POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

Rapport d'architecture ISA

Carte multi-fidélités

Équipe F

BAILET Ludovic CHABANIER Aurélia GAZZEH Sourour
HERRMANN Matis YAHIAOUI Imène

Sommaire

1. Introduction

Dans le cadre de ce cours, nous avons été chargés de créer un logiciel de cartes multi-fidélités pour les villes partenaires afin d'aider les petits commerçants à lutter contre les centres commerciaux. Le présent rapport a pour objectif de vous présenter notre conception de ce logiciel.

1.1. Présentation du sujet

Les cartes multi-fidélités permettent aux adhérents d'obtenir des cadeaux dans les magasins contre des points de fidélité et de profiter de plusieurs avantages dans la ville, comme des places de parking gratuites pendant un certain temps, ou des tickets de bus offerts. Les clients ayant souscrit à ce programme pourront devenir des Very Faithful Person (VFP), et obtenir de meilleurs avantages.

2. ISA

2.1. Diagramme de composants

Vous trouverez notre diagramme en suivant ce lien : Diagramme de composants

AdminManager

Le composant **AdminManager** est chargé de créer et gérer les comptes administratifs tels que les comptes shop, shopkeeper et admin. En effet, la création de ces comptes nécessite une validation préalable de l'administrateur et leur gestion implique l'attribution de privilèges spécifiques en fonction des rôles de chaque utilisateur. La séparation de cette fonctionnalité dans un composant distinct permet donc une meilleure organisation et un contrôle plus efficace sur les différents types de ces comptes.
Ce composant s'occupe donc de la création et de la suppression de ces comptes, ainsi que de l'envoi des mails de promotion et des sondages. Il implémente ces interfaces :

- **AdminRegistration** : Cette interface permet à un administrateur de créer ou supprimer un autre compte administrateur.

```
public interface AdminRegistration {
    public AdminAccount createAdminAccount(Form form)
            throws MissingInformationException, AlreadyExistingAdminException;
    public void deleteAdminAccount(AdminAccount account);
```

- **AdminFinder** : Son rôle est de récupérer un compte administrateur avec l'identifiant passé en paramètre.

```
public interface AdminFinder {
    AdminAccount findAdminByMail(String mail);
    Optional<AdminAccount> findAdminById(Long id);
```

- **ShopKeeperRegistration**: Cette interface permet à un administrateur d'ajouter, de créer ou de supprimer un compte pour un employé qui travaille dans un des magasins partenaires.

```
public interface ShopkeeperRegistration {
    ShopKeeperAccount createShopKeeperAccount(Form form, long shopId)
            throws MissingInformationException, AlreadyExistingMemberException, UnderAgeException;
    void deleteShopKeeperAccount(ShopKeeperAccount account);
```

- **ShopRegistration** : Cette interface permet à un administrateur d'ajouter ou supprimer un magasin partenaire au programme multi-fidélités.

```
public interface ShopRegistration {
    Shop addShop(String name, String address) throws MissingInformationException;
    void removeShop(Shop shop);
```

- **MailSender** : Cette interface permet à un administrateur d'envoyer des mails aux clients.

```
public interface MailSender {
    boolean sendMail(List<MemberAccount> members, Mail mailToSend);
    boolean sendSurvey(List<MemberAccount> members, Survey survey);
```

MemberManager

Le composant **MemberManager** gère tout ce qui est création, mise à jour, archivage et suppression des comptes des membres. Il permet aussi de restaurer un compte archivé à la réinscription d'un ex-membre et de retrouver un compte à partir de son identifiant et la recharge de la carte d'abonnement.
Il gère aussi la mise à jour des statuts vfp, l'archivage des comptes expirés ainsi que la suppression des comptes archivés depuis plus de deux ans d'une façon automatique grâce à l'annotation Spring **Scheduled.**

Les interfaces implémentées par ce composant sont :

- **MemberHandler :** L'interface MemberHandler permet de gérer les comptes des membres en créant, archivant, restaurant ou supprimant des comptes.
  Elle permet aussi au membres de recharger leurs cartes en communiquant avec le composant **BankProxy**
  Elle met également à jour les statuts des comptes en fonction du nombre d'achats effectués à chaque période définie.

```
public interface MemberHandler {
    MemberAccount createAccount(String name, String mail, String password, LocalDate birthDate)
            throws MissingInformationException, AlreadyExistingMemberException, UnderAgeException;
    void archiveAccount(MemberAccount memberAccount) throws AccountNotFoundException;
    void restoreAccount(MemberAccount memberAccount) throws AccountNotFoundException;
     void deleteAccount(MemberAccount memberAccount) throws AccountNotFoundException;
    void updateAccount(MemberAccount memberAccount,Form form) throws AccountNotFoundException;
    void updateAccountsStatus();
    void updateAccountStatus(MemberAccount memberAccount, AccountStatus status) throws AccountNotFoundException;
    void renewMembership(MemberAccount memberAccount) throws AccountNotFoundException, TooEarlyForRenewalException;
    void archiveOrDeleteExpiredAccounts() throws AccountNotFoundException;
    void chargeMembershipCard(MemberAccount memberAccount, double amount, String creditCard)
            throws AccountNotFoundException, PaymentException;
```

- **MemberFinder :** Son rôle est de retrouver un compte membre avec l'identifiant passé en paramètre

```java
public interface MemberFinder {
    Optional<MemberAccount> findById(Long id);
    Optional<MemberAccount> findByMail(String mail);
    List<MemberAccount> findAll();
```

Au début, le **memberManager** était surtout un composant CRUD, puis on a incorporé du métier en implémentant la fonctionnalité VFP, la gestion automatique des comptes expirés et la recharge de carte. Nous avons choisi de regrouper toutes les fonctionnalités liés à la gestion des comptes member dans ces composants afin de pouvoir en implémenter plus dedans plus facilement sans avoir besoin de créer des composants et des liaisons supplémentaires.

ShopManager

Le composant **ShopManager** est responsable de la gestion des informations d'un magasin, telles que la modification de l'adresse et des horaires d'ouverture. Il permet aussi de retrouver les magasins et les comptes des marchands depuis leurs identifiants.

- **ShopHandler :** Son rôle est de gérer les “responsabilités des vendeurs”, cela inclut la modification des différentes informations sur les magasins ainsi que l’envoie de notifications au membres par mail (voir la liaison shopManager - mailSender dans le diagramme de composants)

```java
public interface ShopHandler {
    void modifyAddress(Shop shop, String address);
    void modifyPlanning(Shop shop, WeekDay day, LocalTime OpeningHours, LocalTime ClosingHours);
     void modifyName(Shop shop, String name);
```

- **ShopFinder :** Son rôle est de récupérer un magasin avec l’identifiant passé en paramètre.

```java
public interface ShopFinder {
    Optional<Shop> findShopById(Long id);
```

- **ShopKeeperFinder :** Son rôle est de récupérer un magasin avec l’identifiant passé en paramètre.

```java
public interface ShopkeeperFinder {
     Optional<ShopKeeperAccount>findShopKeeperAccountByName(String name);
     Optional<ShopKeeperAccount> findShopkeeperAccountById(Long id);
    ShopKeeperAccount findShopkeeperAccountByMail(String mail);
```

Catalog

Le composant **Catalog** est responsable de la gestion du catalogue de produits et cadeaux

- **CatalogEditor :** Cette interface permet de modifier le catalogue global du programme multi-crédit.

```java
public interface CatalogEditor {
    void editShopCatalog(Shop shop, List<Product> addedProducts, List<Product> removedProducts)
            throws AlreadyExistingProductException, ProductNotFoundException;
    void addProductToCatalog(Shop shop,Product product) throws AlreadyExistingProductException;
    void removeProductFromCatalog(Shop shop,Product product) throws ProductNotFoundException;
    void addGift(Shop shop, Gift gift)throws AlreadyExistingGiftException;
    void removeGift(Shop shop, Gift gift) throws GiftNotFoundException;
    void addDiscountToProduct(Shop shop, Product product, double discount)throws ProductNotFoundException;
```

- **CatalogFinder:** Cette interface permet aux marchands de modifier la section correspondant à leurs magasins dans le catalogue.

```java
public interface CatalogFinder {
    Optional<Product> findProductById(Long id);
    Optional<Gift> findGiftById(Long id);
```

Le choix de mettre en place un composant shopManager et un composant catalog permet une meilleure gestion des autorisations et des privilèges d'accès. Par exemple, l'accès au composant catalog peut être limité aux utilisateurs ayant un certain niveau de privilèges, tandis que l'accès au composant shopManager peut être accordé à d'autres utilisateurs ayant des privilèges différents. De plus . En séparant ces préoccupations en deux composants distincts, nous pouvons les développer et les gérer indépendamment les uns des autres. Par exemple, si nous devions apporter des modifications à la façon dont les produits/cadeaux sont gérés dans le catalogue, nous pourrions le faire sans affecter le code du composant shopManager.

ParkingManager

Le composant **ParkingManager** gère l'utilisation dès 30 minutes de parking par les membres avec le statut VFP. Il communique avec le service externe **IKnowWhereYouParkedLastSummer** développé sous Rust pour l'enregistrement dans les parkings.

- **ParkingHandler :** Cette interface permet aux membres *VFP* de profiter de leur temps de parking offert en communiquant leur numéro d'immatriculation à la plateforme **IsawWhereYouParkedLastSummer**

```java
public interface ParkingHandler {
    void useParkingTime(MemberAccount memberAccount, String carRegistrationNumber, int parkingSpotNumber)
            throws NotVFPException;
    ISWUPLSDTO[] getParkingInformation(String carRegistrationNumber);
```

Au début, nous avons mis la gestion de parking dans le composant **memberManager** car on voyait cette fonctionnalité comme une gestion de comptes des membres. Cependant, après réflexion nous avons conclu que la gestion des parkings, même si elle utilise l'interface **memberFinder** afin de vérifier les statuts des utilisateurs était une responsabilité à part de la gestion des comptes et donc méritait son propre composant.

Cashier et TransactionHandler

Le composant **Cashier** est chargé de la gestion des paiements effectués par les clients et communique avec la banque: le système de paiement externe.

Dans le cadre de la gestion des paiements effectués par les clients , il est appelé dans le composant **TransactionHandler** en implémentant l'interface **payment** qui se concentre sur la gestion des transactions d'achat et de l'utilisation des points de fidélité.
Ce composant à aussi un rôle dans l'attribution du statut VFP. Quand un membre fait son n-ème achat qui lui permet d'avoir ce statut, **TransactionHandler** appelle l'interface **MemberHandler** afin d'effectuer la modification du statut.

- **TransactionExplorer** : Cette interface permet aux commerçants de consulter les achats effectués et les cadeaux qui ont été offerts dans n'importe quel magasin partenaire

```
public interface TransactionExplorer {
    Optional<Transaction> findTransactionById(Long id);
    List<Transaction> findAllTransactions();
    List<Transaction> getStatisticsOnClientUsage(MemberAccount memberAccount);
    List<Transaction> getStatisticsOnClientUsageAtShop(Shop shop, MemberAccount memberAccount);
```

- **TransactionProcessor** : Cette interface permet aux commerçants de gérer les transactions d'achat et des points de fidélité.

```
public interface TransactionProcessor {
    Purchase processPurchaseWithCreditCard(MemberAccount memberAccount, Purchase purchase, String card)
            throws PaymentException, AccountNotFoundException;
    Purchase processPurchaseWithMemberCard(MemberAccount memberAccount, Purchase purchase)
            throws PaymentException, AccountNotFoundException;
    Purchase processPurchaseWithCash(MemberAccount memberAccount, Purchase purchase)
            throws PaymentException, AccountNotFoundException;
    UsePoints processPointsUsage(MemberAccount memberAccount, UsePoints usePoint)
            throws DeclinedTransactionException, InsufficientPointsException ,AccountNotFoundException;
```

- **PointTrader :** Le rôle de cette interface est d'attribuer ou enlever des points à un membre à la suite d'une transaction.

```
public interface PointTrader {
    void removePoints(MemberAccount memberAccount, UsePoints usePoints) throws InsufficientPointsException;
    void addPoints(MemberAccount memberAccount, Purchase purchase);
```

Le but de regrouper l'achat et attribution des cadeaux dans le même composant est de faciliter le parcours des transactions par les shopKeepers afin d'avoir des statistiques sur l'utilisation de l'abonnement. On peut ainsi récupérer toutes les transactions ainsi que les achats ou les utilisations des points séparément en utilisant le même composant.

2.2. Services Externes

En plus du service de la banque qui nous a été fourni avec le projet "**Simple Cookie Factory**", nous avons implémenté deux services externes pour l'envoi des mails et l'enregistrement des parkings. Il s'agit de deux serveurs **REST** simples implémentés en **Rust.**

Notre backend communique avec ces deux serveurs grâce au composants **ISWUPLSProxy** et **MailProxy** qui implémente les interfaces suivantes:

ISWUPLS : interface proxy du service "**I Saw Where You Parked Last Summer**"

```java
public interface ISWUPLS {
    4 usages  1 implementation  ImeneYAHIAOUI
    boolean startParkingTimer(String carRegistrationNumber, int parkingSpotNumber);

    1 usage  1 implementation  ImeneYAHIAOUI
    ISWUPLSDTO[] getParkingInformation(String carRegistrationNumber);
}
```

MailSender : interface proxy du service mail.

```java
public interface MailSender {
    8 usages  1 implementation  Matis
    boolean sendMail(List<String> membersMail, Mail mailToSend);
    1 usage  1 implementation  Matis
    boolean sendSurvey(List<String> membersMail, Survey survey);
}
```

2.3. Diagramme de classes

Vous trouverez notre diagramme en suivant ce lien : Diagramme de classes

Les trois classes **MemberAccount**, **AdminAccount** et **ShopkeeperAccount** héritent toutes de la classe abstraite **Account**. La classe “MemberAccount” représente les comptes des membres du système, la classe "AdminAccount" représente les comptes administrateurs, et la classe "ShopkeeperAccount" représente les comptes des propriétaires de magasins.

Cette approche permet de regrouper les fonctionnalités spécifiques à chaque type de compte dans leur propre classe respective. En utilisant cette terminologie, l'architecture permet de représenter efficacement les différents types de comptes et de leurs fonctionnalités associées. Elle permet également de différencier les privilèges et les fonctionnalités disponibles pour chaque type de compte en fonction de leurs besoins spécifiques.

L'utilisation de l'héritage entre les classes "MemberAccount", "AdminAccount" et "ShopkeeperAccount" et la classe abstraite "Account" permet également d'éviter la duplication de code et de simplifier la maintenance du système en centralisant les fonctionnalités communes dans la classe abstraite. Il y a également une hiérarchie de classes pour les transactions. La classe abstraite "Transaction" est la classe parent pour les classes "UsePoints" et "Purchase". La classe **UsePoints** représente les transactions dans lesquelles les membres utilisent des points pour gagner un cadeau, tandis que la classe **Purchase** représente les transactions d'achat de produits avec de l'argent réel. Cette approche permet de différencier les types de transactions et de gérer les opérations liées à chaque type de manière spécifique
En plus de ces classes, la classe **Gift** dispose de deux sous-classes, à savoir "**CityGift**" et "**ShopGift**", qui représentent les types de cadeaux proposés.
Chaque "ShopkeeperAccount" dispose d'une liste de **Shop** (magasin) associée, tandis que chaque "Shop" dispose d'une liste de "**Product**" (produits) et "Gift" (cadeau) associée. Cette modélisation orientée objet permet une gestion précise des informations relatives aux produits vendus par chaque magasin.

Enfin, la classe **Survey** possède une liste de **Question** qui représente un sondage, et la classe **Mail** représente un mail.
Nous avons utilisé différentes stratégies pour intégrer la persistance dans notre projet.

Pour les classes **UsePoints** et **Purchase**, qui héritent de la classe **Transaction**, nous avons choisi d'utiliser une stratégie par jointure sur les 3 tables. Chaque classe possède sa propre table, ce qui nous permet de récupérer facilement toutes les transactions effectuées peu importe son type, mais aussi récupérer chaque type de transaction (de points ou monétaire) depuis leur propre table.
Pour les classes qui héritent de la classe **Account** (soit **AdminAccount**, **MemberAccount** et **ShopkeeperAccount**) nous avons utilisé une stratégie par classe, chaque classe héritée à sa propre table afin d'éviter les conflits entre les différents types de comptes existant sur notre application.
Pour la classe **Gift**, nous avons adopté une stratégie par table unique, donc une seule table pour stocker les différents types de cadeaux (**CityGift** et **ShopGift**).
Enfin, toutes les autres classes ont leurs propres tables puisqu'elles n'héritent d'aucune autre classe.
En appliquant la méthode de la cascade de la persistance dans un système de gestion de magasin en ligne, la suppression d'un "Shop" ou d'un "ShopKeeper" entraîne la suppression de tous les produits, cadeaux et transactions liés à ce magasin ou vendeur.

2.4. Forces et faiblesses

Forces :

- La conception de notre architecture s'est appuyée sur le principe de la responsabilité unique, qui consiste à attribuer une seule responsabilité à chaque composant. Cela a permis d'obtenir une architecture plus modulaire, évolutive et facile à maintenir.

Faiblesses :

- Cette forte séparation des responsabilités engendre souvent une ambiguïté sur la répartition des responsabilités dans les composants.
- Une complexité accrue dans la communication entre les différents composants

2.5. Répartition des points

prénom NOM	points
Sourour GAZZEH	100 points
Aurélia CHABANIER	100 points
Ludovic BAILET	100 points
Imène YAHIAOUI	100 points
Matis HERRMANN	100 points

4. Conclusion

En conclusion, ce rapport a présenté l'architecture logicielle de notre application, en mettant l'accent sur les différentes couches et composants qui la composent. Nous avons vu que cette architecture suit les principes de conception, ce qui nous a permis d'obtenir une application modulaire, facile à maintenir et à étendre. De plus, nous avons implémenté toutes les fonctionnalités requises avec succès. En somme, nous sommes confiants que cette architecture logicielle répond parfaitement aux besoins de notre projet et qu'elle permettra à notre application de croître et de s'adapter facilement aux futurs changements.