

Les trois patterns du DDD

...

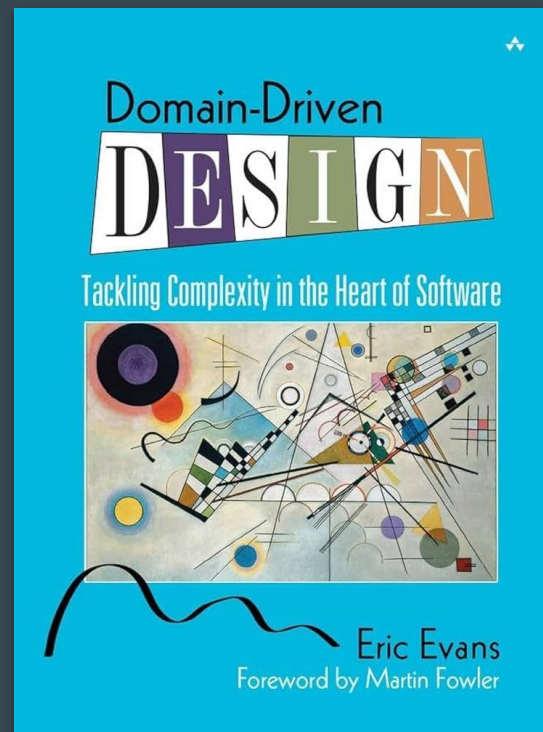
Sommaire

- Introduction
 - Qu'est ce que c'est ?
 - Origine
 - But
- Les 3 patterns techniques
 - Value Objects
 - Entities
 - Aggregates
- Problématique 1
 - Solution “naïve”
 - Solution avec le DDD
- Problématique 2
- Limites/ Avantages
- Live coding

Domain-Driven Design: d'où vient-il ?

- Créateur: Eric Evan (2003)
- Conception pilotée par le domaine
- Technique de conception logicielle
- Mettre le métier au centre du développement
- Ubiquitous language (Langage commun)
- 3 patterns techniques

20 ans de
best-practices de
la POO



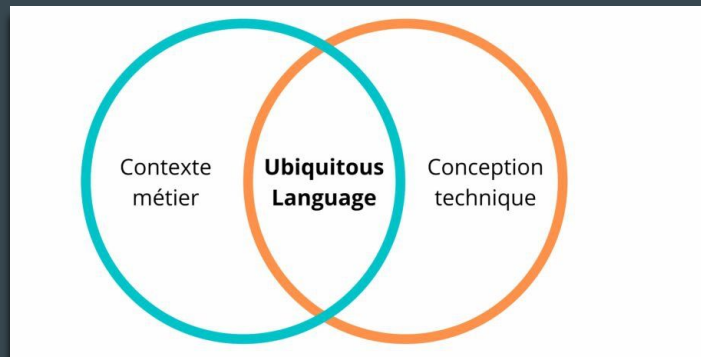
Qu'est ce qu'un pattern :

- Patron de conception
- Donne des solutions aux problèmes communs
- Solutions standards dans la création de logiciels



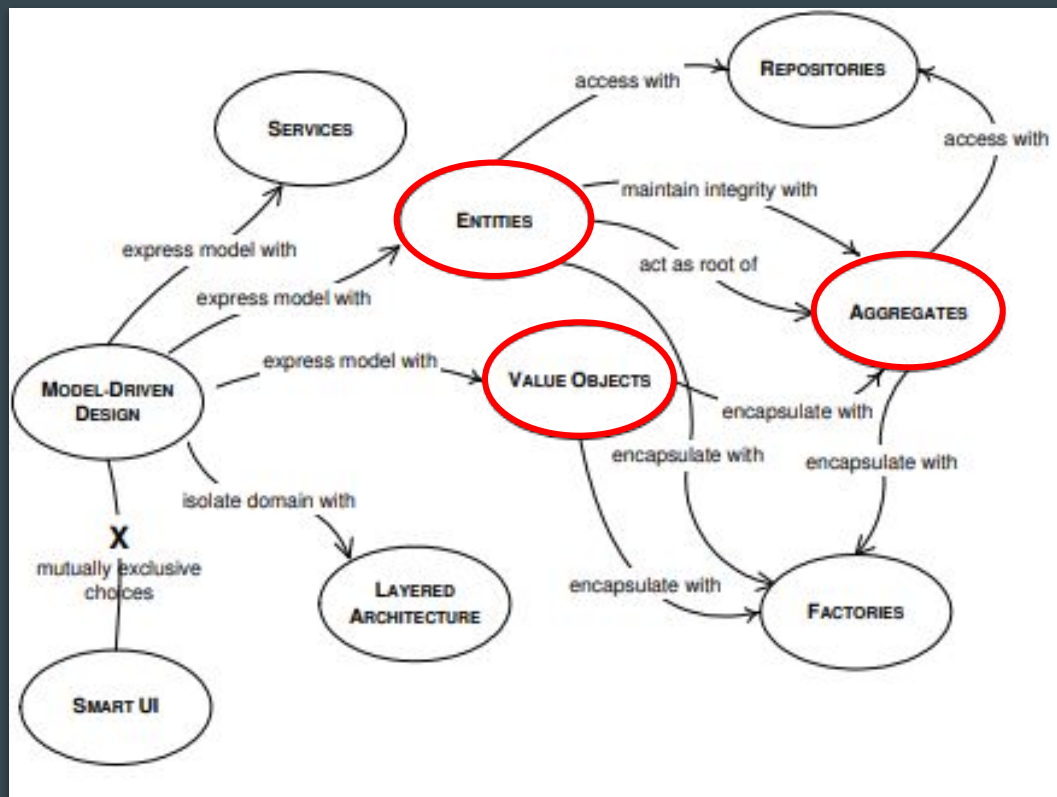
DDD: Quand l'utiliser

- Métier ou application complexe
- Langage métier :
 - Le langage métier est définie avec le client et les développeurs
- Métier avec des risques



Les 3 patterns techniques du DDD

- Value Object
- Entity
- Aggregate



Value Object

- Pas d'identifiant
- Immutable
- N'aura pas de setter
- Peut être remplacé par une autre instance
- Représente un attribut (position, couleur, adresse...)

Entity

- Possède un identifiant
- Possède des attributs (Value Objects)
- Mutable
- Ex: `Personne(id, nom, prénom)`

Aggregate

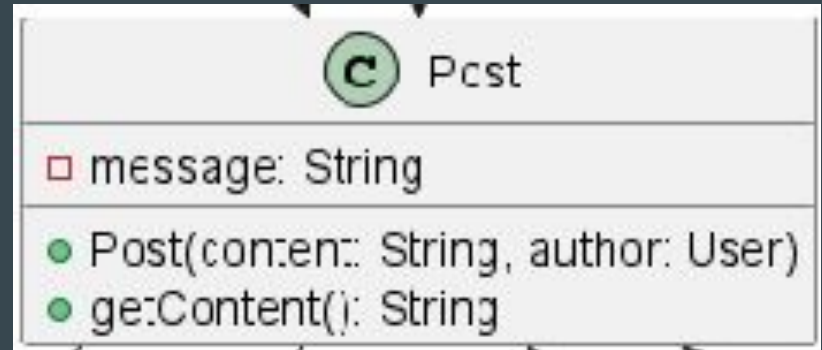
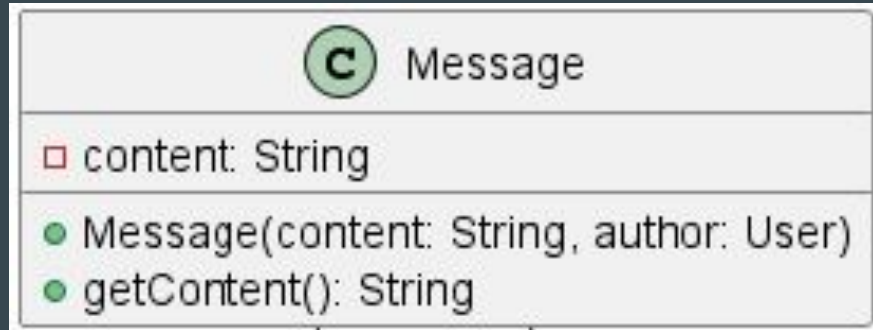
- Ensemble d'Entities et de Value Objects
- Permet la cohérence des objets
- Dispose d'une Racine / Root (dite racine d'agrégat)
- Ex: Une commande et ses articles

Problématique 1: Réseau social

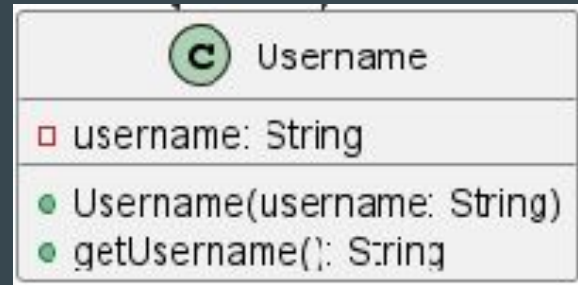
- Utilisateurs
- Voir des messages
- Envoyer/modifier des messages



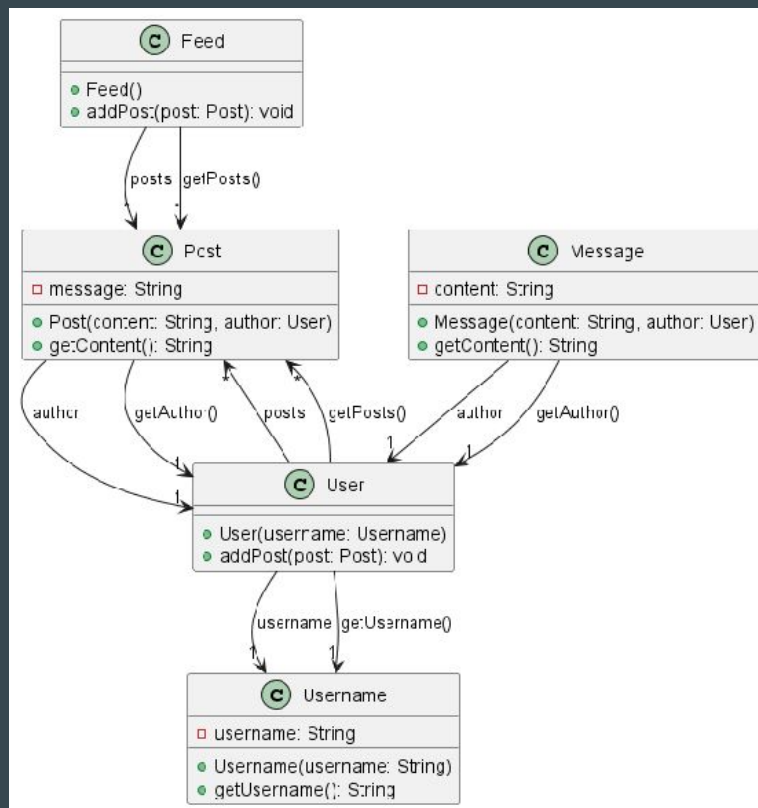
Première solution



Première solution



Première solution



Première solution : les inconvénients

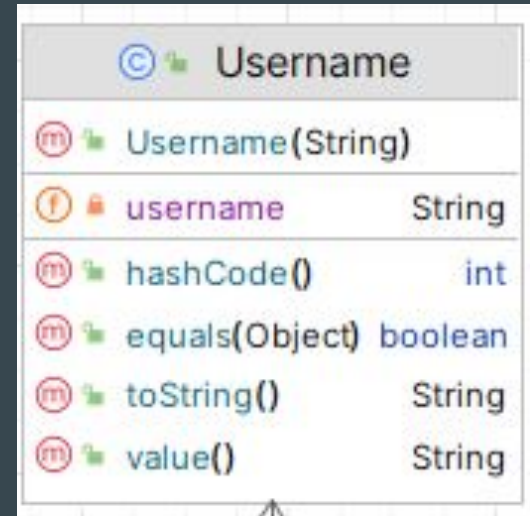
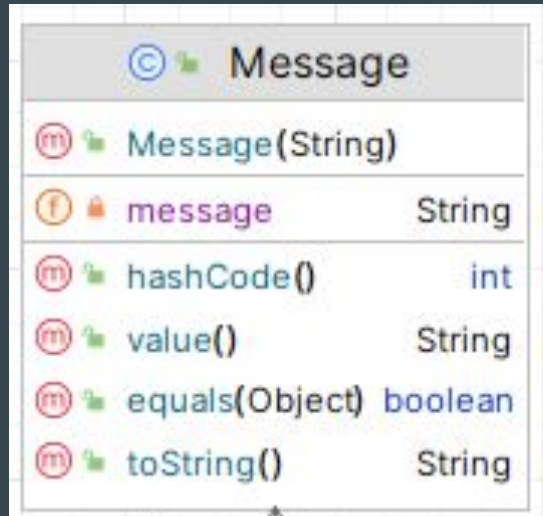
- Moins modulaire et moins explicite dans la modélisation des domaines.
- Manque de structure modulaire
- Maintenir la Cohérence des Données
- Ajouter de nouvelles fonctionnalités
- Difficulté à gérer les erreurs



Solution avec DDD

Nos Value-Objects :

Nom d'utilisateur et message



Nos Entities:















Utilisateur et publication du
réseau

User	
User(long, Username)	
username	Username
id	long
changeName(Username)	void
name()	Username
equals(Object)	boolean
hashCode()	int
toString()	String

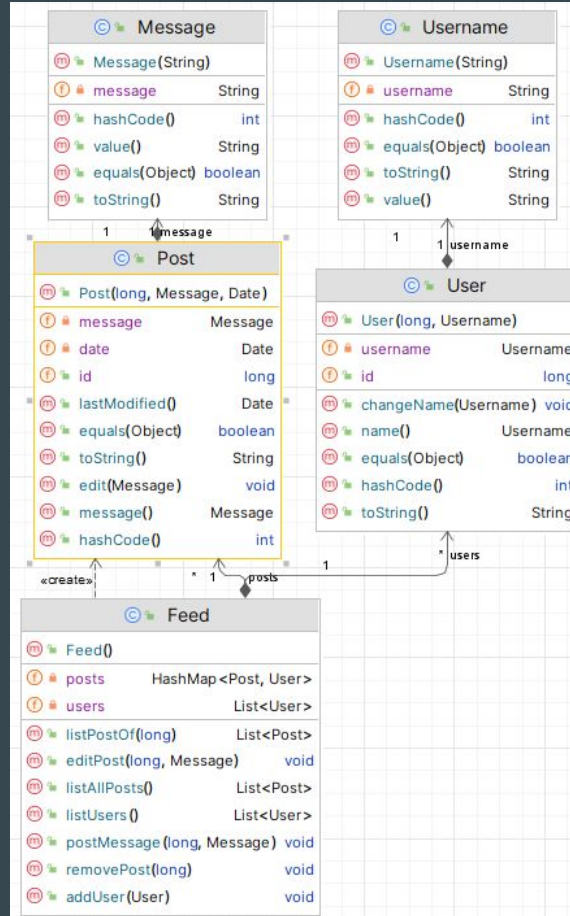
Post	
Post(long, Message, Date)	
message	Message
date	Date
id	long
lastModified()	Date
equals(Object)	boolean
toString()	String
edit(Message)	void
message()	Message
hashCode()	int

Notre Aggregate:

Ensemble des posts et des utilisateurs du réseau

Feed		
		Feed()
		posts HashMap<Post, User>
		users List<User>
		listPostOf(long) List<Post>
		editPost(long, Message) void
		listAllPosts() List<Post>
		listUsers () List<User>
		postMessage (long, Message) void
		removePost(long) void
		addUser (User) void

Solution :



Limite du DDD

- Coût énergétique
- Difficile à appréhender
- Coûteux en temps



Avantages du DDD

- Equipe intégré au contexte métier
- Code simple à appréhender
- Architecture modulaire
- Permet des mise à jour facile



Principes SOLID :

- SRP
- OCP



Live coding



Source :

- <https://thetribe.io/glossaire-domain-driven-design/>
- <https://khalilstemmler.com/articles/typescript-domain-driven-design/entities/>
- <https://medium.com/codex/ddd-entity-and-value-types-ad08c2962fd>
- <https://www.jamesmichaelhickey.com/domain-driven-design-aggregates/>
- <https://alexsoyes.com/ddd-domain-driven-design/>
- <https://lesdieuxducode.com/blog/2019/7/introduction-au-domain-driven-design>

QCM : Kahoot !