Rapport intermédiaire

Léo Bouvier, Alex Delagrange, Vincent Chazeau, Farah Seifeddine

Mars 2023

Table des matières

1 Schéma conceptuel des données

Vous trouverez en annexe le fichier StarUML de notre schéma conceptuel.

2 L'implémentation en classes d'entité JAVA

Pour l'implémentation, nous avons procédé en suivant l'UML:

Pour chaque classe UML, il existe une classe Java avec les attributs partagés entre eux. Pour ce qui est des relations inter-classes, ces dernières sont définies comme des attributs dont le type correspond à la classe cible. Dans le mapping, on précisera bien qu'il s'agit d'une relation avec un décorateur NtoN (voir partie 3).

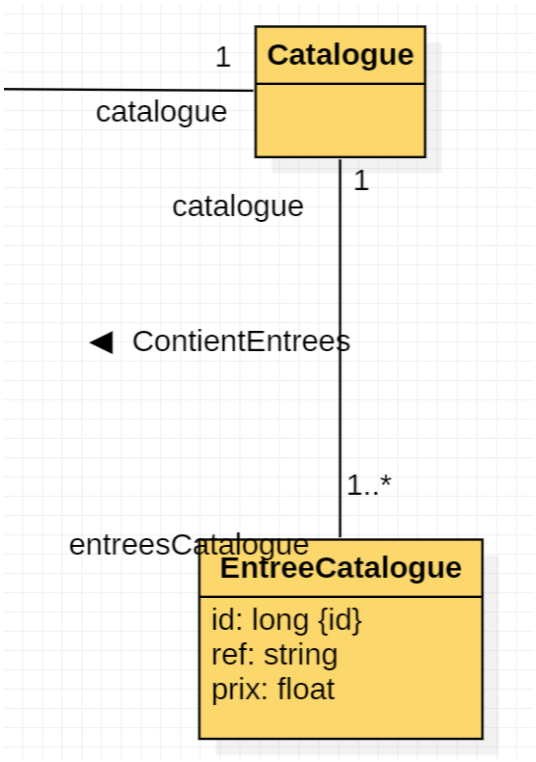

Figure 1: Exemple avec les classes Catalogue et EntreeCatalogue:

Sur l'UML sont représentés les différents attributs de EntreeCatalogue ainsi que la relation "ContientEntrees" qui lie cette classe à Catalogue.

```java
@Entity
public class EntreeCatalogue {

    @Id
    @GeneratedValue
    @Column(name="id")
    private Long id;

    @Column(name="ref")
    private String ref;

    @Column(name="prix")
    private float prix;

    // (UML) Relation ContientEntrees
    @ManyToOne
    private Catalogue catalogue;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

Figure 2: Exemple du mapping de la classe EntreeCatalogue

Dans le l'implémentation Java, on retrouve ces mêmes attributs. On retrouve aussi des getters et setters pour chaque l'attribut, exemple ici avec les méthodes getId() et setId() pour l'attribut id. En effet, les attributs sont propres à l'objet (visibilité Private), d'où la nécessité d'implémenter ces méthodes.

3 Une première version du mapping JPA

Pour chaque classe JAVA qui représente une classe de l'UML, nous avons des attribus associés qui sont mappés avec le nom qu'on souhaite leur attribuer dans notre base de données ex : @Column(name = "Id")

Chaque class JAVA représente une entité/table (@Entity) dans notre base de données. Chaque attribut (hors association) est une colonne (@Column) de notre table. Puis en fonction de l'association, nous avons des variables en plus, par exemple pour une associations 0..* - 1 (@ManyToOne) exemple : (voir figure 2 ci-dessus)

Ce qui servira à créer une table temporaire pour une eventuelle jointure entre les tables.

4 Les choix de conceptions/implémentations

On a conceptualisé les différents types d'impression comme une relation d'héritage, les classes Cadre, Tirage, Calendrier et Album héritent toute de la classe Impression qui contient les relations et attributs communs entre tous les types d'impression comme l'id de l'impression ou le client qui est propriétaire de l'impression. En ce qui concerne l'implémentations java de cette relation d'héritage on a choisi la stratégie TABLE PER CLASS car elle nous permet d'avoir une table par classe et aussi parce que la classe Impression est abstraite et ne peut pas être instanciée. Cette stratégie nous permet donc de ne pas créer de table pour Impression.

On a choisi de définir des types énumérés pour la qualité des cadres, la qualité des articles, la taille des cadres et le format des photos (le format est contenu dans Article). Ainsi on pourra définir des ref pour les articles de manière plus simple et efficace en concaténant simplement le type d'impression au format puis à la qualité. Cela permet aussi d'avoir un contrôle sur les valeurs afin de ne pas se retrouver avec un format ou une qualité qui n'existent pas.

Figure 3: Image de la representation de la classe abstraite Impression