UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

Rapport d'APP

APP1 – Modèles de conception
GIF350

Présenté à
Domingo Palao Munoz

Présenté par
Samuel Bilodeau – BILS2704
Benjamin Chausse – CHAB1704

Sherbrooke – 5 mai 2023

Table des matières

1 Diagrammes de classes de *MenuFact02*

FIXME: Enlever le commentaire nofile avant de remettre le projet.

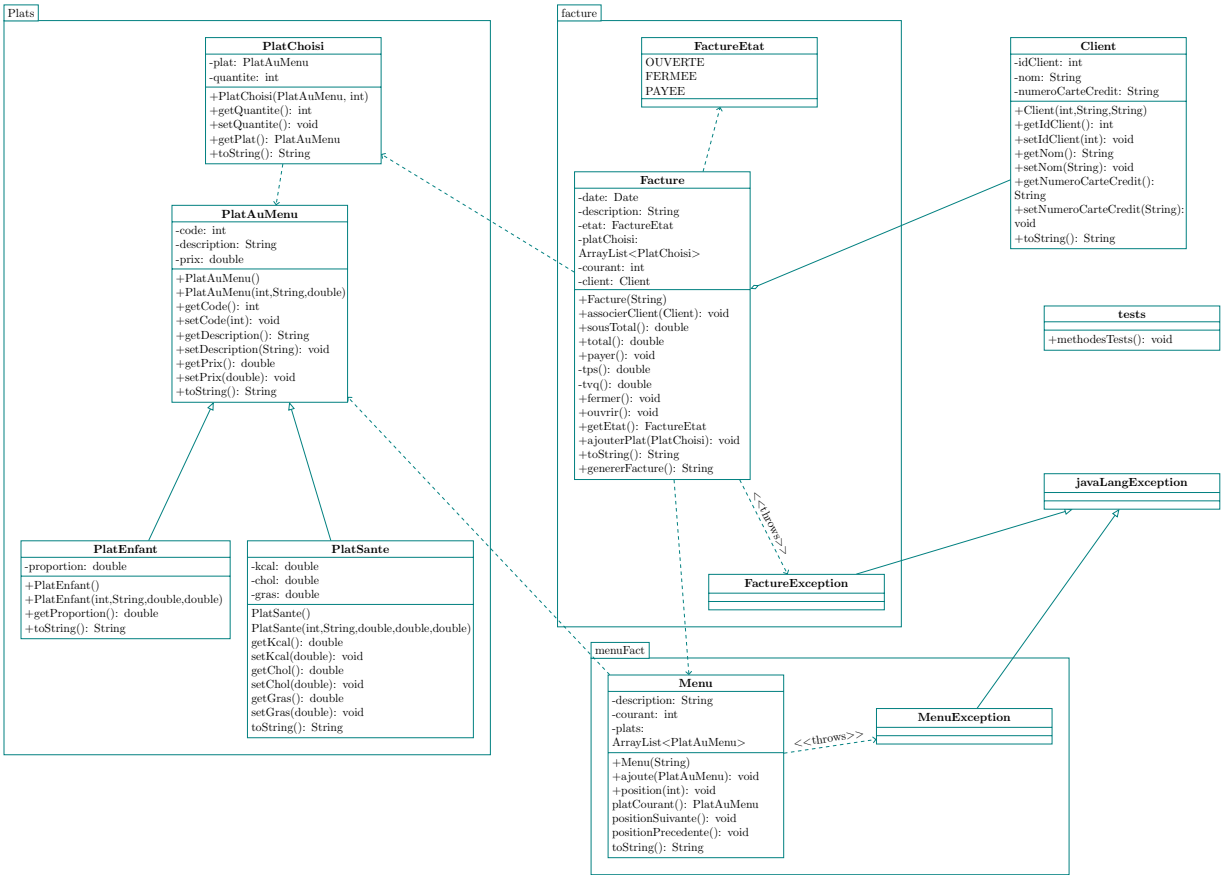

FIGURE 1-1 – Diagramme de classes

2 Choix de conception

TABLE 2-1 – Justification des modèles de conception

Modèle de conception	**Justification**
Modèle-vue-contrôleur (MVC)	Permet de séparer la logique de l'application, l'interface, et l'information stockée par l'application. `Terminal` est la vue (peut être changée pour une interface web ou graphique sans changer la logique) , `Controller` est le cerveau de l'application et `DataBroker` est le modèle qui stoque le Menu, les factures, les Clients, les Chefs, etc...
Flyweight	Permet de réduire la mémoire utilisée par l'application en partageant le prix de tout les plat d'une même gamme de prix. (voir section **??**)
States	Permet de gérer l'état des factures. Une facture peut être dans l'état Ouverte, Fermée ou Annulée. L'état détermine les actions possibles. Par exemple, il est impossible d'ajouter des plats à une facture fermée.
Observer	La classe `IngredientWatcher` permet de notifier les classes qui s'y abonnent à chaque fois que les quantitées de divers ingrédients passent sous un seuil critique. Par exemple, le `Controller` pourrait s'abonner à cette classe pour retirer du menu les plats qui ne peuvent plus être préparés.
Factory	La classe `PlatFactory` permet de créer le bon type de plat selon les paramètres qui lui sont passés. Par exemple, si les ingrédients utilisés sont santé, la factory va créer un `PlatSante`. Dans le cas où les proportions sont petites, la factory va créer un `PlatEnfant`. La gamme de prix est aussi déterminée par les ingrédients utilisés.
Singleton	Diverses classes sont implémentées en tant que `Singleton` pour éviter divers problèmes de conception. Par exemple, il n'y a qu'un seul `Menu` pour tout le restaurant (deux clients ne devrait pas avoir deux menus différents). Il n'y a qu'un seul `DataBroker` pour gérer les données globales de l'application. On ne veut pas avoir deux bases de données différentes pour l'application. Finallement, il n'y a qu'un seul `Controller` pour gérer l'application. On veut éviter des problèmes de synchronisation où deux contrôlleurs modifient la même information simultanément. Cela n'empêche pas l'application d'avoir plusieurs `Terminal`s qui sont en communication avec le même `Controller`.

2.1 Note sur l'utilisation du *Flyweight*

Note: Je sais pas si créer une section juste pour ça est une bonne idée.

Comme mentionné lors du turorat, l'utilisation de patrons de conception était surchargée pour le projet. On nous a demandé de les implémenter pricipalement à des fins éducationnelles. Ceci dit, pour implémenter le patron *Flyweight*, nous avons pensé à l'idée d'un restaurant dans lequel les Plats sont regroupés par gammes de prix (écono, régulier, deluxe). Ainsi, chaque gamme de prix détient un *flyweights* contenant le prix unique de tout les plats dans cette gamme. Une économie de mémoire est donc réalisée puisque la valeur du prix n'est pas dupliquée pour chaque plat d'une même gamme. Bien sûr, une implémentation réaliste de ce patron contiendrait généralement plus qu'un seul attribut dans le *flyweights*. Ces classes seraient donc nommées comme suit :

- `Flyweight` : Classe abstraite
- `EconoFlyweight` : Classe concrète
- `RegularFlyweight` : Classe concrète
- `DeluxeFlyweight` : Classe concrète

3 Utilité des modèles de conception

TODO: Écrire cette section

4 Utilité des tests unitaires

TODO: Écrire cette section