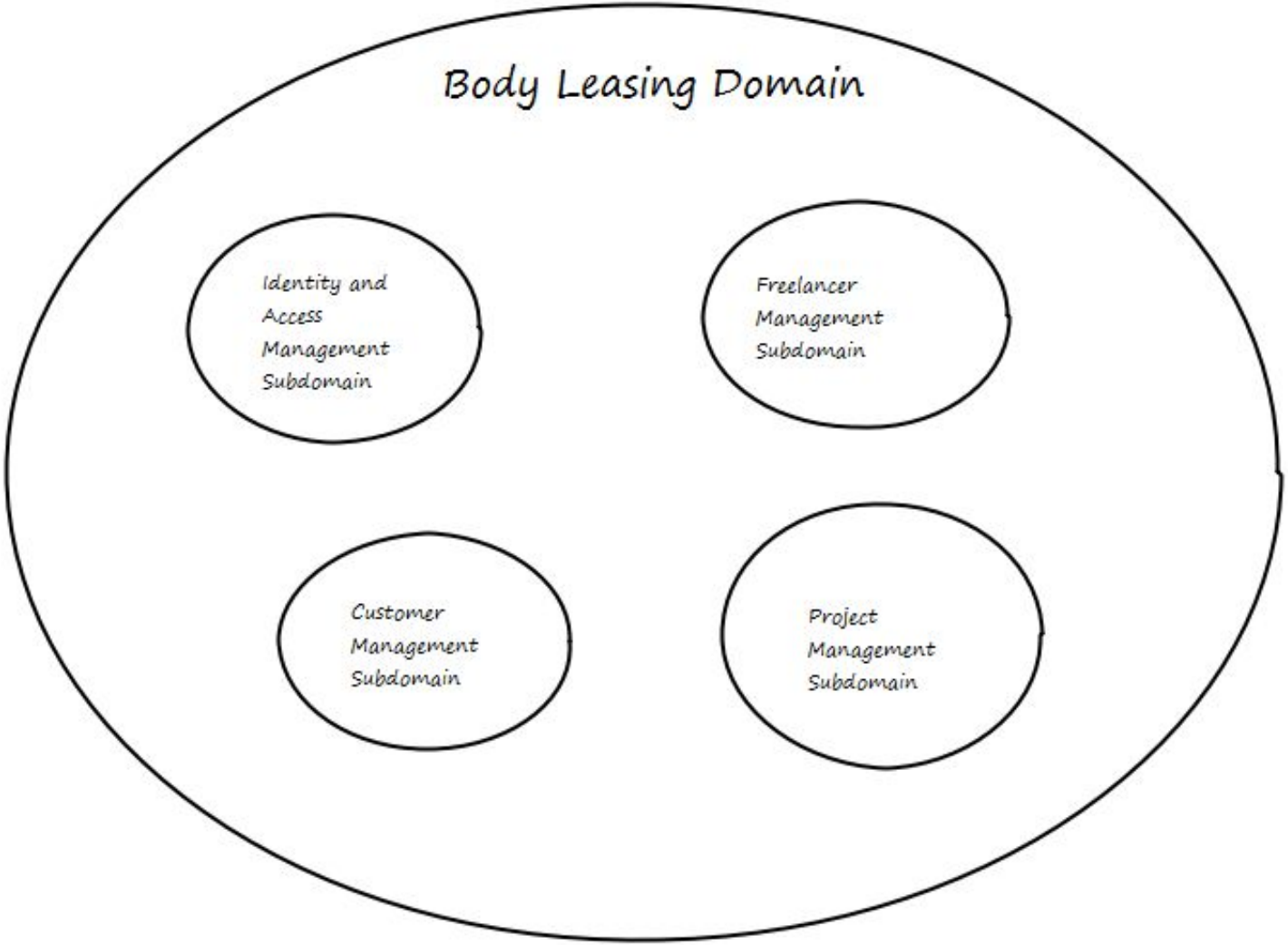
- adhésion acteurs métiers : convaincre de réaliser l'ubiquitous language et affiner la connaissance métier

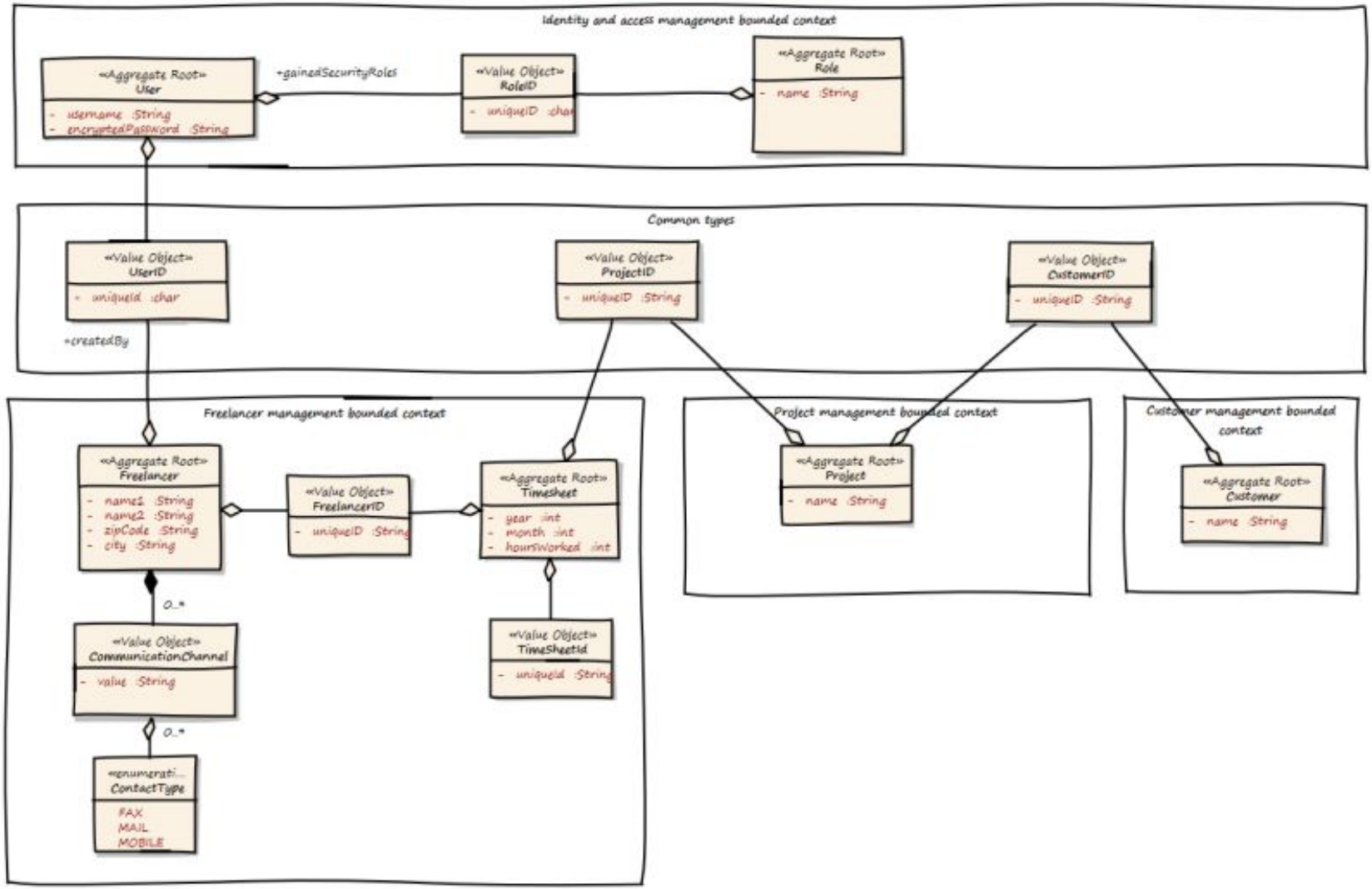
- apprentissage : il faut apprendre du langage et domaine avant de coder

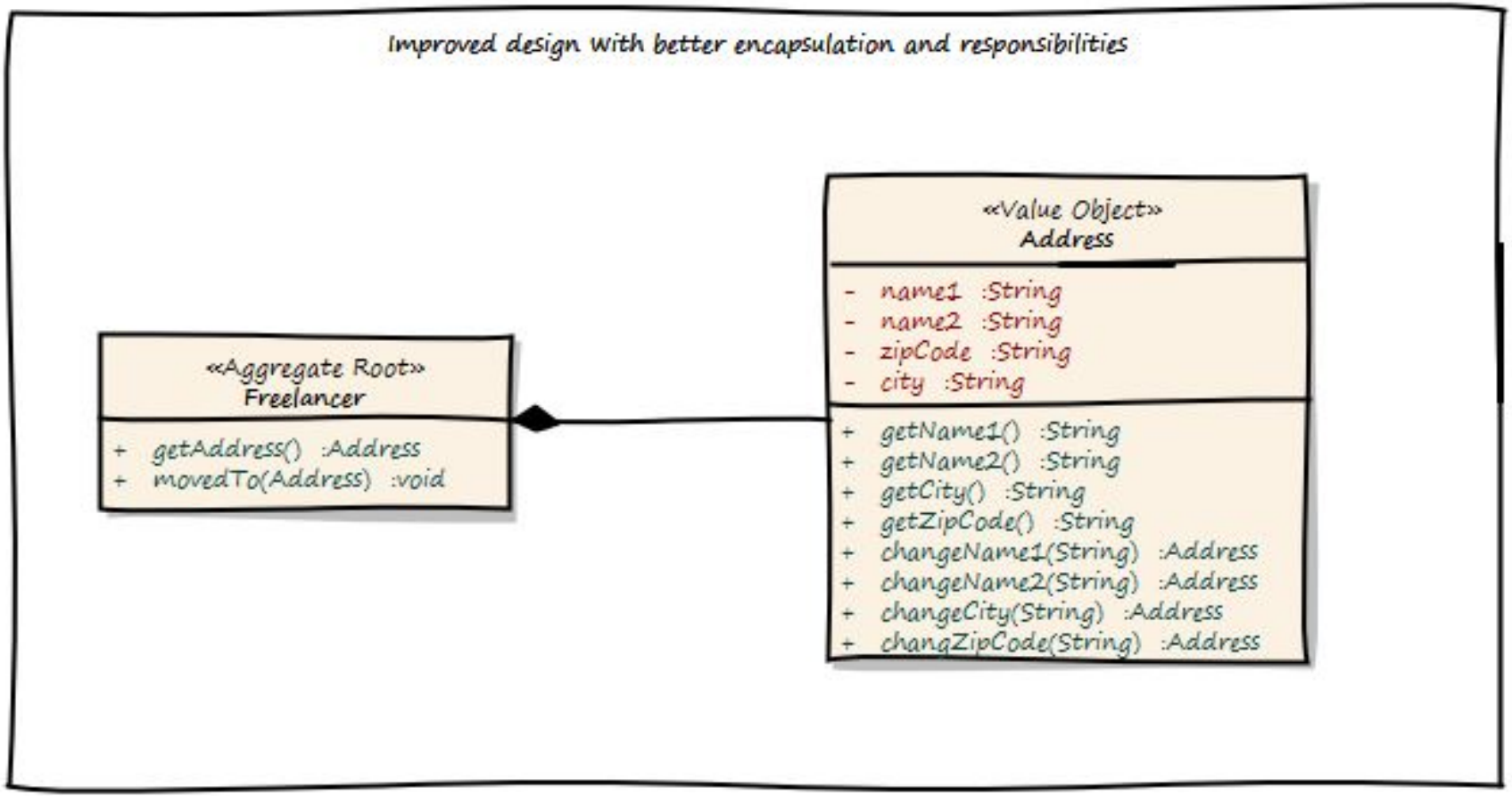
- représenter le domain model : de très grandes feuilles !!

- le modèle doit rester couplé au code : évolution de code ⇐⇒ évolution du domain model (équivalent à maintenir une documentation correcte et précise)


Freelancer, Customer and Project Solution
User
- username :String
- encryptedPassword :String
+gainedSecurityRoles
Role
- name :String
+createdBy
+createdBy
Freelancer
- id :String
- name1 :String
- name2 :String
- zipCode :String
- city :String
+contactInformation
0..*
ContactInformation
- id :String
- value :String
+contactType
ContactType
- id :String
- isEmail :boolean
- isPhone :boolean
- isMobile :boolean
Timesheet
- id :String
- year :int
- month :int
- hoursWorked :int
0..*
Project
- id :String
Customer
- id :String


Body Leasing Domain
Identity and Access Management Subdomain
Freelancer Management Subdomain
Customer Management Subdomain
Project Management Subdomain


Identity and access management bounded context
«Aggregate Root»
User
- username :String
- encryptedPassword :String
+gainedSecurityRoles
«Value Object»
RoleID
- uniqueID :char
«Aggregate Root»
Role
- name :String
Common types
«Value Object»
UserID
- uniqueId :char
+createdBy
«Value Object»
ProjectID
- uniqueID :String
«Value Object»
CustomerID
- uniqueID :String
Freelancer management bounded context
«Aggregate Root»
Freelancer
- name1 :String
- name2 :String
- zipCode :String
- city :String
«Value Object»
FreelancerID
- uniqueID :String
«Aggregate Root»
Timesheet
- year :int
- month :int
- hoursWorked :int
0..*
«Value Object»
CommunicationChannel
- value :String
«Value Object»
TimeSheetId
- uniqueId :String
0..*
«enumerati...
ContactType
FAX
MAIL
MOBILE
Project management bounded context
«Aggregate Root»
Project
- name :String
Customer management bounded context
«Aggregate Root»
Customer
- name :String


Improved design with better encapsulation and responsibilities
«Aggregate Root»
Freelancer
+ getAddress() :Address
+ movedTo(Address) :void
«Value Object»
Address
- name1 :String
- name2 :String
- zipCode :String
- city :String
+ getName1() :String
+ getName2() :String
+ getCity() :String
+ getZipCode() :String
+ changeName1(String) :Address
+ changeName2(String) :Address
+ changeCity(String) :Address
+ changZipCode(String) :Address