Rapport de projet NFA019
Etienne LARROUMETS
CNAM 2025

Présentation du projet

L'application que j'ai développée et que j'ai baptisée Jbrasserie est divisée en deux parties.

La première (Jbrasserie Shop) consiste en la gestion du stock et du fichier client d'un magasin de bières. Elle permet également de consulter la liste des commandes effectuées par les clients.

La deuxième partie (Jbraseerie Client) concerne la consultation du catalogue de bières par les clients et la possibilité de passer commande.

Pour ce projet, j'ai décidé de partir sur l'idée de deux fenêtres tournant sous la même JVM. Cette solution a été choisie dans le but de faciliter l'utilisation de l'application.

Chaque fenêtre possède trois onglets.

Dans la fenêtre magasin (Jbrasserie Shop) un onglet **Clients** permet de consulter la liste des clients et d'éditer leurs informations si besoin. Un deuxième onglet **Catalogue** permet de consulter le stock de bières du magasin et de le modifier si besoin. Le troisième onglet **Commandes** permet de consulter la liste des commandes effectuées et d'en voir le détail.

Dans la fenêtre client (Jbrasserie Client) un onglet **Profil** permet au client de se connecter et d'éditer ses informations. Un deuxième onglet **Catalogue** permet au client de consulter la liste des bières en stock et de les mettre dans le panier. Le troisième onglet **Panier** permet de valider sa commande ou d'en supprimer des articles.

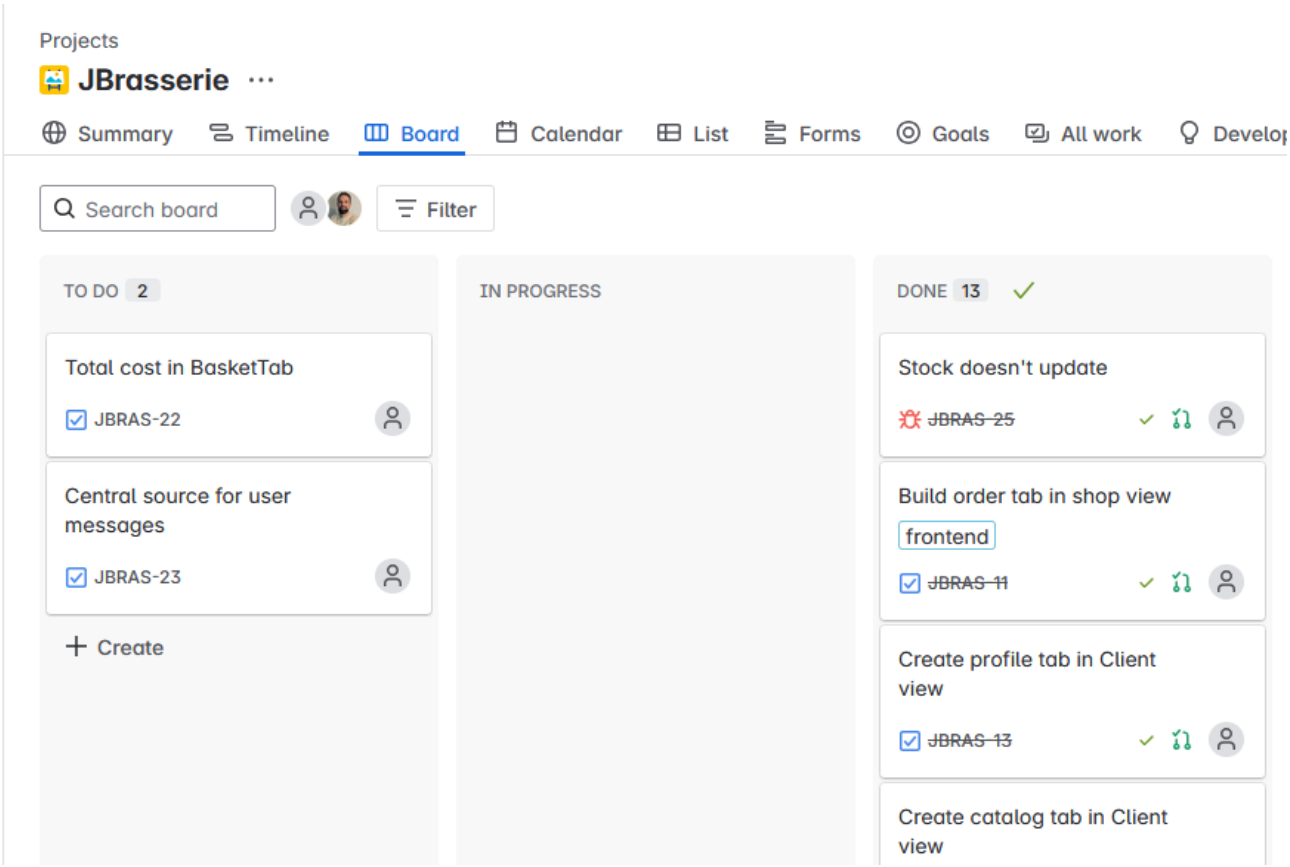
Environnement technique

Pour réaliser cette application j'ai utilisé l'IDE **Eclipse 2025-03** avec **JDK 21**. J'ai utilisé **Maven** pour construire le projet, ceci afin de faciliter la gestion des dépendances et l'import du projet.

J'ai utilisé une base de données **MariaDB** en local avec un serveur **WAMP64**. J'ai utilisé un connecteur **JDBC MariaDB** en version **3.5.3** fourni par le dépôt Maven.

Côté gestion de projet, j'ai utilisé Jira de Atlassian pour créer une board Kanban. Sur cette board j'ai répertorié les tâches de travail. J'ai ensuite couplé mon instance Jira avec mon dépôt Github afin de créer une nouvelle branche pour chaque tâche. Cela m'a grandement facilité la gestion du travail.

Ma board Jira :



Organisation du projet

Les fichiers

J'ai regroupé les classes par packages en fonction de leur nature. Il y a un package pour les vues, un pour les contrôleurs, les modèles, etc.

Voici un aperçu de l'arborescence du projet (dans **src/main/java/org.cnam.jbrasserie**):

- **App.java** : c'est la classe principale qui contient la méthode *main*
- **beans/** : contient les modèles métier
- **config/** : contient le fichier *config.properties* contenant la configuration de la base de données
- **controlers/** : les contrôleurs
- **dao/** : contient les DAO et la Factory pour toutes les requêtes en base de données
- **exceptions/** : contient des exceptions personnalisées pour l'affichage de messages utilisateurs
- **observers/** : contient les classes utilisées pour faire le lien entre la partie client et la partie magasin et également à passer des informations entre contrôleurs.
- **session/** : contient le fichier Session.java qui conserve les données utilisateurs durant la session d'utilisation

- **tableModels**/ : contient les classes héritées de AbstractTableModel, servant à gérer les données des tableau Swing JTable.
- **views**/ : contient les vues du programme

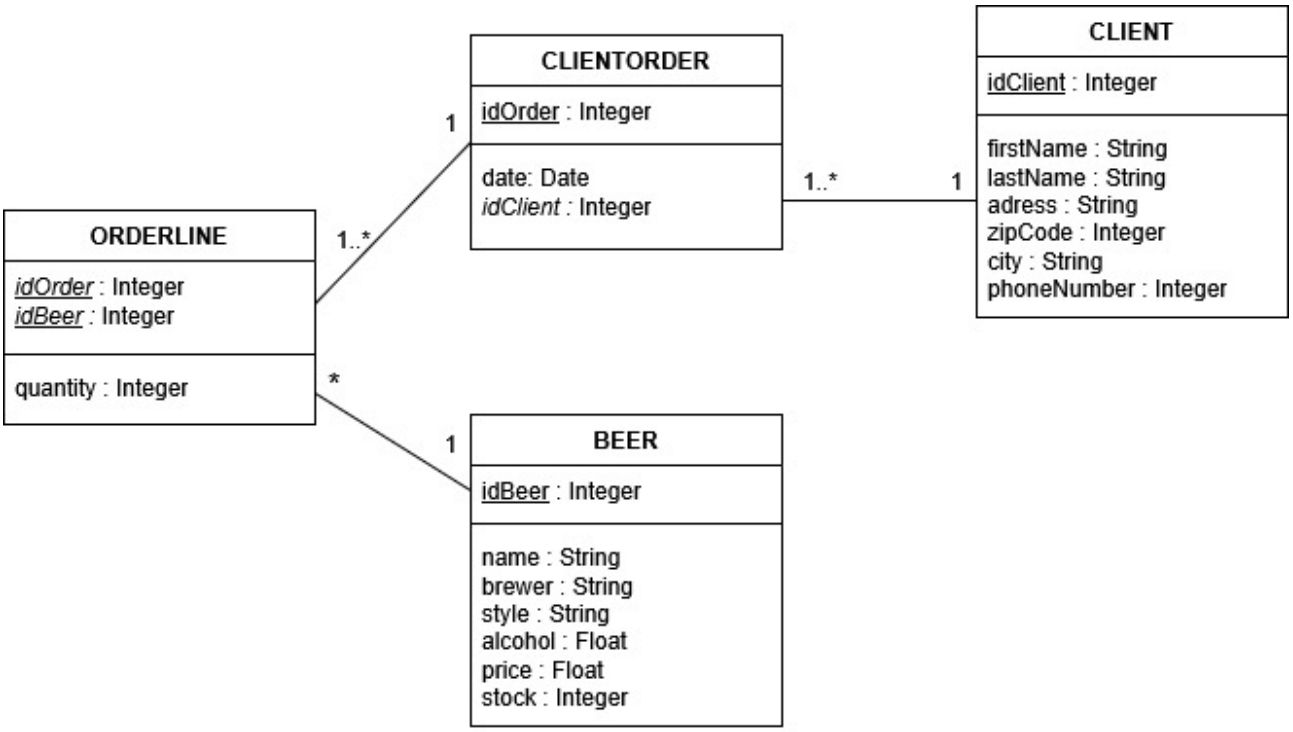
La base de données

Voici le diagramme UML représentant les tables de la base de données.

Nous avons une relation **1:N** entre **ClientOrder** et **Client**. En effet, un client peut effectuer plusieurs commandes et chaque commande correspond à un seul client.

Nous avons ensuite une relation **N:N** entre **ClientOrder** et **Beer** qui est représentée par la table pivot **OrderLine**. Une commande est composée d'une ou plusieurs bières et une bière peut faire partie d'aucune ou plusieurs commandes.

Ce diagramme représente le modèle logique des données, celui qui représente les tables en base de données. Si l'on avait représenté le modèle conceptuel des données, **ClientOrder** serait directement liée à **Beer** et l'entité **OrderLine** n'apparaîtrait pas



Architecture de l’application

Les modèles

La première partie du projet a consisté à mettre en place toute la mécanique d’accès aux données.

Cela a commencé avec la création des classes métier.

Concernant ces classes il n’y a rien de particulier. Ce sont des Java Beans classiques que l’on encapsule (attributs en private + accesseurs).

Pour le modèle **OrderLine.java** qui représente une ligne d’une commande, on effectue une vérification dans le setter de la quantité afin de s’assurer que le stock pour l’article en question est suffisant.

Dans le modèle **Client.java**, on effectue une vérification du numéro de téléphone. En effet celui ci est de type S*tring* afin de pouvoir conserver le « 0 » devant le numéro (le type *Integer* retire automatiquement le « 0 »). Mais on ne veut pas que le numéro soit composé de lettres ou autres caractères. On vérifie donc cela à l’aide d’une expression régulière.

Pour ces deux vérifications, en cas d’anomalie, on lève une exception *BeanException* que l’on fait remonter jusqu’aux contrôleurs qui donnent l’ordre aux vues d’afficher un message d’erreur pour avertir l’utilisateur.

Accès aux données

La connexion à la base de données se fait dans le fichier **DBConnection.java** du package **database**. La configuration de la base de données se fait dans le fichier **config.properties** du package **config**.

Pour persister et récupérer les données, j’ai fait appel au design pattern DAO. Je trouve que c’est un bon moyen d’isoler la couche d’accès aux données du reste de l’application.

Pour chaque classe métier on implémente une classe DAO : **BeerDaoImplDB.java**, **ClientDaoImplDB.java**, etc. Ces classes sont elles mêmes héritées d’une interface **BeerDao.java**, **ClientDao.java**, etc. Toutes les requêtes SQL vers la base de données sont effectuées au sein des DAO.

J’ai ensuite utilisé une Factory pour séparer l’instanciation des DAO des contrôleurs.

Pour faire simple, la classe **FactoryDao.java** (dans le package **dao**) permet d’instancier automatiquement les objets DAO en fonction du type de datasource que l’on souhaite utiliser (base de données, fichier xml, csv, etc.). Le type de datasource que l’on souhaite utiliser est défini dans le fichier **datasource.properties** (package **dao**). Il est par défaut défini à « DB » (DataBase).

Ainsi les contrôleurs n’ont plus à savoir quel type de datasource l’application utilise, on se contente de faire appel à la Factory avec un simple :

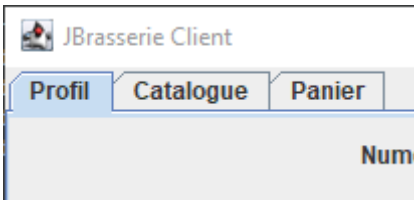
```
ClientDao clientDao = FactoryDao.getClientDao();
```

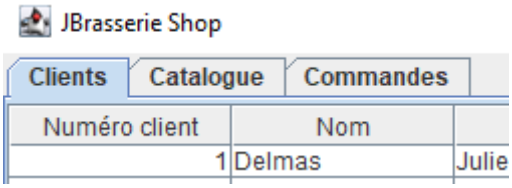
D’où l’utilité d’utiliser une interface pour les Dao : peu importe le type d’implémentation que l’on utilisera, elles seront toutes de type xxxxDao (ClientDao dans l’exemple au dessus).

Je parle plus en détail de la Factory dans le document « Mise en lumière ».

Les vues

Chaque fenêtre représente une vue (**ClientView.java** et **ShopView.java**). Chacune d'elles est constituée de trois onglets représentant chacun une vue :



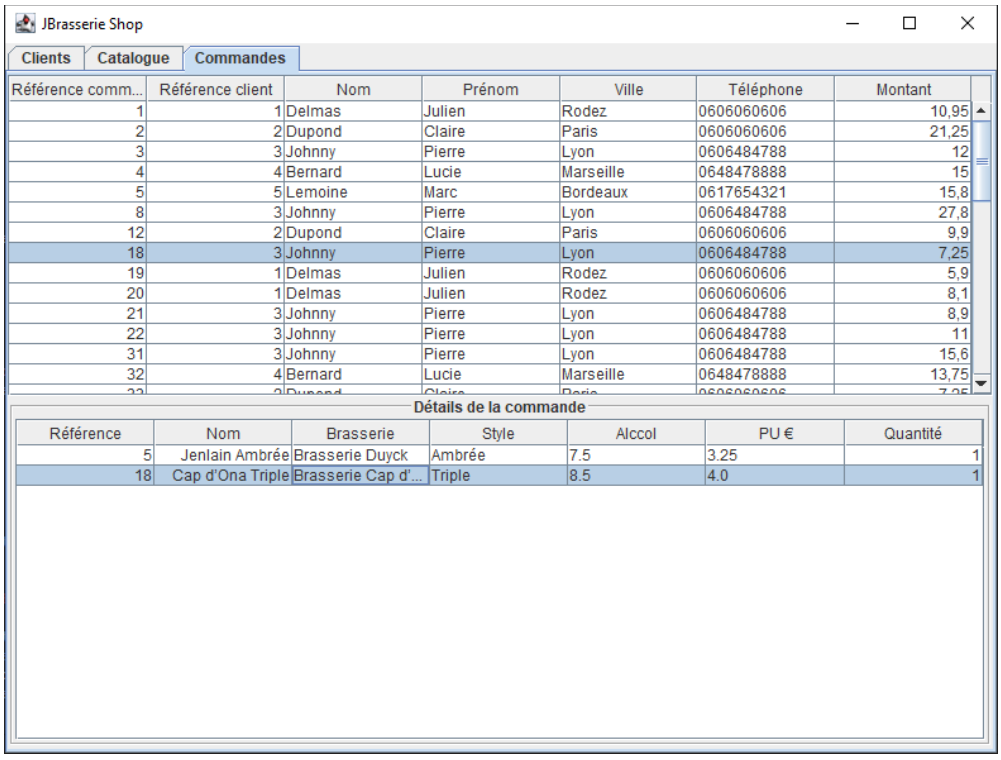


Petite particularité : la fenêtre JBrasserie ne compte qu'un onglet *Profil* au démarrage. En effet, il faut se connecter avec un numéro de client valide pour accéder au catalogue et au panier.

Pour placer les éléments Swing, j'ai utilisé le layout GroupLayout. Ce layout étant un peu complexe à appréhender, j'ai d'abord agencé les éléments à la main puis je me suis aidé de l'extension WindowBuilder fournie par Eclipse pour corriger les anomalies.

Il n'y a que l'onglet *Commandes* de la fenêtre magasin pour lequel je n'ai pas utilisé ce layout car ce n'était pas nécessaire (on affiche juste deux tables).

J'ai beaucoup utilisé les composants JTable de Swing dans mes vues. Ces composants ont la particularité de ne pas «directement» gérer les données affichées. Pour cela on utilise des classes héritées de AbstractTableModel que l'on passe en paramètre au constructeur de la JTable au moment de son instanciation. On retrouve donc dans le package **tablesModels** toutes les classes que j'ai créées pour gérer les données des différentes JTable.



Les vues se chargent uniquement de l'affichage, tout la logique est déléguée aux contrôleurs. J'ai choisi de lier les boutons des vues (JButton) aux contrôleurs via des classes anonymes. Chaque bouton est instancié en appelant la classe abstraite **AbstractAction** qui implémente elle même l'interface **ActionListener**.

J'ai découvert cette technique avec ce projet et je l'ai trouvé très bien pensée car ainsi le contrôleur n'a plus besoin d'implémenter **ActionListener** et donc plus besoin de faire de *switch case* pour connaître la source de l'action ou de lier un contrôleur par bouton. On se contente d'appeler la méthode souhaitée du contrôleur depuis la vue.

Extrait de la vue **ClientTab.java** :

```
// Buttons
this.updateButton = new JButton(new AbstractAction("Valider") {
    @Override
    public void actionPerformed(ActionEvent e) {
        clientControler.updateClient();
    }
});

this.deleteButton = new JButton(new AbstractAction("Supprimer") {
    @Override
    public void actionPerformed(ActionEvent e) {
        clientControler.deleteClient();
    }
});

this.newButton = new JButton(new AbstractAction("Nouveau client") {
    @Override
    public void actionPerformed(ActionEvent e) {
        clientControler.newClient();
    }
});
```

Les contrôleurs

On compte un contrôleur par onglet, six au total. Leur rôle est simple : ils captent les actions envoyées depuis les boutons des vues, font appel aux DAO pour persister ou récupérer des données et mettent à jours les vues.

Ils chargent et initialisent les données au démarrage de l'application en récupérant les données client, le catalogue et la liste des commandes depuis la base de données.

Les exceptions

Pour pouvoir avertir l'utilisateur en cas d'erreur ou de succès lors d'une saisie ou d'un enregistrement, j'ai implémenté un affichage de messages. Ce système repose en partie sur les exceptions.

Au delà des exceptions natives Java, j'ai créé trois exceptions personnalisées :

- **BeanException.java** : cette exception est levée en cas d'anomalie sur une donnée métier (ex : quantité demandée supérieur au stock)

- **DaoException.java** : cette exception est levée depuis les DAO pour avertir l'utilisateur qu'une entrée d'une table de la base ne peut pas être supprimée. C'est le cas par exemple pour un client ayant déjà effectué une commande ou une bière faisant déjà partie d'une commande.
- **FormException.java** : cette exception est levée depuis les contrôleurs en cas de mauvaise saisie de l'utilisateur dans un des formulaires (exemple : une chaîne de caractère au lieu d'un nombre sur le champs « *Prix* » ou *« Stock »).*

Les observers

Les observers ont été utilisés dans cette application pour notifier certaines vues de changements déclenchés par d'autres contrôleurs que les leurs.

Le mieux est d'illustrer le principe par un exmple :

Quand un client a fini de remplir son panier et qu'il veut confirmer la commande, il faut que la vue « Commandes » côté magasin soit mise à jour.
Pour cela, la vue **OrderTab.java** (qui affiche la liste des commandes passées dans la partie JBrasserie shop) implémente l'interface *OrderObserver.* Ainsi, on peut l'ajouter à la liste des *observers* de la classe **OrderNotifier.java**.
Quand le client valide sa commande, le contrôleur associé au bouton de validation de commande déclenche la méthode *newOrderSubmitted()* de tous les *observers.* Ces derniers se chargent ensuite d'appeler leurs contrôleurs pour mettre à jour les vues.

J'ai utilisé ce mécanisme pour mettre à jour les vues de manière automatique. Quelques exemples :

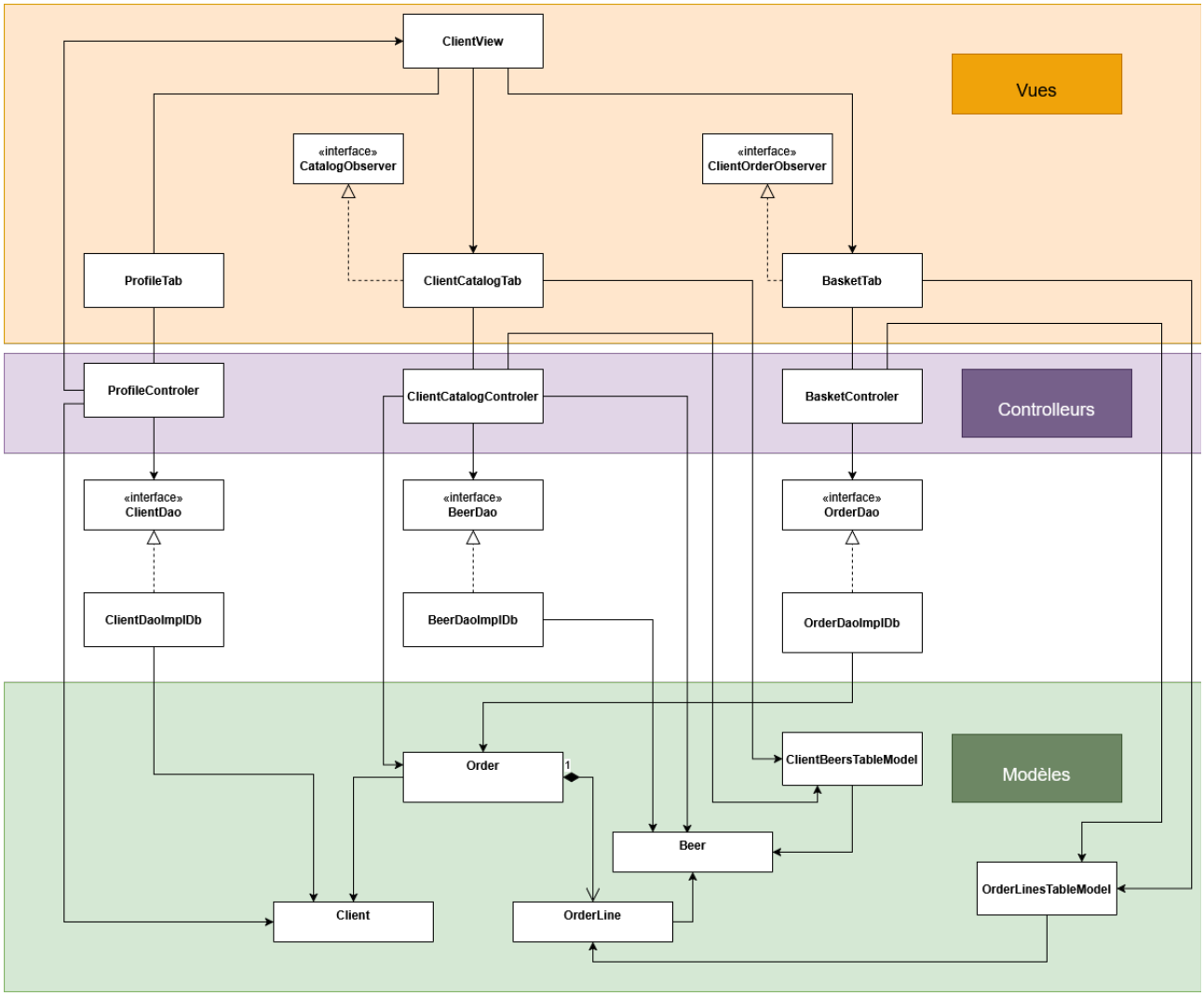
- un client valide une commande, le stock est mis à jour dans le catalogue magasin
- un client modifie ses infos, le fichier client du magasin est mis à jour
- le magasin ajoute/modifie/supprime une bière du catalogue, le catalogue client est mis à jour

Diagrammes UML

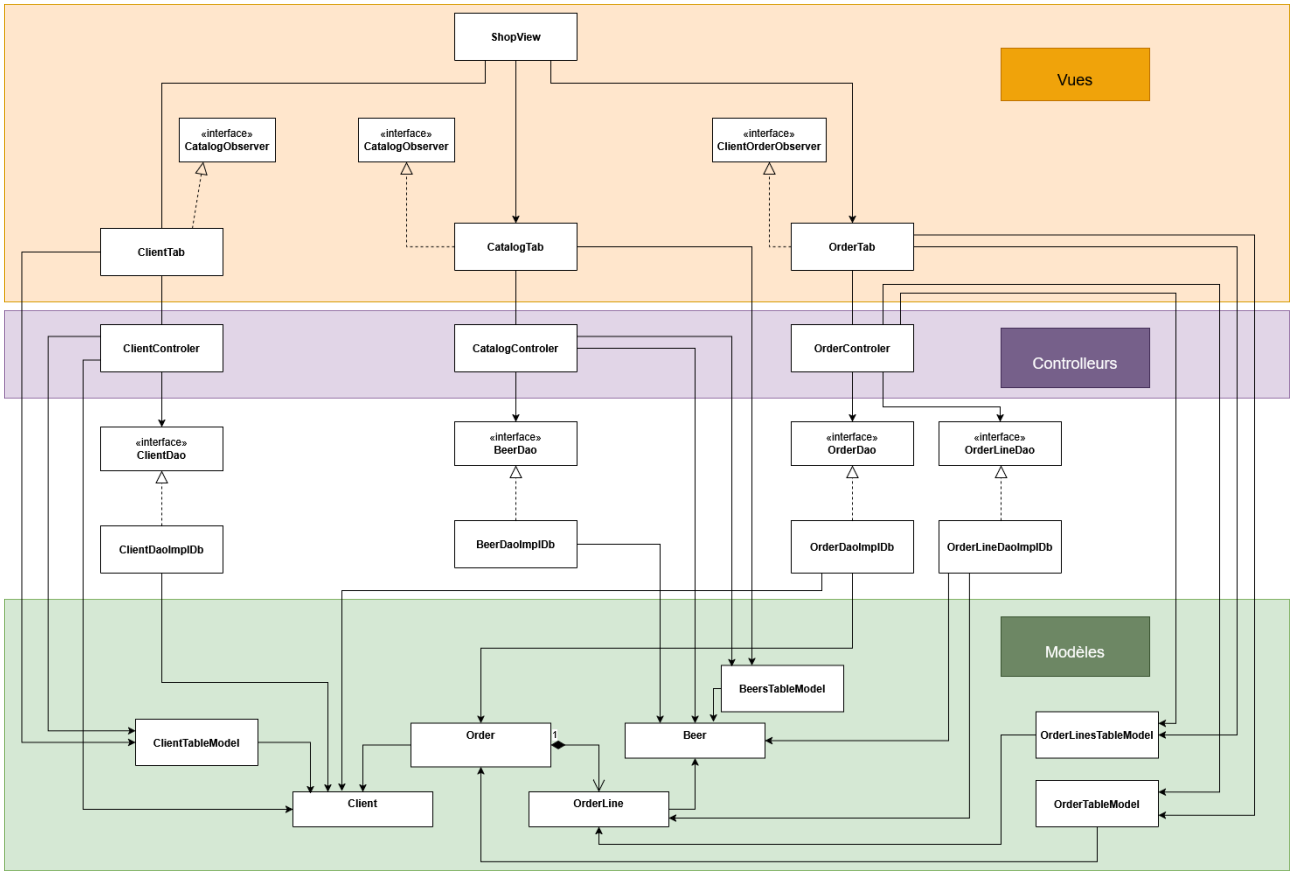
Dans les diagrammes UML de l'application, j'ai volontairement omis les attributs et méthodes des classes pour ne pas surcharger un affichage déjà bien fourni !

Je fourni en pièces jointes du projet les diagrammes au format PNG pour une meilleure lisibilité.

Le diagramme UML de la partie client :



Le diagramme UML de la partie magasin :



Conclusion

Ce projet m’a permis d’approfondir et de mettre en pratique l’architecture MVC et les composants Swing. L’utilisation de certains design patterns m’a permis d’aller plus loin dans la démarche.

Le plus dur pour moi a été la création des interfaces IHM car je n’ai pas l’habitude de travailler avec Swing. En tant normal, j’ai plus l’habitude de pratiquer sur des applications web de type Java EE.

L’architecture MVC m’était plus familière notamment grâce à l’apprentissage de frameworks comme Java EE, Laravel ou encore CodeIgniter.

Mon but principal tout au long du projet a été d’élaborer une application qui soit fonctionnelle et finie. J’ai réussi à respecter la quasi totalité des objectifs que j’avais fixés dans mon cahier des charges pour la version 1.0. La seule fonction prévue au cahier des charges et que je n’ai pas implémentée est la possibilité pour un nouveau client de se créer un compte depuis la partie client.

Le code aurait demandé un travail de refactorisation. En effet, on retrouve certains morceaux de code semblables ou ayant des fonctionnalités proches. J’aurais pu utiliser plus le polymorphisme, notamment en ce qui concerne les IHM. Le temps était compté et j’ai préféré prioriser la fonctionnalité du code.