

Systeme de Vente en Ligne

Explication des Choix de Conception et d'Implémentation

Hilal

12 mars 2025

1 Introduction

Ce projet implémente un **systeme de vente en ligne** où un utilisateur peut ajouter des **produits vendables** (Disque, Nourriture) dans un **panier** et en calculer le prix total.

L'architecture respecte les **principes de la programmation orientée objet (POO)** avec une séparation des responsabilités et l'utilisation d'une **interface** pour assurer l'extensibilité.

2 Justification des choix de conception

2.1 Modélisation orientée objet

- **Encapsulation** : Les attributs sont en `private` et accessibles via des méthodes publiques.
- **Modularité** : Chaque classe a une responsabilité bien définie.
- **Extensibilité** : L'interface `Vendable` permet d'ajouter de nouveaux types de produits facilement.

2.2 Interface `Vendable` : un contrat pour les objets vendables

L'interface `Vendable` garantit que tout objet vendable possède une méthode `getPrixUnitaire()`. Elle respecte la contrainte de l'énoncé : *Disque et Nourriture n'héritent que de Object*.

Listing 1 – Interface Vendable

```
package com.Vente;  
  
public interface Vendable {  
    double getPrixUnitaire();  
}
```

3 Explication des classes et leur rôle

3.1 Classe Disque

Cette classe représente un disque vendable (CD, vinyle).

Listing 2 – Classe Disque

```
package com.Vente;  
  
public class Disque implements Vendable {  
    private String titre;  
    private String artiste;  
    private double prixUnitaire;  
  
    public Disque(String titre, String artiste, double prixUnitaire) {  
        this.titre = titre;  
        this.artiste = artiste;  
        this.prixUnitaire = prixUnitaire;  
    }  
  
    @Override  
    public double getPrixUnitaire() {  
        return prixUnitaire;  
    }  
}
```

3.2 Classe Nourriture

Cette classe représente un aliment vendable avec une date d'expiration.

Listing 3 – Classe Nourriture

```
package com.Vente;  
  
import java.time.LocalDate;  
  
public class Nourriture implements Vendable {  
    private String nom;  
    private LocalDate dateExpiration;  
}
```

```

private double prixUnitaire;

public Nourriture(String nom, LocalDate dateExpiration, double prixUnitaire) {
    this.nom = nom;
    this.dateExpiration = dateExpiration;
    this.prixUnitaire = prixUnitaire;
}

@Override
public double getPrixUnitaire() {
    return prixUnitaire;
}
}

```

3.3 Classe Produit

Un Produit est composé d'un objet vendable et d'une quantité.

Listing 4 – Classe Produit

```

package com.Vente;

public class Produit {
    private Vendable chose;
    private int quantite;

    public Produit(Vendable chose, int quantite) {
        if (quantite <= 0) {
            throw new IllegalArgumentException("La quantite doit tre positive");
        }
        this.chose = chose;
        this.quantite = quantite;
    }

    public double getPrixTotal() {
        return quantite * chose.getPrixUnitaire();
    }
}

```

3.4 Classe Panier

Gère une liste de produits et le calcul du prix total.

Listing 5 – Classe Panier

```

package com.Vente;

import java.util.ArrayList;

```

```

import java.util.List;

public class Panier {
    private List<Produit> produits;

    public Panier() {
        this.produits = new ArrayList<>();
    }

    public void add(int quantite, Vendable chose) {
        if (quantite <= 0) {
            throw new IllegalArgumentException("Quantit  invalide.");
        }
        produits.add(new Produit(chose, quantite));
    }

    public double calculerTotal() {
        return produits.stream().mapToDouble(Produit::getPrixTotal).sum();
    }
}

```

4 Explication des mesures défensives

Le code inclut plusieurs mécanismes pour garantir la fiabilité du programme :

- **Exceptions** pour éviter des erreurs critiques (ex. `IllegalArgumentException` si la quantité ≤ 0).
- **Assertions** (`assert`) pour les préconditions et postconditions.

5 Conclusion

Pourquoi ce modèle est-il efficace ?

- **Interface Vendable** → Respecte la contrainte de l'énoncé.
- **Encapsulation et séparation des responsabilités.**
- **Extensible** → Ajouter de nouveaux objets vendables est simple.
- **Robuste** → Vérifications et tests garantissent un bon fonctionnement.