WATTIN Tristan - FURNON Claire
Master 1 ILSEN Classique

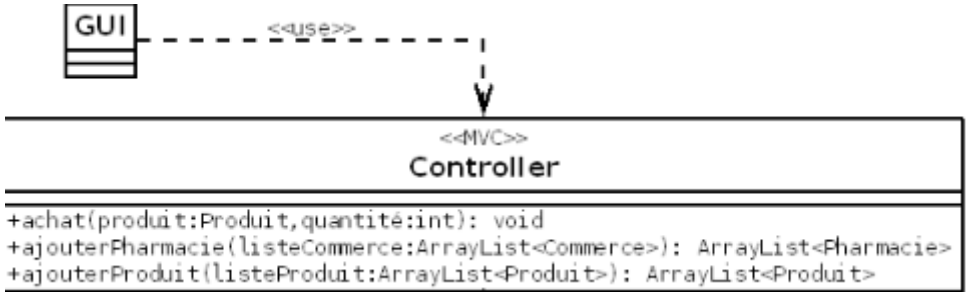
Compte-Rendu Application de gestion de pharmacies

Lien GitHub :
https://github.com/NekuHarp/pharmacie

Liste des Design Patterns

- MVC ;
- Flyweight ;
- Singleton ;
- State ;
- Strategy ;
- Template Method ;
- Observer ;
- Memento (non implémenté).

Pattern MVC

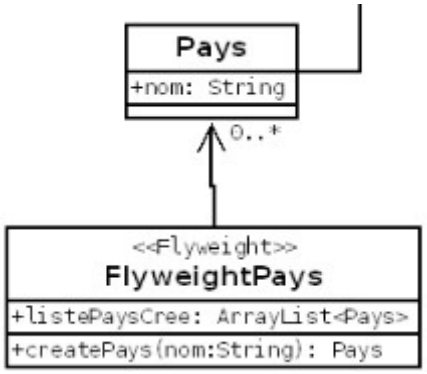


Le pattern MVC (Modèle Vue Contrôleur) est obligatoire dans le cadre de ce mini-projet.

Nous avons donc la GUI (la vue) qui utilise le « controller » pour fonctionner, qui est lui-même relié au « Modèle » (c'est-à-dire au reste de l'application).

Afin de garder une visibilité correcte, la capture du MVC ci-dessus n'inclut pas la partie « modèle » car elle prendrait énormément de place.

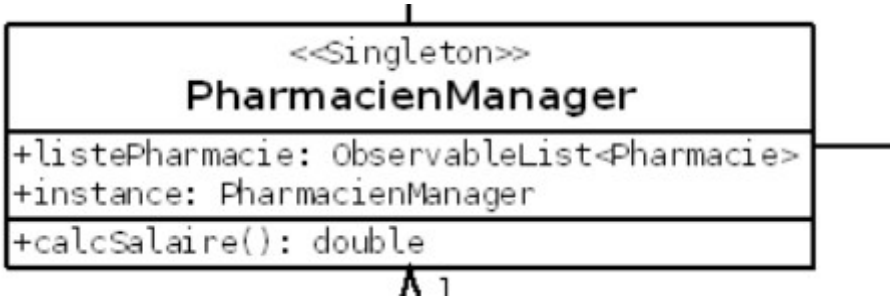
Flyweight



Le pattern Flyweight est utilisé dans notre cas pour la gestion des pays, dans le cadre de la gestion du réseau bancaire de type Visa.

Ce pattern nous permet de pouvoir sélectionner des pays depuis une liste de pays, et d'en créer de nouveaux s'ils n'existent pas déjà, ce qui nous permet d'économiser de la mémoire. De plus, cela nous permet d'appliquer certaines contraintes pour certains pays, comme demandé dans le sujet dans le cadre du réseau Visa, et donc d'avoir par défaut certains pays déjà créés.

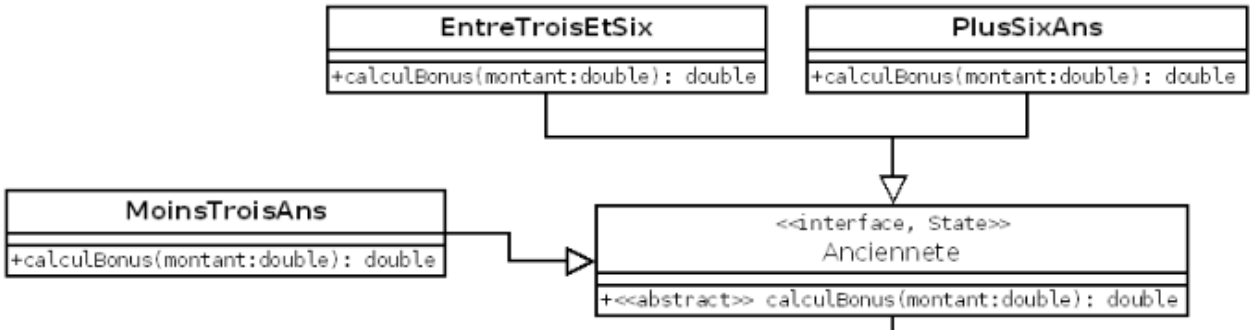
Singleton



Le pattern Singleton est utilisé dans notre cas pour le Pharmacien Manager, c'est-à-dire le pharmacien utilisateur de notre application, et gérant des pharmacies concernées.

Ce pattern nous permet de garantir qu'il n'existe qu'une seule instance de pharmacien manager, ce qui est logique puisque notre application n'a qu'un seul utilisateur simultané, et qu'il n'y a qu'un seul manager par pharmacie.

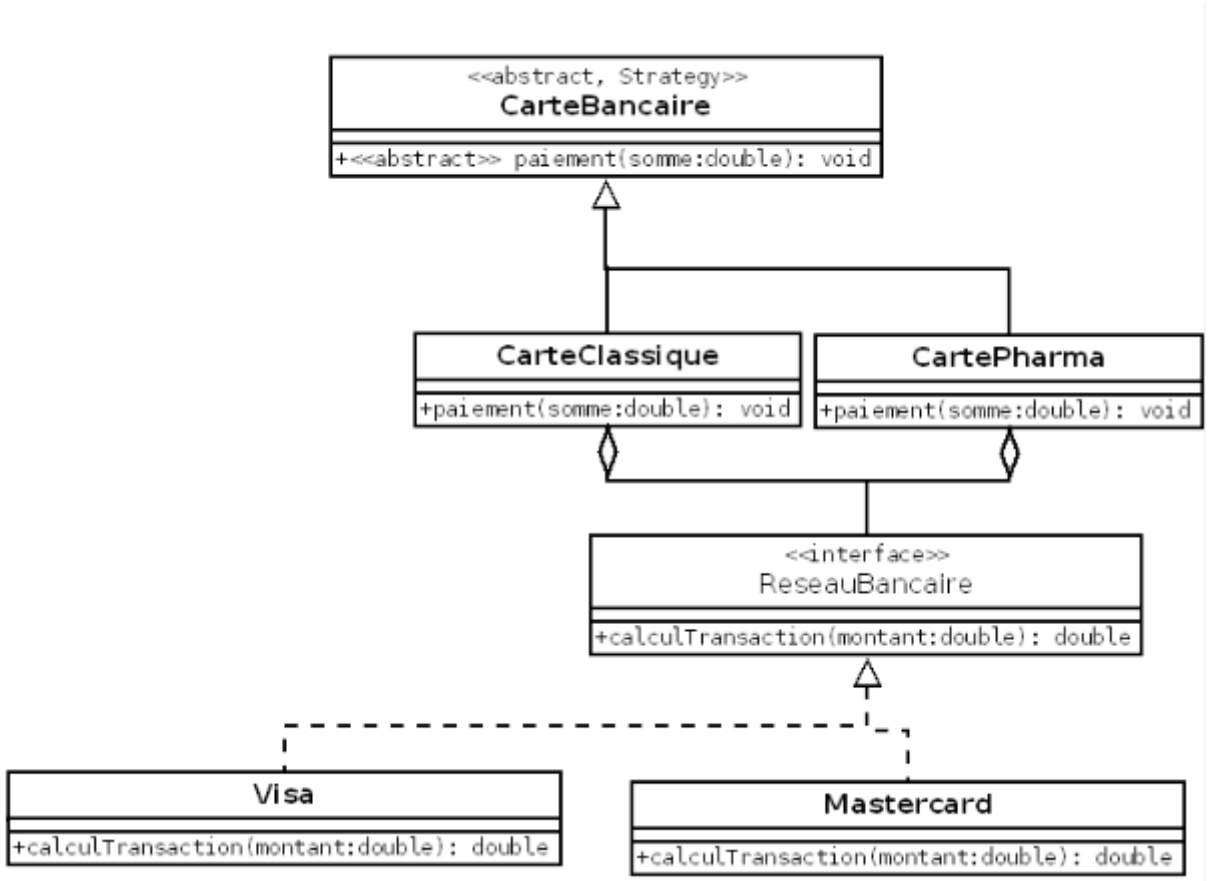
State



Le pattern State est utilisé dans notre cas pour la gestion de l'ancienneté, pour les préparateurs de commande dont le salaire dépend de leur ancienneté.

Ce pattern nous permet de gérer les différents états possibles de l'ancienneté sans modifier l'interface Anciennete elle-même. De plus, si le système d'ancienneté est mis à jour et que les états changent (ou qu'ils y a de nouveaux états ajoutés au système), alors ce pattern nous permet de mettre à jour l'application aisément.

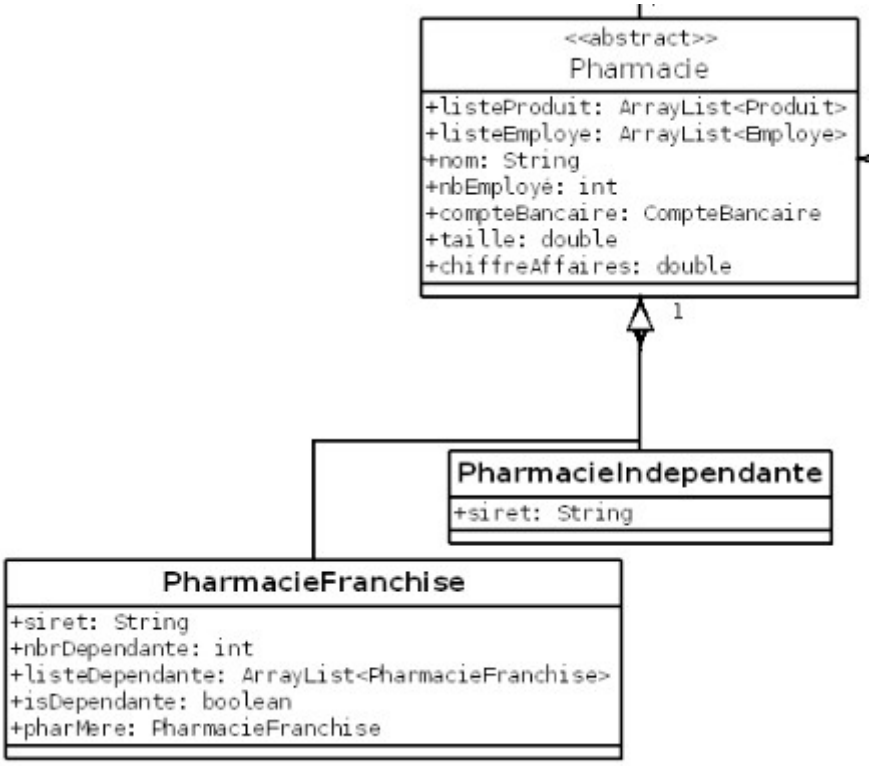
Strategy

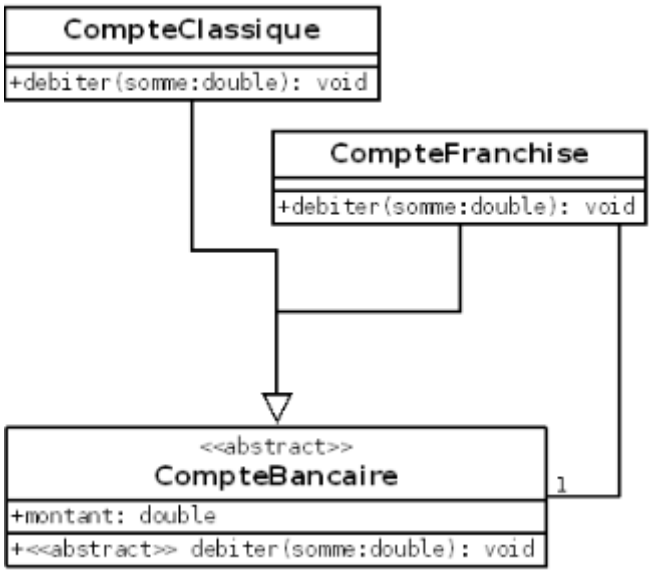


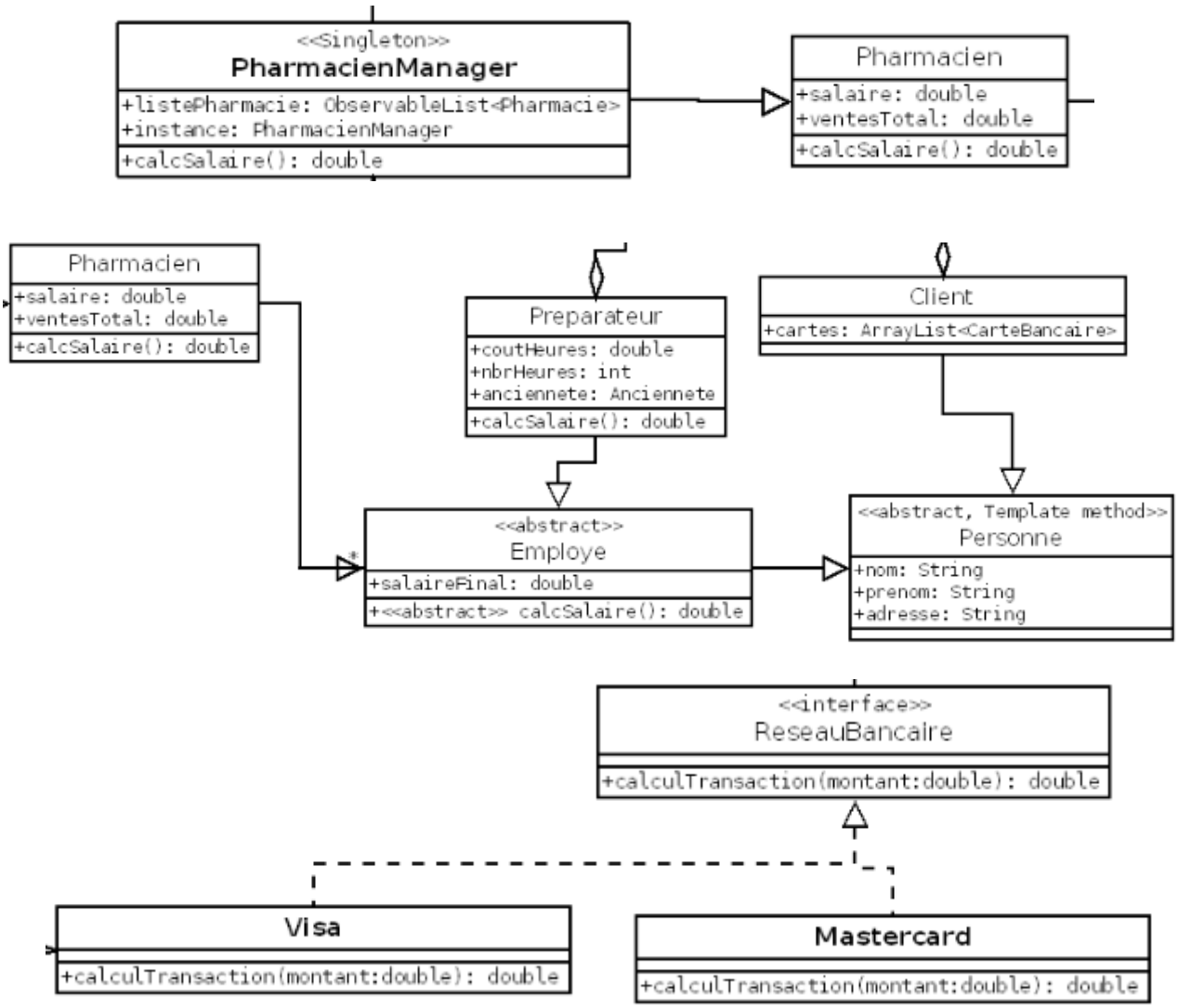
Le pattern Strategy est utilisé dans notre cas pour la gestion des différents systèmes de paiement (et de remboursement) des différents types de cartes bancaires (ici, cartes classiques et cartes de pharmacies) à partir de la classe abstraite CarteBancaire.

Ce pattern nous permet aisément, d'une façon similaire au pattern State vu précédemment, d'ajouter ou de modifier différents types de cartes bancaires, tout en ayant tous ces types reliés au même système de réseau bancaire.

Template Method







Le pattern Template Method est utilisé dans notre cas à plusieurs reprises afin de gérer l'héritage de classes, telles que le réseau bancaire (Mastercard ou Visa), les personnes (clients ou employés), les employés (préparateurs ou pharmaciens), les pharmaciens (un pharmacien manager étant un pharmacien), les comptes bancaires (classiques ou franchisés), et les pharmacies (franchisées ou indépendantes).

Ce pattern nous permet tout simplement de gérer les différentes classes qui implémentent une même classe abstraite. Il s'agit d'héritage simple.

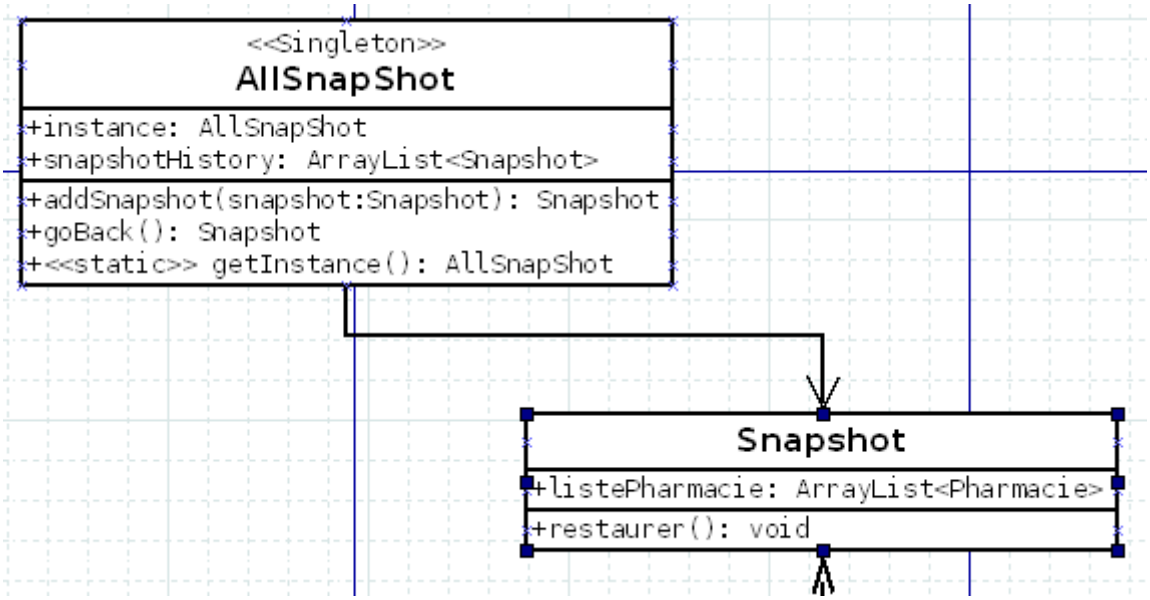
Observer

Le pattern Observer, non-représenté sur le diagramme, est utilisé dans notre cas pour mettre à jour la vue de l'utilisateur lorsqu'une modification est apportée aux données.

À chaque achat / vente de la part d'un client, les fonctions concernées appellent une fonction du contrôleur pour rafraîchir la vue, afin que les informations visibles à l'utilisateur soient toujours d'actualité.

Cela nous permet d'avoir une vue toujours à jour pour l'utilisateur.

Memento



Malheureusement, par manque de temps, ce pattern n'a pas été implémenté. Il aurait servit à garder une trace de chaque modification, et à revenir en arrière à n'importe quel moment.