ISA/DevOps

Carte multi-fidélités

Équipe H

(comme hibou)

Buquet Antoine

-

Karrakchou Mourad

-

Imami Ayoub

-

Bonnet Kilian

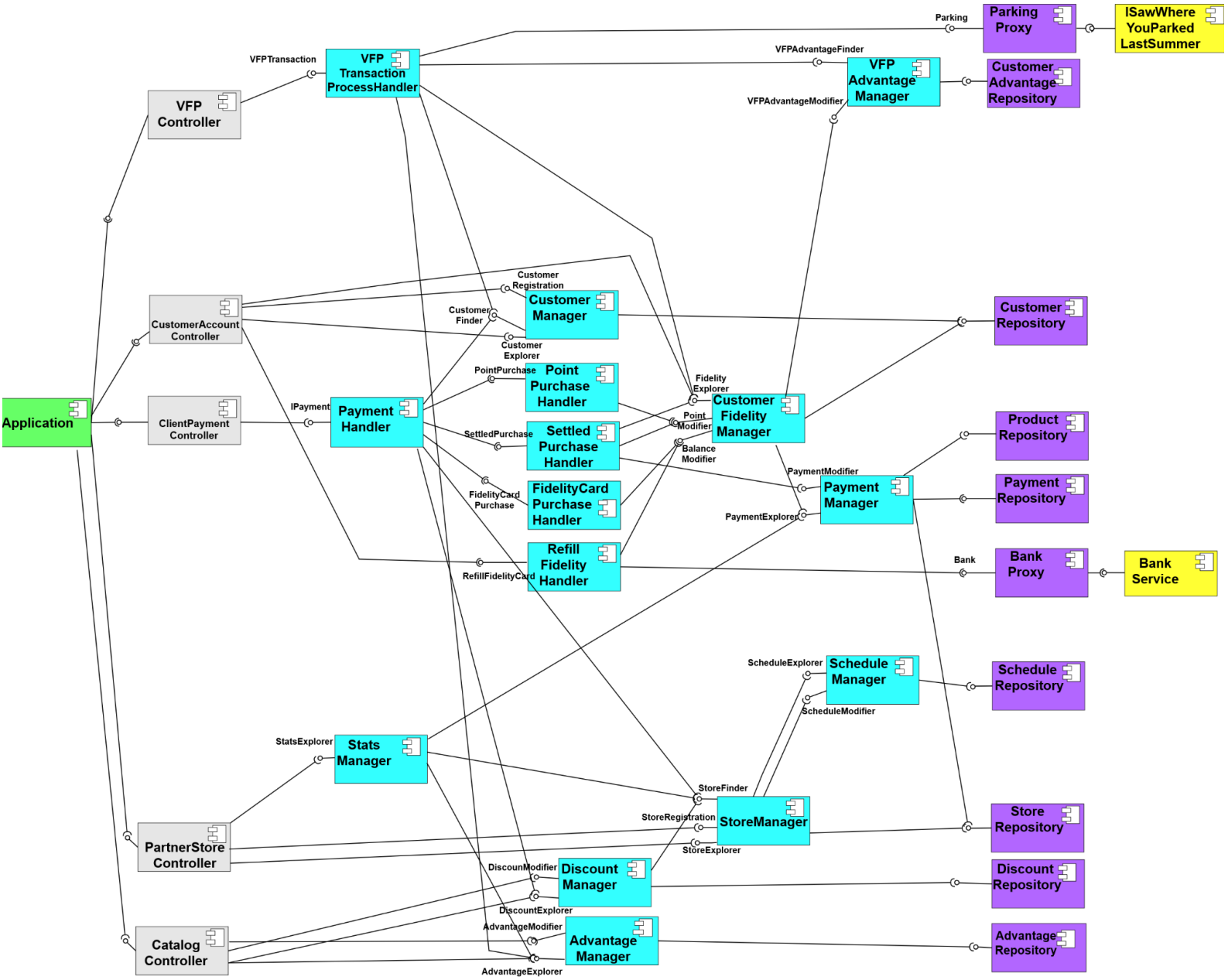
-

Le Bihan Léo

Buquet Antoine - Karrakchou Mourad - Imami Ayoub - Bonnet Kilian - Le Bihan Léo

Buquet Antoine - Karrakchou Mourad - Imami Ayoub - Bonnet Kilian - Le Bihan Léo

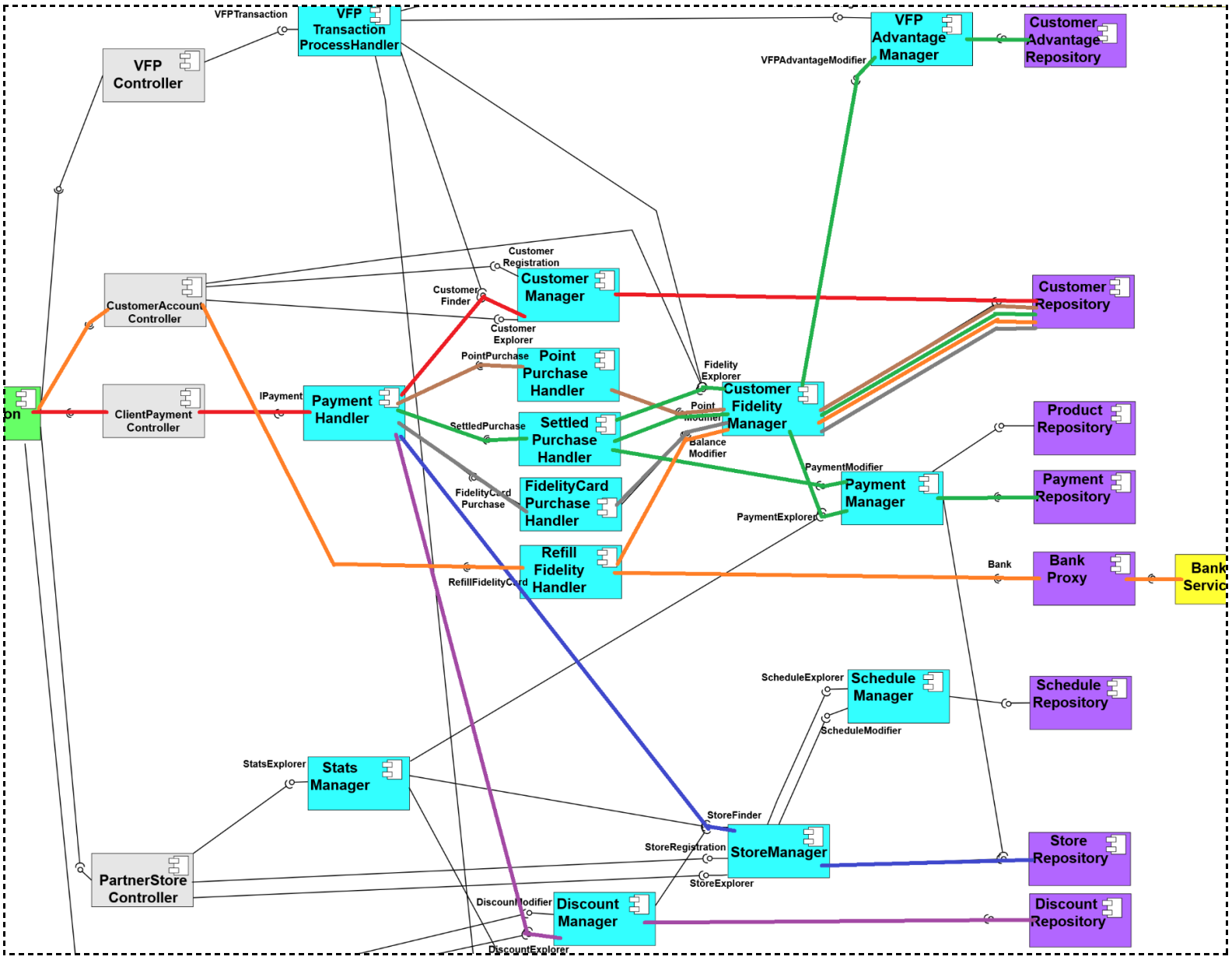
I - Diagramme de composants



1 - Interfaces métier

Pour une meilleure lecture et compréhension du rapport, les **Composants** sont écrits en gras et les *Interfaces* sont écrites en italique.

Paiements



Le système de paiement correspond à la partie centrale de notre application. C'est pour cela qu'il s'agit également de la fonctionnalité la plus développée et complexe de notre système. Il est possible de payer de deux façons différentes et chacune d'elles va être clairement expliquée ci-dessous en s'appuyant sur les différents chemins de couleur qui sont représentés sur le diagramme ci-dessus.

Paiement réglé en magasin : c'est le moyen de paiement le plus basique et le plus souvent utilisé.

Chemin rouge : Une requête contenant l'id du customer, l'id du store ainsi qu'un set d'Item (un Item contient une quantité et un objet Buyable qui peut être un Product ou un Discount cf. diagramme du modèle métier) est interceptée par le **ClientPaymentController.** Ce dernier transmet les informations au composant **PaymentHandler** à travers son interface *IPayment* qui fournit deux méthodes différentes en fonction du moyen de paiement. Le composant **PaymentHandler** va en premier lieu utiliser l'interface *CustomerFinder* afin de récupérer l'objet Customer depuis le **CustomerRepository**.

Chemin bleu : Une fois le Customer récupéré, il faut faire de même avec l'id du Store reçu en paramètre. Le **PaymentHandler** fait donc appel à l'interface *StoreFinder* pour récupérer le Store depuis le **StoreRepository**.

Chemin violet (optionnel) : Le **PaymentHandler** vérifie également s'il y a un objet Discount parmis la shopping list (le set d'Item passé en paramètre), le cas échéant, il doit s'assurer à l'aide de l'id de la Discount de l'Item que la Discount existe bel et bien dans la base de données du système. Pour cela il doit communiquer avec l'interface *DiscountExplorer* du **DiscountManager**.

Chemin marron (optionnel) : Une fois que la vérification est faite, **PaymentHandler** utilise la seule méthode buyWithPoint fourni par l'interface *PointPurchase*. Cette méthode prend en paramètre l'objet Customer précédemment récupéré du repository et le set d'Item. Le composant **PointPurchaseHandler** calcule la somme du prix en points de tous les objets Discount présents dans le set d'Item, puis il vérifie que le nombre de points sur le compte fidélité du Customer est supérieur au nombre de points requis pour pouvoir profiter de la Discount. Si tout est correct alors **PointPurchaseHandler** envoie le nombre de points à soustraire au compte de fidélité ainsi que le Customer, à la méthode decrementPoints de l'interface *PointModifier* qui va s'occuper d'effectuer le changement avant de le sauvegarder dans le **CustomerRepository**.

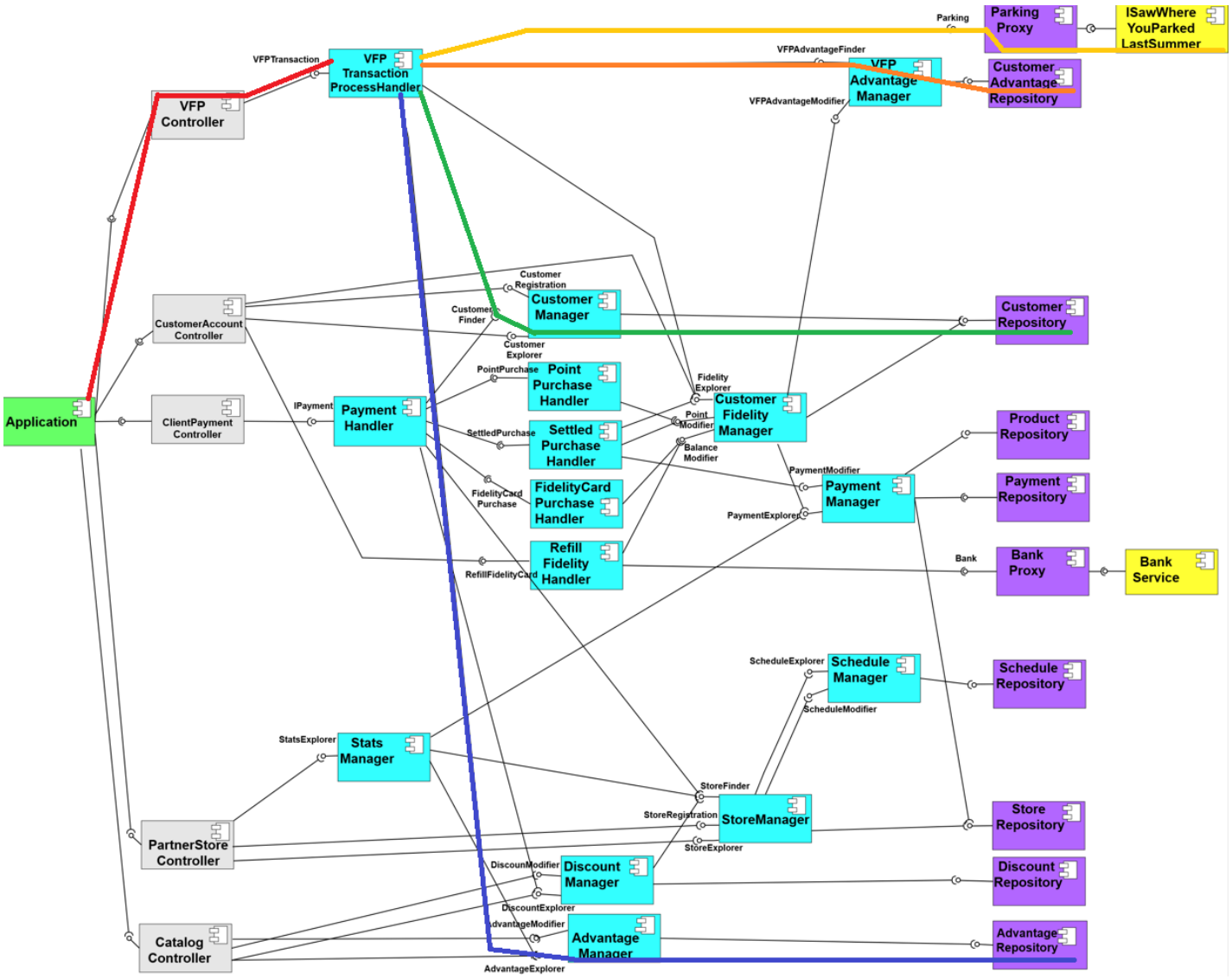
Chemin vert : Après que l'achat des Discount ait été géré (si nécessaire), **PaymentHandler** doit créer l'objet Payment et le faire valider à travers l'interface *SettledPayment.* Cette interface est en charge de mettre à jour sur le compte fidélité du client les points qu'il vient d'acquérir en faisant son achat. **SettledPurchaseHandler** utilise donc la seconde méthode fournie par l'interface *PointModifier* qui est incrementPoints. Le Customer est alors mis à jour et sauvegardé dans le repository correspondant. Il faut également sauvegarder l'objet Payment en passant par l'interface *PaymentModifier* implémentée par le composant **PaymentManager**. Enfin, il ne reste plus qu'à vérifier si le Customer est éligible à devenir VFP. Pour cela **SettledPurchaseHandler** utilise une des méthodes de l'interface *FidelityExplorer* qui, à partir du Customer passé en paramètre et de l'interface *PaymentExplorer,* vérifie si le Customer a atteint le palier de dix paiements. Si c'est le cas, alors le **CustomerFidelityManager** transmet le Customer au **VFPAdvantageManager** via l'interface *VPFAdvantageModifier* qui va enregistrer un nouvel objet CustomerAdvantage dans le **CustomerAdvantageRepository**.

Paiement par carte de fidélité : il est également possible de déposer de l'argent sur le solde de son compté fidélité. Cela permet de pouvoir effectuer des achats sans payer par carte bancaire par exemple.

Chemin orange : Avant de pouvoir faire un achat avec ce moyen de paiement, il est obligatoire d'avoir au préalable mis de l'argent sur son compte fidélité. Pour cela, c'est le **CustomeAccountController** qui reçoit une requête avec les informations nécessaires tels que l'id du Customer, sa carte de crédit ainsi que le montant à créditer sur son compte fidélité. C'est ensuite au travers de l'interface *RefillFidelityCard* du **RefillFidelityHandler**, qui va communiquer à son tour avec l'interface *Bank* du **Bank proxy**, que la connexion jusqu'au service externe de la banque est effectuée. Une fois la transaction validée par le service externe de la banque, **RefillFidelityHandler** ajoute les fonds au solde du compte fidélité grâce à l'interface *BalanceModifier*.

Chemin gris : Ce chemin s'insère dans la procédure normale d'un paiement décrit précédemment, entre le chemin bleu (ou marron s'il y a au moins une Discount dans la shopping list) et le chemin vert. Il s'agit de l'étape supplémentaire à effectuer lorsque le moyen de paiement choisi est la carte de fidélité. Il faut alors communiquer avec l'interface *FidelityCardPurchase* qui utilise le Customer et la shopping list afin de calculer le montant total des objets Product présents dans la shopping list. Pour terminer l'opération il faut décrémenter le solde présent sur le compte fidélité du Customer grâce à l'interface *BalanceModifier*.

VFP - Very Faithful Person



Pour récompenser certains clients, un système d'avantages est mis en place, il s'agit du statut de VFP, un statut obtenu après un certain nombre d'achats

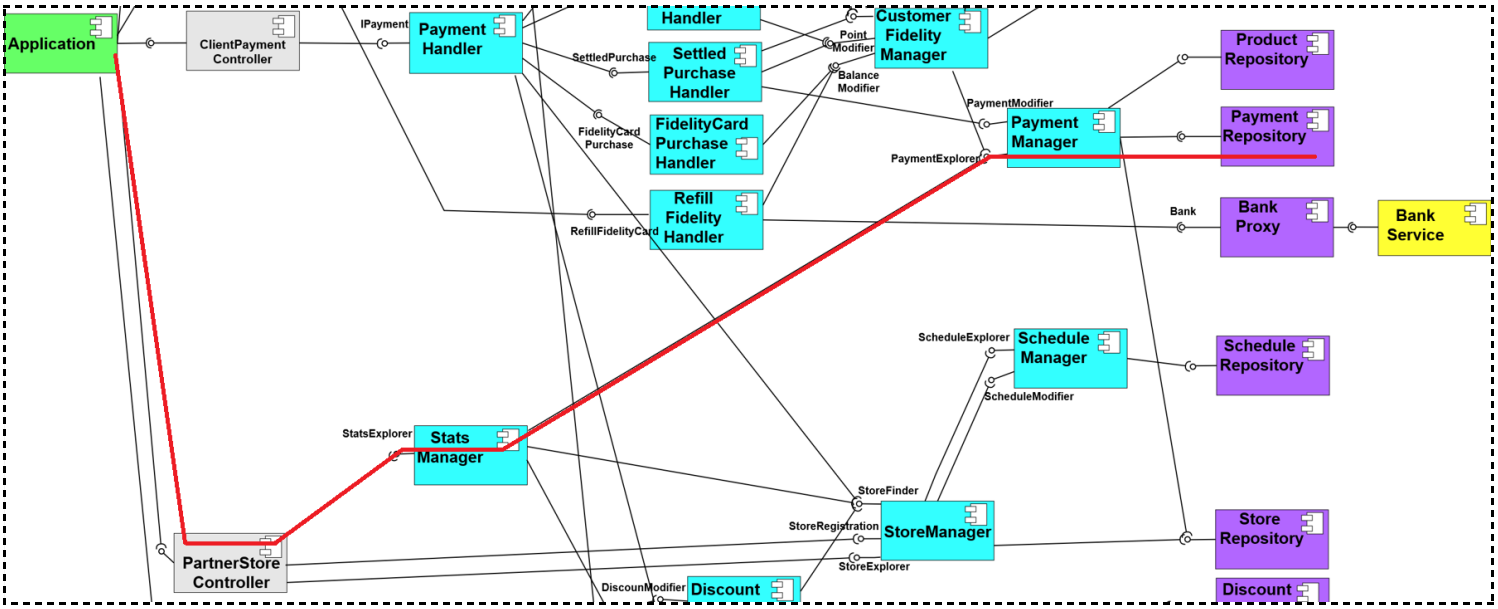
En rouge, l'**Application** fait des appels au backend en passant par le **VFPController** qui utilise l'interface *VFPTransaction* afin de communiquer avec le **VFPTransactionProcessHandle.** Ce dernier gère l'utilisation des avantages, dont notamment l'avantage parking.
Pour utiliser un avantage, le **VFPTransactionProcessHandler** va dans un premier temps, effectuer quelques vérifications. Pour cela, il utilise les interfaces suivante :

- **En vert**, *CustomerFinder* pour communiquer avec le **CustomerManager** qui, quant à lui, communique avec le **CustomerRepository** afin de trouver le client bénéficiant de l'avantage.
- **En bleu**, *AdvantageExplorer* pour communiquer avec l'**AdvantageManager** qui, à son tour, communique avec l'**AdvantageRepository** afin de vérifier si l'avantage existe.
- **En orange**, *VFPAdvantageFinder* pour communiquer avec le composant **VFPAdvantageManager** qui, enfin, utilise le **CustomerAdvantageRepository** afin de vérifier si le client est bénéficiaire du statut.
- **En jaune**, *Parking*, dans le cas d'un avantage de parking, pour communiquer avec le **ParkingProxy.** Ce composant communique avec le service externe ISawWhereYouParkedLastSummer afin de vérifier s'il reste des places disponibles dans le parking souhaité.

Enfin, le **VFPTransactionProcessHandler** communique à nouveau avec le **VFPAdvantageManager** à travers l'interface *VFPAdvantageFinder*. Ce dernier a la responsabilité de vérifier si la personne a déjà utilisé l'avantage. Le cas échéant, si la date de la dernière utilisation est supérieure à 24 heures, alors on met à jour le **CustomerAdvantageRepository** pour ajouter l'avantage qu'on vient de consommer à la liste des avantages consommés.

Cette architecture suit une structure en couches qui permet une bonne organisation des différentes responsabilités de l'application. Les différents composants ont chacun leur propre responsabilité. Nous pouvons ainsi effectuer différentes vérifications, adaptées à différents avantages. De plus, cette architecture nous permet d'implémenter très facilement de nouveaux avantages liés, ou non, à des services externes.

Statistiques

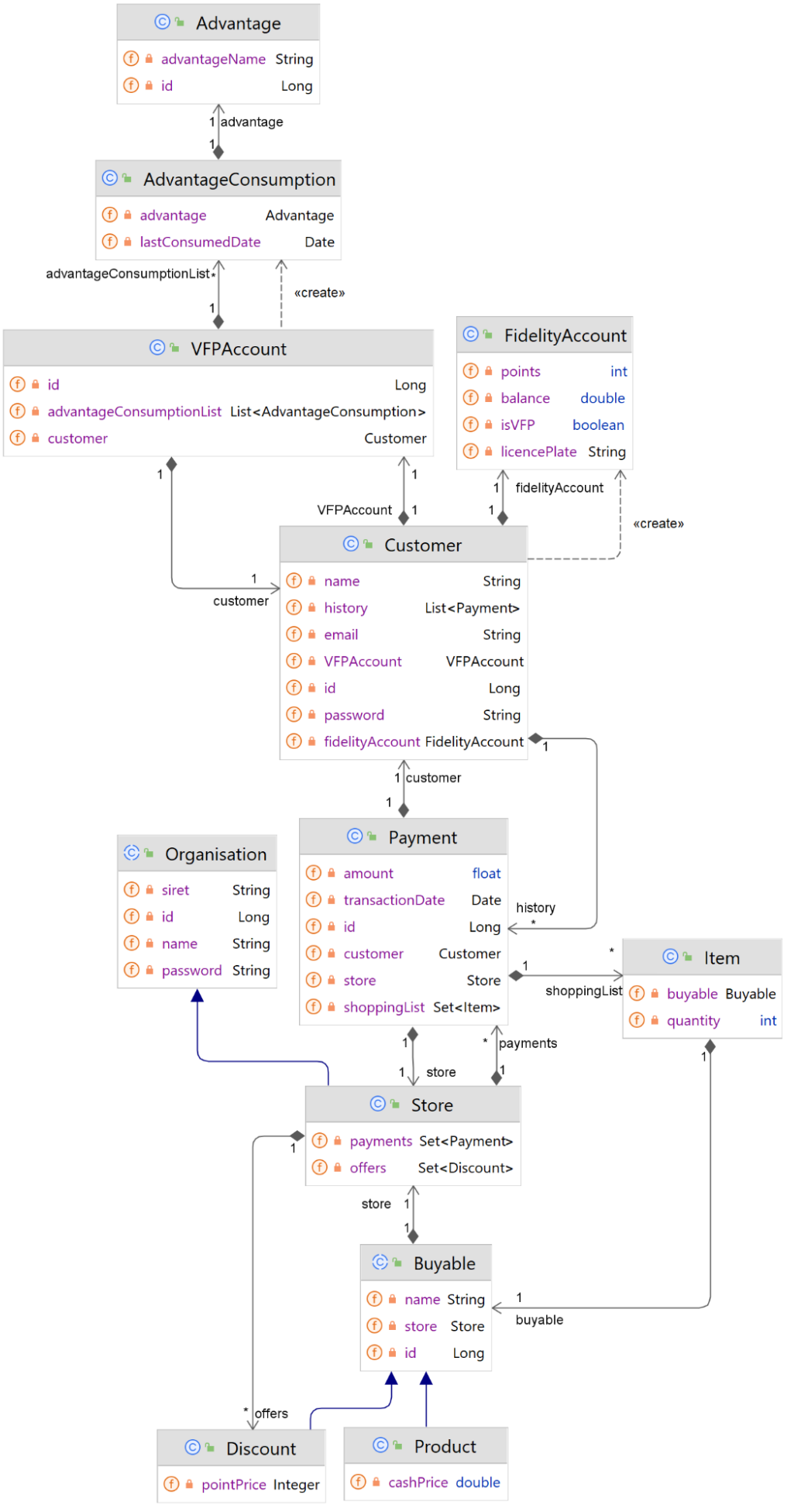


Dans le cadre de ce système de carte multi-fidélités, il convient d'obtenir des indicateurs chiffrés montrant le coût des avantages offerts aux clients, comparé au volume de ventes généré par ces clients.

Pour cela, l'**Application** fait appel au backend en passant par le **PartnerStoreController** qui utilise l'interface *StatsExplorer* afin d'accéder au composant **StatsManager**. Ce composant à la responsabilité de calculer les statistiques à partir de données comme des paiements. De ce fait, il utilise l'interface *PaymentExplorer* pour communiquer avec le composant **PaymentManager** dont la responsabilité, ici, est de retrouver des paiements dans le **PaymentRepository**. Ainsi le **StatsManager** peut calculer les indicateurs cités précédemment.

Ces indicateurs sont utiles pour connaître la rentabilité des avantages et surveiller s'il n'y a pas une sur-évaluation ou une sous-évaluation des avantages. De plus, grâce à cette architecture, il est facile d'implémenter de nouvelles statistiques, en effet, le **StatsManager** utilise aussi les interfaces *StoreFinder* et *AdvantageExplorer,* lui permettant de calculer plusieurs statistiques relatives aux magasins et aux réductions.
Cette implémentation permet donc de séparer les responsabilités et de transmettre les données utiles, facilitant ainsi l'ajout de nouvelles statistiques mais aussi de nouveaux composants qui permettraient d'agrandir le champ des possibles.

2 - Diagramme de classes du modèle métier



Au cours de l'avancement du projet, nous avons apporté plusieurs modifications significatives à notre diagramme de classes du modèle métier. Nous avons décidé de ne pas implémenter certains concepts tels que Schedule, TerritorialCommunity et TownHall, mais notre architecture permet d'ajouter facilement ces éléments si nécessaire. Nous avons également supprimé certaines entités comme BankTransaction, Employee et Role car elles étaient soit inutiles, soit hors sujet pour notre système. Nous avons dû également faire évoluer notre modèle métier en raison d'une mauvaise conception initiale de certaines entités. Les avantages par exemple, étaient initialement associés à des dates via une HashMap pour gérer le FidelityAccount, mais nous avons dû revoir complètement la manière dont notre application prenait en compte les avantages afin de retirer ces mauvaises pratiques. C'est pour cela que nous avons créé les entités VFPAccount, AdvantageConsumption et Advantage. Enfin, le dernier changement majeur se trouve au niveau des Item / Product / Discount. Auparavant, un Item avait un Product et une Discount était simplement une sous-classe de Product. Nous avons choisi de clairement séparer les Discount des Product en introduisant une classe mère commune Buyable aux Product et Discount. Discount n'est donc plus le nom le plus adapté, il s'agit plus d'une offre plutôt qu'une promotion.

3 - Choix d'implémentation de persistance

Pour une meilleure lecture et compréhension du rapport, les **Entités** sont écrites en gras et les *Relations* sont écrites en italique.

Les **Customer** sont rendus persistant et possèdent une relation *@OneToOne* en cascade avec leur **VFPAccount** et *@Embedded* pour leur **FidelityAccount**. De cette façon, la cohérence des données est assurée, un **Customer** dispose d'un seul et unique **VFPAccount** par exemple. Enfin, la relation en cascade garantit que le **VFPAccount** soit effacé en même temps que le Customer.

Les **Payment** sont également des entités importantes dans notre système et se doivent d'être persistées. Ils ont une relation *@ManyToOne* avec les **Customer** car chaque paiement est relié à un seul client mais également avec les **Store** qui peuvent avoir besoin de connaître tous les paiements qui leur sont associés. Enfin un Paiement contient également une *@ElementCollection* pour son attribut shoppingList qui est un Set d'Item, de cette manière, la liste d'éléments est stockée comme une collection d'éléments intégrés dans la table Payment.

Buyable représente un achat potentiel dans un magasin. Les instances de cette classe seront stockées dans une table unique de la base de données avec une colonne discriminante nommée "buyable_type". La classe abstraite est définie avec l'annotation @Inheritance(strategy = InheritanceType.SINGLE_TABLE) pour indiquer que la hiérarchie des classes qui héritent de cette classe sera stockée dans une seule table. Les classes **Discount** et **Product** héritent de cette classe.

Nous avons choisi cette implémentation de persistance car nous devons garantir que les données soient disponibles pour les utilisateurs à tout moment. Cette implémentation permet également la prise en compte de l'historique des données comme les paiements et facilite la création de rapports statistiques, utiles aux commerçants surveillant les différents indicateurs.

De plus, certaines opérations comme le contrôle du statut VFP par exemple, passent par plusieurs étapes de vérifications, il est donc important d'utiliser des transactions, qui sont prises en charge par la persistance, afin de garantir l'intégrité des données.

II - Architecture

1 - Forces

L'une des principales forces de cette architecture est sa flexibilité, qui permet une évolution facile et rapide. En effet, la séparation des responsabilités entre les différents composants de l'architecture permet une maintenance et une évolution plus facile de chacun d'entre eux indépendamment, sans affecter les autres parties du système. Cela permet d'ajouter ou de supprimer des fonctionnalités sans avoir à repenser l'ensemble de l'architecture.

Une autre force de cette architecture est sa modularité, qui permet de facilement intégrer de nouveaux composants ou de remplacer des composants existants. Cela permet également de facilement ajouter de nouvelles fonctionnalités sans affecter l'ensemble du système.

Enfin, cette architecture est conçue pour être facilement scalable, elle peut facilement supporter une augmentation du nombre d'utilisateurs, et peut ainsi s’adapter à différentes zones géographiques, ou supporter une augmentation de données traitées, que ce soit pour la gestion des points, des réductions, des avantages ou encore des indicateurs permettant de réaliser des statistiques.

2 - Faiblesses

Bien que cette architecture présente de nombreux avantages, elle peut également présenter certaines faiblesses qui doivent être prises en compte.

La séparation des responsabilités peut entraîner une complexité accrue de l'architecture, car elle nécessite une coordination étroite entre les différents composants pour assurer une communication et une intégration efficaces. Si cette coordination n'est pas gérée correctement, cela peut entraîner des erreurs.

De plus, dans notre architecture initiale, nous utilisions des identifiants (id) pour la communication des données entre les composants, ce qui nous obligeait à utiliser des structures de données telles que les HashMaps. Cependant, l'un des avantages de l'utilisation de Spring est que cela évite l'utilisation de ces structures. Par conséquent, nous avons dû faire évoluer notre architecture et notre modèle métier pour ne plus avoir les identifiants lorsqu’il est nécessaire d’utiliser des objets afin de bénéficier des avantages offerts par l'utilisation de Spring.

Notre système de paiement utilise plusieurs composants qui vont communiquer afin de traiter le paiement par points, par carte de fidélité ou un achat déjà réglé. Précédemment chaque étape majeure allait sauvegarder l'état de l'utilisateur dans la base de données. Cela a pour avantage de ne pas rendre dépendant les composants entre eux, cependant cela crée une grosse faiblesse: Si une étape échoue au milieu du processus de paiement certaines informations de l'utilisateur auront déjà été mises à jour. Nous avons résolu ce problème en ajoutant l'annotation *@Transactional* au système de paiement. De ce fait, si une erreur se produit au cours de la phase de paiement, aucun changement n'aura été ajouté à la base de données.

3 - Capacité d'évolution vis-à-vis du sujet

Notre architecture a été conçue avec une grande flexibilité pour permettre une évolution facile au fil du temps. En outre, nous avons pris en compte les aspects métier pour nous assurer que l'architecture répond aux besoins et aux contraintes de la carte multi fidélité, tout en étant capable d'évoluer. Par conséquent, nous sommes en mesure de facilement ajouter ou supprimer des fonctionnalités en fonction du sujet. Notre architecture nous offrirait par exemple la possibilité d'intégrer de nouveaux services externes. Mais aussi de nouveaux types d'organisations qui ajouteraient des avantages au système de fidélité. Ces ajouts ne nécessitent pas un refactor complet du système.

III - Répartition des points

Buquet Antoine	**100**
Karrakchou Mourad	**100**
Imami Ayoub	**100**
Bonnet Kilian	**100**
Le Bihan Léo	**100**