DDD : et si on reprenait tout par le bon bout ? tout simplement

Speakers : Thomas Pierrain et Jérémie Grodziski
Format : Conférence
Date : 22 avril 2016

Thomas et Jérémie commencent par faire référence au Blue Book Domain Driven Design d'Eric Evans. Livre salué lors de sa sortie. Mais 13 ans après, très peu font du DDD au boulot
Pourquoi le DDD n'a-t-il pas pris ?
Trop de mots compliqués alors que le DDD « c'est simple »
C'est avant tout se concentrer sur la valeur métier, et cela avant la technique

Le 1er boulot du développeur, c'est de comprendre le Domaine sur lequel il va travailler.
Qu'est-ce que le Domaine ?
Exemple 1 : Comptabilité en partie double. Concepts : Compte, Débit, Crédit, Montant.
Problèmes résolus : traçabilité.
Exemple 2. Domaine IDE. Concepts : projets, fichiers, analyse, refactoring, vcs, debugger.

Le DDD est une super boîte à outils agnostique à la technologie. Très pérenne.
Le dernier framework JS ne va pas le démoder.

Niveau 1 : le code / le binôme

Exemple de code illisible.
Zoom sur les code smells.

- Les magic number sans sens
- Duplication de code
- Primitive obsession : usage intensif du BigDecimal. Ne porte pas beaucoup d'informations
- Préoccupations mélangées avec du SQLException qui remonte, du contrôleur
- Terminologies foireuses : i, EJB

Au niveau métier, on cherche les implicites

- Livraison gratuite ?
- Cout de la livraison ?
- Devise en euros ?

- Poids en grammes ? Coût variable pour chaque Kg expédié ?
- Est-ce des options de livraison ?

Beaucoup de lignes de code pour beaucoup de questions.

Le DDD cherche à lever les implicites, donner du sens. Quelle est l'intention ?

- Signature de la méthode explicite
- Magic number en constante
- Méthode chargée de calculer le coût de livraison
- Utilisation du value type Amount à la place du BigDecimal

Un expert du Domaine doit être capable de comprendre le sens du code.

Un **Value Type** : l'important est la somme des attributs, pas qui ils sont. Exemple : billet de 10 euros échangés entre les 2 speakers (sauf pour la BDF). L'importance c'est le montant.
Caractéristiques :

- Immutable par définition
  - Constructeur dit transactionnel => contraintes renforcées
  - Fail fast
  - Pas de setteu
- Riches en logique du domaine
- Sur tous les attributs
  - Egalité forte : equals
  - Unicité forte : hashCode
- Fonctions combinables
  - Au lieu de changer l'état, on compose puis on réaffecte. Lors d'une addition, un nouveau montant est retourné. On retrouve le paradigme fonctionnel

Permet d'aller à l'essentiel : complexité essentielle, complexité accidentelle

Pourquoi en DDD on appelle cela Value Object ? C'est un oxymore car l'objet bouge. Pas très bien choisi à l'époque. On parle désormais de Value Type.

Pour les speakers, **le Value Type est le meilleur ROI du DDD** pour les raisons suivantes :

- Exprime le domaine
- Donne l'exemple
- Introduction progressive sur la codebase
- Avaleur de complexité
- Simplifie le raisonnement (immutable, thread-safe, testable)

« Plante la graine du Domaine dans le code »

Jérémie a envie de shooter les classes nommées en Manager, les helpers, un user ….

La solution du DDD : utiliser le même langage que les utilisateurs. C'est ce qu'on appelle l'Ubiquitous Language. Dans certains cas, l'utilisateur est multiple. Le DDD répond à çà par le Contexte.

Le Contexte : groupe de personnes qui vont utiliser un mot ou une expression.
En DDD, on parle de **Bounded Context**. L'importance est de déterminer les groupes de personnes qui vont utiliser l'application.

Exemple sur Amazon : quelles sont les contextes ?

- Mise en avant
- Catalogue de produits
- …

Attention au modélisme « Modéliser pour modéliser ». Les modèles sont aux services des usages.

Niveau 2 : l'appli / l'équipe

Pour éviter le plat de spaghettis ou l'effet mikado, utiliser l'**Architecture Hexagonale**.
Le monde se divise en 2 catégories :

1. **Le dedans** : le métier
2. **Le dehors** : infrastructure, code technique, API, SPI …

Il faut isoler les 2. Hexagonal <> Layers.

Un autre problème courant : arriver dans le bourbier du legacy.
Question courante : « On refait tout ou pas ? »
A l'intérieur d'un contexte legacy, on peut créer un Bubble Contexte DDD bien pensé. On ajoute un anti-corruption layer.

Niveau 3 : le SI

Dans une entreprise, il va falloir communiquer : entres équipes et entre systèmes.
Il y'a du pouvoir dans l'entreprise.
Il faut expliciter les rapports de pouvoir qu'il y'a. Le DDD utilise la métaphore du courant de rivière. Si quelqu'un fait quelque chose, qui est impacté ?
Pour se protéger des changements du upstream, on peut mettre en place le pattern anti-corruption pattern.
Innovation du DDD : Strategic Design.

Conclusion

Le DDD est beaucoup plus facile et applicable que ce que l'on peut penser.
Cette conférence n'a donné qu'une vue partielle de tous les patterns rendus accessibles.

Let’s reboot DDD.
Communauté ouverte de praticiens : @DDDreboot + #slack
Populariser le DDD en étant plus accessibles : faire du DDD puis dire après ce que c’est.
Etendre sa boîte à outils.