Rapport Architecture Logicielle #2

CastexSki

Semestre 8 - 2021

Équipe C

BERTOLOTTO Loïc
FACQ Antoine
LECAVELIER Maëva
MAZURIER Alexandre
POUEYTO Clément

Université Côte d'Azur
Polytech Nice Sophia
SI4 - ISA

Table des matières

Diagramme cas d'utilisation 3
Le client 3
L'employé de la station 3
Diagramme de classes 4
Persistance 5
Diagramme de composants 6
Interfaces 7
Gestion des cartes 7
Gestion des paniers 8
Gestion du paiement 8
Gestion des clients 8
Gestion de la station 9
Gestion des écrans d'affichage 9
Gestion des portiques de sécurité 10
Gestion des statistiques 10
Evolution 11
Conclusion 11

Diagramme cas d'utilisation

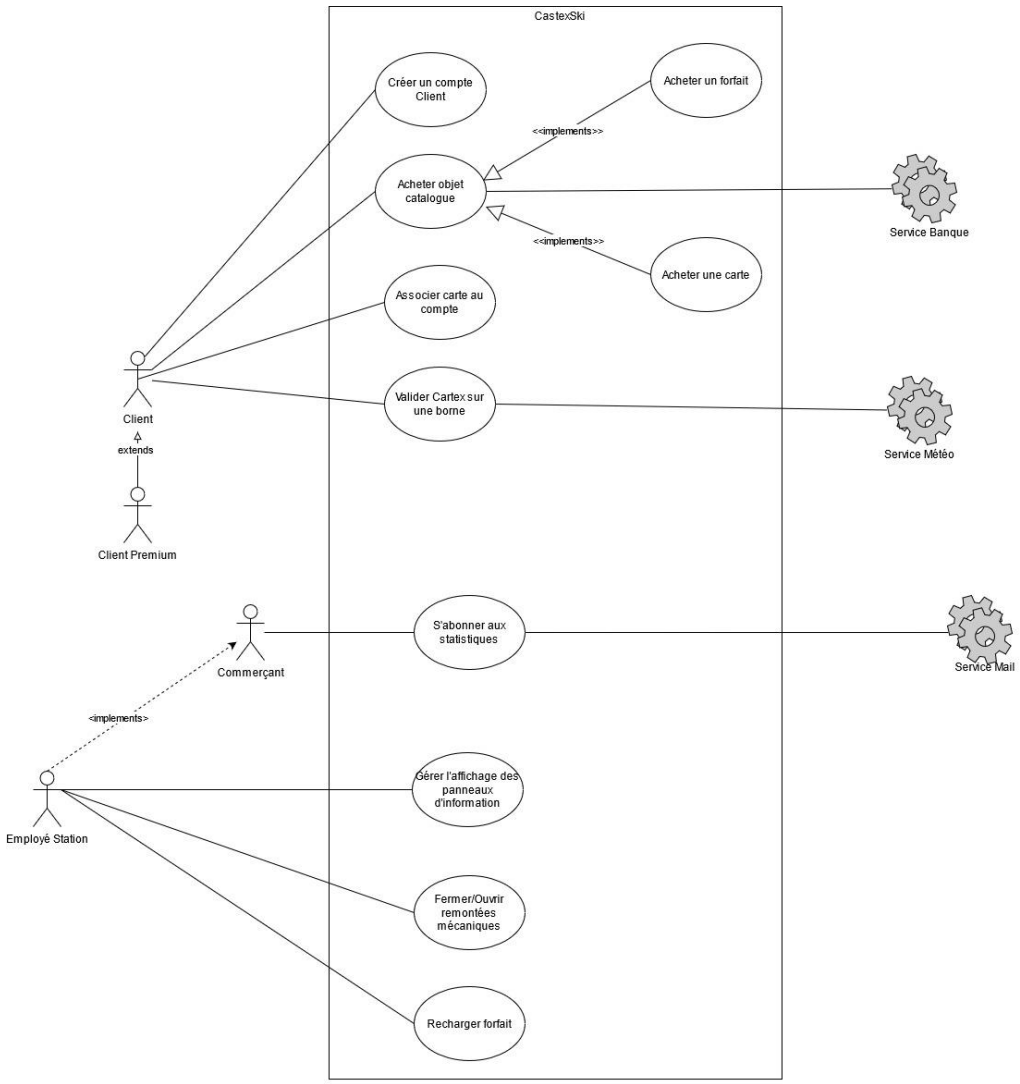
Notre projet CastexSki possède deux principaux acteurs :

Le client

Il peut créer un compte, ajouter des forfaits et des cartes à son panier, payer le contenu de son panier, associer une carte à un forfait, associer une carte à son compte et passer sa carte à une borne afin d'utiliser une remontée mécanique.
Un client devient Premium lorsqu'il possède une SuperCartex liée à son compte.
Il reçoit alors des notifications de la météo.

L'employé de la station

Il peut gérer l'affichage de messages sur des panneaux, gérer l'ouverture et la fermeture de stations et de remontées mécaniques.
S'abonner à la newsletter des statistiques de vente de forfait et de fréquentation de la station.
Obtenir les statistiques de vente de forfait ou de fréquentation à un jour donné.



Justification:

- Le client doit pouvoir créer son compte pour devenir premium ou enregistrer sa carte
- Un client peut accéder au contenu des offres de forfait et carte et faire un achat
- Avant d'emprunter les remontées, le client doit passer sa carte afin de vérifier sa validité
- Un perchiste peut alerter lors de sur fréquentations
- Le commerçant peut s'inscrire pour recevoir les dernières statistiques d'affluences

Avec plus de temps nous aurions pu implémenter un service d'envoi de message en plus de celui de l'envoi de mail. Nous aurions pu aussi implémenter un système de remboursement de forfait en cas de mauvaise météo par exemple.

Diagramme de classes

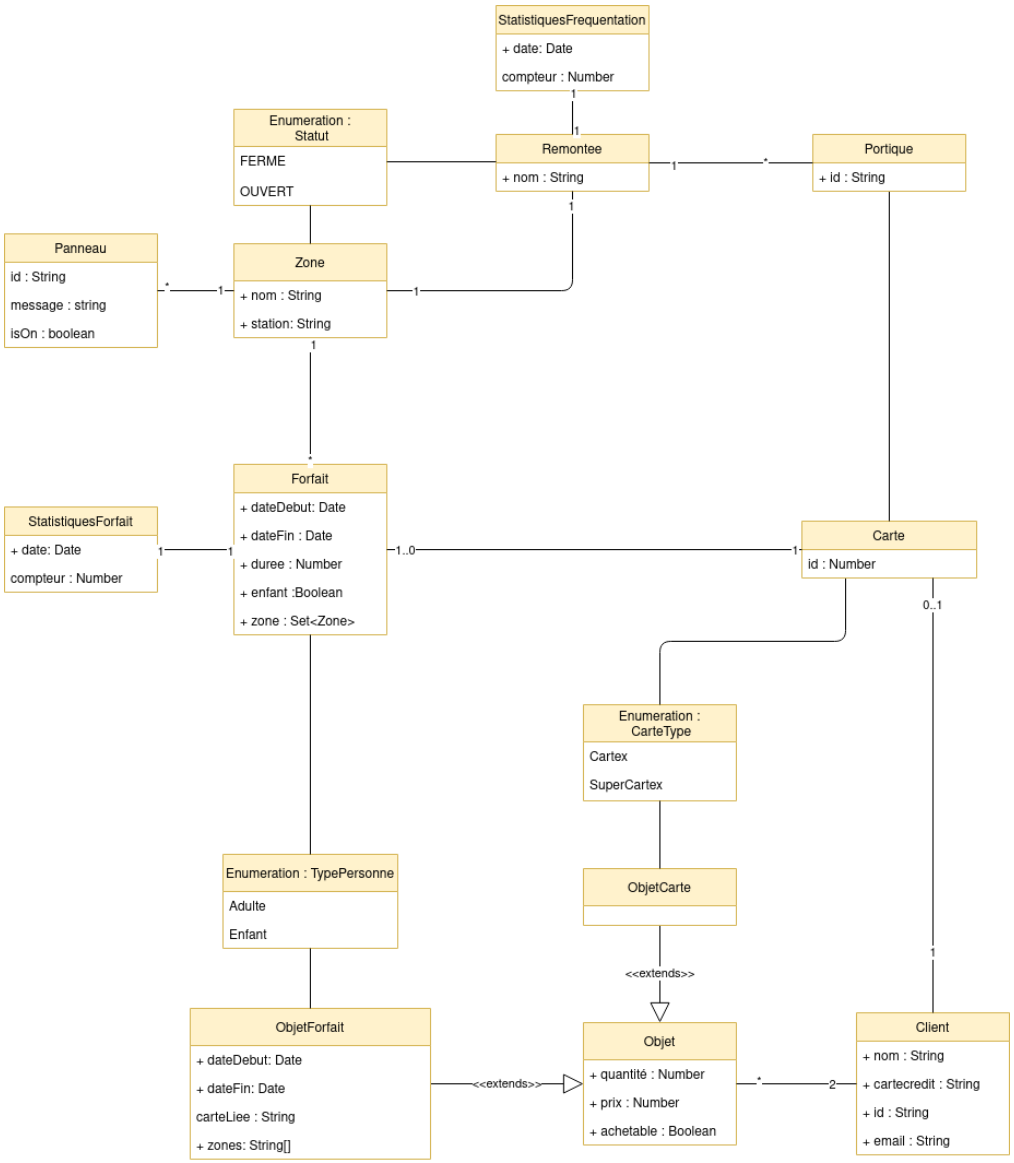
Pour construire notre modèle relationnel, nous sommes partis du diagramme de classe pour connaître les objets métier à sauvegarder dans la base de données. Nous avons dû faire des choix de relation entre les objets.

Un client est identifiable par son email et possède un panier contenant les objets qu'il souhaite acheter et des informations bancaires. Il peut aussi posséder une carte.

Une carte possède un forfait. Un forfait est modulable en fonction de la date de départ, de la date de fin, des zones, du type de forfait et une distinction adulte / enfant.

Une remontée mécanique possède plusieurs portiques et se situe dans une zone.

Nous avons deux types de statistiques différents : des statistiques de vente de forfait et de fréquentation d'une remontée mécanique pour un jour.



Notre diagramme de classe a beaucoup évolué depuis le 1er rapport.

En effet, lors de l'implémentation de la persistance nous sommes tombés face à un problème d'architecture de certaines classes. Au départ nous avions un objet « Carte » et « Forfait » qui possédaient deux rôles :

- L'objet commandé qui possédait les caractéristiques lors de l'achat comme le prix et la quantité
- L'objet final qui était utilisé par les autres composants et stocké dans la base de données

Nous avons donc séparé respectivement ces rôles en deux classes (*CardItem, PassItem*) et (*Card et Pass*). La première servant pour la commande, l'autre pour l'objet métier.

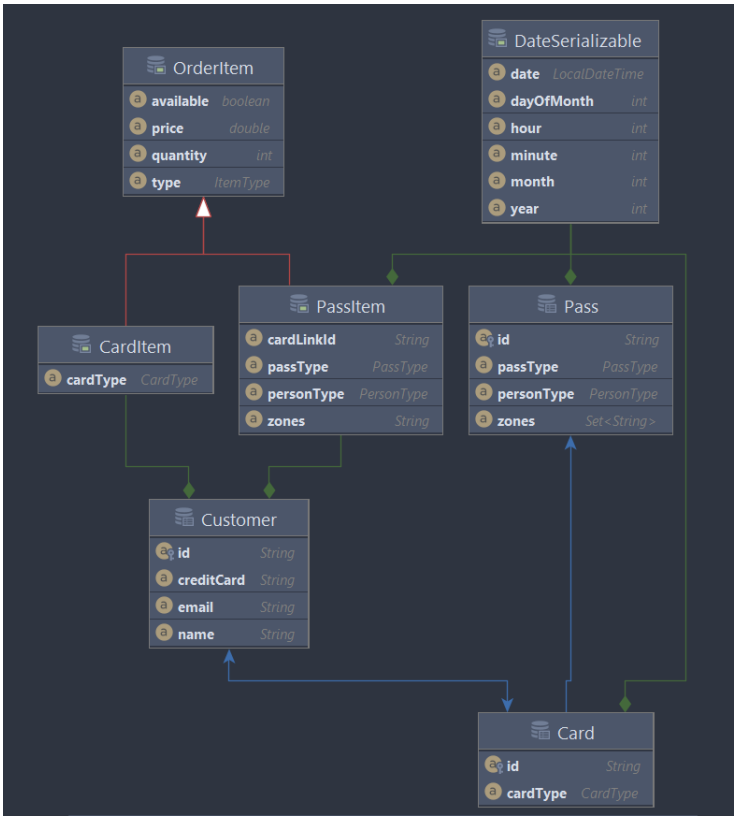
Ainsi l'ensemble des objets du catalogue ne sont pas directement stockées dans une table puisqu'ils ne sont utiles que pour l'instanciation des objets métiers et ne servent plus une fois la commande passée.

Cependant chaque carte et chaque forfait sont propres à un achat effectué par un client. Ils doivent donc être enregistrés dans une table puisqu'ils ne sont pas forcément liés à un compte client.

Enfin les panneaux, zones, remontées mécaniques et portiques sont des entités physiques qui possèdent un état et ont donc chacun besoin d'une table au sein de la base de données.

Persistance

La persistance a été mise en place la semaine suivant la présentation de notre MVP. Nous utilisons pour cela la librairie Java Persistance API, basé sur HSQLDB, qui nous a permis d'abstraire la liaison avec la base de données, et de ne pas utiliser directement des requêtes SQL dans notre code. Voici les diagrammes de persistance, avec liaisons entre les différents objets.



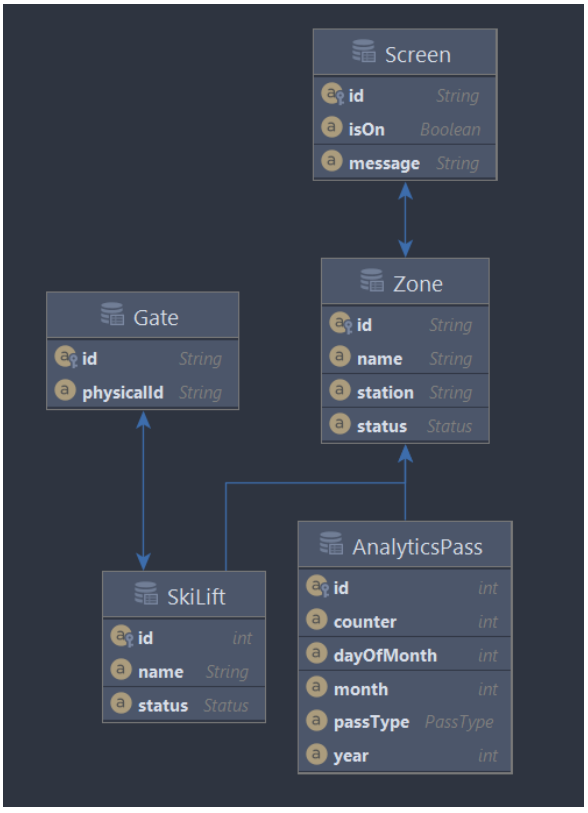


Diagramme de composants

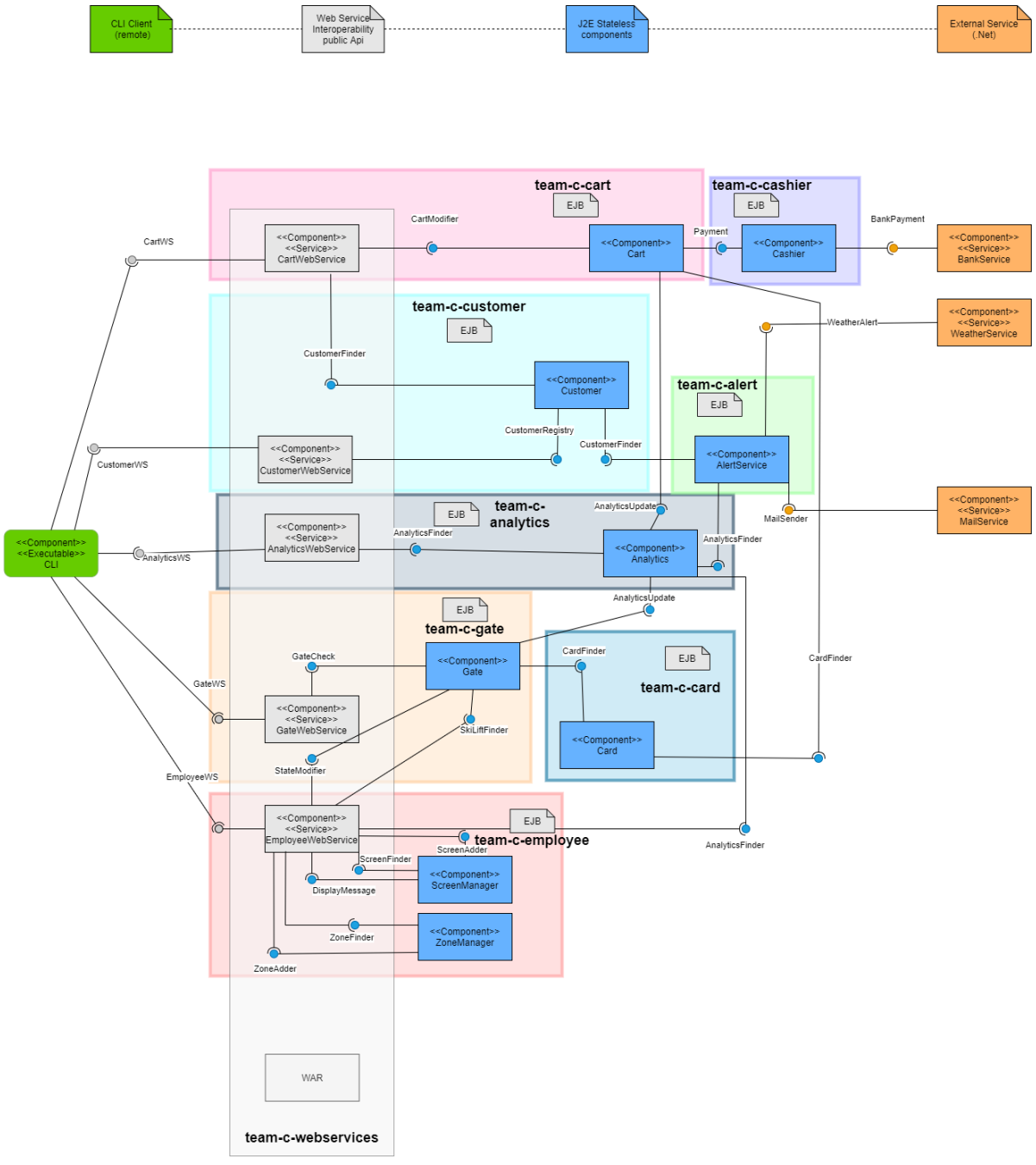
Le projet est divisé en 10 modules :

Dont un module $Entities$ contenant l'ensemble des objets métier du projet et un module $WebServices$ permettant une exposition des services en SOAP.

Nous avons découpé notre projet en plusieurs modules représentant chacun une partie logique du métier. Ainsi chaque module possède son composant, ses interfaces et son webservice si besoin.

Cela permet de ne pas dupliquer des responsabilités entre les composants au sein du projet et de garder une forte cohérence entre les appels de composants.

Tous les composants sont *Stateless* car les composants n'ont pas besoin de garder en mémoire des informations.



Le composant $Customer$ est un composant majeur possédant plusieurs interactions avec d'autres composants. Il permet d'enregistrer le compte d'un client et de retrouver ses informations via son email ou encore de lier sa carte et son forfait à son compte.

Le module $Cart$ utilise les informations du client afin de modifier et valider son panier puis délègue la charge du paiement au module $Cashier$ qui communique avec un service externe bancaire.

Le composant $Gate$ est appelé lorsqu'un client passe sa carte à un portique pour emprunter une remontée mécanique. Ce composant récupère les informations du forfait d'un client grâce à la puce de la carte à partir du composant $Card$. Il va ensuite s'assurer de la validité du forfait et de l'état de la remontée mécanique.

Le module $alertService$ dépend de plusieurs composants et services externes car il utilise les statistiques et les données de la météo pour envoyer des mails à un client premium.

Nous n'avons pas eu le temps d'utiliser des composants *« oriented message »*. Cependant on peut imaginer une utilisation par l'utilisateur possédant une SuperCartex lorsqu'il passe par une remontée mécanique. Celui-ci enverrait un message de passage qui serait traité plus tard par le serveur pour débiter une journée ou bien pour l'envoie de mail aux utilisateurs.

Les statistiques ont été implémentées dans le module $Analytics$ en utilisant les intercepteurs. Elles sont mises à jour par le module $Cart$ (après un paiement) et le module $Gate$ (après un passage validé), ce qui permet une implémentation orthogonale après l'exécution réussie d'une action.

Le module $Employee$ regroupe les interactions des employés de la station (perchiste, commerçant) comme la gestion de l'affichage des panneaux, de l'état des zones et des remontées mécaniques.

Il permet aussi l'inscription aux newsletters des statistiques.

Avec du recul, ce module aurait pu être découpé en deux autres :

- Un module **Station** permettant de gérer l'ouverture de zones et de remontées mécanique
- Un module **Panneaux** permettant de gérer l'affichage des panneaux.

Interfaces

Gestion des cartes

CardFinder

Permet de gérer les cartes et les forfaits achetés et de les lier. Les cartes sont retrouvées par l'id physique de la puce.

```
void putCard(Card card);

void putPass(Pass pass);

void linkPassToCard(Card card, Pass pass);

Optional<Card> getCardById(String id);

Optional<Pass> getPassById(String id);

List<Card> getAllCards();
```

```
List<Pass> getAllPasses();
```

Gestion des paniers

CartModifier

Cette interface est dédiée à la gestion et la validation du panier d'un client. $OrderItem$ permet de regrouper les informations d'un objet comme le prix, la quantité et le type de l'objet à traiter.

```
boolean addItem(Customer customer, OrderItem item);

boolean removeItem(Customer customer, OrderItem item);

void payByCb(Customer customer) throws PaymentException, EmptyCartException,
InvalidCardIdException, UnknownCardException;

Set<OrderItem> contents(Customer customer);
```

Gestion du paiement

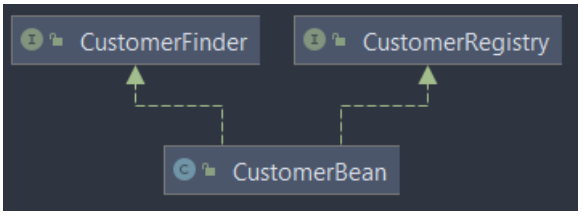
Payment

Traite du paiement d'un panier et renvoie une confirmation de paiement.

```
double getOrderPrice(Set<OrderItem> items);

boolean payByCb(Customer customer, Set<OrderItem> items) throws PaymentException,
InvalidCardIdException, UnknownCardException;
```

Gestion des clients



CustomerFinder

Permet de retrouver un utilisateur via un de ses attributs

```
Optional<Customer> findById(String id);

Optional<Customer> findByEmail(String email);
```

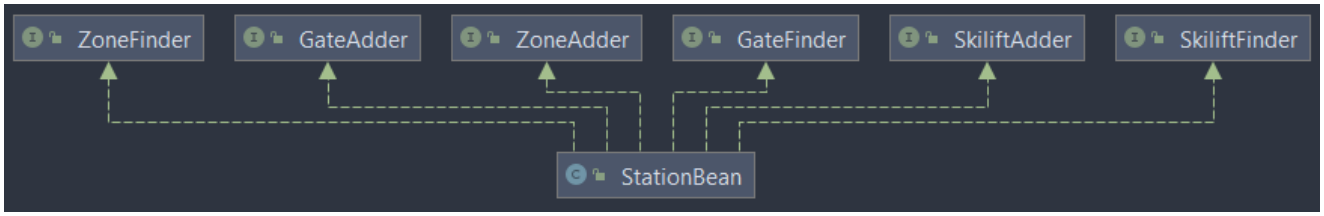
CustomerRegistry

Cette interface a pour rôle d'effectuer des actions liées au compte client comme la création d'un compte et l'association d'une carte à un client.

```
void registerCustomer(String email, String name, String creditCard) throws
AlreadyExistingCustomerException;

void associateCardToCustomer(Customer customer, Card card) throws
AlreadyAssociatedCard;
```

Gestion de la station



ZoneFinder & ZoneAdder

Ces interfaces permettent de trouver et d'ajouter des zones dans la base de données.

```
Optional<Zone> getZoneById(String id);
void addZone(String name);
```

SkiLiftFinder & SkiliftAdder

Ces interfaces permettent de trouver et d'ajouter des remontées mécaniques dans la base de données.

```
Optional<SkiLift> getSkiLiftByName(String name);

void addSkilift(String name, String zone) throws AlreadyExistingSkiliftException,
UnknownZoneException;
```

GateFinder & GateAdder

Ces interfaces permettent de trouver et d'ajouter des portiques dans la base de données.

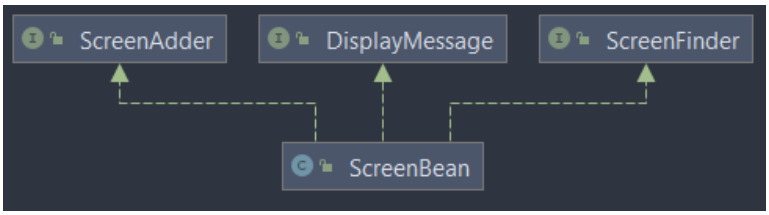
```
Optional<Gate> getGateByPhysicalId(String physicalId);

void addGate(String physicalId, String skiliftId) throws UnknownSkiliftException,
AlreadyExistingGateException;
```

Gestion des écrans d'affichage

ScreenFinder & ScreenAdder

Ces interfaces permettent de trouver et d'ajouter des écrans d'affichage dans la base de données.



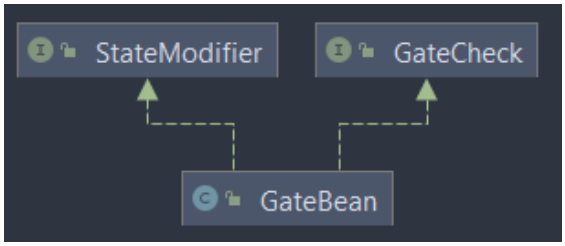

```
Screen findById(String id)throws ScreenNotFoundException;
List<Screen> findAll();
String addScreen(String zoneName);
```

DisplayMessage

Cette interface permet d'afficher des messages sur les panneaux d'une station.

```
void showMessage(String message, String …ids);
void showMessageOnZonesScreens(String message, String …ids);
```

Gestion des portiques de sécurité



GateCheck

Cette interface permet de valider le passage d'une carte et de son forfait à une remontée mécanique.

```
PassageResponse isPassageValid(Card card, Gate gate) throws UnknownCardException,
UnvalidatedPassException;
```

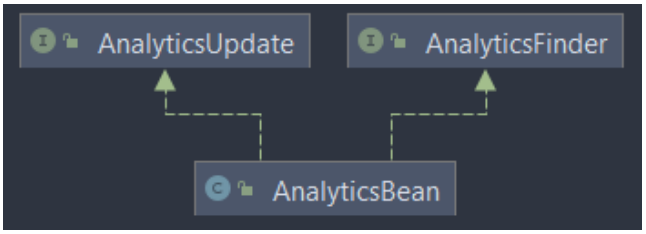
StateModifier

Cette interface permet de modifier l'état d'ouverture d'une zone / d'une remontée mécanique.

```
void setZoneState(Zone zone, Status status) throws UnknownZoneException,
AlreadyHaveStatusException;

void setSkiLiftState(String idLift, Status status);
```

Gestion des statistiques



AnalyticsUpdate

Cette interface a pour objectif de mettre à jour les statistiques de vente de forfait et de passage à une remontée mécanique. Elle est utilisée par les intercepteurs.

```
void updateAnalytics(AnalyticsVisit analytics)

void updateAnalytics(AnalyticsPass analytics)

void updateAnalytics(Set<OrderItem> cart)
```

AnalyticsFinder

Cette interface permet d'obtenir des statistiques sur une journée. Elle est utilisée par le service d'alertes pour envoyer un récapitulatif des statistiques d'une journée afin d'être envoyée aux commerçants abonnés.

```
String getDailyAnalytics(int day, int month, int year);

List<AnalyticsPass> getPassAnalyticsByDate(int day, int month, int year);

List<AnalyticsVisit> getVisitAnalyticsByDate(int day, int month, int year);
```

Evolution

Une des évolutions possibles de notre projet est la mise en place de la communication par message pour les services d'alertes mails pour le rapport de fréquentation aux employés de la mairie et d'alertes météo par sms aux clients disposant d'une SuperCartex. En effet, l'utilisation de Message Driven Bean dans ce cas-là se justifie par le besoin d'un envoi massif de mails ou SMS sans surcharger les services externes qui se chargent de les envoyer.

Actuellement, les remontées mécaniques se ferment automatiquement lorsque la météo est mauvaise. Nous pourrions mettre en place un automatisme similaire en utilisant l'affluence à certaines remontées, pour répartir les skieurs sur la station et ainsi éviter des attroupements autour de certaines zones. Pour cette idée, il suffirait juste de traiter les données de statistiques que nous récoltons déjà.

Conclusion

Tout au long de ce projet, l'architecture de ce projet a évolué pour permettre une meilleure interopérabilité entre les composants. En découpant le code par module, nous avons séparé la couche de persistance, de métier et de modèle. Ce qui pourrait permettre par la suite de changer de technologie de base de données sans changer la logique d'interactions des composants. Ce découpage permet aussi une implémentation de nouveaux services plus facilement en créant un nouveau module sans impacter les autres composants.

Rédiger ce document nous a permis de prendre du recul sur l'architecture finale de notre projet puisqu'en effet, le découpage du module Employee aurait pu être affiné davantage. Enfin nous aurions pu diviser l'exécutable en différents clients afin de coller à un projet plus réaliste qui posséderait plusieurs points d'entrées comme une console : pour le client, pour passer à une borne et pour les actions des employés.