POLYTECH
NICE-SOPHIA

Université Nice Sophia Antipolis
Membre de UNIVERSITÉ CÔTE D'AZUR

Rapport de projet ISA

Drone Delivery

Team B

Alexandre LONGORDO
Alexis LEFEBVRE
Betsara MARCELLIN
Jason HAENLIN
Younes Abdennadher

Table des matières

I) Diagramme de cas d'utilisation

1) Diagramme de cas d'utilisation actuel

La figure 1 représente les cas d'utilisations couverts par notre système actuel.

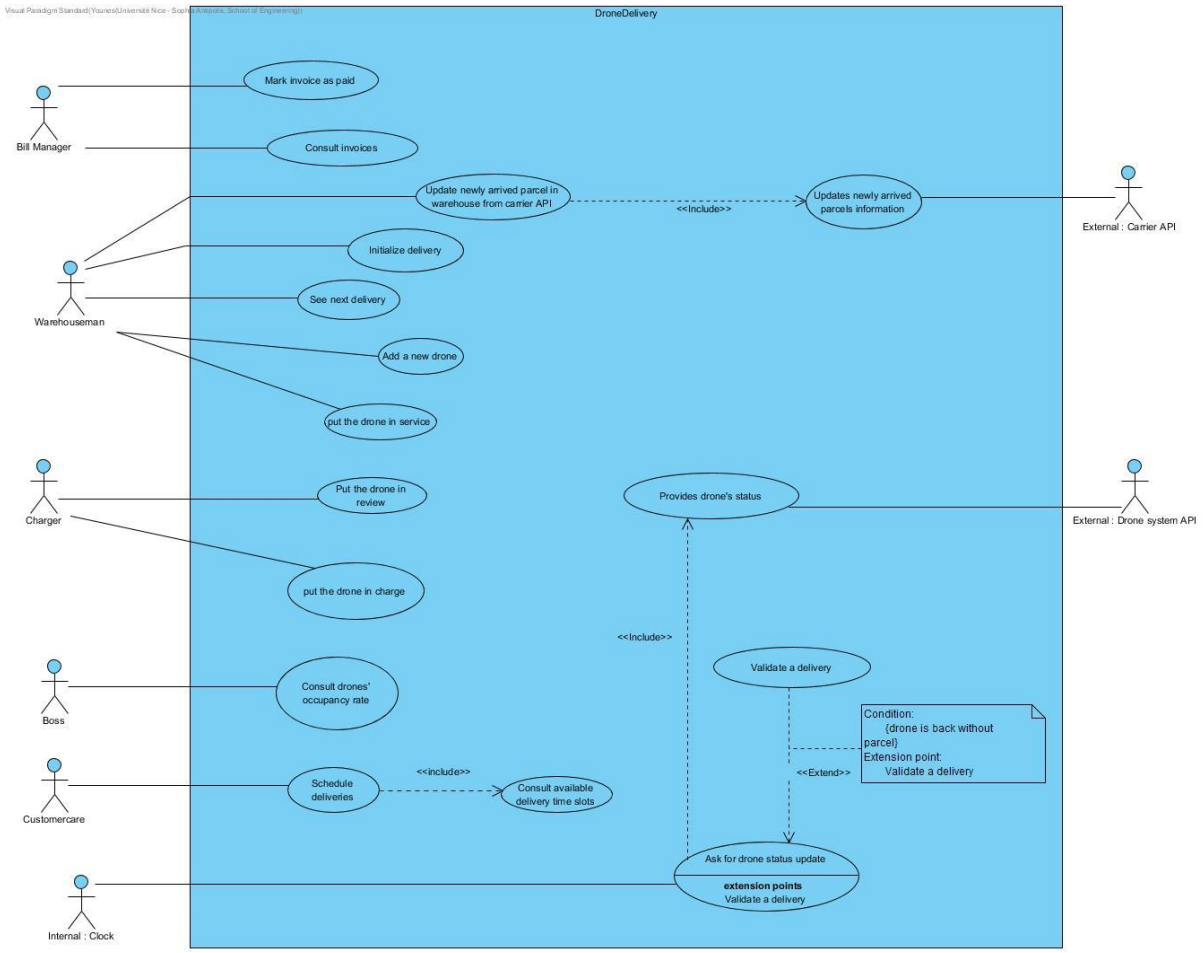

Figure 1 : Diagramme de cas d'utilisations couverts par le système

Nous pouvons noter trois acteurs spéciaux dans ce diagramme :

- Le système de drone externe (*External drone system API*) fournit au système via son API (le système de drone n'interagit donc pas directement avec le système) le statut du drone.
- Le système externe de transporteur (*External carrier API*) fournit au système via son API (il n'interagit pas directement avec le système) les nouveaux colis qui arrivent à l'entrepôt.
- L'horloge (*Internal clock*) qui s'occupe de demander régulièrement au système de drone si son statut a changé lorsqu'une livraison est en cours. Cette horloge permet à notre système d'être au courant en temps réel du statut du drone une fois que celui-ci est de retour à l'entrepôt après une livraison. C'est l'horloge qui s'occupe de synchroniser le statut du drone dans le système avec le statut du système de drone externe et de mettre à jour le statut de la livraison au retour du drone.

Différences par rapport aux cas d'utilisations initialement prévus

Ce diagramme de cas d'utilisations possède moins de cas d'utilisations que ce qui a été initialement prévu (cf Figure annexe 1). Les cas d'utilisations que nous couvrons sont en effet des cas qui apportent plus de valeur que ce qui a été retiré. Passons en revue certains cas notables qui ne sont plus couverts par le diagramme :

- Le chargeur ne consulte plus le temps de vol des drones via le système, ni le niveau de batterie. Cette action du chargeur peut être faite directement en regardant le drone, et sans passer par le système. L'implémentation d'accesseurs directement dans le système n'est donc pas un ajout de valeur.
- L'acteur client n'existe plus. Il n'est pas nécessaire qu'il interagisse avec le système puisque le service client peut déjà faire ce travail.
- Nous avons supprimé la possibilité pour le client de donner son avis pour une livraison, le boss ne peut donc pas consulter les avis clients.

2) Scénario actuel couvert

Nous allons détailler dans cette partie un scénario couvert par le périmètre actuel de notre projet.

- Etant donné un point d'accès bureau, un point d'accès entrepôt, un drone d'ID 001 avec 19 heures de vol et une API fournie par l'un des transporteurs.
- Marcel reçoit les nouveaux colis de l'un des transporteurs dans l'entrepôt et met à jour le système depuis le client de l'entrepôt. Le système se met à jour en fonction des données présentes sur l'API du transporteur. Le système crée également une facture correspondant à l'arrivée de colis par le transporteur.
- Les clients appellent Clissandre pour prévoir des horaires de livraison toute la journée. Clissandre note les rendez-vous dans le système jusqu'à remplir la journée de livraisons (il faut noter que l'horaire de livraison correspond à l'horaire de départ du drone et non à l'horaire d'arrivée du colis).
- Marcel ajoute un nouveau drone 002 dans le système. Clissandre peut maintenant planifier de nouvelles livraisons pour la journée.
- Marcel regarde son écran pour avoir les références des prochains colis à charger ainsi que le numéro du drone associé, attache le colis à ce drone, et appuie sur le bouton Initialiser pour que le système envoie le signal de lancement au drone avec l'heure prévue de départ. Le drone sait lui-même quand il doit partir et gère son lancement en fonction de l'heure reçue.

- Au retour du drone, Charlène le récupère, et en fonction des informations qu'elle lit directement sur le drone, soit le met en charge, soit l'apporte à Garfield pour révision et change son état depuis son interface pour avertir qu'il est en révision, soit l'apporte à Marcel pour la mission suivante. Le retour du drone vide (donc colis livré) ou non vide déclenche une mise à jour du statut de la livraison (livrée ou ratée).
- Lorsque Marcel reçoit le drone de la part de Garfield ou de Charlène, il le pose sur son emplacement afin que le système passe son état en prêt pour la prochaine livraison.

- Tous les soirs, Gisèle émet les factures. Elle peut les voir, elle a alors les informations de transporteur, les colis associés, s'ils ont été livrés, si la facture est payée ainsi que les dates de création et de paiement de la facture.
- Gisèle peut indiquer qu'une facture a été payé.
- Bob peut accéder quand il veut au taux d'occupation du drone pour la journée.

II) Choix par rapport au métier

1) Interactions avec l'extérieur

Notre système interagit avec les employés de bureau (le comptable, le patron et le service client), les employés d'entrepôt de l'entreprise (le chargeur et le manutentionnaire), les drones ainsi que les transporteurs. Nous avons choisi de représenter le métier des employés par l'intermédiaire de deux applications console : *OfficeCLI* et *WarehouseCLI*.

Nous recevons les informations des colis que le transporteur livre à l'entrepôt via son API REST, Carrier API. L'interaction de notre système avec les drones est une communication avec une API REST qui dirige la flotte de drone. Pour avoir une vue globale de l'interaction avec l'extérieur, se référer à la figure 3).

Avoir deux clients différents nous permet de séparer en deux environnements bien distincts les services auxquels ils ont accès. Les employés de bureau n'ont pas à interagir avec ce qui se passe dans l'entrepôt et réciproquement. Par exemple, il ne serait pas judicieux qu'un garagiste puisse planifier une livraison lui-même, de même, un chargé de clientèle ne devrait pas pouvoir décider de mettre un drone en charge.

Une autre solution serait d'utiliser une authentification pour chaque employée, ainsi via le même client afficher une interface adaptée à chacun. Cette solution demande cependant un effort supplémentaire de la part de l'utilisateur.

La première action dans le processus de déroulement de la livraison des colis c'est l'enregistrement des colis à livrer. Dans notre système, un employé peut mettre à jour les colis enregistrés via une interface à sa disposition. L'interface récupère ses données depuis l'API du transporteur, cette API indique tous les colis qu'on a reçu du transporteur. Cette solution nous évite d'enregistrer les colis à l'unité dans notre système, ce qui est un avantage lorsque l'on a un flux important de colis arrivant dans l'entrepôt. Cependant, nos clients devront avoir une API qui soit conforme à nos interfaces (cf. interface *CarrierMonitor*).

Ensuite, une communication avec la flotte de drone physique est nécessaire pour coordonner la livraison des colis. Nous avons donc prévu que le système communique avec le système externe des drones. Cette commination se traduit par une API REST qui nous fournit le statut d'un drone et qui permet au système d'envoyer des informations aux drones. L'avantage de se baser sur un système externe de drone permet à notre système de ne pas avoir à gérer le contrôle des drones. Lors de l'initialisation d'une livraison par exemple, il suffit au système de fournir l'heure à laquelle le drone doit décoller et le système externe s'occupe de faire voler le drone à l'heure exacte.

2) Diagramme d'objets métier

La figure 2 correspond au diagramme d'objets métier.

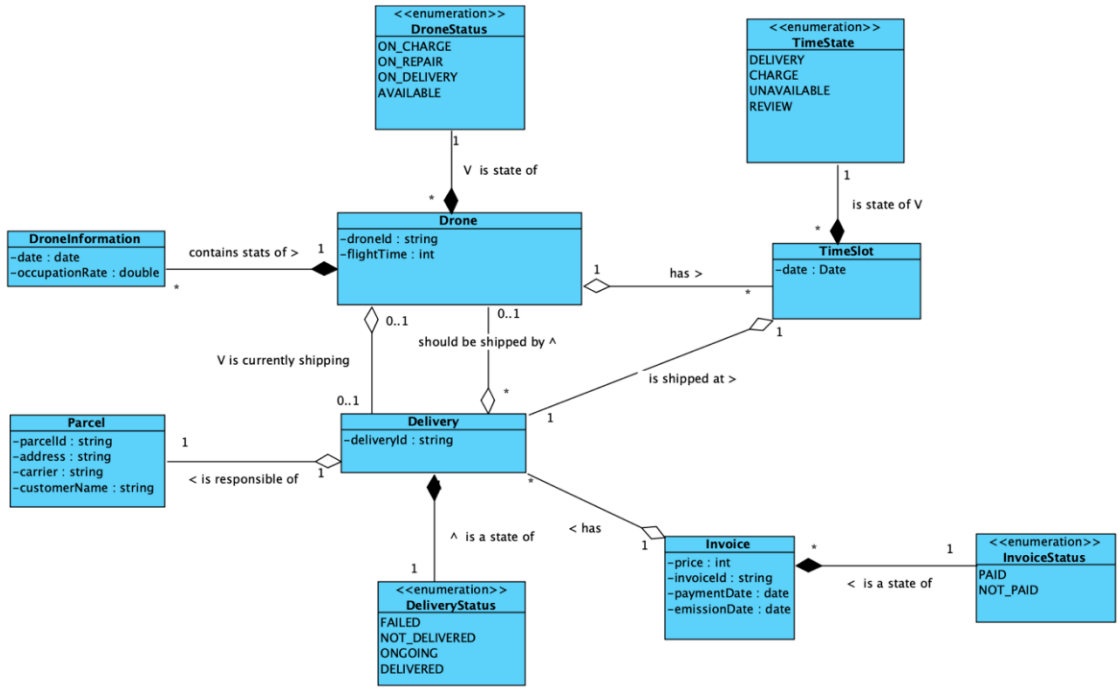

Figure 2 : Diagramme d'objets métiers actuel

A) Choix relatifs au modèle

Le drone est le centre du modèle, il possède des attributs comme un identifiant et une durée de vol. Cette durée de vol permet actuellement de savoir si le drone doit être mis en révision, c'est une donnée qui pourrait servir à bien d'autres choses (statistiques par exemple).

Nous avons associé un statut à un drone sous forme d'énumération, en effet les statuts sont toujours les mêmes et permettent de savoir dans quel état il se trouve actuellement. Le drone est également associé à une liste de *DroneInformation*, il s'agit en fait d'une liste de statistiques.

Nous avons deux liens entre Drone et Delivery, d'abord une Delivery possède le drone qui va la livrer et le Drone possède la Delivery qu'il est en train de livrer. Plusieurs Delivery peuvent posséder le même drone mais une seule Delivery peut être actuellement livrée.

Un *Timeslot* représente l'état planifié d'un drone à une heure donnée, nous avons fixé la durée de celui-ci à 15 minutes pour faciliter l'algorithme. Ce créneau peut être une charge, une *review*, une livraison ou une indisponibilité. S'il s'agit d'une livraison alors le *TimeSlot* est associé à cette livraison. Nous avons décidé de cette approche afin d'avoir un accès simplifié au planning d'un drone, cette architecture permet également d'obtenir le planning global du système en bouclant sur les drones et leurs créneaux.

L'objet Delivery est l'objet utilisé par le système tout au long de la livraison d'un colis. Cette livraison contient le colis (*Parcel*), elle possède un statut qui correspond à son état (livrée, non livrée, échouée ou en cours de livraison). En effet le système suit et met à jour cet objet à chaque étape de la livraison.

Une facture est représentée par l'objet *Invoice* qui possède un prix, une référence, une date d'émission et un statut de paiement. La facture contient des livraisons (une ligne dans la facture est une livraison) ce qui permet d'accéder aux informations du colis (transporteur, référence du colis …).

L'objet *DroneInformation* représente une statistique à une date d'un drone, pour le moment cet objet ne possède que le taux d'occupation. Un Drone possède plusieurs de ces objets, cela permet d'avoir des statistiques sur le drone à une date précise et donc si un besoin d'avoir des statistiques sur un intervalle ou sur les mardis cela est possible.

B) Persistance des objets métier

Pour la gestion de nos objets métiers, il a été difficile de bien comprendre les rouages mêlés entre les objets et la notion de base de données. Il faut traiter les objets plutôt comme des tables en base de données. Le point où nous avons dû faire le plus attention concerne les relations entre Delivery, Drone et *TimeSlot*. Une Delivery est associée à un Drone qui est lui-même associé à la Delivery actuelle et chaque Drone possède une liste de *TimeSlot* qui est associée à une Delivery lorsque celle-ci est utilisée pour effectuer une livraison.

De ce fait, il faut faire attention car nous avons une dépendance circulaire assez forte. De ce fait, nous avons fait en sorte de mettre toutes ces relations en *LAZY Loading* pour éviter de tout charger d'un seul coup. En effet, si on souhaite avoir le drone associé à une livraison, il sera chargé dynamiquement par le manager lorsque nous souhaitons y accéder. Par ailleurs, dans certains cas où il faut des éléments provenant de ces 3 relations, nous avons effectué des détachements pour couper la dépendance avant que le manager ne demande une autre information. Par exemple, si nous récupérons tous les Drones, nous récupérons tous les *TimeSlots* associés. Or si on récupère les *Deliveries* des *TimeSlots*, alors nous créons une dépendance circulaire. De ce fait, juste avant de récupérer les *Deliveries*, il faut détacher les Drones, cela permet de récupérer les *Deliveries* mais d'avoir la propriété Drone à *null* de l'entité.

De la manière, nous avons décidé de faire des objets persistés car toutes nos entités doivent être gardé en mémoire pour pouvoir les réutilisés plus tard. Ça reste cohérent avec le fait que nos composants soient uniquement Stateless car nous n'avons pas besoin de garder leurs états. Un composant Statefull n'est utile que pour garder une session active pour plusieurs appels de commande, ce qui n'est pas notre cas. Une requête répond à une demande et nous persistons nos objets pour nous permettre de récupérer leurs états dans un composant qui en a besoin.

III) Choix d'implémentation des fonctionnalités

La figure 3 représente notre diagramme de composant global actuel. La figure 4 représente le diagramme de composant tel qu'il était à la sortie du produit minimal viable.

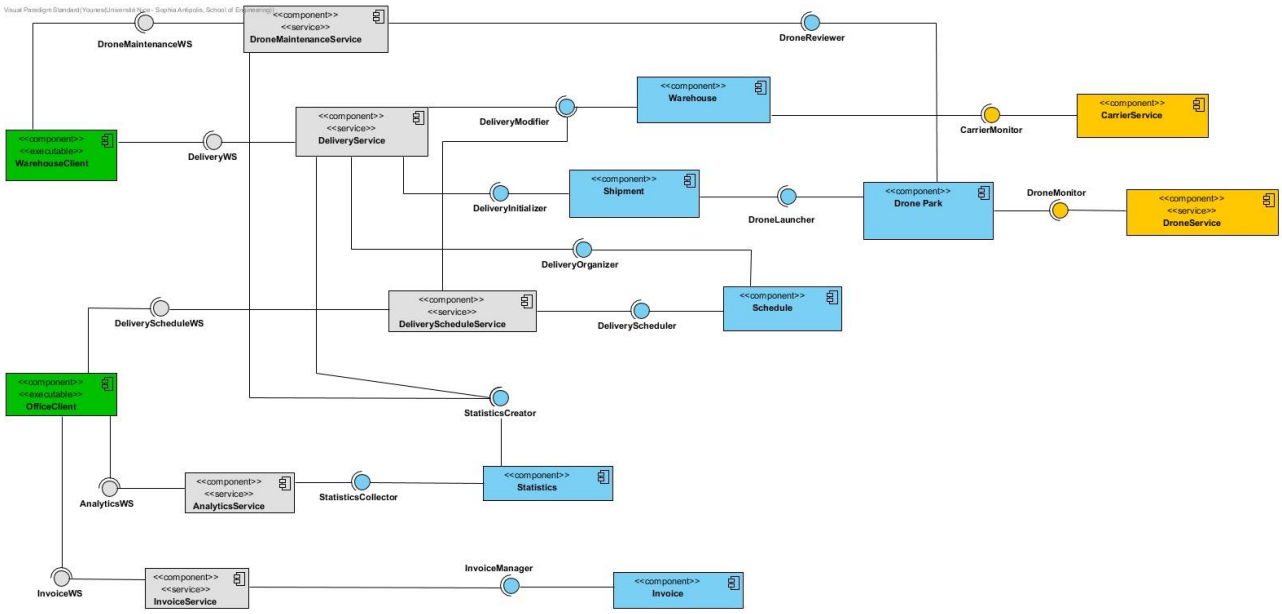

Figure 3 : Diagramme de composants actuel

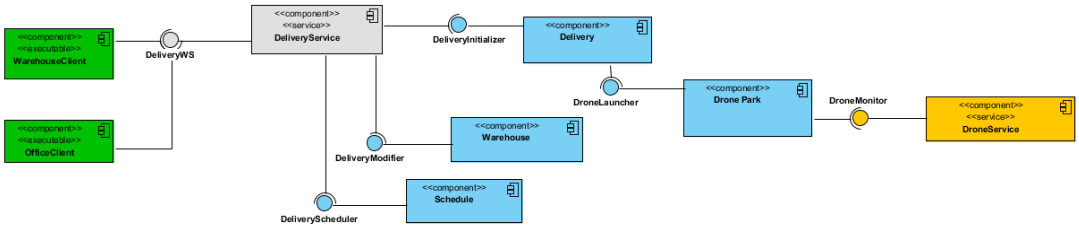

Figure 4 : Diagramme de composant du MVP

Sur la figure 4, on voit que nous avons décidé d'implémenter, pour le MVP, un produit fournissant le plus de fonctionnalités pertinentes possible, tout en gardant chacune de ces fonctionnalités minimales. Cela se traduit par exemple par la présence de deux clients pour rester au plus proche de ce que demande le métier (deux interfaces bien distinctes pour les employés qui ne travaillent pas au même endroit).

Sur la *figure annexe 2* figure le diagramme de composants initialement prévu. Celui-ci peut être comparé au diagramme de composant actuel en figure 3.

Tous nos composants sont des composants *stateless*. Aucun d'entre eux n'ont besoin de garder en mémoire la session d'un utilisateur. Chaque action client nécessite une seule suite d'action dans le système. Les composants communiquent entre eux les informations dont ils ont la responsabilité de récupérer de la base de données.

1) Gestion des drones

La gestion des drones de l'entrepôt se fait exclusivement par le composant *DronePark* et via l'interface *DroneReviewer*. *DronePark* est responsable de l'ajout des drones, du changement de leur état, de leur lancement lors de l'initialisation d'une livraison et de leur arrivée après une livraison.

A) Méthodes de l'interface DroneReviewer

L'interface *DroneReviewer* fournit les méthodes permettant la maintenance du parc de drones. Parmi ces méthodes on retrouve *addDrone(id)* qui permet d'ajouter un drone au système. Une méthode permettant d'obtenir les drones *getDrones()* est également présente. En effet la responsabilité de création et de récupération de drones revient à DronePark.

On y trouve également trois méthodes permettant de changer l'état du drone :
setDroneInCharge(droneId), putDroneInRevision(droneId) et setDroneAvailable(droneId).

B) Prise de recul

DronePark est le seul composant en relation avec les API externes de gestion de drones. Il est également le seul composant à changer les états et le nombre de drones. Le bénéfice d'une telle architecture est que si l'API du drone vient à changer le seul composant qui devra être mis à jour est *DronePark* sans affecter le reste de l'architecture.

2) Gestion des expéditions

La gestion des expéditions est gérée par trois composants. *Schedule*, Shipment et DronePark via les trois interfaces décrites en partie A ci-dessous.

A) Méthodes des interfaces DeliveryOrganizer, DeliveryInitializer et DroneLauncher

Lorsque Marcel cherche à avoir la prochaine livraison à charger, il passe par le service *DeliveryService* qui va d'abord se connecter à l'interface *DeliveryOrganizer* qui demande à *Schedule* la prochaine livraison dans le planning.

Une fois que l'on initialise la livraison, *DeliveryInitializer* permet d'interagir avec le composant *Shipment*. Ce composant est chargé de récupérer la bonne livraison puis interagit lui-même avec le composant *DronePark* via la méthode *initializeDroneLaunching(drone, date, delivery)* de l'interface *DroneLauncher*. *DronePark* n'a alors plus qu'à envoyer les informations reçues à l'API de drone externe qui va se charger de lancer le drone au bon moment.

C) Prise de recul

Nous nous sommes posé la question de la pertinence de l'existence du composant *Shipment*. Ce composant n'est en effet qu'un intermédiaire entre le service Delivery et le composant *DronePark*. Il est cependant, selon nous, nécessaire pour ne pas donner plus de responsabilité au composant *DronePark* que ce qu'il faut. Ce composant n'a qu'un rôle : gérer les drones.

Avoir un composant chargé uniquement de la logique d'expédition nous donc semble important.

3) Gestion de l'entrepôt

La gestion de l'entrepôt est faite par le composant Warehouse. Celui-ci permet de gérer les colis et la création des livraisons. C'est ce composant qui s'occupe de faire le lien entre les API des transporteurs et notre système. Dans notre cas, nous avons une API (*CarrierAPI*) qui permet de simuler l'interaction avec un seul transporteur. Ce composant va sur demande, récupérer les colis depuis l'API pour préparer les livraisons pour la journée. Il va donc instancier et persister les nouvelles livraisons qui pourront par la suite être liée à un *Timeslot* qui n'est pas gérer par le même composant. Le composant Warehouse va après la création des livraisons, notifier le composant *Invoice* pour lui demander de générer une nouvelle facture pour le transporteur avec la liste des livraisons.

A) Évolution depuis la planification initiale

On remarque en premier lieu que le composant Warehouse qui n'est pas présent sur le diagramme initial, a été ajouté au cours du MVP pour stocker temporairement des livraisons mockées directement dans le système.

Actuellement, le composant *Warehouse* est responsable des entrées des colis dans le système, son rôle est de faire entrer les *Parcel* et les assigner dans une nouvelle *Delivery.* Ainsi le colis est gardé avec toutes ces informations dans le système mais notre système ne va toucher qu'à la livraison qui a été nouvellement créée. Le composant *Warehouse* est connecté à une API externe qui est l'API d'un transporteur. Nous avons décidé de créer ce composant pour lui donner une responsabilité précise : faire l'interface avec les transporteurs externes. Ainsi si d'autres transporteurs doivent être ajoutés il s'agira de modifier uniquement ce composant.

Warehouse est un composant *Stateless* qui ne conserve pas d'état, en effet il traite directement les colis reçus par l'API et les transforme en Delivery qu'il envoie directement en base de données.

B) Méthodes de l'interface DeliveryModifier

L'interface *deliveryModifier* est composée de deux méthodes.

La première méthode est *checkForNewParcels()* celle-ci est chargée de récupérer les colis auprès des transporteurs et de les transformer en livraison, c'est Marcel qui est responsable en début de journée de déclencher cette arrivée de colis une fois les camions arrivés dans l'entrepôts.

La seconde méthode *findDelivery(String deliveryId)* permet de récupérer une livraison.

C) Prise de recul

Le composant Warehouse est l'unique composant qui gère l'entrée des colis dans le système. Étant donné qu'il peut recevoir un grand nombre de colis à la fois il pourrait devoir faire un grand traitement en cas de grosse charges (beaucoup de colis par transporteurs et plusieurs transporteurs). Le composant pourrait donc bloquer le système quelques instants le temps de *parser* tous les colis et créer toutes les *Delivery*. Il serait judicieux d'utiliser un *Message Driven Bean* qui pourrait recevoir les données de(s) API et envoyer un message qui serait reçu par le composant *Warehouse.* Cela permettrait de ne pas bloquer le système, de plus l'ordre de récupération des colis n'importe pas.

De plus dans notre architecture, *Warehouse* n'est capable de fonctionner qu'avec une seule API de transporteur, pour cela il aurait fallu faire une structure d'héritage, *CarrierAPI* serait une classe abstraite et on aurait une classe par transporteur différente. Lors de l'appel on parcourrait tous les transporteurs et on appellerait *generatingInvoice* avec le nom du transporteur.

4) Gestion du planning

Le composant *Schedule* est en charge de gérer le planning de tous les drones, il prévoie toutes les livraisons, les temps de charge et les temps de maintenance. Il comporte tout l'algorithme de planification mais on pourrait être capable d'appeler une API externe qui offre un algorithme de planification.

Notre algorithme est très naïf, nous partons du principe qu'une journée c'est toujours 3 livraisons 4 temps de charge de manière répétitive. Lorsque aucune livraison n'est planifiée parmi les 3 slots avant le temps de charge, ce temps de charge est « Reservé pour la charge ».

La figure 5 représente un planning tel qu'il est généré pour un drone au début de la journée s'il n'a aucune révision prévue dans la journée.

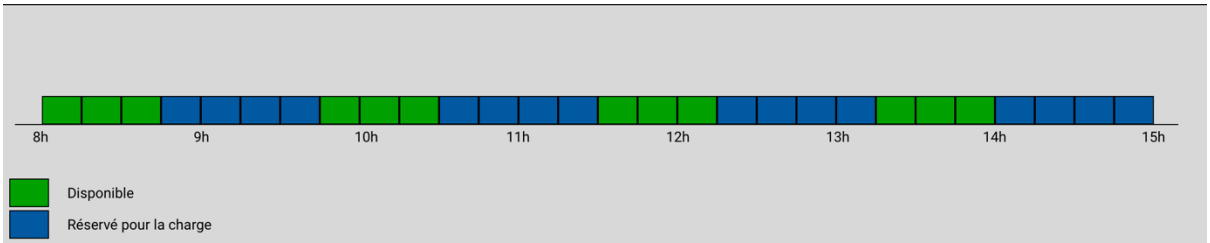

Figure 5 : Planning au début de la journée

Admettons que Clissandre veuille mettre une livraison pour 9h45 notre algorithme mettra la livraison et les temps de charge correspondant (fig. 6).

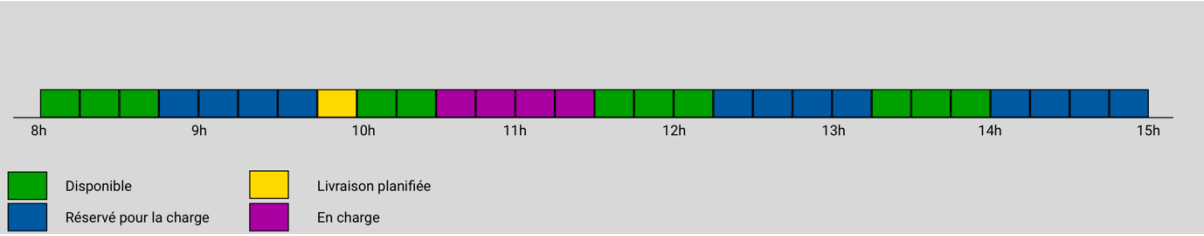

Figure 6 : Planning après avoir planifier une livraison

Maintenant, dans une autre journée, le drone a besoin d'une révision après 1h de vol dans la journée, on aura donc 3h de disponible, les temps réservés pour la charge, une case disponible et 12 cases pour la maintenance (fig. 7).

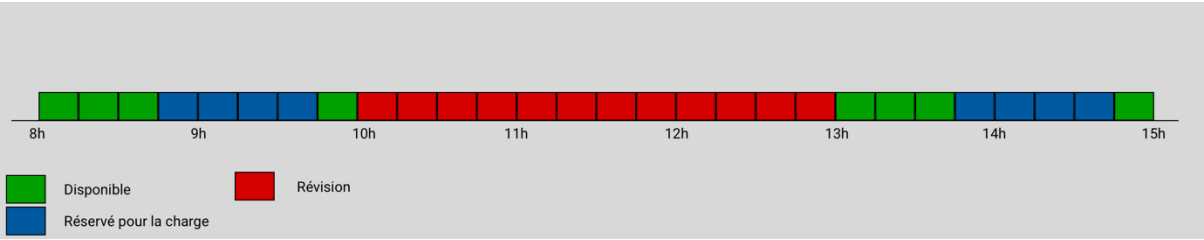

Figure 7 : Journée avec une maintenance à effectuer

Comme on le constate on part du principe qu'une livraison dure au maximum 15 minutes aller/retour, même si la livraison est effectivement plus rapide, on ne démarrera pas la prochaine livraison à son heure prévue.

Cet algorithme nous permet d'avoir un comportement minimal. Cela nous a permis de nous concentrer sur l'architecture du projet plutôt que sur le développement d'un algorithme bien optimisé.

A) Méthodes des interfaces DeliveryOrganizer et DeliveryScheduler

DeliveryOrganizer et *DeliveryScheduler* sont deux interfaces produites par le composant *Schedule*.

DeliveryOrganizer a une méthode *getNextDelivery* qui indique la prochaine livraison à effectuer au manutentionnaire dans le client *warehouse*.

DeliveryScheduler a deux méthodes *Schedule*Delivery qui planifie une livraison à une date donnée si le créneau est disponible et *getplanning* qui renvoie le planning d'un drone.

C) Prise de recul

Le composant *Schedule* a une classe dieu puisque tout l'algorithme est dans une même classe et nous aurions pu découper afin d'améliorer la lisibilité du code. Par exemple créer des objets internes afin de découper la responsabilité pour chaque partie de l'algorithme.

Notre *Schedule* passe directement dans la base de données pour récupérer les drones et leur planning. Nous aurions pu faire un lien avec *DronePark* pour demander à *DronePark* la liste de drones ou un drone en particulier.

Dernière amélioration possible, pour rendre le projet plus ouvert à l'amélioration, nous aurions pu ajouter une classe abstraite, un moyen de transport, que le drone hériterait. « *Schedule* » deviendrait alors indépendant des drones et recevrait uniquement un moyen de transport, cela permet de rendre *Schedule* réutilisable pour d'autres moyens de livraison comme par exemple la livraison par voiture autonome.

Toutes

getplanning devrait ne pas avoir la notion de drone puisque le client office ne s'occupe pas des drones.

5) Gestion des factures

La gestion des factures est gérée par le composant *Invoice*, celui-ci émet une interface *DeliveryBilling* qui a une fonction *generatingInvoice*, celle-ci est appelé par Warehouse qui donne la liste des livraisons à effectuer pour générer la facture.

Invoice attend des livraisons d'un unique transporteur et génèrera une facture à chaque appel de l'API externe.

La génération est très simple, on créé une facture avec sa date d'émission, son prix, et son statut.

A) Méthodes de l'interface InvoiceManager

InvoiceManager possède deux méthodes *getInvoices* qui permet de récupérer la liste des factures et *confirmInvoicePayment* qui permet de confirmer que le paiement d'une facture a bien été exécuté et passer la facture dans le statut « payé ». Elle met aussi la date de paiement de la facture.

B) Prise de recul

Le composant est assez rudimentaire, ses tâches sont basiques. On aurait cependant pu ajouter la notion de transporteur passé en paramètre de *generatingInvoice* et de l'ajouter à la facture. On pourrait aussi améliorer avec une gestion des factures par transporteur, filtrer les factures non payées.

6) Gestion des statistiques

Les statistiques sont gérées par le composant *Statistics,* ce composant est de type stateless car nous avons décidé que les statistiques sont gardées en base de données et récupérées une fois demandées. Le composant s'occupe des entrées et sorties de statistiques. Expliquer dans cette partie la manière d'implémenter et pourquoi

A) Méthodes des interfaces StatisticsCollector et StatisticsCreator

Les deux interfaces StatisticsCreator et StatisticsCollector sont respectivement des points d'entrée et de sortie de données dans le composant.

Prenons le cas du taux d'occupation des drones, à chaque fois qu'une action est effectué sur un drone un intercepteur va demander la branche StatisticsCreator et plus *précisément* la méthode

addOccupancy(droneId, date, duration) qui va se charger de recalculer le taux d'occupation du drone en fonction de la durée d'occupation à la date précisée et l'envoyer en base de données.

Enfin pour récupérer les statistiques il suffit d'appeler la méthode getOccupancyRate(droneId) de StatisticsCollector qui va renvoyer un taux d'occupation du drone.

B) Prise de recul

Le composant statistique permet actuellement de gérer uniquement le taux d'occupation des drones. Si l'on décide d'ajouter une statistique supplémentaire il suffira de fournir l'information par le biais de l'interface *StatisticsCreator* et il faudra également fournir la fonction de sortie dans *StatisticsCollector* ainsi le composant *Statistics* est un composant qui s'occupe entièrement du traitement des *Statistics* avec en entrée des données et en sortie des statistiques.

IV) Perspectives d'évolution

Certains points restent à améliorer dans notre structure. Nous pouvons notamment parler ces quelques éléments :

- La liaison entre *Schedule* et *DronePark* devrait être renforcée.

  Les informations de *Schedule* pour une heure X ne correspondent actuellement pas forcement à l'état d'un drone pour l'heure X. Si le *Schedule* indique qu'un drone doit être en charge à 10h, il ne l'est pas forcément dans la réalité.

- Modifier le processus d'arrivée des colis et le rendre asynchrone

  A l'aide d'un Message Driven Bean, on peut rendre asynchrone le processus d'arrivée des colis afin de ne pas bloquer le système pendant qu'il traite l'arrivée des colis. Et pourra permettre de notifier directement lorsque le drone revient.

- Améliorer les dépendances entre composants

  Nos objets métiers *Drone*, *Delivery* et *TimeSlot* dépendent beaucoup les uns des autres et créent une dépendance circulaire. Cette dépendance peut poser problèmes dans certains cas. De ce fait, nous ne sommes pas convaincus de l'approche actuelle des objets ou de leurs utilisations.

- Abstraire le côté drone.

  Le but ici est d'abstraire l'existence des drones pour que les composants qui communiquent avec *DronePark* ne dépendent pas de l'entité Drone et soient réutilisables quelque-soit le moyen de transport.

- Le client office ne doit pas être au courant de l'existence des drones

  Le *getplanning* demande un drone en paramètre pour afficher le planning de celui-ci. Une amélioration possible c'est de combiner les plannings de tous les drones en mettant juste des horaires disponibles et indisponible ou un horaire disponible a au moins un drone qui peut faire la livraison. Bien sûr comme *schedule* travaille avec un composant abstrait ceci ne poserait aucun problème.

- Ajouter la possibilité de se greffer à plusieurs API externe de transporteur grâce à une structure d'héritage.

ANNEXES

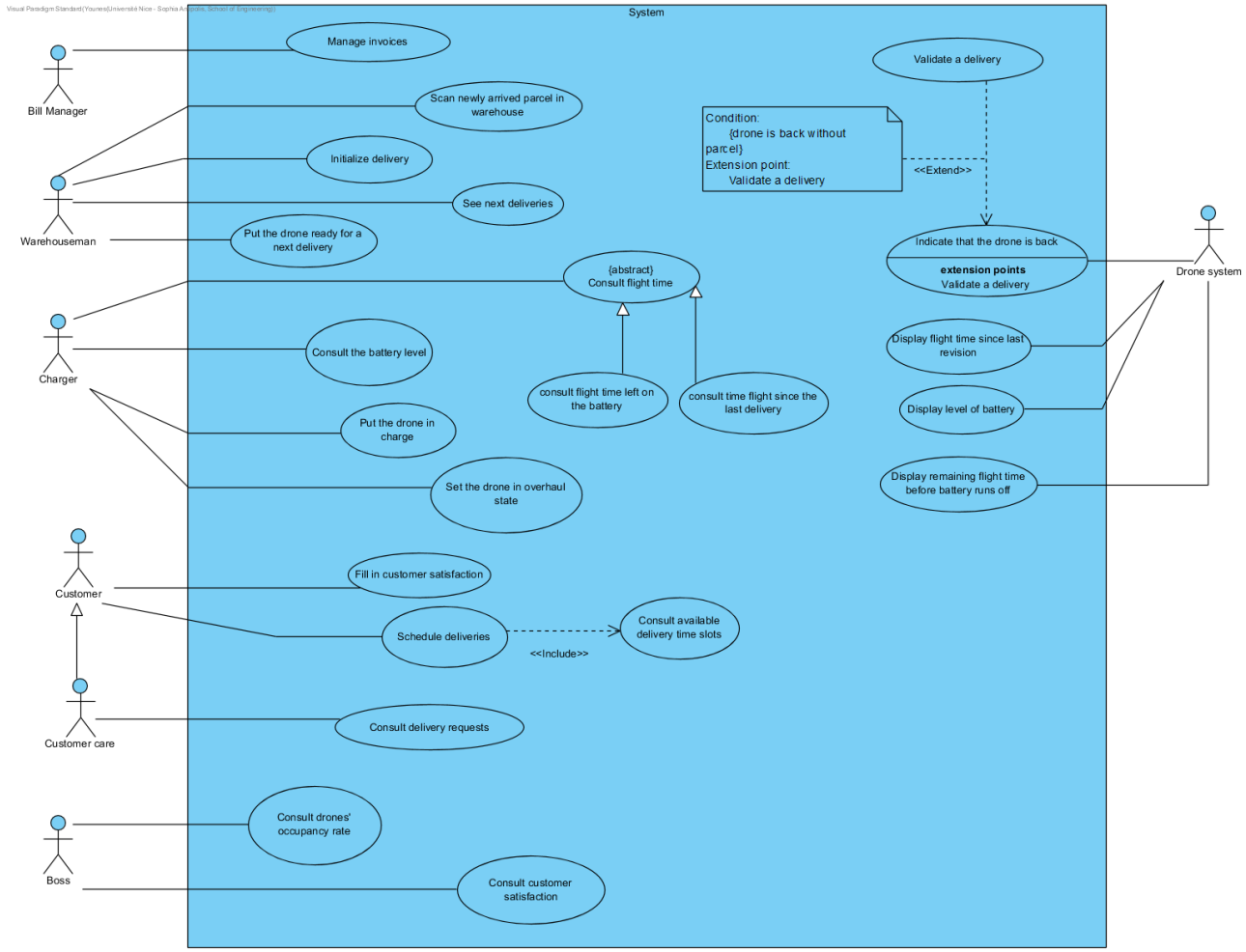

Figure annexe 1 : Diagramme de cas d'utilisation initial

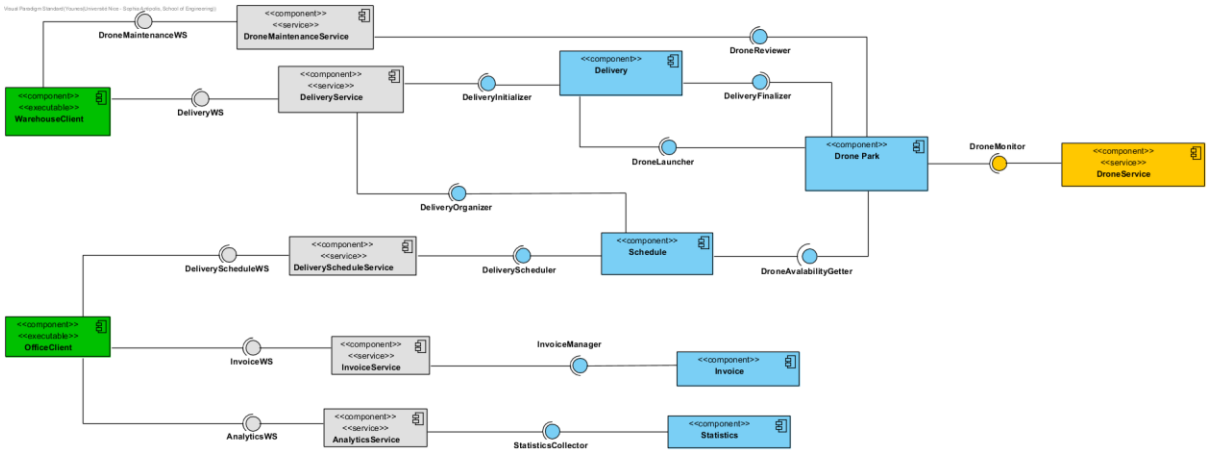

Figure annexe 2 : Diagramme de composants initial