POLYTECH
NICE SOPHIA

Rapport d'architecture ISA

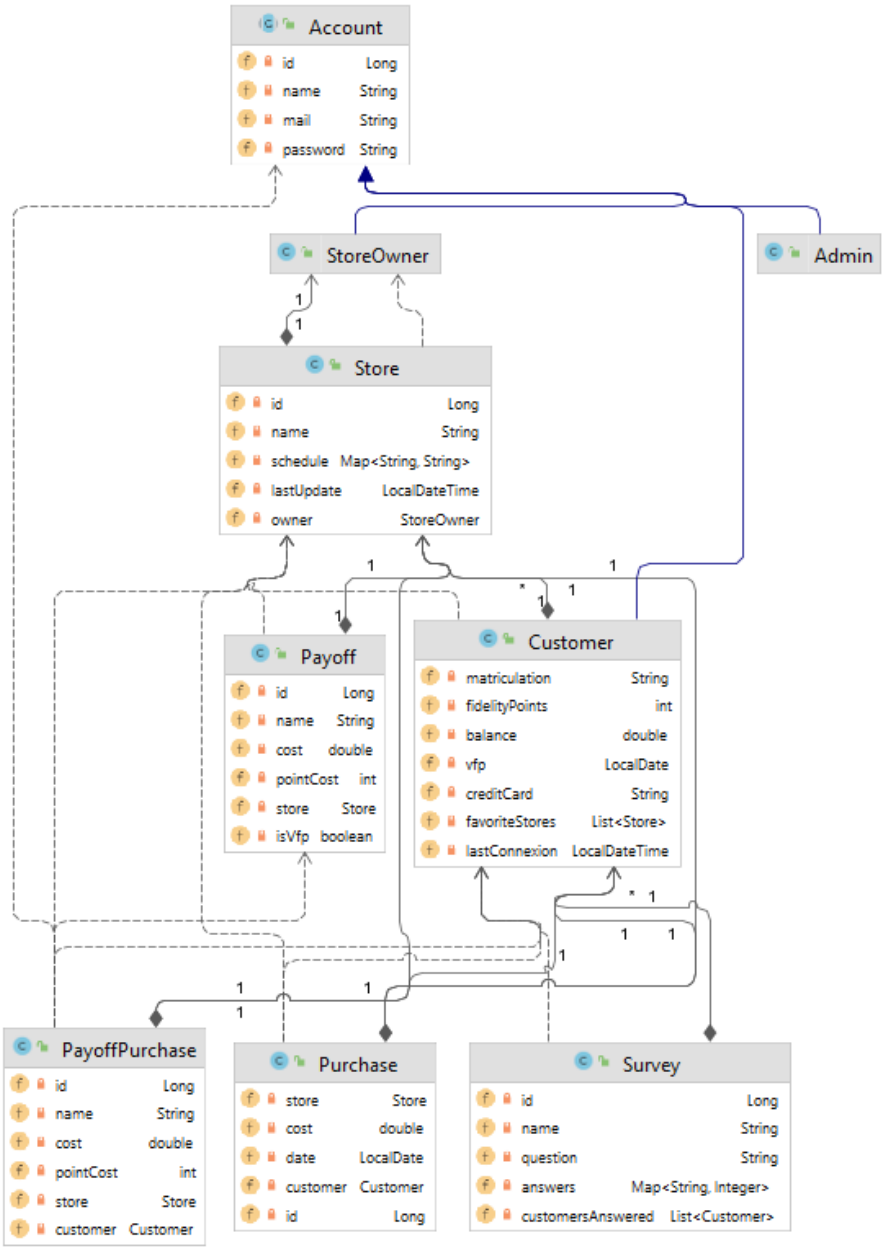
Equipe B

Bevan Tom
Bourdeau Quentin
Correia Ambre
Faucher Vinh
Jeannes Théo

POLYTECH NICE SOPHIA

Sommaire

1. Objets métiers



1. Account

Account est une super classe abstraite qui représente un compte. Elle n'a pas vraiment de valeur métier en tant que telle et permet simplement de centraliser les informations communes à tous les comptes, à savoir le nom, l'email, et le mot de passe. Cette classe n'est pas censée évoluer beaucoup, car les évolutions potentielles du projet seront probablement portées sur ses sous-classes.

2. Admin et StoreOwner

Ces deux classes implémentent la superclasse *Account*. Nous avons décidé de tout de même séparer ces deux classes car malgré leur similarité, car elles représentent deux objets métiers bien distincts, avec des comportements et des permissions très différentes.

3. Customer

Customer implémente également la superclasse *Account*. Elle est plus complète que les autres puisque le *customer* est au centre du projet. Elle contient des informations telles que : le numéro de carte bancaire, le solde actuel du compte, le statut VFP, les points de fidélité, la plaque d'immatriculation et les boutiques favorites du customer.

Nous avons choisi de représenter le statut VFP comme une date, ce qui permet de mettre à jour facilement le statut. Puisque l'attribut *vfp* correspond à la date d'expiration du statut VFP, il suffit de définir l'attribut à une date dans le futur lorsque nous souhaitons donner le statut VFP. A l'inverse, il n'est pas nécessaire de modifier le statut VFP pour le faire expirer, une fois la date dépassée, le client n'apparaîtra plus VFP.

La classe *customer* ne contient cependant pas l'historique de ses achats ou des récompenses qu'il a récupéré car ces informations sont détenues par des objets dédiés, ce qui permet de découper les responsabilités, et donc d'éviter que le *customer* devienne une classe dieu.

4. Store

Le *store* contient toutes les informations liées au magasin, ainsi que son *storeOwner* correspondant. Il comporte un nom de magasin, des horaires et une date de mise à jour des horaires. Cette date de mise à jour peut être comparée avec la dernière connexion d'un *customer* qui aurait mis ce *store* dans ses favoris, afin de savoir s'il est utile de prévenir le client si les horaires du *store* ont changés.

5. Payoff

Les *payoffs* sont les récompenses récupérables par les *customers* en utilisant les points de fidélité. Ils contiennent leurs coûts, en points et en euros, ce qui permet au magasin de savoir ce que lui coûte le programme de fidélité, le *store* qui propose l'offre ainsi que le statut VFP, pour savoir s' il est nécessaire d'être VFP pour accéder à ce *payoff*. La *payoff* représente donc une offre dans le catalogue, et non pas une instance d'achat de l'objet.

6. Purchase

La *purchase* représente une transaction faite “en dehors” du cadre de l’application. c'est-à-dire une transaction en argent réel entre le store et le customer. Pour éviter tout abus nous avons supposé que cette transaction était initiée par le store et non par le *customer*. La transaction contient un *customer*, un *store*, un coût (servant à faire l’attribution de points) et une date (pour pouvoir connaître l’activité d’un *customer* sur une période, et ainsi lui accorder le statut VFP). Elle est stockée en base de données afin de servir d’historique de transactions.

7. PayoffPurchase

La *payoffPurchase* est l’entité représentant une transaction d’un *customer* récupérant une *payoff*. Elle a un nom, un coût en points et en argent réel, ainsi qu’un *store* et qu’un *customer*.

Il y a une duplication d’informations volontaire entre *payoffPurchase* et *payoff*. En effet une référence vers la payoff achetée ne serait pas suffisante pour l’enregistrer avec un *payoffPurchase*, puisque une *payoff* est susceptible d’être modifiée. Une référence impliquerait d’enregistrer tous les états ayant existé à n’importe quel moment de l’application de tous les différents *payoffs*. En supposant que les *stores* changent d’offres régulièrement, font des promotions temporaires sur certaines offres, il serait nécessaire de stocker énormément de *payoffs* différentes pour ne pas fausser les historiques d’achats, alors même que ces *payoffs* ne seraient plus disponibles. Pour cette raison, nous avons choisis une duplication d’informations, afin que le *payoffPurchase* soit créé selon une *Payoff*, sans pour autant être dépendant de cette *Payoff,* ce qui assure que l’historique d’achat reste correct à tout moment.

8. Survey

Le *survey* décrit un sondage simplifié. Il n’y a qu’une question à laquelle nous pouvons répondre par oui ou par non. Il contient une map qui permet de savoir combien de personnes ont répondu une réponse donnée (stockée de manière anonyme). Il contient aussi la liste des gens ayant répondu pour pouvoir notifier les customers qui n’ont pas encore répondu. Cela permet aussi d’empêcher les *customers* de répondre plusieurs fois au même *survey*.

3. Persistance

Tous nos objets métiers sont des entités possédant un @Id auto généré avec @GeneratedValue.

1. Account

L'*account* nécessite un choix d'implémentation pour la persistance, qui est de savoir comment stocker les classes héritées *Customer, Store Owner* et *Admin* en base de données.

La classe *account* n'est pas utilisée pour chercher un objet en base de données, donc nous avons décidé de choisir l'implémentation TABLE_PER_CLASS qui crée une table à part entière pour chaque classe enfant.

En effet, dans notre conception la classe abstraite *account* ne représente pas réellement un objet concret, et permet juste d'éviter la répétition de quelques champs d'informations. Il n'est donc pas incohérent qu'elle n'apparaisse pas dans le schéma de base de données.

2. Customer

La majorité des dépendances du *customer* ont été relayés à d'autres objets, il n'a donc à s'occuper que de ses *stores* favoris grâce à un @ManyToMany, puisqu'un *customer* peut avoir plusieurs *stores* en favoris, et un *store* peut être dans les favoris de plusieurs *customers*.

3. Admin et StoreOwner

Ces deux classes sont restées inchangées avec l'implémentation de la persistance, elles sont simplement convertis en @Entity.

4. Store

Le *store* contient ses horaires, qui sont sous la forme d'une map, avec l'annotation @ElementCollection. Il contient également son *storeOwner*. Un *StoreOwner* peut toutefois avoir plusieurs *stores*. La relation est donc un @ManyToOne. Ceci implique que le *store* doit être supprimé si son *storeOwner* est supprimé, alors que l'inverse n'est pas vrai. Nous avons utilisé l'option @OnDelete permet de respecter cette contrainte.

5. Payoff

Le cas du *payoff* est similaire à celui du *store*, le *store* n'a pas de référence vers ses *payoff*, mais le *payoff* à une référence vers son unique *store*. La solution choisie est donc la même que pour le *store*, c'est-à-dire un @ManyToOne avec l'annotation @OnDelete.

6. Purchase

Toujours dans le même cas, puisque nous avons essayé d'être constant dans les choix d'implémentations, la *purchase* référence à la fois son unique *store*, et son unique *customer* et pas l'inverse. Nous avons donc la même solution à savoir un @ManyToOne et une annotation @OnDelete pour les deux références, puisqu'un *store* et un *customer* peuvent être liés à plusieurs *purchase*.

7. PayoffPurchase

Le *payoffPurchase* possède les mêmes problématiques que les *payoffs* de par leur redondance d'informations. Les références vers le *store* et le *customer* sont donc encore une fois gérées par un @ManyToOne avec @OnDelete.

8. Survey

Cette classe possède deux modèles de données problématiques pour la persistance. Le premier est la map de réponses qui ne contient que des String et des Integers, elle peut donc être gérée avec @ElementCollection. La deuxième est la liste de *customers* ayant répondus au *survey*. Les *customers* peuvent avoir répondu à un ou plusieurs *survey*, et chaque *survey* a obtenu une réponse de potentiellement plusieurs *customers*. C'est donc une association @ManyToMany.

4. Composants

Le diagramme de composant se trouve à la fin du document, en format paysage, pour une meilleure lisibilité.

1. Registries

Les registries sont les composants qui s'occupent notamment des opérations CRUD de l'objet métier auxquels ils sont liés. Leur responsabilité est de modifier et d'enregistrer les objets métiers auxquels ils sont associés.

StoreOwnerRegistry : Il s'occupe uniquement des opérations CRUD du *storeOwner.*

CatalogRegistry : Il s'occupe des opérations CRUD des *payoffs*. Il a également la responsabilité de filtrer les *payoffs* disponibles pour un *customer* spécifique, en vérifiant le statut VFP du *payoff*, le solde courant du *customer* pour vérifier qu'il peut acheter la *payoff*, et de s'il a déjà acheté dans le magasin proposant le *payoff*.

AdminRegistry : Il s'occupe uniquement des opérations CRUD de l'*admin*.

CustomerRegistry : Il s'occupe des opérations CRUD du *customer*. Il permet également de trouver un *customer* selon des filtres précis. De plus, il est appelé pour mettre à jour les informations des *customer*, comme par exemple son statut VFP. C'est un des composants centraux du projets, qui intervient dans de nombreux scénarios. Cependant, sa seule responsabilité est de modifier et sauvegarder les *customers.* Découper cette unique responsabilité n'aurait pas de sens, nous l'avons donc gardé comme un seul composant.

PurchaseRegistry : Il s'occupe des opérations CRUD des *purchase*. Il permet de trouver les listes de *purchases* appartenant à un *store*, ou à un *customer*.

PayOffPurchaseRegistry : Il s'occupe des opérations CRUD des *payoffsPurchase*. Il permet par exemple de rechercher des *payoffsPurchase* selon le *store* auquel ils appartiennent.

SurveyRegistry : Il s'occupe des opérations CRUD des *surveys*. Il permet de trouver tous les *surveys* auxquels un customer spécifique n'a pas encore répondu. Il permet également de modifier un survey, avec la réponse d'un *customer,* par exemple.

2. Handlers

Les handlers sont les composants qui s'occupent majoritairement de la logique métier. Tout comme les registries, cette règle n'est pas absolue et ils peuvent aussi contenir les opérations CRUD liées à leur objet métier s'il y en a un.

Store Handler : Il s'occupe des opérations CRUD des *stores*. Il est notamment responsable de la mise à jour des horaires des *stores*, ce qui doit également modifier la date à laquelle cette modification à été faite. Cette information est ensuite utilisée pour notifier les *customers* du changement d'horaires d'un *store*.

PayOffHandler : Il sert à gérer la transaction liée à la récupération d'un *payoff* par un *customer*. Il est chargé de contacter le *catalogExplorer* pour vérifier la disponibilité du payoff, le *parkingProcessor* dans le cas où le payoff concerne le parking, et le *customerBalancesModifier* pour retirer les points de fidélité consommés par le *customer*.

ParkingHandler : Lorsqu'un client veut réclamer la *payoff* parking, elle est redirigée vers ce composant après avoir été validée. Ce composant a donc la responsabilité de vérifier que le *customer* ait bien une plaque d'immatriculation pour pouvoir accéder à ce service, et de gérer l'application de la récompense, en appelant la méthode du proxy *Parking* pour communiquer avec le service externe ISawWhereYouParkedLastSummer.

PaymentHandler : Il sert à gérer les scénarios où le *customer* veut remettre de l'argent sur sa carte de fidélité. Ce composant gère uniquement la logique des transactions liés au service externe de la banque.

TransactionHandler : Toutes les *purchases* passent par ce composant. Il détient la responsabilité de vérifier la validité de la transaction, et de faire les modifications nécessaires lorsque celle- ci est acceptée. A chaque transaction, les points de fidélité du client, ainsi que son solde s'il a payé via l'application, sont mis à jour. Le composant a également la charge de mettre à jour le statut VFP du client, si cela est nécessaire. Une fois validée, la *purchase* est ensuite envoyée au *purchaseRegistry*.

DataGatherer : Ce composant a la responsabilité de regrouper les informations pertinentes concernant les magasins à l'aide de différents finders, et de les agréger sous forme de tableau de bord.

5. Interfaces

1. les *Finder*

Ces interfaces n'ont qu'un seul rôle, celui de trouver l'objet métier associé selon les critères fournis par les *handler* ou les *registry* qui les utilisent. Elles sont implémentées par les *registry*.

2. les *Modifier*

Toutes les interfaces de notre package *modifier* sont implémentées par nos composants *Registry*.

Les interfaces *Registration* sont utilisées pour inscrire et supprimer un utilisateur ayant besoin d'un compte et sujet à modification.

Les interfaces *Recording* sont utilisées pour enregistrer des objets métiers non modifiables, donc notamment les transactions ayant eu lieu.

Les interfaces *Modifier* regroupent les différentes opérations de modifications possibles sur nos objets métier modifiables, comme par exemple, modifier le statut VFP d'un client ou les horaires d'un *store*.

3. Autre

Bank : Implémentée par *BankProxy*, elle permet de communiquer avec le service externe de la banque.

Parking : Implémentée par *ParkingProxy*, elle permet de communiquer avec le service externe ISawWhereYouParkedLastSummer.

ParkingProcessor : Implémentée par le *ParkingHandler*, elle permet de s'assurer que les conditions spécifiques pour la *payoff parking* sont remplies, avant d'utiliser le *ParkingProxy*.

Payment : Implémentée par *PaymentHandler*, cette interface permet de recharger la carte bancaire virtuelle d'un *customer*.

PayoffProcessor : Implémentée par *PayoffHandler*, elle permet de vérifier les conditions pour recevoir une *payoff* et de l'obtenir dans le cas échéant.

StoreDataGathering : Implémentée par *DataGatherer*, elle permet de rassembler les informations pertinentes pour un *storeOwner* et de les retourner sous forme de *dashboard*.

TransactionProcessor : Implémentée par *TransactionHandler*, elle permet la validation des transactions, payées ou non via l'application.

6. Forces, Faiblesses et Capacités d'évolution

Une des forces de notre architecture est son découpage fin. Cela nous a certainement pris un peu plus de temps lors de l'étape de conception, et la mise en place initiale au démarrage du projet a demandé un peu plus de travail, mais cela nous a permis d'éviter l'implémentation d'une super-classe qui aurait eu trop de responsabilité. Ce découpage, qui nous semble pertinent, nous a également permis de définir clairement la responsabilité de chaque composant, rendant les modifications et ajouts de fonctionnalités plus faciles à implémenter. Cela présente également l'avantage d'avoir une architecture facile à faire évoluer, puisque que le couplage entre les composants est limité.

Par exemple, lors de l'un de nos refactor, l'ajout du *storeOwnerController*, a été grandement simplifié par notre architecture. De même, lorsque nous avons modifié le système de notification, cela n'a pas eu beaucoup d'impact sur l'ensemble de notre projet. Nous pensons que notre architecture a donc été bénéfique en nous évitant les répercussions majeures et en cascade lors de changements majeurs dans l'architecture.

Le nombre important d'interfaces et de composants pourrait être vu comme une faiblesse au premier abord, dans l'hypothèse ou un collaborateur rejoindrait le projet en cours d'implémentation. Mais notre catégorisation des composant en *Registry* ou *Handler*, ainsi que la classification de nos interfaces entre *Finder*, *Registration*, *Recording*, *Modifier*, *Processor*, ou simplement associé à une utilité précise (comme l'interface *Bank*) aide à comprendre la structure que nous avons choisi pour notre implémentation du projet.

La gestion des erreurs au niveau de l'api REST est gérée de façon globale par un Global Controller Advice, ce qui permet de facilement relier chaque erreur à un code d'erreur, et rend la distribution des exceptions plus facilement maintenable.

Une de nos faiblesses est la gestion des autorisations. En effet, un *customer* n'as pas le droit de faire les mêmes actions qu'un *storeOwner* ou qu'un *admin*. Actuellement, cette notion est très binaire, l'utilisateur connecté est un *customer* ou non, un *storeOwner* ou non. Il n'y a pas vraiment de notion d'une échelle de droit, ou de nuance. Nous pensons que cette partie serait à améliorer pour une potentielle évolution du projet, avec un véritable système d'autorisation. Notamment car les *storeOwner* et les *admin* ont certains droit en commun, comme modifier le catalogue. Nous pensons qu'implémenter également un type de compte pour les employés de magasin serait pertinent, car ils pourraient avoir besoin d'utiliser certaines fonctionnalités, sans pour autant être des *storeOwner*.

La gestion de nos sondages pourrait grandement être améliorée. D'une part, différentes fonctionnalités sont manquantes pour gérer complètement un système de sondage. Il faudrait être capable de faire des sondages avec plusieurs questions, permettre aux utilisateurs de répondre partiellement à un sondage pour le revendre plus tard, laisser la possibilité à l'*admin* de modifier ou d'ajouter des réponses à une question. Toutes ces actions impliquent de créer de nouveaux objets, comme un objet Question par exemple, mais également l'apparition de nouveaux composants.

Notre système de notifications est également améliorable. En effet, les notifications sont seulement stockées dans les *controllers*, de façon temporaire. Une amélioration serait de gérer différemment ces notifications, en créant un composant dédié à leur gestion par exemple.

7. Répartition des points

Tom	Quentin	Ambre	Vinh	Theo
96	101	101	96	106

Diagramme de Composants