La Cave de BOUTIN :

Ce rapport s'inscrit dans le cadre du projet de fin de semestre en Génie Logiciel.
L'objectif principal de ce projet est la mise en œuvre des concepts étudiés durant le semestre, en particulier les design patterns ainsi que les principes fondamentaux de la programmation orientée objet (POO).

🎯 Objectif du projet :

Une application Java en console qui permet :

- À l'utilisateur de choisir des produits alcoolisés (vin, bière, whisky, etc.)
- D'ajouter ses produits à un panier
- De passer à la caisse en saisissant des infos de paiement fictives
- De recevoir une facture générée dynamiquement

Rappel des fonctionnalités que tu veux en console :

- Affichage des produits alcoolisés par catégorie
- Sélection d'alcools par l'utilisateur → ajout au panier
- Simulation d'un paiement fictif (nom + n° carte)
- Génération et affichage d'une facture en console

Fonctionnement :

L'utilisateur lance le programme, il peut alors choisir une catégorie (comme "Bières") puis choisir un produit à mettre dans son panier. Une fois terminé, il entre ses informations et valide son achat fictif. Une facture est alors générée.

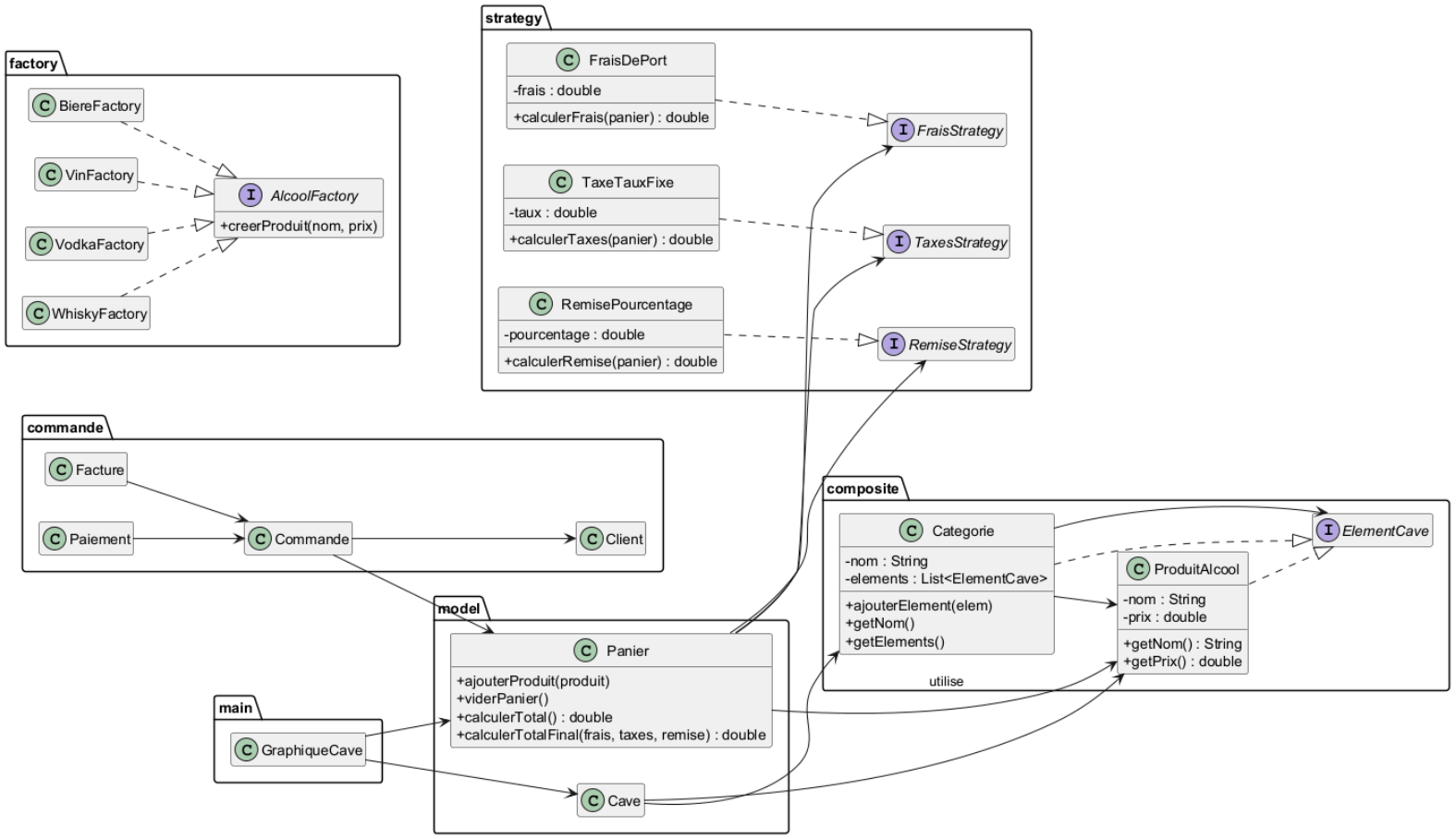
Structure du projet :

Le code est organisé en différents "packages", chacun ayant un rôle précis :
- commande : gère les clients, les commandes, les paiements et les factures.
- composite : permet d'organiser les produits dans des catégories. Par exemple, la catégorie "bières" contient toutes les bières.
- factory : sert à fabriquer différents types de bouteilles automatiquement selon leur type (ex : whisky).
- model : contient les objets concrets comme la classe “Bouteille", qui représente une vraie bouteille.
- main : contient le programme principal qui lance l'application graphique.

Arborescence du projet :

```
src/
├── commande/
│   ├── Client.java
│   ├── Commande.java
│   ├── Facture.java
│   └── Paiement.java
│
├── composite/
│   ├── Categorie.java
│   ├── ElementCave.java
│   └── ProduitAlcool.java
│
├── factory/
│   ├── AlcoolFactory.java
│   ├── BiereFactory.java
│   ├── VinFactory.java
│   ├── VodkaFactory.java
│   └── WhiskyFactory.java
│
├── main/
│   └── GraphiqueCave.java
│
├── model/
│   ├── Cave.java
│   └── Panier.java
│
└── strategy/
    ├── FraisDePort.java
    ├── FraisStrategy.java
    ├── RemisePourcentage.java
    ├── RemiseStrategy.java
    ├── TaxeTauxFixe.java
    └── TaxesStrategy.java
```

Diagramme UML :



Designs Patterns :

Qu'est-ce qu'un design pattern ?

Un design pattern c'est une solution générale, réutilisable et éprouvée à un problème courant en programmation orientée objet. On parle ici d'une manière standardisée d'organiser son code pour qu'il soit plus lisible, modulaire, et facile à maintenir.

Design Patterns utilisés dans le projet :

1. Le pattern Composite :
Ce pattern a été utilisé pour organiser les produits d'alcool dans des catégories comme Bières, Vins, Whiskys, etc.

Chaque catégorie contient directement plusieurs produits (par exemple, la catégorie Whiskys contient des références comme "Red Label" ou "Jack Daniel's").

Même si notre projet ne contient pas de sous-catégories, ce pattern reste utile. Il permet de manipuler les catégories et les produits de la même manière, sans faire de différences dans le code. De plus, si on souhaite un jour ajouter des sous-catégories, le code est déjà prêt à l'accepter, sans besoin de le modifier.

2. Le pattern Factory :

Le pattern Factory a été utilisé pour créer les produits d'alcool de manière simple et centralisée. Chaque type d'alcool possède sa propre "usine" comme ***VinFactory, WhiskyFactory,*** etc. qui sait comment créer les bons objets avec leur nom et leur prix.

Ce modèle nous permet d'éviter de répéter du code, de rendre le programme plus lisible, et de faciliter l'ajout de nouveaux types d'alcools sans impacter le reste du projet.

3. Le pattern Strategy :
Ce pattern est utilisé pour appliquer dynamiquement différentes règles de calcul sur le panier (frais de port, taxes, remises).

Plutôt que de coder ces calculs en dur dans la classe Panier, on délègue leur logique à des objets Strategy spécifiques, comme ***FraisDePort, TaxeTauxFixe ou RemisePourcentage.*** Cela permet de changer facilement la stratégie de calcul sans modifier le panier lui-même. Par exemple, on peut ajouter un nouveau mode de calcul de taxes ou de remises sans toucher au reste du code.

Points positifs et négatifs des patterns :

Points positifs :

- **Organisation du code plus claire** : chaque partie du programme a un rôle bien défini, ce qui facilite la compréhension générale du projet.

- **Séparation des responsabilités** : les calculs, la création des objets, ou la structure des données sont séparés dans des modules spécifiques, ce qui évite de tout mélanger.

- **Facilité d'évolution** : nous pouvons ajouter de nouveaux types de produits ou de nouvelles règles de calcul (remises, taxes...) sans devoir modifier les classes principales.

- **Réutilisabilité** : certaines parties du code, comme les stratégies de calcul ou les factories, peuvent être ré utilisées dans d'autres projets similaires.

Points négatifs :

- **Complexité initiale** : au début du projet, mettre en place les patterns demande un peu plus de réflexion et de structure, ce qui peut être difficile quand le projet est encore petit.

- **Trop pour des cas simples** : certains patterns, comme Composite, peuvent sembler un peu "trop" si le projet ne nécessite pas toutes leurs possibilités (comme la gestion de sous-catégories qui n'existent pas encore dans notre cas).

- **Besoin de bien comprendre la programmation orientée objet** : leur bonne utilisation nécessite de maîtriser des notions comme les interfaces, l'abstraction, et la délégation.

Conclusion :

L'utilisation des design patterns dans ce projet a été un vrai atout.
Même si le projet reste relativement simple, ces modèles nous ont permis de structurer notre application de manière propre, cohérente et professionnelle.

Ils ont apporté une meilleure lisibilité du code, une organisation plus claire, et surtout une capacité d'évolution importante. Par exemple, il serait très facile d'ajouter une nouvelle stratégie de remise, un nouveau type de produit, ou une sous-catégorie sans modifier la structure existante.

Nous avons donc pu expérimenter concrètement l'intérêt des design patterns dans ce projet, et comprendre qu'ils ne sont pas réservés aux grandes applications : même dans un petit projet, ils améliorent la qualité du code et facilitent le travail en groupe.