POLYTECH NICE SOPHIA

22/02/2023

Rapport d'architecture ISA

Equipe B

Bevan Tom
Bourdeau Quentin
Correia Ambre
Faucher Vinh
Jeannes Théo

1. Introduction

Ce rapport présente la conception du projet de carte multi-fidélités. Ce système de carte permet aux usagers d'utiliser une carte de fidélité dans plusieurs magasins d'une même zone, pour accumuler des points de fidélités, qui leur permet de profiter de divers avantages auprès des magasins, comme un café gratuit, ou auprès des municipalités, comme la gratuité d'un ticket de bus par jour.

Hypothèses de travail :

Nous avons fait différentes hypothèses dans notre interprétation du sujet :

- Lorsque le sujet mentionne *“un administrateur du système peut l’interroger pour obtenir des informations sur les habitudes de consommation des utilisateurs (en achat “brut” et en cadeaux récupérés).”*, cela veut dire que les habitudes de consommations en achat brut seraient la consommation d’un client moyen dans les différents magasins, tandis que les habitudes de consommations en cadeaux récupérés seraient représentés par le coût lié aux offres promotionnelles offertes par magasin par client.
- Lorsque le sujet mentionne *“un administrateur [...] peut envoyer des offres promotionnelles à la demande du service “Citoyen Numérique” ou de l’association des commerçants, ou encore lancer des sondages de satisfaction aux usagers”,* cela signifie que l’administrateur devait avoir possibilité d’envoyer des mails aux utilisateurs, mais que les potentielles réponses aux sondages étaient géré par un système externe, en dehors du périmètre du projet.
- Lorsque le sujet mentionne “une commerçante partenaire [...] peut aussi obtenir des indicateurs sur l’utilisation du programme auprès des autres partenaires, pour savoir si sa participation est surévaluée ou non”, cela implique que le commerçant doit avoir la possibilité de connaître son volume de vente lié à la carte multi-fidélité et les coûts liés aux avantages offerts, mais également sa position par rapport aux autres commerces, sans pour autant divulguer les volumes de ventes ou les coûts des autres partenaire du programme.
- Lorsque le sujet mentionne “Pour obtenir un cadeau, il faut avoir fait au moins un achat antérieur dans la boutique concernée, et présenter sa carte lors du paiement de l’achat en cours (on ne peut pas juste venir prendre un cadeau)”, cela signifie que l’utilisateur doit avoir acheter préalablement dans la boutique et présenter sa carte pour retirer un avantage fidélité.
- De manière générale, les avantages fidélités sont consommés lors de leur achat, et ne peuvent pas être stockés par l’utilisateur.

2. Cas d’utilisation

Acteurs primaires

Le client est représenté par les personnas de Jacques, d’Alison et de Pierre. Il utilise l’application pour bénéficier d’avantage grâce à leurs achats.

Le gérant d'enseigne est représenté par la personna de Laura. Le gérant utilise l'application pour attirer des clients et rivaliser avec la concurrence des plus grandes enseignes.

L'administrateur système est représenté par la personna de Frank. Il supervise le projet et a des droits privilégiés, notamment celui de créer des comptes administrateur et gérant, en plus d'ajouter les cadeaux de la ville au catalogue.

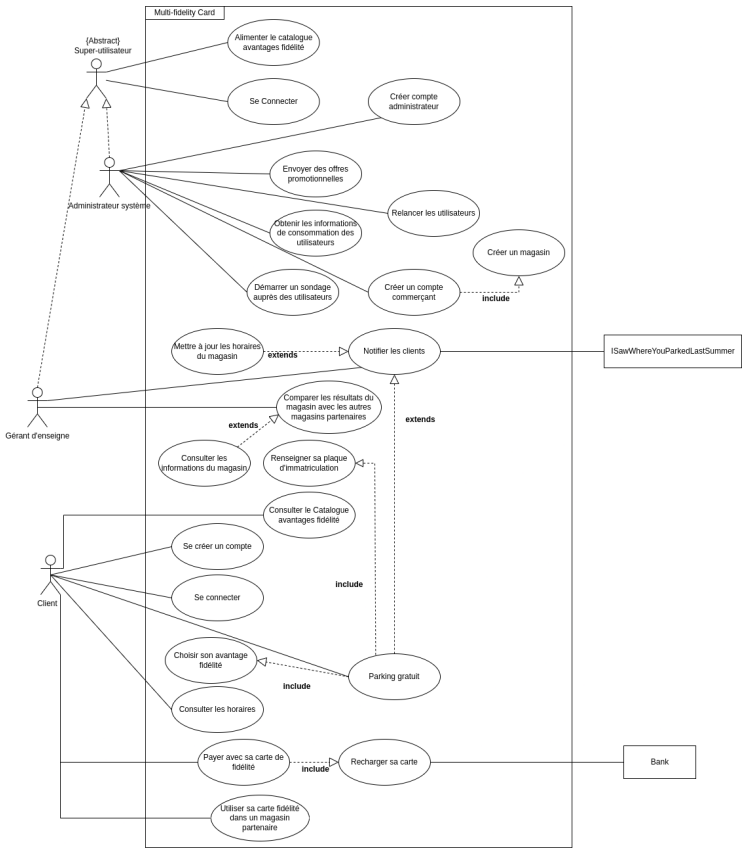
Acteurs secondaires

La banque est appelée lorsqu'un utilisateur veut recharger sa carte pour pouvoir payer.

ISawWhereYouParkedLastSummer est appelé lorsqu'un client utilise l'avantage d'un parking gratuit.

Diagramme de cas d'utilisation

Chacune des personas fournies dans le sujet ont donné lieu aux différents cas d'utilisation décrits dans notre diagramme de cas d'utilisation. Cela a donné lieu à des cas d'utilisation plus ou moins complexes selon les cas. Les explications et justifications pour les cas les plus complexes peuvent être trouvés ci-dessous, mais également en annexe.



Utiliser sa carte fidélité dans un magasin partenaire : Ceci est le cas de base de l'application. Un client doit avoir la possibilité d'utiliser sa carte multi-fidélité dans tous les magasins partenaires après ses achats pour accumuler des points avantages.

Consulter les informations du magasin : Le gérant d'enseigne peut consulter les informations de son magasin comme le volume de ventes effectuées à des clients possédant une carte multi-fidélité ainsi que les coûts associés. Le gérant peut ensuite, s'il le veut, comparer ces données avec les autres magasins.

Comparer les résultats du magasin avec les autres magasins partenaires : Le gérant peut comparer ses résultats de ventes avec les autres magasins partenaires. Les données sont anonymes et le gérant est présenté avec des statistiques comme son pourcentage des ventes globales effectuées à des clients possédant une carte multi-fidélité par les magasins partenaires ou encore sa fréquence de visite comparée aux autres.

Choisir son avantage fidélité : Avec les points obtenus par le client lors de ses achats, ce dernier peut choisir dans le catalogue de cadeau un avantage à obtenir selon les magasins ou les communes (par exemple une place de parking gratuite pour 30 min)

Parking gratuit : Afin de choisir d'avoir une place de parking gratuite, l'utilisateur doit au préalable avoir choisi de prendre un avantage fidélité et doit également avoir renseigné son numéro de plaque d'utilisation (d'où les différents include sur le diagramme). De plus, cet avantage est lié par un extends au cas d'utilisation *notifier les clients* car ce dernier est notifié lorsqu'il ne reste plus que 5min, et lorsque ce que le temps est écoulé.

Notifier les Clients : Le client peut être notifié à différents moments. Par exemple, lorsque le magasin change d'horaire, alors le client peut être notifié de ce changement, d'où le extends. De plus, lors de l'utilisation du parking gratuit, le client est notifié lorsqu'il ne reste plus que 5 min, et lorsque le temps est écoulé. Tout cela est géré par un acteur qui est ISawWhereYouParkedLastSummer.

Recharger sa carte : Le client a la possibilité d'utiliser sa carte de fidélité pour payer ses achats. Pour se faire, il doit recharger sa carte de fidélité en ajoutant un montant donné, il effectue cette action en ligne en payant par carte bancaire. De ce fait, ce cas d'utilisation est lié à un système de paiement externe "Bank" qui est chargé de savoir si le paiement est valide ou non.

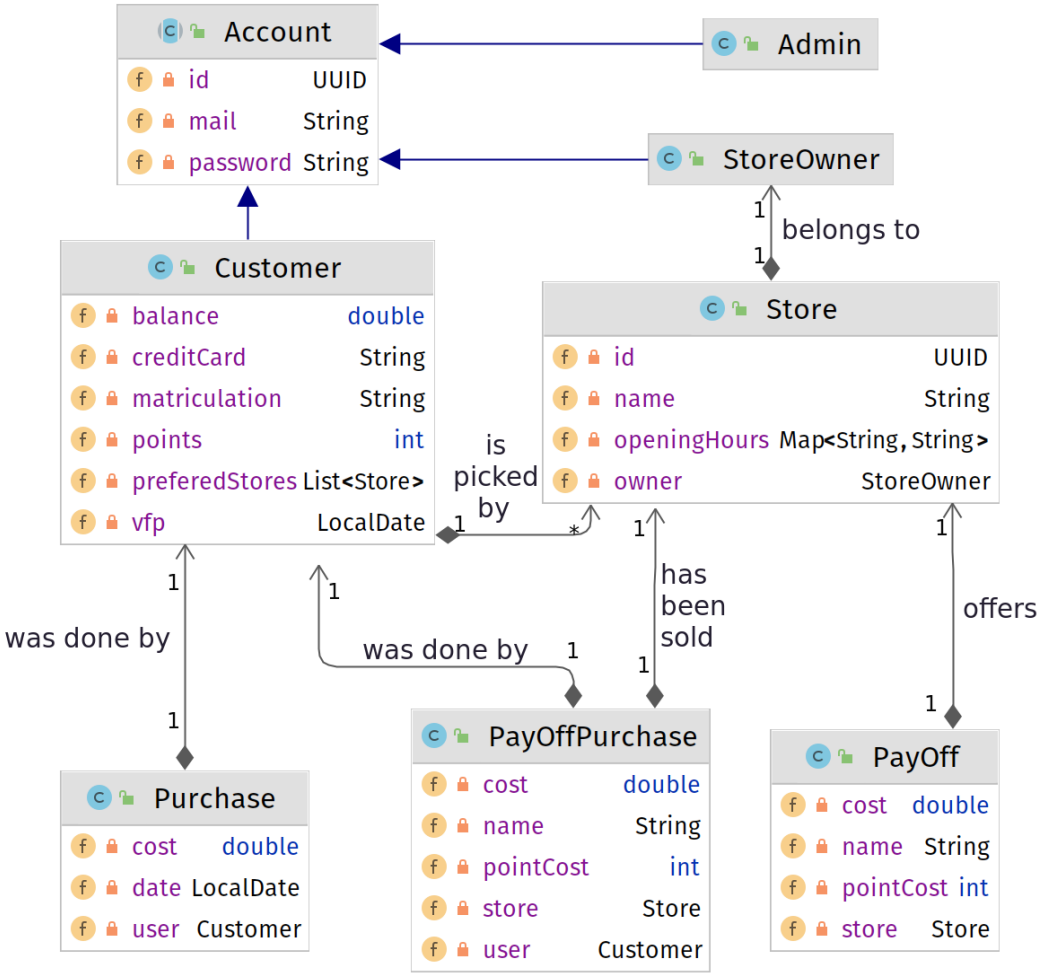
Relancer les utilisateurs : Lorsqu'un client est sur le point de perdre son statut VFP, l'administrateur peut le relancer en l'informant qu'il doit utiliser sa carte dans des magasins partenaires s' il ne souhaite pas perdre ses avantages VFP.

3. Objets métiers

Les premiers objets métiers représentent les utilisateurs du système, à savoir l'administrateur, le gérant de magasin, et le client.

Le client a besoin de nombreux arguments, pour pouvoir effectuer toutes ses actions. Il doit par exemple avoir la possibilité de renseigner sa carte bancaire pour recharger son compte, ou encore de renseigner sa plaque d'immatriculation pour bénéficier de l'application *ISawWhereYouParkedLastSummer.* Le statut VFP est représenté par une date, ce qui permet de simplement vérifier si l'utilisateur est VFP en comparant la date d'expiration dans son compte avec la date du jour.
Nous avons ensuite décidé de représenter les commerces, qui doivent pouvoir modifier leurs horaires, ce qui nécessite d'avoir des horaires. Chaque magasin est possédé par un propriétaire, il y a donc une composition qui s'impose.
De plus, les magasins peuvent proposer des offres pour les clients fidèles. Nous avons donc dû représenter les offres de fidélité avec la classe *PayOff*.
Pour plusieurs fonctionnalités, notamment les statistiques pour l'administrateur ou les commerçants, il y a un besoin de connaître les achats et les avantages offerts. Nous avons donc modélisé les classes *Purchase* et PayOffPurchase dans cet objectif. Les avantages indiquent leur coût réel ainsi que du nombre de points offerts, ce qui permet à *PayOffPurchase* de connaître le prix réel et les points dépensés à chaque transaction, ce qui permettra aux commerçants de connaître leur participation exacte au programme.

Le diagramme de classe peut être trouvé ci-dessous :



4. Interfaces

Les interfaces que nous avons identifié comme nécessaire au projet sont les suivantes :

AdminDataGathering

```
public interface AdminDataGathering {
   Map<Store,Double> inquireConsumptionHabitsSells(Admin authorization);

   Map<Store,Double> inquireConsumptionHabitsPayOffs(Admin authorization);
}
```

Cette interface permet à un administrateur de récupérer les habitudes de consommations des utilisateurs, que ce soit concernant les achats ou les avantages fidélités réclamés.

AdminFinder

```
public interface AdminFinder {
    Optional<Admin> findAdminByMail(String mail, String password);

    Optional<Admin> findAdminById(UUID id, String password);
}
```

Cette interface permet d'accéder au compte d'un administrateur avec son email ou son id, et son mot de passe.

AdminNotifier

```
public interface AdminNotifier {
        boolean notify(List<Customer> targets, String message, Admin
authorization);
}
```

Cette interface permet à un administrateur de notifier tous les clients souhaités avec un message, comme un sondage ou une offre promotionnelle par exemple.

AdminRegistration

```
public interface AdminRegistration {
    Admin register(String mail, String password, Admin authorization) throws
AlreadyExistingAccountException;
}
```

Cette interface permet à un administrateur de créer un nouveau compte administrateur, et lève une erreur si l'adresse mail est déjà utilisée.

Bank

```
public interface Bank {
   boolean pay(Customer customer, double balance) throws PaymentException;
}
```

Cette interface permet de communiquer avec la banque pour valider le paiement d'un client et lève une exception si le paiement est refusé.

CatalogExplorer

```
public interface CatalogExplorer {
   Set<PayOff> availablePayoffs(Customer customer);
   Set<PayOff> exploreCatalogue(Customer customer, String search);
}
```

Cette interface permet à un client de consulter le catalogue, et de consulter les récompenses auxquelles il peut prétendre.

CatalogModifier

```
public interface CatalogModifier {
   boolean addPayOff(String name, double cost, int pointCost, Store store,
StoreOwner authorization) throws NegativeCostException,
NegativePointCostException, CredentialsException;

   boolean editPayOff(PayOff payOff, Store store, double cost, int pointCost,
StoreOwner authorization) throws NegativeCostException,
NegativePointCostException, CredentialsException;

   boolean editPayOff(PayOff payOff, Store store, double cost, StoreOwner
authorization) throws NegativeCostException, CredentialsException;

   boolean editPayOff(PayOff payOff, Store store, int pointCost, StoreOwner
authorization) throws NegativePointCostException, CredentialsException;

   boolean addPayOffAdmin(String name, double cost, int pointCost, Admin
authorization) throws NegativeCostException, NegativePointCostException;

   boolean editPayOff(PayOff payOff, Store store, double cost, int pointCost,
Admin authorization) throws NegativeCostException,
NegativePointCostException;

   boolean editPayOff(PayOff payOff, Store store, double cost, Admin
authorization) throws NegativeCostException;

   boolean editPayOff(PayOff payOff, Store store, int pointCost, Admin
authorization) throws NegativePointCostException;
}
```

Cette interface permet à l'administrateur et au commerçant d'ajouter des offres au catalogue ou de les modifier. Des exceptions seront levées si un commerçant essaye de modifier une offre qui ne lui appartient pas, et si les coûts, réels ou en points, des offres sont négatifs.

ContributionGathering

```
public interface ContributionGathering {
   Map<String, Double> inquireComparisonContribution(Store current);

   Map<String, Double> inquireComparisonSell(Store current);
}
```

Cette interface permet au magasin de se comparer aux autres partenaires du programme.

CustomerBalancesModifier

```
public interface CustomerBalancesModifier {
   Customer editBalance(Customer customer, double balanceChange) throws
InsufficientBalanceException;

   Customer editFidelityPoints(Customer customer, double balanceChange)
throws NegativePointCostException;
}
```

Cette interface permet aussi de modifier le solde et le nombre de points associé à chaque client, lors d'un achat ou lorsque l'utilisateur sélectionne une récompense. Des exceptions seront levées, si les paiements mènent à des soldes négatifs, que ce soit pour les points fidélités, ou pour l'argent stocké sur la carte.

CustomerFinder

```
public interface CustomerFinder {
   Optional<Customer> findCustomerByMail(String mail, String password);

   Optional<List<String>> findCustomersMailByStore(Store store);

   Optional<Customer> findCustomerById(UUID id, String password);
}
```

Cette interface permet de récupérer les clients, avec leurs adresses mails ou IDs, et leurs mots de passe. Elle permet également au *NotifierHandler* de récupérer les adresses mails des clients qui ont le magasin choisi dans leurs favoris.

CustomerProfileModifier

```
public interface CustomerProfileModifier {
   Customer recordMatriculation(Customer customer, String matriculation);

   Customer recordCreditCard(Customer customer, String creditCard);

    Customer recordNewFavoriteStore(Customer customer, Store store) throws
StoreNotFoundException;

     Customer removeFavoriteStore(Customer customer, Store store) throws
StoreNotFoundException;

     Customer recordNewFavoriteStores(Customer customer, Set<Store> store)
throws StoreAlreadyRegisteredException;

     Customer removeAllFavoriteStores(Customer customer, Set<Store> store)
throws StoreAlreadyRegisteredException;
}
```

Cette interface permet de modifier les attributs du client, que ce soit si le client veut saisir sa plaque d'immatriculation, enregistrer un magasin dans ses favoris, ou saisir son moyen de paiement pour recharger ce compte. Des exceptions seront levées lors de l'ajout de magasins dans les favoris, s'ils sont déjà dans la liste, et pendant la suppression si les magasins ne sont pas présents dans les favoris de l'utilisateur.

CustomerRegistration

```
public interface CustomerRegistration {
   Customer register(String mail, String password) throws
AlreadyExistingAccountException;
}
```

Cette interface permet de créer un compte client, et lève une exception si l'adresse mail est déjà utilisée.

Parking

```
public interface Parking {
   boolean park(String matriculation);
}
```

Cette interface permet de communiquer avec le service externe *ISawWhereYouParkedLastSummer*, et lève une exception si la communication avec le service échoue.

ParkingNotifier

```
public interface ParkingNotifier {
   boolean notify(Customer target, int remainingTime);
}
```

Cette interface permet de notifier un client avec le temps restant pour la gratuité de son stationnement, lors de l'avantage "Parking Gratuit".

ParkingProccesor

```
public interface ParkingProcessor {
   boolean useParkingPayOff(Customer user) throws NoMatriculationException,
ParkingException;
}
```

Cette interface permet à l'utilisateur d'utiliser son avantage "Parking Gratuit", et lève une exception si l'utilisateur n'a pas de plaque d'immatriculation renseignée dans son profil.

Payment

```
public interface Payment {
   boolean refillBalance(Customer user, double amount) throws
NoCreditCardException,NegativeRefillException, PaymentException;
}
```

Cette interface permet à un client de payer, pour pouvoir recharger son compte, et lève une exception si le paiement est refusé, ou s'il n'a pas renseigné de carte bancaire.

PayOffProcessor

```
public interface PayOffProcessor {
   Customer claimPayOff(Customer user, PayOff payOff) throws
InsufficientBalanceException, VFPExpiredException;
}
```

Cette interface permet de traiter la demande d'avantage d'un client, et lève une exception si le client n'a pas assez de points, ou s'il demande un avantage VFP alors qu'il ne l'est pas.

PayOffPurchaseFinder

```
public interface PayOffPurchaseFinder {
   Set<PayOffPurchaseFinder> lookUpPayOffPurchasesByStore(Store store);

   Set<PayOffPurchaseFinder> lookUpPayOffPurchases();
}
```

Cette interface permet de récupérer l'historique de tous les avantages réclamés, et de trier par magasins, si nécessaire.

PayOffPurchaseRecording

```
public interface PayOffPurchaseRecording {
   boolean recordPayOffPurchase(PayOff transaction, Customer user);
}
```

Cette interface permet d'enregistrer une demande d'avantage, lorsque celle-ci a été validée préalablement.

PurchaseFinder

```
public interface PurchaseFinder {
   Set<Purchase> lookUpPurchasesByStore(Store store);

   Set<Purchase> lookUpPayPurchases();

   Set<Purchase> lookUpPurchasesByCustomer(Customer customer);
}
```

Cette interface permet de récupérer l'historique de tous les achats, et de trier par magasins ou par client si nécessaire.

PurchaseRecording

```
public interface PurchaseRecording {
   boolean recordPurchase(Purchase purchase, Customer user);

}
```

Cette interface permet d'enregistrer un achat, lorsque celui-ci a été validé.

StoreDataGathering

```
public interface StoreDataGathering {
     Map<String, Double> inquireOwnContribution(Store current, StoreOwner
authorization) throws CredentialsException;

        Map<String, Double> inquireOwnSells(Store current, StoreOwner
authorization) throws CredentialsException;
}
```

Cette interface permet au magasin de récupérer son volume de ventes et ses contributions en avantages fidélités de son magasin auprès du programme.

StoreFinder

```
public interface StoreFinder {
   Optional<Store> findStoreByName(String name);

   Optional<Store> findStoreById(UUID id);

   Optional<Map<String,String>> findOpeningHoursOfStore(Store store);
}
```

Cette interface permet de trouver un magasin avec son nom ou son ID, ce qui permet au client de l'ajouter à ses magasins favoris, et de consulter les horaires d'un magasin choisi.

StoreModifier

```
public interface StoreModifier {
   boolean updateOpeningHours(Store store, Map<String, String> openingHours,
StoreOwner storeOwner) throws CredentialsException;
}
```

Cette interface permet pour un commerçant de modifier les horaires de son magasin. Une exception sera levée si un commerçant essaye de modifier les horaires d'un magasin qui ne lui appartient pas .

StoreNotifier

```
public interface StoreNotifier {
   boolean notify(Map<String, String> openingHours);
}
```

Cette interface permet de notifier les clients qui ont le magasin en favoris lorsque le magasin change ses horaires.

StoreOwnerFinder

```
public interface StoreOwnerFinder {
   Optional<StoreOwner> findStoreOwnerByMail(String mail, String password);

   Optional<StoreOwner> findStoreOwnerById(UUID id, String password);
}
```

Cette interface permet de trouver un commerçant avec son adresse mail ou son id, et son mot de passe.

StoreOwnerRegistration

```
public interface StoreOwnerRegistration {
   StoreOwner register(String mail, String password, Admin authorization)
throws AlreadyExistingAccountException;
}
```

Cette interface permet à un administrateur de créer un nouveau compte commerçant, et lève une erreur si l'adresse mail est déjà utilisée.

StoreRegistration

```
public interface StoreRegistration {

   Store register(Map<String, String> openingHours, StoreOwner storeOwner,
String name) throws AlreadyExistingStoreException;

}
```

Cette interface permet de créer un magasin, avec le propriétaire du magasin, le nom et les horaires d'ouverture, et lève une exception si un autre magasin du même nom existe.

TransactionProcessor

```
public interface TransactionProcessor {
   Customer purchase(Customer user, double cost);

   Customer purchaseFidelityBalance(Customer user, double cost) throws
InsufficientBalanceException;
}
```

Cette interface permet de gérer les achats des clients, qu'ils payent avec un moyen de paiement externe ou avec leur compte client. S'ils payent avec leur compte client et que le solde n'est pas suffisant, une exception sera levée.

5. Composants

StoreOwnerRegistry : Le composant StoreOwnerRegistry gère les comptes des gérants de magasin, il sert notamment à créer un compte commerçant (avec

l'interface *StoreOwnerRegistration)*, mais aussi à créer le magasin en lien avec le propriétaire grâce à *StoreRegistration*.

AdminRegistry : Le composant AdminRegistry sert à créer (avec l'interface *AdminRegistration*) et à retrouver (avec l'interface *AdminFinder*) les comptes administrateurs du système.

CatalogRegistry : Le composant CatalogRegistry permet de modifier l'état du catalogue, c'est à dire d'ajouter et retirer des offres (avec l'interface *CatalogModifier*), et de renvoyer le catalogue pour un client donné (avec l'interface *CatalogExplorator*) , en prenant en compte les magasins qu'il a déjà visité et les cadeaux journaliers déjà récupérés.

DataGatherer : Le composant DataGatherer est un composant qui sert à récupérer et à agréger les données demandées par l'administrateur ou le gérant de magasin (avec les interfaces *StoreDataGathering*, *ContributionGathering* et *AdminDataGathering*). Il a donc accès aux historiques de transactions, que ce soit des achats grâce à l'interface *PurchaseFinder* ou des avantages fidélités avec l'interface *PayOffPurchaseFinder* ,et à la responsabilité de préparer les données avant de les transmettre aux utilisateurs.

PurchaseRegistry : Le composant PurchaseRegistry sert à enregistrer (grâce à l'interface *PurchaseRecording*) et récupérer l'historique d'achats des consommateurs (grâce à l'interface *PurchaseFinder*) .

PayOffPurchaseRegistry : Le composant PayOffPurchaseRegistry sert à enregistrer (grâce à l'interface `PayOffPurchaseRecording`) et récupérer l'historique des avantages fidélités réclamés par les utilisateurs (grâce à l'interface *PayOffPurchaseFinder*) .

StoreHandler : Le composant StoreHandler permet de modifier (grâce à l'interface *StoreModifier*) et d'ajouter des magasins (grâce à l'interface *StoreRegistration*), ainsi que de faire les actions en lien avec un magasin, comme changer les horaires ou les offres de celui-ci. Il peut prévenir d'un changement d'horaire en utilisant l'interface *StoreNotifier.*

TransactionHandler : Le composant TransactionHandler permet de gérer les transactions faites par le client (grâce à l'interface *TransactionProcessor*), en vérifiant la validité de celles-ci. Si les transactions sont autorisées, il a la responsabilité de modifier le compte du client (grâce à l'interface *CustomerModifier*) avec les informations de la transaction. Il pourra ensuite l'enregistrer grâce à l'interface *PurchaseRecording.*

NotificationHandler : Le composant NotificationHandler permet de notifier les clients, il est utilisé par les magasins lors des changements d'horaire (grâce à l'interface *StoreNotifier*) pour informer les utilisateurs qui ont le magasin dans leurs favoris en les trouvant grâce à l'interface *CustomerFinder*, lors de la perte d'un statut VFP, lorsqu'un stationnement gratuit est sur le point d'expirer (grâce à l'interface *ParkingNotifier*) ou lors de la demande d'un administrateur (grâce à l'interface *AdminNotifier*).

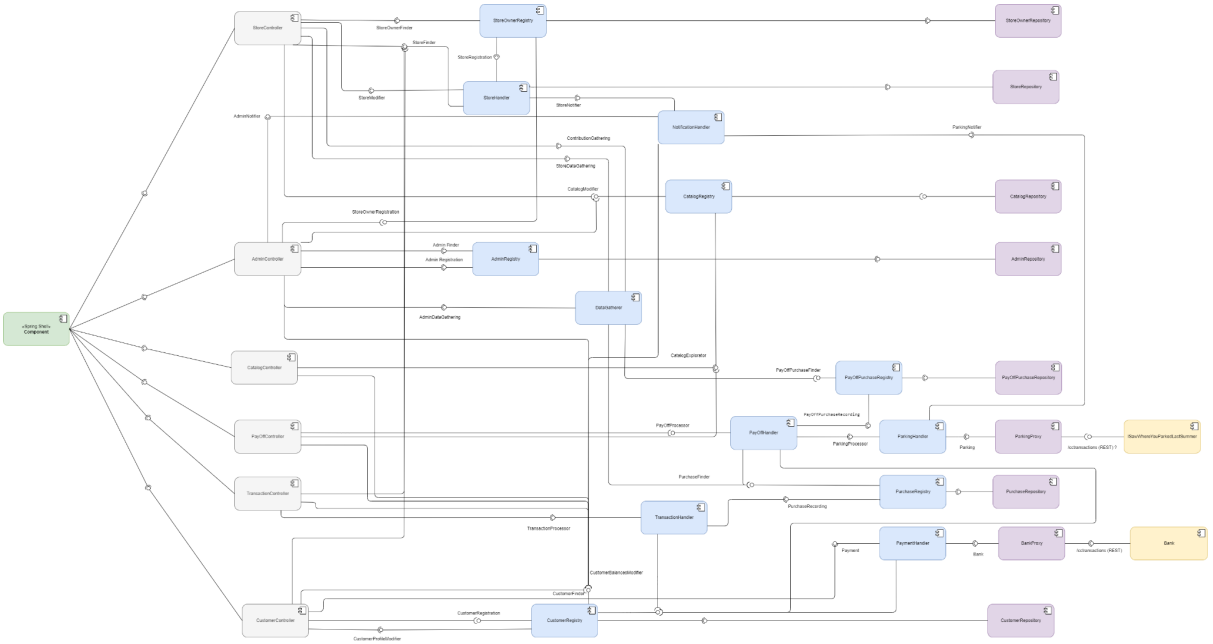
PayOffHandler : Le composant PayOffHandler permet de vérifier la validité de ce que le client demande (grâce à l'interface *PayOffProcessor)* en vérifiant par exemple son statut VFP et il a la responsabilité d'appeler le ParkingHandler grâce à l'interface *ParkingProcessor* si la récompense choisie est une place de parking et que la transaction est valide. Si la récupération de la récompense s'est bien passée, il doit aussi enregistrer toutes les récompenses qui ont été réclamées grâce à l'interface *PayOffRegistration,* et modifier les points du client grâce à l'interface *CustomerModifier.*

CustomerRegistry : Le CustomerRegistry permet de créer (grâce à l'interface *CustomerRegistration)* et modifier les informations clients (grâce à l'interface *CustomerModifier)* tel que le numéro d'immatriculation ou le numéro de carte bancaire.

ParkingHandler : Le composant ParkingHandler permet de gérer la connexion avec le service externe ISawWhereYouParkedLastSummer grâce à l'interface *Parking,* et gère les demandes de stationnement gratuit des utilisateurs lorsqu'ils utilisent cet avantage (grâce à l'interface *ParkingProcessor*).

***Payment*Handler :** Le *PaymentHandler* permet de gérer la connexion avec la banque grâce à l'interface *Bank*, nécessaire lors du rechargement du compte client, et gère les paiements de l'utilisateur (grâce à l'interface *Payment)* .

Le diagramme des composants, résumant ces explications se trouve ci-dessous, mais aussi en annexe :



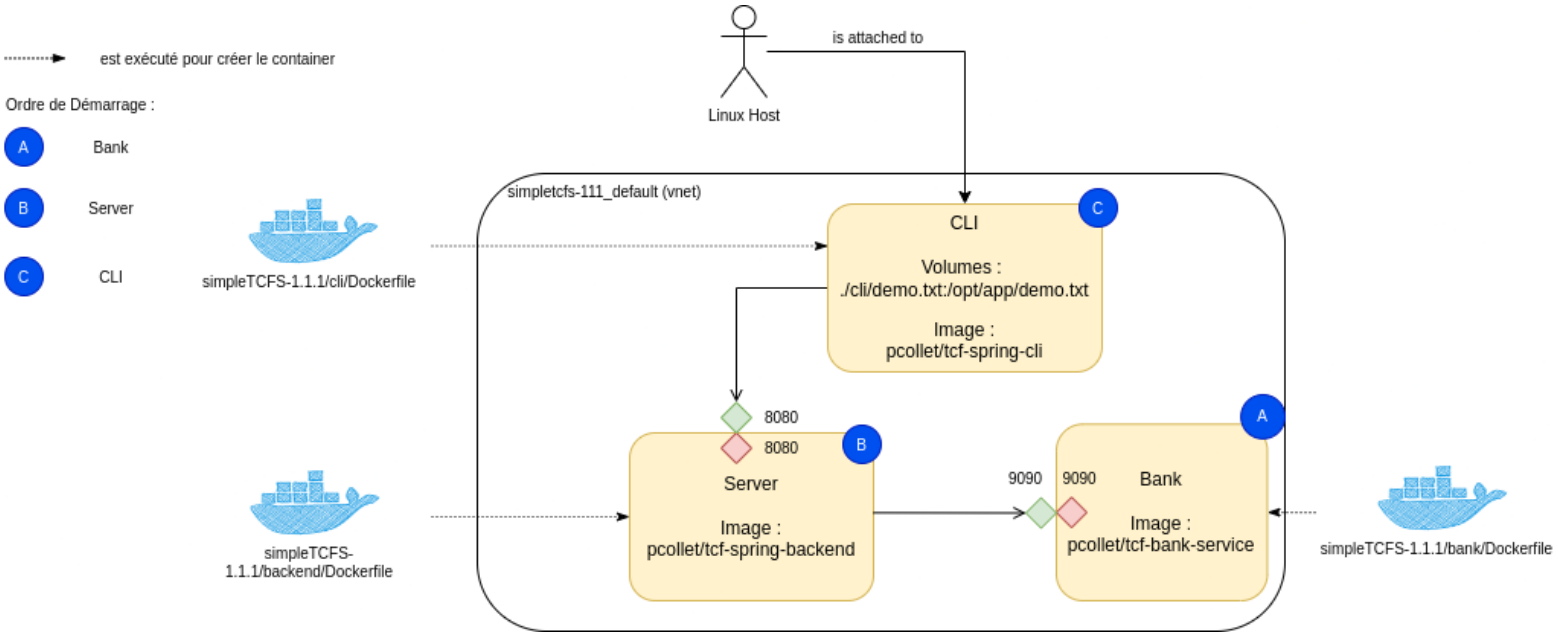
6. Scénarios MVP

Nous envisageons un ensemble de scénarios pour construire un MVP et ensuite l'étendre pour couvrir l'ensemble des fonctionnalités du programme de carte multi-fidélité:

- L'administrateur enregistre un commerçant dans le système (CLI→AdminController→ StoreOwnerRegistry → StoreHandler → StoreRepository)(Création du magasin réussi dans StoreRepository → StoreHandler → StoreOwnerRegistry → StoreOwnerRepository pour creer le compte commercant)
- L'inscription d'un client (CLI → CustomerController → CustomerRegistry → CustomerRepository)
- Le magasin enregistre un achat d'un client qui utilise sa carte fidélité (CLI → TransactionController → TransactionHandler → PurchaseRegistry → PurchaseRepository)
- Le magasin alimente le catalogue fidélité (CLI→ StoreController → CatalogRegistry →CatalogRepository)
- Le client consulte le catalogue (CLI → CatalogController →CatalogRegistry →CatalogRepository)
- Le client utilise ses points pour acheter un avantage fidélité (CLI → PayOffController → PayOffHandler → PayOffPurchaseRegistry → PayOffPurchaseRepository)

7. Docker Chart de la Cookie Factory fournie

L'hôte ne passe par le réseau pour interagir avec les containers. Avec la commande *docker attach cli*, il attache l'entrée standard, la sortie standard et la sortie d'erreur de son terminal aux entrées et sorties de la CLI. De cette façon, il peut interagir avec le système à travers la CLI sans utiliser le réseau. Un schéma montrant les containers avec leurs images, leurs volumes et leurs interactions à travers le réseau est :



Annexes

Tous les diagrammes sont trouvables en .png dans le dossier “doc” sur le github classroom correspondant.

Diagramme des cas d’utilisations