Rapport final ISA

Équipe G

Bezes Bastien

Devictor Pauline

Dubois Quentin

El Kateb Sami

Latapie Florian

10 avril 2023

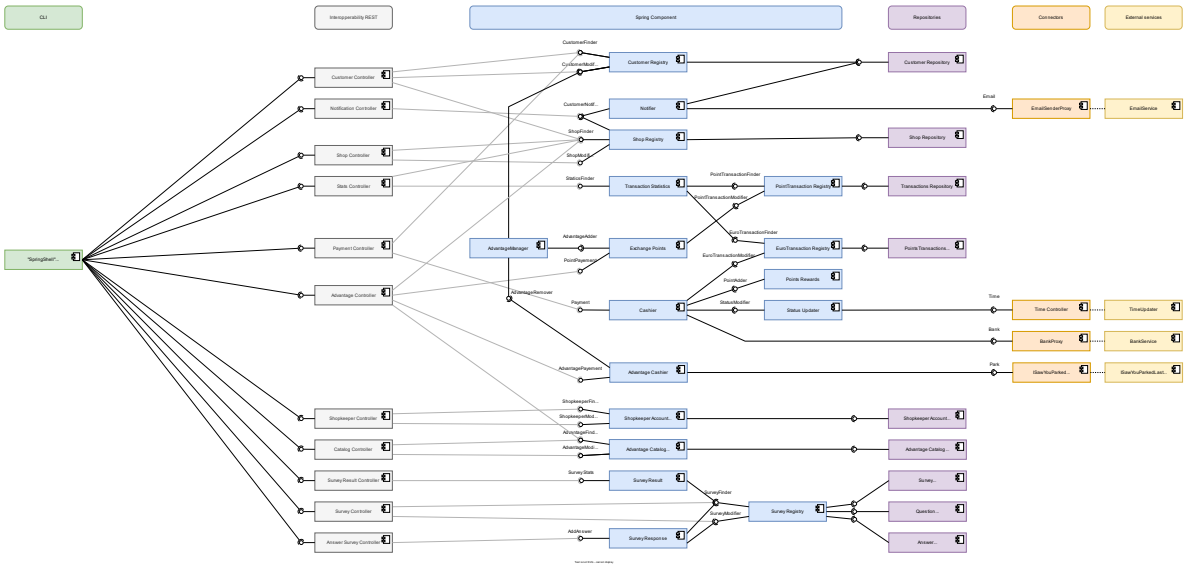
Université Côte d'Azur - Polytech Nice – SI4-ISA/DevOps

2022 – 2023

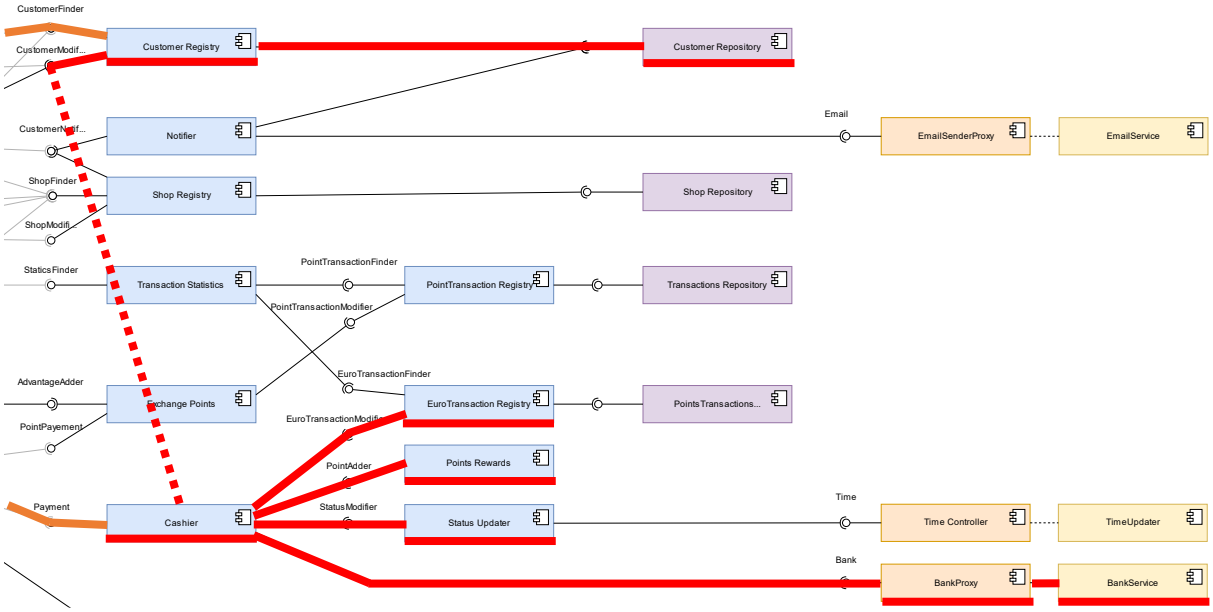
Table des matières

1. Architecture composants

Nous avons deux sous-systèmes coexistant à l'intérieur de notre application : un concernant le système multi-fidélité, l'autre le système de sondages. Ces deux systèmes sont indépendants et ne communiquent pas entre eux au niveau de leurs composants métier. Cependant, ils partagent des données en commun telles que les Customer.



Gestion d'un paiement en euro



Choix de conception

Nous avons décidé que la somme en euro à débiter à un Customer est indiquée par un commerçant, (émetteur de la commande). Nous ne nous préoccupons pas du détail des achats du client, car cela est en dehors de la portée du projet. De plus, nous n'avons pas prévu de faire des statistiques sur ces données. Par conséquent, le détail des achats nous serait inutile et consommerait inutilement de la mémoire pour leur stockage.

Cashier

Interface : Payment

Notre composant principal est le composant Cashier. Son objectif est de débiter uniquement des sommes d'argent en euro aux clients.

```
EuroTransaction payWithLoyaltyCard(Euro amount, Customer
customer, Shop shop, Date date) throws
NegativePaymentException, NegativeEuroBalanceException;
```

Il peut débiter directement la somme en euro du Customer qui lui est passée en paramètre. Cela est possible puisque nous stockons, ce que nous appelons le CustomerBalance dans le Customer. Celui-ci se compose de la quantité de points accumulée au fil des achats, des euros chargés sur sa carte de fidélité, ainsi que les avantages achetés par le client.

```
EuroTransaction payWithCreditCard(Euro amount, Customer
customer, Shop shop, String creditCard, Date date) throws
PaymentException, NegativePaymentException;
```

Dans le cas d'un paiement par carte bancaire, le composant Cashier s'adresse au Bank Proxy afin de débiter le client grâce à son numéro de carte bancaire. Cela est possible, car nous demandons le numéro de carte bancaire en paramètre de la fonction.

Si la transaction réussit, le client est débité soit directement sur sa carte bancaire, soit sur sa carte de fidélité. Le Cashier contacte alors le composant PointRewards, qui va s'occuper de calculer et d'ajouter les points au Customer. Enfin, le Cashier enregistre la nouvelle transaction dans le EuroTransactionRegistry, afin de la sauvegarder. Nous enregistrons les transactions, car nous en aurons besoin plus tard pour le calcul de statistiques concernant l'utilisation du système de carte multi-fidélité.

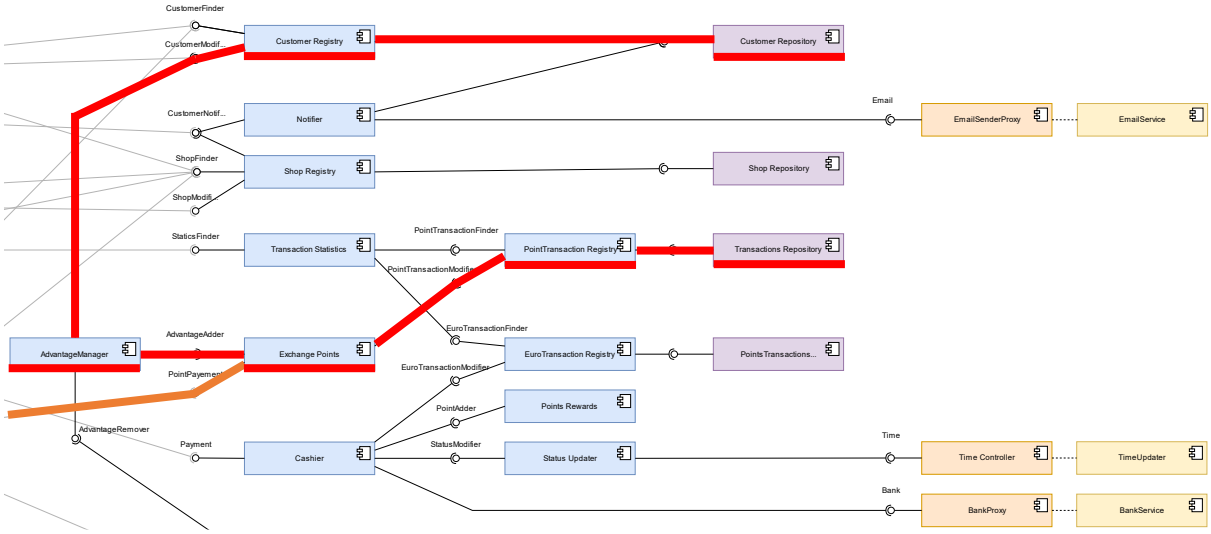
Point rewards

Interface : PointAdder

```
Point gain(Customer customer, Euro amount);
```

Ce composant s'occupe de calculer le nombre de points gagné par le client en fonction de la quantité d'euro dépensée. Il ajoute également les points gagnés au Customer.

Achat d'avantages



AdvantageManager

Interfaces : AdvantageAdder, AdvantageRemover

AdvantageRemover

```
void removeAdvantage(Customer customer, AdvantageItem
item)throws CustomerDoesntHaveAdvantageException;
void checkAdvantagePresence(Customer customer, AdvantageItem
item)throws CustomerDoesntHaveAdvantageException;
```

AdvantageAdder

```
void addAdvantage(Customer customer, AdvantageItem item);
void checkAdvantagePresence(Customer customer, AdvantageItem
item)throws CustomerDoesntHaveAdvantageException;
```

Ce composant a la responsabilité de gérer les avantages d'un Customer. Il permet d'ajouter un avantage avec l'interface AdvantageAdder. De même, il permet d'enlever un avantage avec l'interface AdvantageRemover. Pour rappel, le Customer contient un CustomerBalance comprenant la liste des avantages détenus par le client. Les deux interfaces contiennent aussi la méthode checkAdvantagePresence permettant de vérifier la présence d'un avantage dans la liste des avantages détenus par le client.

Nous avons fait le choix de créer le composant AdvantageManager afin de rassembler au sein d'un même composant toute la logique concernant la gestion des avantages d'un client. Dans le cas contraire, nous aurions réparti la logique au sujet de la gestion des avantages dans ExchangePoints et

dans AdvantageCashier. Ce choix aurait conduit à des problèmes de maintenabilité et de compréhension du code.

Exchange Points

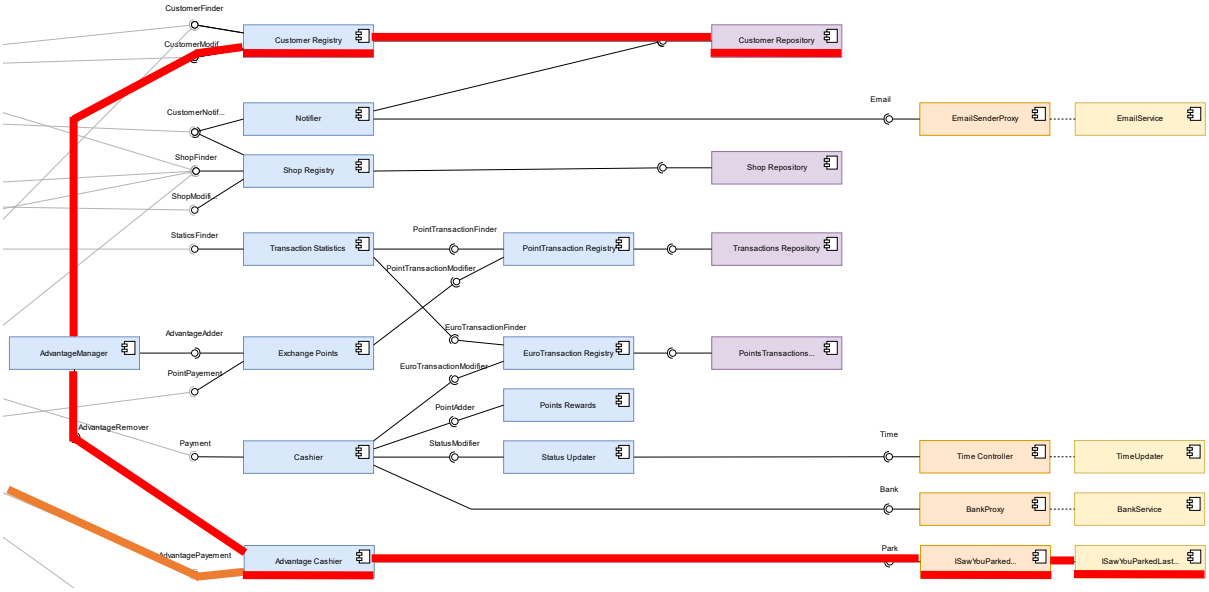
Interface : PointPayment

```
PointTransaction payPoints(Customer customer, AdvantageItem item) throws NegativePointBalanceException;
```

Le composant ExchangePoint est utilisé lors de l'achat d'un avantage par un client. Il a pour responsabilité de débiter du compte du Customer le nombre de points correspondant à l'AdvantageItem, puis utilise l'interface AdvantageAdder du composant AdvantageManager pour ajouter l'avantage au compte du client.

Ce composant utilise également le PointTransactionRegistry pour enregistrer cette transaction, ce qui nous est utile pour calculer les statistiques du système de carte multi-fidélité.

Utilisation d'un avantage



Advantage Cashier

Interface : AdvantagePayment

```
AdvantageTransaction debitAdvantage(Customer customer, AdvantageItem item) throws CustomerDoesntHaveAdvantageException, ParkingException;
```

```
List<AdvantageTransaction> debitAllAdvantage(Customer
customer, List<AdvantageItem> items) throws
CustomerDoesntHaveAdvantageException, ParkingException;
```

Le composant AdvantageCashier a pour responsabilité de contacter tous les services externes en lien avec les avantages, par exemple le service externe de parking.

Afin de vérifier si un certain avantage est détenu par le client, il contacte le composant AdvantageManager. Ensuite, il contacte le service externe correspondant à l'avantage, s'il y en a un. Enfin, AdvantageManager est appelé dans le but de supprimer l'avantage qui vient d'être utilisé.

Status Updater

Interface : StatusModifier

StatusModifier

```
void updateStatus(Customer customer, EuroTransaction
transaction);
```

Le composant StatusUpdater a pour but de mettre à jour, le statut VFP (Very Faithful Person) du client. Le client gagne le statut VFP lorsqu'il effectue deux achats consécutifs en moins de sept jours. Inversement, il le perd si cette condition n'est pas respectée.

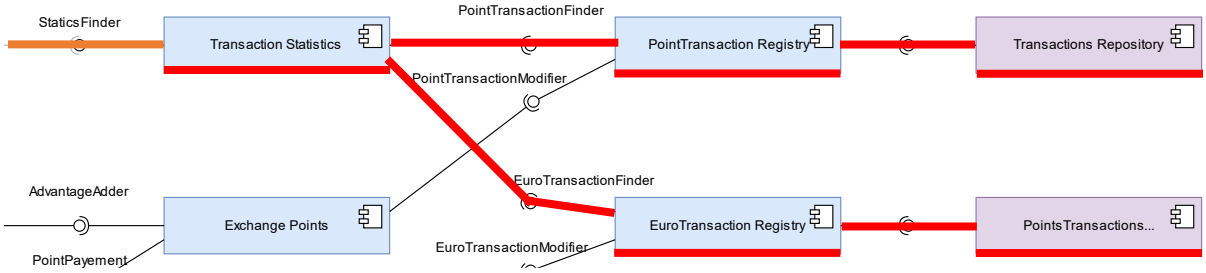
Actuellement, l'ajout et la suppression du statut VFP ont lieu uniquement lorsque le client effectue une nouvelle transaction. Pour l'ajout, cela ne pose aucun problème, car le nouveau statut est obtenu instantanément après la transaction.

En revanche, pour la suppression, un problème survient, puisque le client conserve son statut VFP tant qu'il n'a pas effectué une nouvelle transaction. Ainsi, le client peut acheter un avantage réservé aux VFP alors qu'il aurait dû perdre son statut avant cette transaction.

Pour corriger cela, nous pourrions vérifier le statut du client avant chaque achat en point. Toutefois, cela ne garantirait pas une information à jour pour le client avant d'effectuer une nouvelle transaction.

C'est pourquoi, dès la conception de notre architecture, nous avons prévu d'ajouter un contrôleur connecté à ce composant qui déclencherait la suppression des statuts VFP expirés à intervalles réguliers, par exemple tous les jours à minuit. Nous ne l'avons pas encore implémenté par manque de temps.

Statistiques



Transaction Statistics

Interface : StatsFinder

```
Statistic computeStatsAllShop();
Statistic computeStatsShop(Shop shop);
```

Le composant Transaction Statistics a la responsabilité de calculer les statistiques en lien avec le système de carte de fidélité. Il calcule les statistiques globales du système multi-fidélité ou pour un magasin en particulier.

Ce composant est connecté à deux registres, le EuroTransactionRegistry et le PointTransactionRegistry, cela permet de pouvoir récupérer les informations nécessaires aux calculs des statistiques.

Notifications

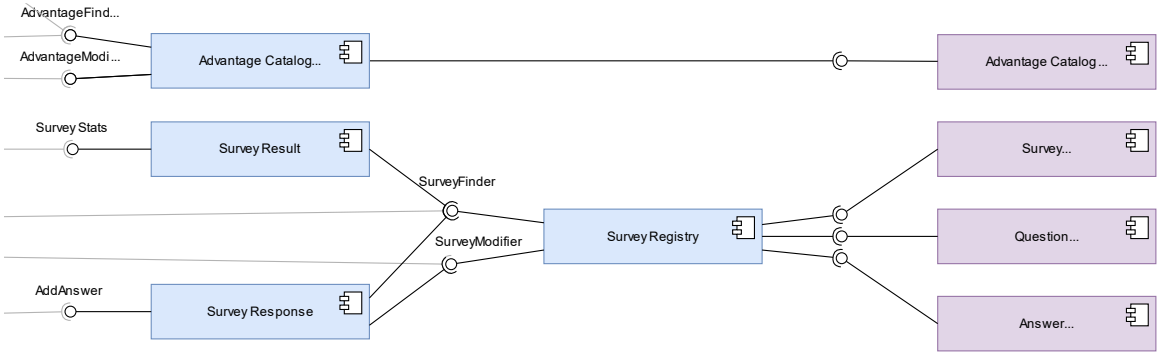
Notifier

Interface : Sender

```
void send(Notification notification);
```

Le composant Notifier permet d'envoyer des messages aux clients. Il est relié à différents services externes (exemple : mail ou sms). La méthode pour envoyer un message est send et elle prend en paramètre une Notification. La classe Notification contient une liste de destinataires (ContactDetails) ainsi qu'un message (NotificationMessage). Nous avons fait le choix d'encapsuler dans un seul objet Notification, l'ensemble de ContactDetails et NotificationMessage. La raison est simple, si plus tard, nous voulons stocker les messages pour les retrouver comme une boite mail, il est plus simple d'avoir un seul objet.

Sondages



Survey Response

Interface : SurveyAddAnswer

```
Survey addAnswer(String answer, Long question, Long surveyID);
```

Ce composant s'occupe de récupérer la réponse d'un client à un sondage. Pour l'instant, nous avons passé des id, mais nous comptons le modifier plus tard avec les classes Question et Answer comme nous l'avons fait dans le service de payement.

Survey Result

Interface : SurveyStats

```
int getNbParticipants(Long surveyID);
```

Ce composant a la responsabilité de récupérer les résultats d'un certain sondage. Pour l'instant, nous renvoyons simplement le nombre de participants ayant répondu au sondage. Comme précédemment, nous comptons changer l'id par un Survey. Nous renvoyons pour l'instant un entier, ce qui est suffisant pour le moment. Dans le cas où nous aurions des statistiques plus complexes, il serait préférable de créer des objets afin de les stocker comme nous l'avons fait pour les statistiques du système de fidélité.

Contrôleurs

Choix de la séparation du Payment Controller et du Advantage Controller

Nous avons décidé d'utiliser 2 contrôleurs distincts, l'un s'occupant des paiements en euro et l'autre des avantages. Nous avons fait ce choix, car nous avons séparé l'action de débiter un avantage de l'action de débiter des euros au niveau des composants du backend. Cela permet également de bien séparer les avantages de la partie paiement en euro.

2. Modèle métier

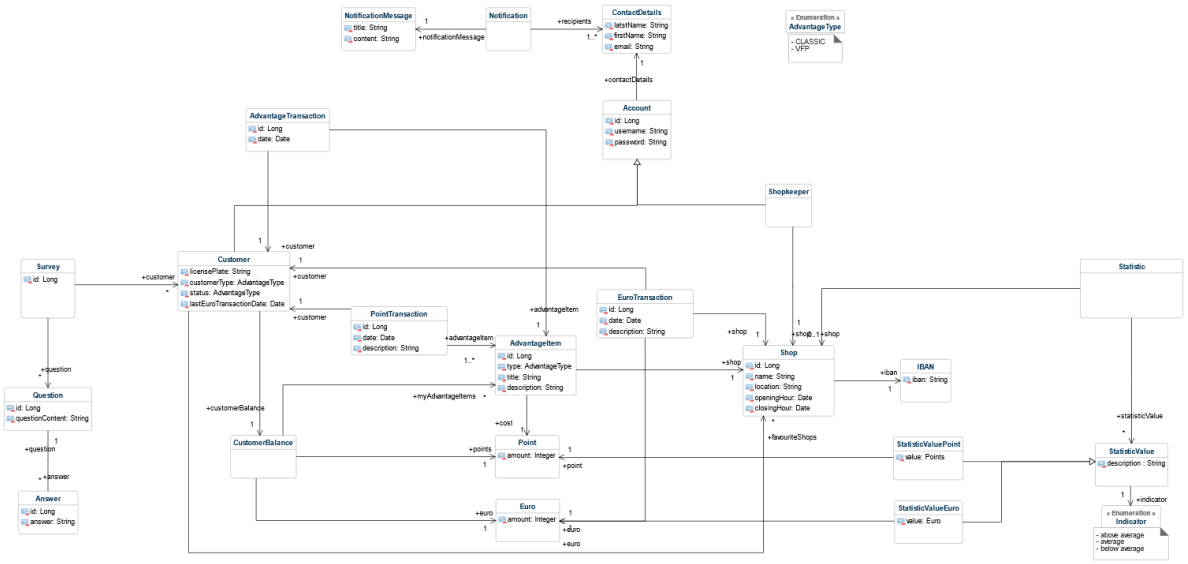


Diagramme au format SVG en ligne

Comptes

ContactDetails

ContactDetails est la classe représentant les informations de contact d'un compte. Elle est utilisée par le service de notification afin de pouvoir envoyer des messages aux utilisateurs. Elle contient le lastName (String), le firstName (String) ainsi que l'adresse mail liée au compte. Dans le cas où nous voudrions ajouter un contact par SMS, nous aurions à rajouter le champ SMS, dans cette classe. Nous avons fait le choix de la séparer de Account car Account modélise l'authentification.

Account

Account est une classe abstraite correspondant à la modélisation d'un compte, contenant les informations utiles pour l'authentification. Cette classe contient un username (String) et un password (String). Il contient également un ContactDetails. Elle est étendue par les classes concrètes Customer et ShopKeeper. Nous avons préféré faire de l'héritage afin d'éviter la duplication de code et de séparer cette partie commune utile à l'authentification, des spécificités des classes filles.

Customer

Customer hérite de Account. Il contient un CustomerBalance qui représente la carte de fidélité du client. Le client a un statut, customerType, qui est une valeur de l'Enum AdvantageType qui est soit VFP, soit CLASSIC. La date de la dernière transaction en euros effectuée par le client, lastEuroTransactionDate (Date), est utilisée pour mettre à jour son statut. Le client possède également une licensePlate (String), qui est utilisée pour l'avantage de parking offert par le système multi-fidélité. Enfin, la classe contient une liste de magasins, correspondant aux magasins préférés du client.

CustomerBalance

Le CustomerBalance contient une quantité de points, une quantité d'euros, ainsi que les avantages détenus par le client. Nous avons séparé CustomerBalance de Customer pour éviter d'avoir

une classe monolithe concentrant trop d'informations disparates. Cela permet de séparer la représentation du Customer et de sa carte de fidélité.

ShopKeeper

ShopKeeper est la représentation d'un commerçant. Elle hérite de Account car elle représente un humain interagissant avec notre système, de plus cette classe est liée à un shop pour indiquer de quel magasin s'occupe ce compte. Cette classe permet de modéliser le commerçant qui a l'autorisation de modifier les informations liées à son magasin.

Magasin

Shop

Shop contient toutes les informations classiques d'un magasin (nom, localisation, horaires d'ouverture et les horaires de fermeture). Shop contient aussi un IBAN permettant au système de payer le magasin lorsque le client utilise de l'argent chargé sur sa carte de fidélité. On suppose que le seul moyen permettant à un magasin d'être rémunéré est par le biais de son IBAN.

Avantage

AdvantageItem

AdvantageItem représente un avantage qu'un client peut acheter avec ses points. Il appartient à un Shop et possède un prix correspondant à montant en Point.

Données de transaction

Toutes les données de transaction qui sont mentionnées ci-dessous contiennent un Customer, ainsi qu'une date correspondant à la réalisation de la transaction.

EuroTransaction

EuroTransaction correspond à une transaction en euro réalisée par un client. Elle contient un shop, le magasin à qui est destiné l'argent.

Cette transaction modélise l'achat en euro d'un produit ou service réel par un client dans le magasin spécifique.

PointTransaction

PointTransaction correspond à une transaction relative à l'achat d'avantages en Point. Elle contient donc une liste d'AdvantageItem. Ainsi, le client possède dans son compte les nouveaux AdvantageItem qu'il pourra ainsi utiliser.

Cette transaction modélise l'achat en point d'un AdvantageItem qui pourra être utilisé dans un magasin spécifique.

AdvantageTransaction

AdvantageTransaction correspond à la transaction liée à l'utilisation d'un avantage par un client. Elle contient un advantageItem. On retire du client l'advantageItem qu'il vient d'utiliser, car il le possède maintenant dans la vraie vie.

Cette transaction modélise l'utilisation d'un AdvantageItem par un client.

Statistiques

Statistic

Statistic est l'objet renvoyé à la CLI. Il contient un shop si la statistique est en rapport à un shop en particulier. Dans le cas où la statistique est globale, elle n'est liée à aucun shop (shop est null). Il contient aussi une liste de StatisticValue.

StatisticValue

StatisticValue est la modélisation d'une donnée statistique. Elle contient une description représentée par une String.

Elle contient aussi un indicateur correspondant à un des éléments de l'Enum Indicator, permettant de donner de l'information sur le positionnement du magasin par rapport aux autres magasins concernant cette métrique.

Cette classe est abstraite et ce sont ces deux classes filles StatisticValuePoint et StatisticValueEuro qui ajoutent la valeur de cette statistique soit en Point, soit en Euro en fonction de la classe.

Notification

Notification

La classe Notification encapsule deux informations. La première est le contenu du message dans un NotificationMessage. La seconde est la destination du message, en possédant une liste de ContactDetails.

NotificationMessage

Un NotificationMessage correspond au message d'une notification. Elle contient un title (String) et un content (String).

Sondage

Survey

Un Survey contient une liste d'utilisateurs ayant répondu au sondage, ainsi que la liste de questions associées.

Pour l'instant, nous avons supposé que chaque client peut répondre à n'importe quel sondage. Il serait possible de rajouter à un survey une liste de personnes éligible à répondre.

Question

Une question contient une liste d'Answer et une String correspond à l'intitulé de la question

Answer

Une réponse contient une String modélisant la réponse. Elle contient également une référence à la question à laquelle elle est associée pour pouvoir la retrouver.

Nous avons choisi de séparer questions et réponses pour pouvoir avoir des questions dont la réponse est libre et pas uniquement des QCM. De plus, cela permet de se rapprocher du modèle réel.

3. Choix d'implémentation de la persistance

Conséquence de la migration à la persistance

Le passage de notre architecture à la persistance n'a pas eu un impact important sur notre modèle métier en ce qui concerne les associations entre les classes.

Nous avons cependant enlevé la relation d'héritage entre la classe Customer et la classe Account. Nous avons fait cela, car, il nous a été conseillé lors du passage à la persistance de ne pas migrer des classes possédant une relation d'héritage. Or, la première classe que nous avons voulue migrer était Customer, puisque c'était la classe qui avait le plus d'importance au niveau métier.

Il ne nous serait pas difficile de remettre cette relation d'héritage, maintenant que nous maitrisons la persistance avec JPA.

Général

Les objets que nous avons rendus persistants sont les comptes des utilisateurs, c'est-à-dire Customer et ShopKeeper. Nous avons aussi les magasins, Shop et leurs avantages, AvantageItem. Nous avons également les PointTransaction et EuroTransaction qui sont persistants que nous utilisons pour calculer les statistiques.

Néanmoins, nous avons décidé de ne pas rendre les AdvantageTransaction persistant. Nous avons fait ce choix, car nous les avons créés non pas dans le but de calculer des statistiques comme pour les autres objets transactions. Mais, dans un but d'informer l'utilisateur que son avantage a bien été utilisé et de lui fournir des informations sur ce dernier.

De même, l'objet Statistic n'est pas stocké, il permet juste d'encapsuler l'information concernant les statistiques qui est ensuite envoyé à l'utilisateur.

Cascading

Nous avons appliqué le « Cascading Delete » aux classes PointTransaction et EuroTransaction sur le champ Customer, stockant le client à l'origine de la transaction. Cela signifie que si un utilisateur est supprimé, toutes ses transactions seront supprimées avec lui. Nous avons pris cette décision pour respecter le RGPD. Cependant, cela peut poser un problème pour le calcul des statistiques, car la suppression d'un utilisateur aura un impact sur les statistiques.

Pour résoudre ce problème, nous pourrions mettre en place un système qui permet de modifier le champ utilisateur contenu dans les transactions lorsqu'un utilisateur est supprimé. Nous pourrions remplacer ce champ par un utilisateur générique, tel que « UserDeleted ». De cette manière, nous pourrions laisser le droit à la suppression des données tout en évitant les calculs erronés des statistiques.

Pour les sondages, nous avons aussi utilisé « Cascading Delete » dans Survey sur le champ questions. De même sur les champs réponses contenu dans Question. Ainsi, lorsque nous supprimons un sondage, ses questions et ses réponses sont supprimées.

Toutefois, nous ne prenons pas encore en compte que la suppression d'un utilisateur entraine la suppression de ses réponses. Pour l'instant, si nous supprimons un utilisateur, ses réponses aux différents sondages ne sont pas supprimées. Néanmoins, cela ne serait pas difficile, il suffirait de rajouter un champ Customer dans Answer avec l'annotation suivante : @OnDelete(action = OnDeleteAction.CASCADE). Cette annotation permet d'indiquer à JPA de supprimer toutes les Answer dans le champ owner correspond au Customer qui est supprimé.

4. Forces et faiblesses de l'architecture

Forces

L'indépendance entre la gestion du débit d'avantages et la gestion du débit des euros, qui nous permet de facilement réaliser ces actions. Pour l'instant, nous faisons deux appels à l'API, lorsque nous voulons débiter un avantage et une somme en euro. Mais cela pourrait très simplement être changé en ajoutant une nouvelle route faisant appel à ces deux composants distincts.

La quantité de composants de notre architecture nous permet d'avoir un bon découpage des responsabilités métiers. Nous sommes particulièrement fiers de notre découpage sur la gestion d'un nouveau paiement en euro, la gestion de l'achat d'un avantage et l'utilisation d'un avantage.

Le découplage entre le sous-système de multi-fidélité et le sous-système de sondage. Ces deux sous-systèmes ne communiquent pas ensemble.

Faiblesses

Le changement du statut d'un client, ne se produit que lors de la réalisation d'une nouvelle transaction par ce même client. Ce qui est un problème puisque le client peut garder le statut VFP pour une durée infinie lorsqu'il ne réalise pas de nouvelles transactions. Il pourrait donc acheter des avantages réserver au VFP alors qu'il ne devrait pas pouvoir le faire. Il faudrait rajouter un service qui fait appel à notre système par un contrôleur qui lance la vérification du statut de tous les utilisateurs. Ce service pourrait être déclenché automatiquement une fois par jour, par exemple.

De plus, la date de la dernière transaction du client est actuellement stockée dans l'objet Customer, ce qui permet de gagner en performance en dupliquant cette information qui est présente dans EuroTransactionRegistry. Cependant, si les conditions d'obtention et de perte du statut VFP venaient à changer, notre solution ne serait plus viable. Dans ce cas, il serait préférable de retirer la date de dernier achat du client et de permettre au composant StatusUpdater de récupérer les informations directement depuis EuroTransactionRegistry. Cela améliorerait l'évolutivité du système.

Enfin, le sous-système de sondage souffre d'un problème d'architecture. Le SurveyRegistry est pour le moment connecté à trois Repository différents, ce qui n'est pas idéal. Il serait préférable que chaque Repository ait son propre Registry dédié. De plus, les interfaces des différents composants du système contiennent des identifiants plutôt que des objets. Ce point devrait être amélioré pour uniformiser l'architecture du système.

5. Répartition des points

Nous sommes tout d'abord fiers du travail que nous avons accompli durant ce projet. Nous avons beaucoup appris que ce soit en ISA et en DevOps. Bien évidemment, certains avaient plus de facilités sur certains domaines que d'autres. Néanmoins, chacun a participé à sa manière dans les différentes tâches du projet. Nous avons donc décidé de répartir équitablement le nombre de points.

NOM Prénom	Nombre de points
BEZES Bastien	100
DEVICTOR Pauline	100
DUBOIS Quentin	100
EL KATEB Sami	100
LATAPIE Florian	100