Comprendre le Domain-Driven Design (DDD) et les Bounded Contexts

Exemples Concrets et Étude de Cas : Netflix

Djebabla Ammar

November 27, 2025

Contents

1 Introduction

Le Domain-Driven Design (DDD) est un ensemble de principes, de pratiques et de modèles visant à concevoir des systèmes logiciels plus robustes, plus cohérents et mieux alignés sur la logique métier. DDD place le **domaine métier** au centre de l'architecture logicielle. Il vise à réduire la complexité, clarifier les responsabilités et faciliter l'évolution.

Dans ce document, nous allons :

- Comprendre les concepts clés de DDD,
- Étudier le concept de **bounded context** en profondeur,
- Illustrer chaque concept avec des exemples,
- Étudier un cas concret de niveau international : **Netflix**.

2 Concepts fondamentaux du DDD

2.1 Le Domaine

Le domaine représente la réalité métier. Pour Netflix, les domaines principaux sont :

- La gestion du catalogue de films et séries,
- Le système de recommandations,
- La gestion des utilisateurs et des profils,
- La gestion des abonnements et de la facturation,
- La diffusion et le streaming.

2.2 Ubiquitous Language (Langage omniprésent)

Tous les acteurs du projet (développeurs, experts métiers, designers) doivent partager le même vocabulaire. Exemples Netflix :

- *Title* = un film ou une série,
- *Profile* = un sous-compte utilisateur,
- *Playback* = un événement de lecture,
- *Recommendation* = une suggestion personnalisée.

3 Bounded Context : le coeur de DDD

3.1 Définition

Un **bounded context** (contexte borné) est une zone clairement délimitée où un modèle métier spécifique s'applique. Chaque contexte possède :

- ses modèles,
- ses entités,
- son langage,
- ses règles métier,
- ses données.

3.2 Pourquoi des bounded contexts ?

Imaginez Netflix sans séparation :

- Le catalogue,
- Le système de facturation,
- Le moteur de recommandation,
- Le système de streaming,

Tout serait mélangé dans une seule base de données, un seul modèle, et chaque changement risquerait de tout casser.

Les bounded contexts permettent :

- Modularité,
- Indépendance des équipes,
- Facilité d'évolution,
- Robustesse,
- Microservices clairement alignés.

4 Les bounded contexts de Netflix

Voici une proposition réaliste et représentative des vrais systèmes de Netflix :

4.1 1. Catalogue Context

Responsable de :

- Films, séries, acteurs, catégories,
- Disponibilité selon la région,
- Métadonnées (résumé, durée, images),
- Versions (4K, doublages, sous-titres).

Entité principale : Title **Value objects :** Language, Region, Quality-Format

4.2 2. User Management Context

Responsable de :

- Comptes utilisateurs,
- Profils multiples,
- Historique de visionnage,
- Contrôles parentaux.

Entité principale : User, Profile

4.3 3. Billing Context

Responsable de :

- Les abonnements,
- Les paiements,
- Les renouvellements automatiques,
- Les factures.

Entité principale : Subscription **Value object :** PaymentMethod

4.4 4. Recommendation Context

Responsable de :

- Algorithmes de personnalisation,
- Similarité entre contenus,
- Classement des suggestions.

Ce contexte **lit** depuis le Catalogue et l'User Management, mais ne modifie rien directement.

4.5 5. Streaming Context

Responsable de :

- Distribution vidéo,
- CDN,
- Optimisation adaptative,
- Événements de lecture (playback events).

5 Relations entre les contexts

Chaque contexte ne doit pas dépendre intimement des modèles des autres. On utilise généralement :

- Des événements de domaine,
- Des DTO,
- Des API contractuelles.

6 Exemple complet : Workflow Netflix

Nous allons suivre un scénario simple : **Un utilisateur regarde un film, et Netflix met à jour son historique et ses recommandations.**

6.1 Étape 1 : Le Catalogue fournit les données du film

Catalogue Context répond :

```
Title(id=43, name="Stranger Things", type="Series", ...)
```

6.2 Étape 2 : Le User Context enregistre l'événement de visionnage

User Management :

```
PlaybackEvent(
    user_id=22,
    profile_id=3,
    title_id=43,
    duration=48min
)
```

6.3 Étape 3 : Le Recommendation Context ajuste l'algorithme

Recommendation Context reçoit l'événement :

```
UserWatched(title_id=43)
```

Il met à jour :

- scores de similarité,
- courbes de préférence,
- classement des suggestions.

6.4 Étape 4 : Le Billing Context n'est pas concerné

Il est complètement isolé — c'est ça, un bounded context bien conçu.

7 Modélisation DDD détaillée : Exemple de Netflix

7.1 Entité du Catalogue : Title

```
class Title:
    id: TitleID
    name: str
    genres: list[Genre]
    formats: list[QualityFormat]
    region_availability: list[Region]
```

7.2 Value Object : QualityFormat

```
class QualityFormat:
    resolution: str   # "HD", "4K"
    sound: str        # "Dolby Atmos"
```

7.3 Aggregate : User

```
class User(AggregateRoot):
    id: UserID
    profiles: list[Profile]
    payment_method: PaymentMethod
```

7.4 Domain Service : RecommendationEngine

```
class RecommendationService:
    def suggest_for_profile(profile_id):
        # règle métier complexe
        return ranked_titles
```

8 Pourquoi Netflix utilise DDD ?

- Les équipes sont grandes et distribuées,
- Le système doit être scalable,
- Les fonctionnalités évoluent indépendamment,
- Le streaming ne doit pas casser la facturation,
- Le catalogue doit être mis à jour sans impacter les recommandations.

9 Conclusion

Le Domain-Driven Design apporte une architecture robuste et orientée métier. À travers l'exemple de Netflix, nous avons vu comment les bounded contexts permettent d'organiser un système massif et complexe tout en minimisant les dépendances.

DDD n'est pas juste une méthode, c'est une philosophie pour mieux structurer et comprendre le domaine métier.