

1 Système de gestion de parking

Le système gère un parking à barrières automatisées.

- Entrée : l'utilisateur arrive, prend un ticket à la borne d'entrée, ce qui ouvre la barrière. Le système enregistre l'heure d'entrée et l'identifiant du ticket.
- Paiement : avant de sortir, l'utilisateur insère le ticket dans une borne de paiement. Le système calcule le montant dû en fonction de l'heure d'entrée. L'utilisateur paie par carte bancaire. Une fois le paiement validé par le système de paiement externe, le ticket est validé.
- Sortie : l'utilisateur insère le ticket validé à la borne de sortie pour ouvrir la barrière.
- Maintenance : un technicien de maintenance doit pouvoir ouvrir les barrières manuellement en cas de panne et extraire un rapport des recettes journalières.

=> Identifier les acteurs du système.

=> Identifier les cas d'utilisation principaux de ce système.

=> Représenter ces acteurs et cas d'utilisation dans un diagramme de cas d'utilisation.

2 Application d'achat de billets de train

Considérons une application de réservation de billets de train. Des échanges avec différents utilisateurs de l'application nous permettent d'établir la description suivante :

- l'utilisateur se connecte
- l'application affiche sur l'écran "effectuer une réservation"
- après avoir cliqué sur le bouton, l'utilisateur est invité à saisir les villes de départ et d'arrivée
- l'autocomplétion des noms de ville affiche en priorité les villes déjà saisies par l'utilisateur
- l'utilisateur sélectionne ensuite une date de départ
- l'application calcule les trajets disponibles (horaires de départ/arrivée, type de train, prix)
- l'utilisateur sélectionne la proposition qui l'intéresse
- il est finalement invité à saisir ses informations de paiement pour procéder à celui-ci
- à tout moment l'utilisateur peut retourner sur l'application pour afficher sa liste de réservations et

télécharger le billet associé

- si les options de réservation le permettent, il peut annuler une réservation
- les utilisateurs ayant effectué plus de 15 achats accèdent aux status d’ambassadeur
- un ambassadeur a accès à des promotions exclusives via une page dédiée dans l’application
- il peut en sélectionner une et procéder au paiement de la même manière que pour une réservation classique

=> Représenter le système à l’aide de diagrammes de cas d’utilisation et de classes.

3 Gestion d’un cadastre

Nous souhaitons réaliser un modèle UML pour une application de gestion cadastrale d’un territoire. L’application doit permettre d’ajouter, supprimer ou découper une parcelle. Elle doit également permettre de fusionner plusieurs parcelles d’une commune. Une commune est découpée en sections cadastrales, qui peuvent elles-mêmes être redécoupées en sous-sections cadastrales, auxquelles sont rattachées les parcelles. Chaque parcelle est l’élément terminal de la hiérarchie et est toujours rattachée à une zone cadastrale (que ce soit une section ou une sous-section).

Pour identifier une parcelle, on concatène le code INSEE de la commune (ex : “58200”) avec le numéro de section (ex : “AD”) et avec celui de la parcelle (ex : 124). Le numéro de section (ex : “AD”) et le numéro de parcelle (ex : 124) sont des identifiants locaux uniques au sein de leur unité cadastrale parente.

L’application doit également permettre de calculer la taxe foncière due par chaque propriétaire d’une parcelle. Le calcul tient compte de la surface de la parcelle ainsi que du nombre et type de bâtiment présent sur celle-ci (bâtiments dits *légers* ou *en dur*).

=> Réalisez les diagrammes de cas d’utilisation et de classes pour l’application à développer.

=> Assurez-vous que votre modèle soit le plus générique possible. Il doit permettre d’intégrer des évolutions sans avoir à tout reprendre (par exemple ajouter des sous-sous-sections).

=> Illustrez votre diagramme de classes à l’aide d’un diagramme d’objets

4 Gestion d’un catalogue de symboles cartographiques

Nous développons une application qui doit gérer l’affichage d’un catalogue de symboles pour la cartographie. Une entité géographique (point, ligne ou polygone) est affichée sur la carte avec un style de rendu précis.

Chaque entité, par exemple, une rivière ou une route départementale, est caractérisée individuellement par :

- Un identifiant unique (ID) ;
- Sa géométrie.

Le style de rendu (couleur, épaisseur de trait, symbole de point) est cependant le même pour toutes les rivières d'un même jeu de données, et le même pour toutes les routes départementales. Ce style est défini par le cartographe et doit pouvoir être mis à jour dynamiquement dans le système (par exemple, changer la couleur de toutes les rivières d'un bleu clair à un bleu foncé).

Nous avons initialement deux types d'entités linéaires : Rivière et RouteDépartementale.

Le système doit permettre de calculer la longueur de chaque entité.

=> Proposez une modélisation UML (diagramme de classe) qui permette à une seule modification du style de mettre à jour toutes les entités de ce type sur la carte.

5 Station météo

Nous réalisons une mini station météo à l'aide d'un Raspberry Pi. Le montage consiste à connecter au Raspberry Pi un premier capteur de température, de pression et d'humidité et un second capteur de luminosité. Pour communiquer avec ces capteurs, le Raspberry Pi utilise le protocole I2C. L'ensemble des données sont collectées à l'aide d'un script `meteo.py` par le Raspberry Pi qui les envoie via sa connexion wifi sur Google Drive dans un document Google Sheet (tableur). Des courbes de tendances sont tracées à partir de ce classeur et exportées chaque semaine sur un poste bureautique classique.

=> Réalisez le diagramme de déploiement de la station météo.

6 Système de suivi GPS de flotte de véhicules

Modélisez un système de suivi de véhicules d'entreprise. Chaque véhicule est équipé d'un Boîtier GPS/4G. Ce boîtier exécute un fichier `firmware_tracker.bin`. Il envoie les positions via le réseau 4G à un serveur central. Le serveur exécute un programme en Java (`ServiceDeFlotte.jar`) et stocke les données dans une base de données. Un opérateur utilise un poste de supervision pour se connecter au serveur et monitorer la position des véhicules.

=> Réalisez le diagramme de déploiement de ce système.

7 Recette de la mousse au chocolat

Proposez un diagramme d'activité pour la recette ci-dessous. Optimisez-le en supposant être assez nombreux pour pouvoir paralléliser au maximum les différentes tâches.

Commencer par casser le chocolat en morceaux, puis le faire fondre. En parallèle, casser les oeufs en séparant les blancs des jaunes. Quand le chocolat est fondu, ajouter les jaunes d'oeuf. Battre les blancs en neige jusqu'à ce qu'ils soient bien fermes. Les incorporer délicatement à la préparation chocolat sans les briser. Verser dans des ramequins individuels. Mettre au frais au moins 3 heures au réfrigérateur avant de servir.

8 Application de navigation GPS

Nous souhaitons réaliser une application de navigation GPS. Cette application doit permettre à l'utilisateur d'afficher une carte. Lorsque la position de la voiture est détectée, la carte sera centrée cette position. L'application doit également permettre d'afficher la vitesse de déplacement de la voiture. Enfin l'application est utilisée avant tout pour guider le conducteur (calculer et afficher des itinéraires).

Pour calculer la position de la voiture, le GPS doit recevoir les signaux d'au moins 4 satellites.

1. Réalisez le diagramme de cas d'utilisation pour ce système.
2. Proposez un diagramme d'activité pour l'activité de guidage du conducteur.
3. Concevez un diagramme de classes permettant de modéliser le système.
4. Complétez votre diagramme d'activité en précisant les entités du système responsable des différentes activités.

9 Réveille-matin

Considérons un réveille-matin simplifié :

- il est possible d'armer ou désarmer l'alarme ;
- quand l'heure courant devient égale à l'heure de l'alarme, le réveil sonne ;
- il est possible d'interrompre la sonnerie ;
- la sonnerie s'interrompt automatiquement après 30 minutes.

Un réveil matin dispose des opérations `armer(heure)`, `desarmer()`, `sonner()`, `arret_sonnerie()` et `heure_courante()`. Il dispose d'un attribut `heure_alarme`.

=> Proposez un diagramme d'états-transitions pour modéliser la dynamique de ce réveille-matin.

10 Cohérence de diagrammes de classes et de séquence

Construisez un diagramme de classe cohérent avec le diagramme de séquence ci-dessous.

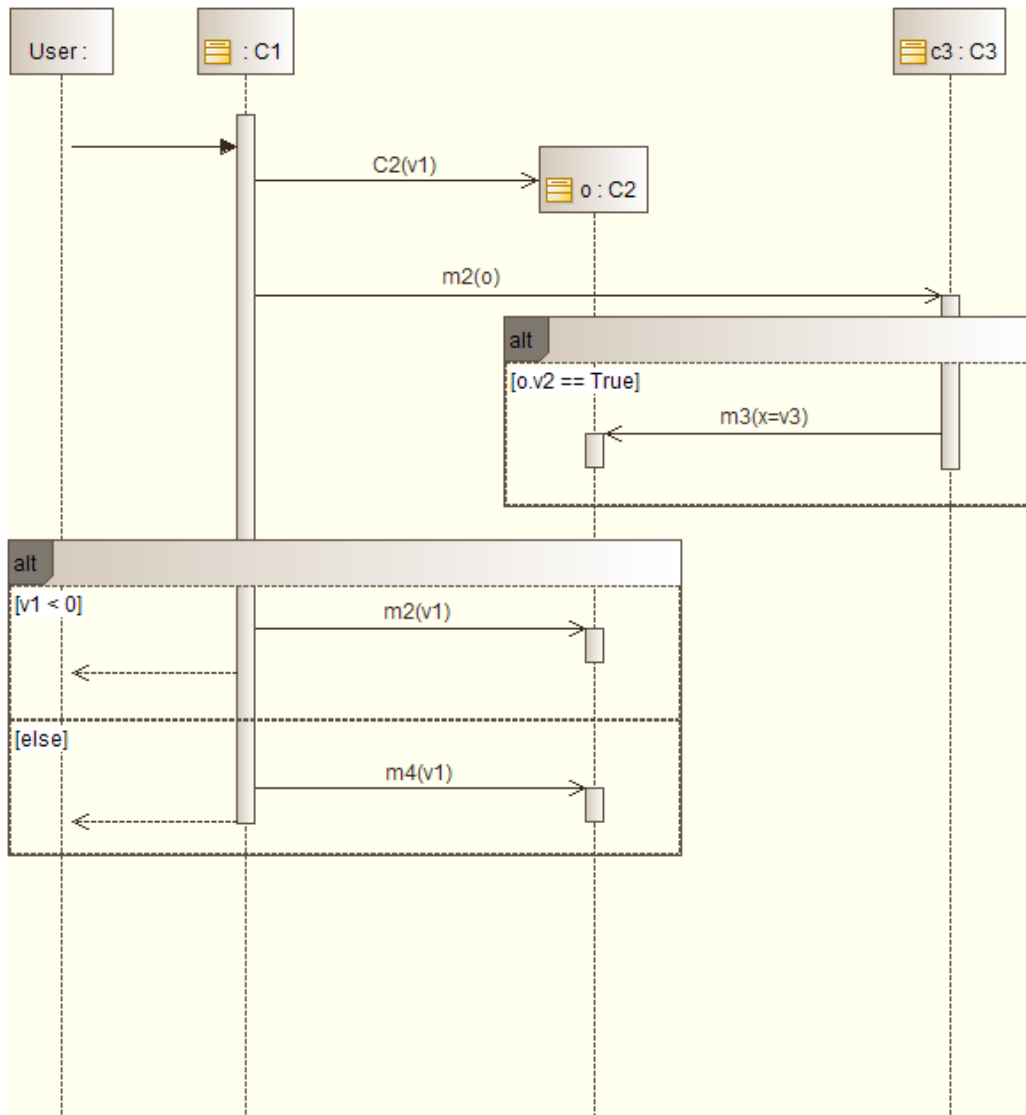


FIGURE 1 – Diagramme de séquence

11 Démineur

Nous nous proposons dans ce sujet de modéliser un jeu de démineur. Le but de ce jeu est de trouver le plus rapidement possible, sans les toucher, toutes les cases d'une grille contenant des mines.

Nous nous contenterons ici d'une version simplifiée du jeu, sans chronomètre. Notre jeu est donc composé

d'un compteur de mines cachées et d'une grille rectangulaire, la grille étant un assemblage de cases. Au début de la partie toutes les cases du plateau sont cachées. A chaque tour, le joueur peut afficher le contenu d'une case. Son contenu peut être : rien, une mine ou un nombre indiquant le nombre de mines dans les cases voisines. Il peut également signaler la position d'une mine en posant un drapeau sur une case (les drapeaux peuvent être enlevés à n'importe quel tour).

Avant de commencer une partie, le joueur peut configurer la partie (nombre de mines et taille de la grille).



FIGURE 2 – Interface du jeu du démineur

Plusieurs scénarios sont possibles en fonction du contenu d'une case découverte :

- une mine : le joueur a perdu, la partie est terminée ;
- un chiffre : il ne se passe rien ;
- case vide : toutes les cases voisines sont dévoilées, sauf celles signalées par un drapeau. Si une des cases voisines ne contient rien, le processus continue à partir de cette case.

Lorsqu'une case est marquée à l'aide d'un drapeau, le compteur décrémente le nombre de mines de un. Une case marquée d'un drapeau ne peut pas être découverte (impossible d'afficher son contenu).

1. Réalisez le diagramme de cas d'utilisation pour notre jeu de démineur.
2. Réalisez un diagramme d'activité ou de séquence pour chaque cas d'utilisation.
3. Ecrivez un diagramme de classe pour le jeu de démineur.
4. Sans chercher à les développer, indiquez quelle(s) autre(s) diagramme(s) UML pourrai(en)t être intéressant(s) pour compléter la modélisation (justifiez votre réponse).

Pour chacun des diagrammes, vous n'oublierez pas d'expliquer vos diagrammes et de justifier les choix effectués.

12 VéliDescartes

La Cité Descartes souhaite mettre en place un système de vélo en libre-service.

Une dizaine de stations doivent être matérialisées sur le campus par les équipes de maintenance. Elles constitueront les seuls lieux où les utilisateurs pourront emprunter et déposer des vélos.

Les utilisateurs du système interagiront avec celui-ci via une application mobile (iOS ou Android).

Avant de pouvoir emprunter un vélo, l'utilisateur devra s'abonner au service. Plusieurs types d'abonnements seront possibles à l'ouverture du service : mensuel ou annuel. Les utilisateurs devront s'acquitter d'une cotisation dont le montant dépendra du type d'abonnement. Une fois cet abonnement payé, les locations seront gratuites.

Un utilisateur abonné peut emprunter un vélo à n'importe quelle station et le retourner dans n'importe quelle autre. Il ne peut emprunter qu'un seul vélo à la fois. Lorsqu'il retourne un vélo, l'abonné peut signaler un problème sur le vélo (pneu crevé, roue voilée, selle manquante...).

Avant d'autoriser l'emprunt d'un vélo, le système s'assurera donc notamment que : - l'utilisation n'a pas d'autre location en cours ; - l'état du vélo permet un emprunt ; - le vélo choisi est bien situé dans le périmètre d'une station.

Les administrateurs du système, qui se connectent via une interface web classique, doivent pouvoir savoir à chaque instant si un vélo est en cours d'utilisation ou s'il est garé. De même ils doivent pouvoir savoir combien de vélo en état de fonctionnement sont disponibles à une station.

1. Proposez une modélisation du système. Votre réponse pourra inclure une représentation des grandes fonctionnalités du système, avec un focus plus important sur certaines fonctionnalités sensibles. Elle pourra également détailler les structures du programme à réaliser, leurs interactions et/ou leur implantation sur les différents composants matériels du système.

13 EncherePasChere

Une société souhaite mettre en place un site web de ventes aux enchères nommé *EncherePasChere* permettant aux utilisateurs d'acheter et vendre n'importe quel objet à travers le monde. Vous avez la charge de réaliser la phase d'analyse du système. Le cahier des charges est le suivant :

Un visiteur anonyme doit pouvoir parcourir le site web d'*EncherePasChere* sans être inscrit. Il faut en

revanche être inscrit pour acheter ou vendre des objets. Le site propose deux types de compte : acheteur “simple” et acheteur-vendeur. Pour s’inscrire, un acheteur doit renseigner ses informations personnelles (nom, prénom, date de naissance, adresse, adresse mail) et choisir un login et un mot de passe. Et pour ce qui est du compte vendeur, *EncherePasChere* propose deux façons d’en ouvrir un :

- en indiquant, via un formulaire sécurisé sur internet, le numéro de compte sur lequel doivent être virés les résultats des ventes ;
- une méthode par courrier postal en joignant un RIB qui prend 2 à 3 jours et permet de recevoir un code de confirmation par courrier au domicile du vendeur.

Le site web *EncherePasChere* propose deux moyens aux utilisateurs pour trouver un objet :

- recherche par catégories : depuis la page d’accueil du site, il est possible de naviguer dans des catégories et sous-catégories d’objets afin de trouver celui recherché ;
- recherche par mot-clé : depuis la page d’accueil du site, il est possible de taper un mot-clé dans un moteur de recherche, les objets correspondants au mot clé étant alors présentés.

Dans tous les cas, les résultats des recherches sont affichés par ordre chronologique inverse de mise en vente (le plus récent en premier).

Lorsqu’un utilisateur a trouvé l’objet de ses rêves, il peut en fonction du format de vente, et s’il possède un compte acheteur, soit enchérir dessus, soit l’acheter directement. S’il enchérit, l’acheteur saisit le montant de son enchère initiale et le montant maximal qu’il est prêt à déboursier pour cet objet. *EncherePasChere* avertira le vendeur si une autre enchère, inférieure à son seuil maximal, est déposée.

A la fin d’une vente (qu’elle soit directe ou par enchère), le vendeur dispose d’une dernière possibilité de refuser la vente. L’absence de réponse 24h après la fin de la vente vaut acceptation.

Si elle est acceptée, l’acheteur doit régler le montant de la vente, soit en effectuant un paiement par carte bleu, soit en utilisant le solde disponible sur son compte acheteur-vendeur s’il en possède un. Le vendeur dispose quand à lui d’un délai de 72h pour expédier le produit ou pour faire accepter une solution de livraison à l’acheteur. Passé ce délai l’acheteur peut effectuer une réclamation auprès d’*EncherePasChere*.

A la réception du colis (confirmation à effectuer en ligne), acheteur et vendeur peuvent s’évaluer réciproquement (note de 0 à 5 étoiles).

Pour vendre un objet, le vendeur (connecté avec un compte acheteur-vendeur) doit remplir un formulaire de mise en vente qui contient :

- un titre clair et accrocheur ;
- une description complète indiquant aussi bien les qualités que les défauts pour éviter les questions et les litiges ;
- une photo fidèle de l’objet ;
- un prix attractif ;
- dans le cas d’une vente par enchère, la durée de la vente (3, 5, 7 ou 10 jours) en pensant à inclure un week-end pour toucher les acheteurs du week-end ;
- les catégories correspondantes à l’objet ;
- les mots-clés correspondants à l’objet.

14 Corrections

14.1 Système de gestion de parking

1. Identification des acteurs : conducteur, technicien de maintenance et système de paiement externe
2. Identification des cas d'utilisation
 1. Entrer dans le parking : gérer l'émission du ticket et l'ouverture d'une barrière d'entrée.
 2. Payer le stationnement : calculer la durée/le coût et interagir avec la borne de paiement.
 3. Sortir du parking : valider le ticket payé et ouvrir une barrière de sortie.
 4. Ouvrir une barrière manuellement : permettre au technicien de forcer l'ouverture d'une barrière (entrée ou sortie).
 5. Éditer rapport de recettes : fournir un bilan des transactions de paiement au technicien.

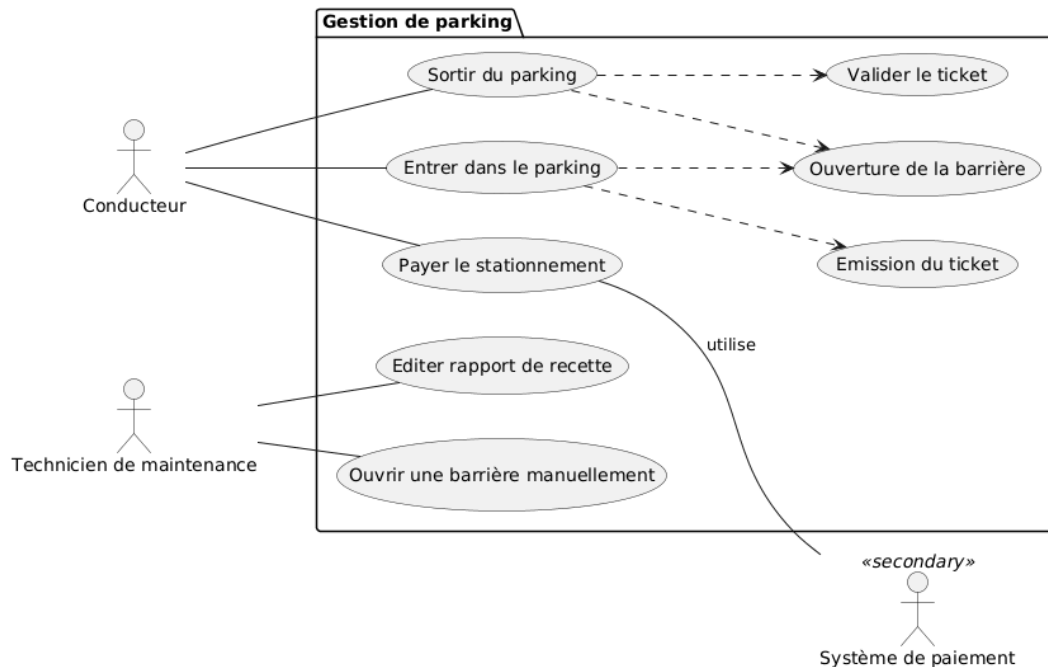


FIGURE 3 – Diagramme de cas d'utilisation - Gestion d'un parking

14.2 Application d'achat de billets de train

Le système étudié est une application d'achat de billet de train. Deux acteurs sont identifiés : le client et l'ambassadeur. L'ambassadeur étant un client particulier. Les trois cas d'utilisation identifiés sont pour les clients : effectuer une réservation, afficher la liste des réservations et annuler une réservation. L'ambassadeur peut effectuer une action additionnelle : afficher les promotions exclusives. Les cas d'utilisation "effectuer une réservation" et "afficher les promotions" partagent un sous-cas d'utilisation "procéder au paiement".

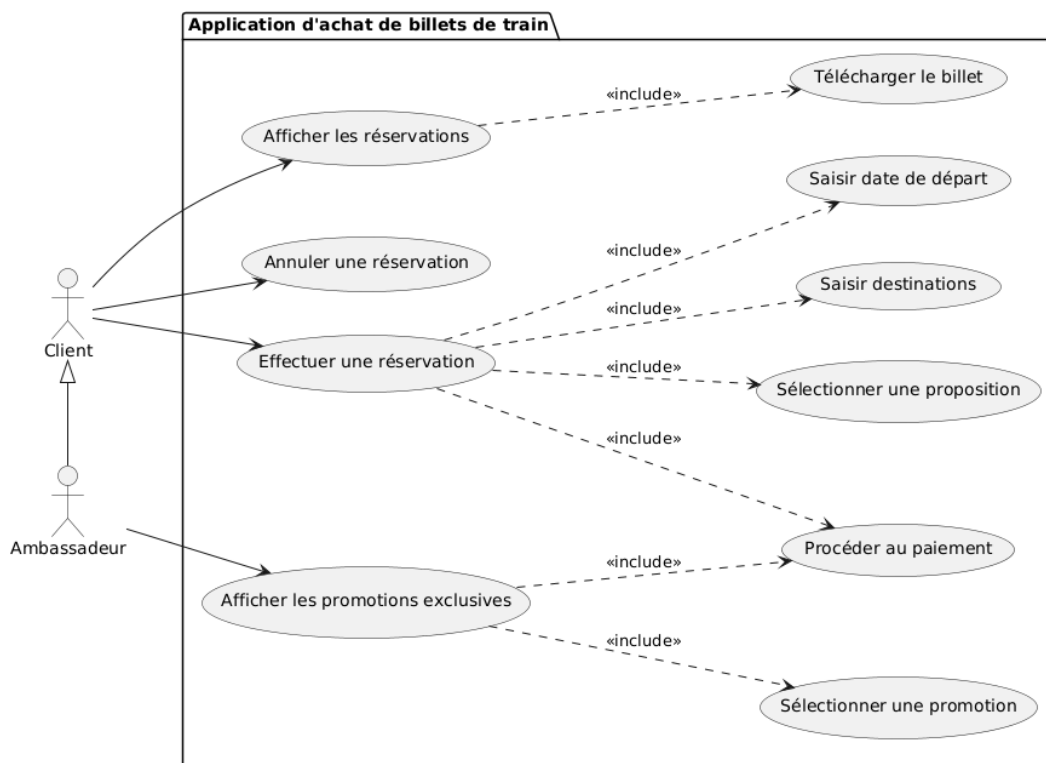


FIGURE 4 – Diagramme de cas d'utilisation - Application d'achat de billets de train

Le premier problème qui se pose lors de la modélisation est de représenter le (les) lien(s) entre trajets et gares : un trajet a une gare de départ, une d'arrivée et peut effectuer des escales dans d'autres gares. On utilisera des liens 1-* pour représenter les liens pour les gares de départ et d'arrivée. Pour les escales, on optera pour un lien -, qui pourrait être complété d'une classe d'association (avec les horaires de l'escale par exemple).

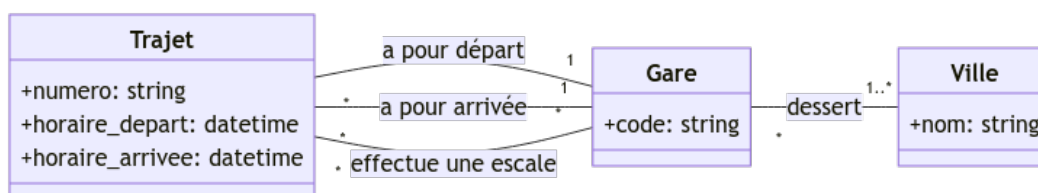


FIGURE 5 – Diagramme de classes - Application d'achat de billets de train

La partie du modèle traitant de la partie réservation (une réservation concerne un trajet) et ensuite ajoutée. Elle utilise trois classes : *Client* (celui qui effectue une réservation), *Reservation* (pour faire le lien entre

le client et le trajet) et *Paiement* qui compose une réservation.

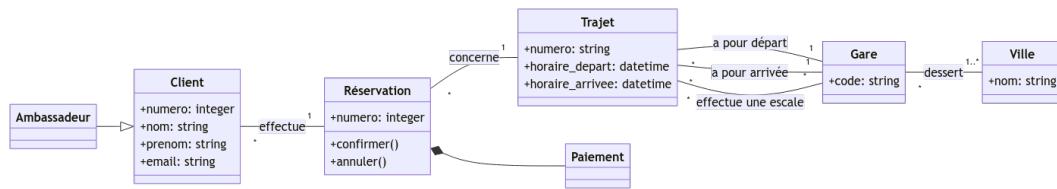


FIGURE 6 – Diagramme de classes - Application d'achat de billets de train

14.3 Gestion du cadastre

Le diagramme de cas d'utilisation comporte deux grandes fonctionnalités : gérer les parcelles et calculer les impôts. La fonctionnalité de gestion des parcelles est décomposée en plusieurs fonctionnalités optionnelles qui correspondent aux différentes tâches exprimées dans l'énoncé.

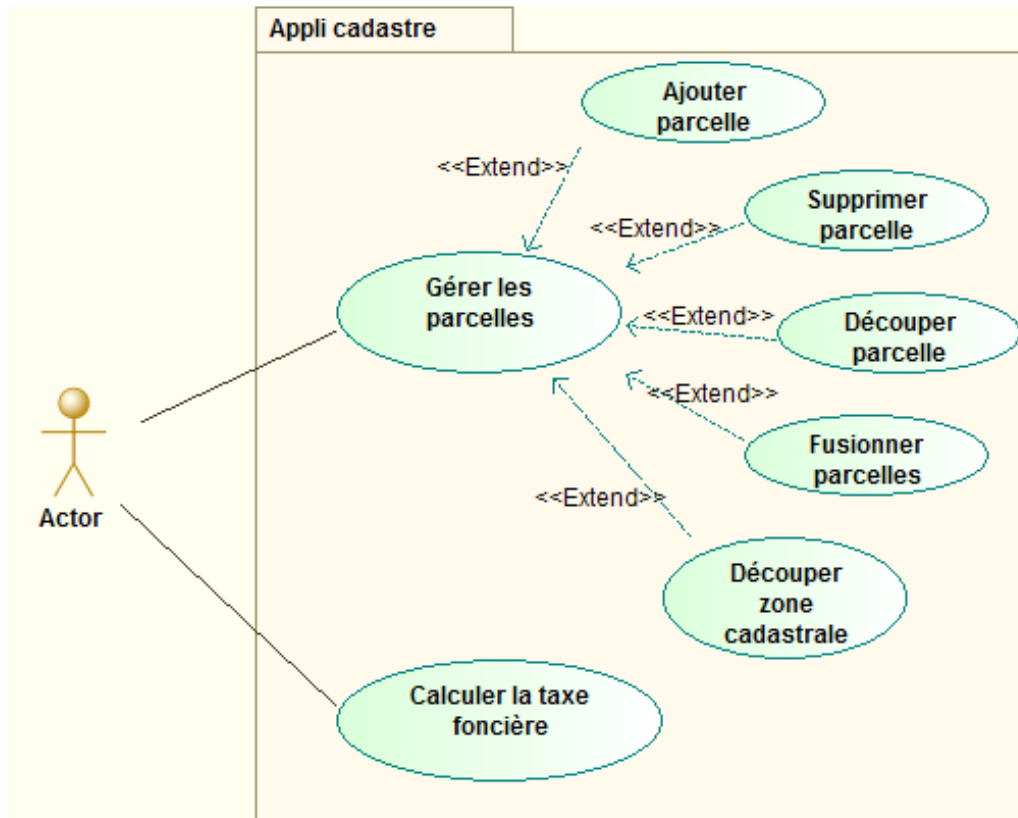


FIGURE 7 – Diagramme de cas d'utilisation - Cadastre

Dans une première approche, nous pouvons proposer le diagramme de classes suivant où nous utilisons des liens d'agrégation entre *Parcelle* et *Section*, et entre *Section* et *Commune*. S'il faut tenir compte de sous-sections, nous pouvons ajouter une classe entre les classes *Section* et *Parcelles*.

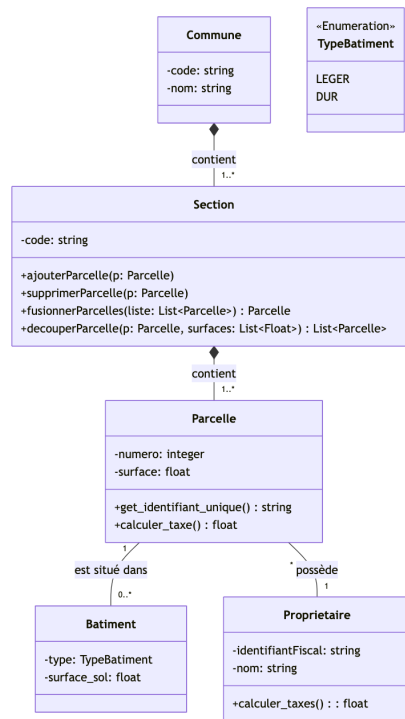


FIGURE 8 – Diagramme de classes v0 - Cadastre

Mais cette version ne répond pas à la problématique de genericité demandée : il est nécessaire d'ajouter de nouvelles classes à chaque sous-niveau que nous voulons prendre en compte. Pour faire évoluer le modèle, nous proposons donc d'introduire une nouvelle classe **ZoneCadastrale** qui possède trois attributs `code`, `niveau` et `nom`, qui représentent respectivement l'identifiant de la zone, son numéro de niveau (1 pour une commune, 2 pour une section, 3 pour une sous-section. ...) et un nom optionnel pour la zone (ex : le nom de la commune).

Par ailleurs, nous indiquons qu'une zone cadastrale est une composition de zones cadastrales. De cette manière, nous conservons les relations de type *une commune est composées de sections, une section est composée de sous-sections*, etc.

Cette modélisation nous permet de reconstruire une hiérarchie dans les différentes zones cadastrales. Le niveau 1 est le plus haut : c'est celui qui contient les communes. Le niveau 2 est composé des éléments contenus dans ceux de niveau 1 (les sections), etc.

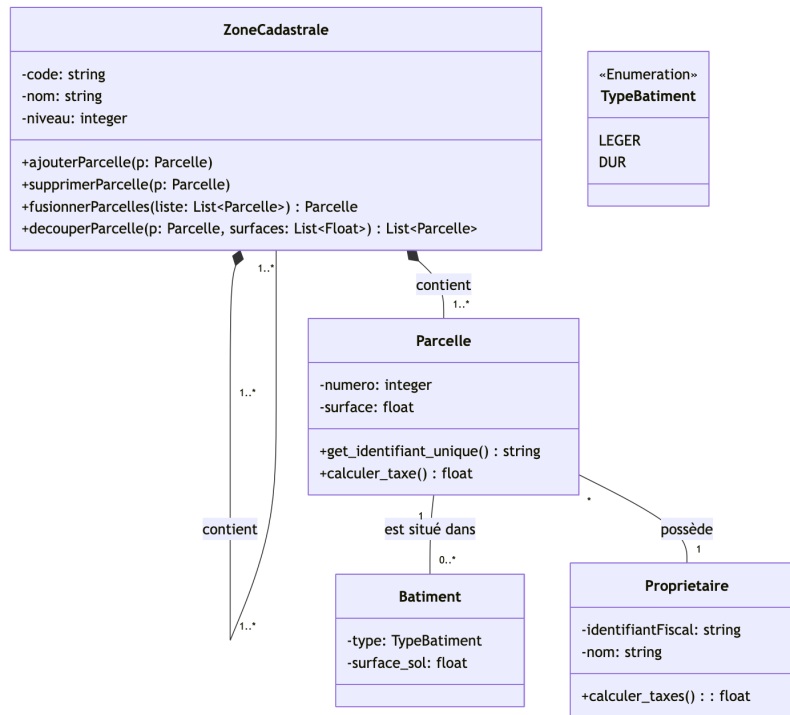


FIGURE 9 – Diagramme de classes - Cadastre

Dans cette deuxième version, nous avons également fait apparaître les méthodes pour manipuler des parcelles depuis une zone cadastrale. Nous faisons le choix de faire porter ces méthodes par la classe zone cadastrale pour représenter le fait qu’une parcelle appartient à une zone cadastrale. D’autres modélisations pourraient ajouter ces méthodes dans une classe “gestionnaire”, ou encore laisser le programme créer les objets `Parcelle` quand il en a besoin.

Des méthodes pour calculer les taxes sont également ajoutées dans les classes `Parcelle` et `Proprietaire`.

Pour s’assurer que le sens de notre modélisation est bien compris, nous utilisons un diagramme d’objets qui illustrera un exemple de situation.

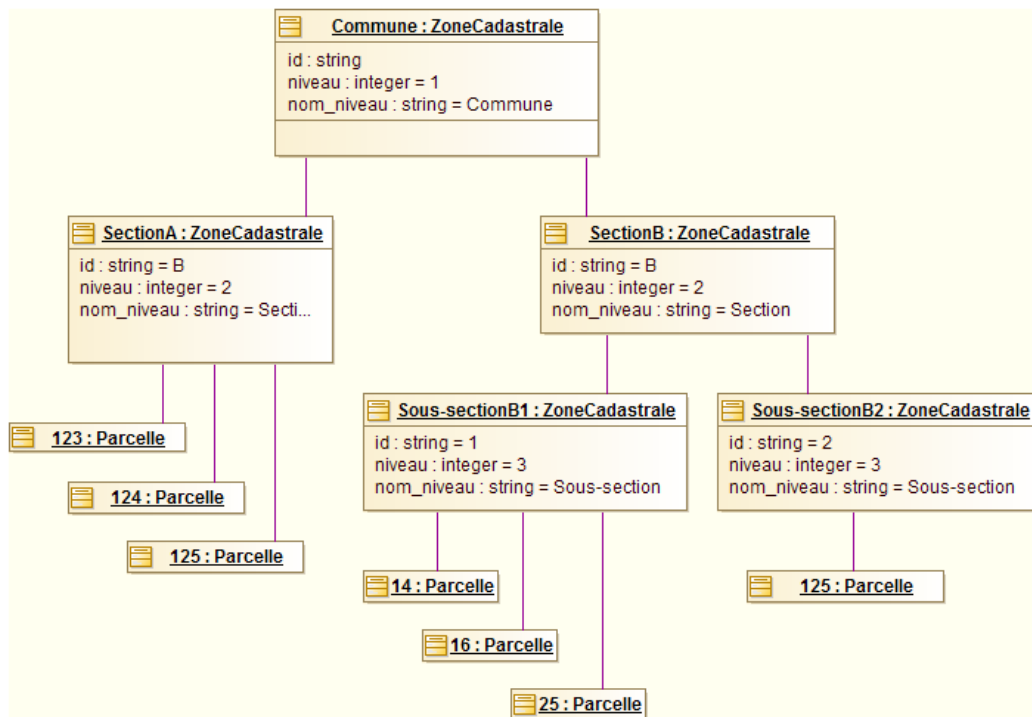


FIGURE 10 – Diagramme d'objets - Cadastre

14.4 Gestion d'un catalogue de symboles cartographiques

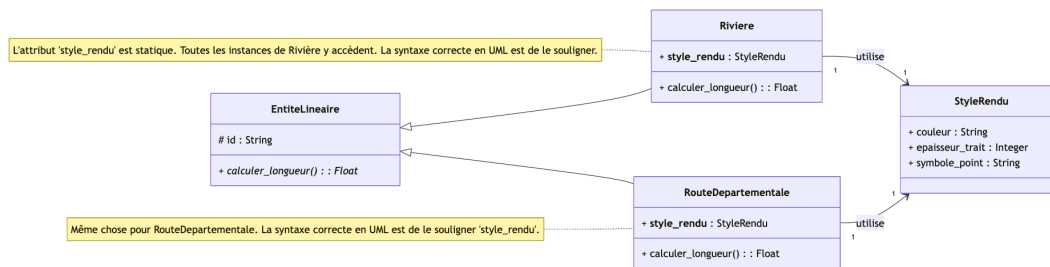


FIGURE 11 – Diagramme de classes - Gestion d'un catalogue de symboles cartographiques

L'objectif de cette modélisation était de représenter, dans un diagramme de classes UML, la situation où toutes les instances d'une classe spécifique (comme **Riviere** ou **RouteDepartementale**) partagent exactement le même objet de spécification de style (**StyleRendu**), sans qu'il soit nécessaire de dupliquer cet objet pour chaque entité individuelle.

La solution consiste à utiliser un attribut de classe (aussi appelé attribut statique), qui est la meilleure pratique UML pour désigner une donnée partagée au niveau de la classe elle-même, et non de l'instance.

Concrètement, nous avons opté pour la modélisation suivante : * isolation de l'ensemble des attributs de style dans une classe **StyleRendu** * ajout de l'attribut `style_rendu` dans les classes **Riviere** et **RouteDepartementale** . * pour indiquer que cet attribut est un attribut de classe (donc partagé par l'ensemble des instances), la notation UML standard consiste à le souligner (ce que le logiciel utilisé ne supporte pas). C'est le marqueur graphique le plus important de cette solution.

14.5 Station météo

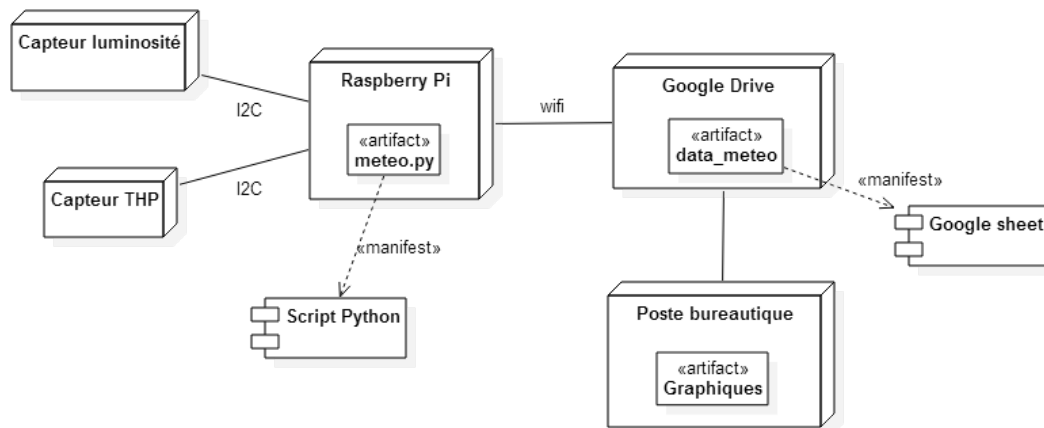


FIGURE 12 – Diagramme de déploiement - Station météo

14.6 Recette de la mousse au chocolat

Pour optimiser la réalisation de la recette, nous parallélisons toutes les tâches qui ne concernent pas les mêmes ingrédients.

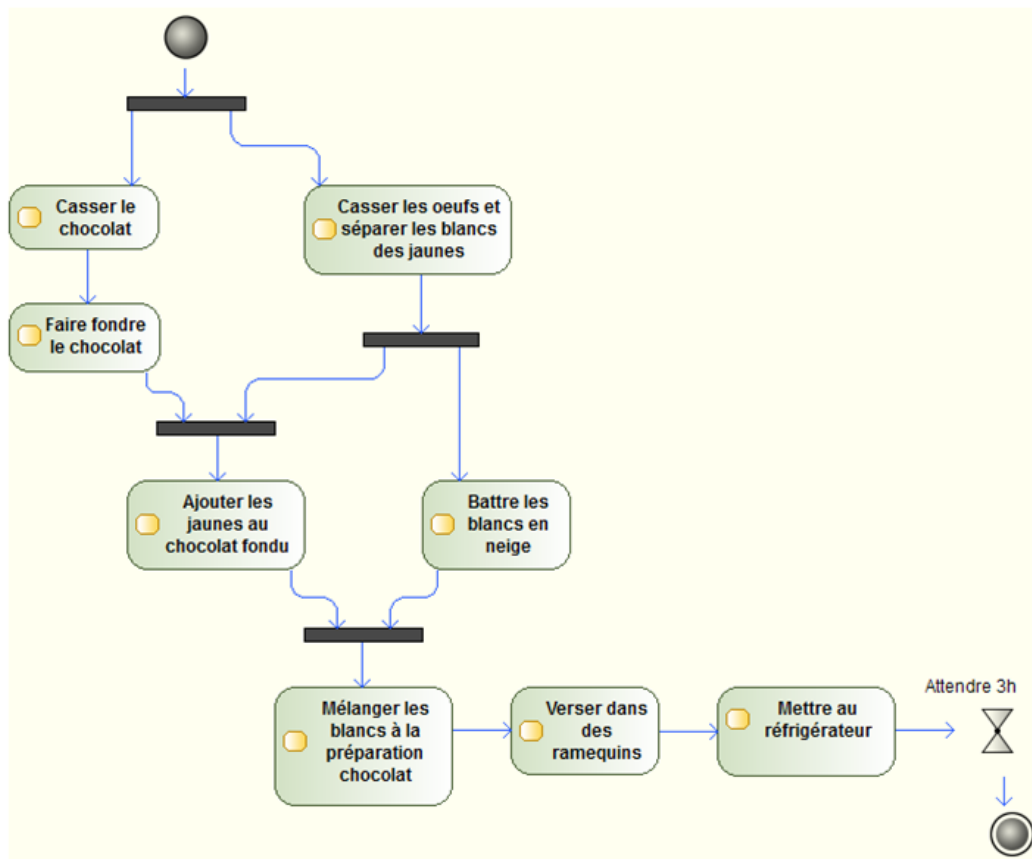


FIGURE 13 – Diagramme d'activité - Recette de la mousse au chocolat

14.7 Application de navigation GPS

Le système étudié est une application de navigation GPS.

Nous identifions trois cas d'utilisation principaux :

- afficher une carte ;
- afficher la vitesse de déplacement ;
- guider le conducteur, ce qui inclut de calculer et d'afficher des itinéraires.

Pour calculer la vitesse de déplacement et des itinéraires, le GPS a besoin de détecter la position de la voiture, ce qui fait appel à des satellites (acteur secondaire du système). La détection de la position de la voiture est une option de la fonctionnalité d'affichage de la carte.

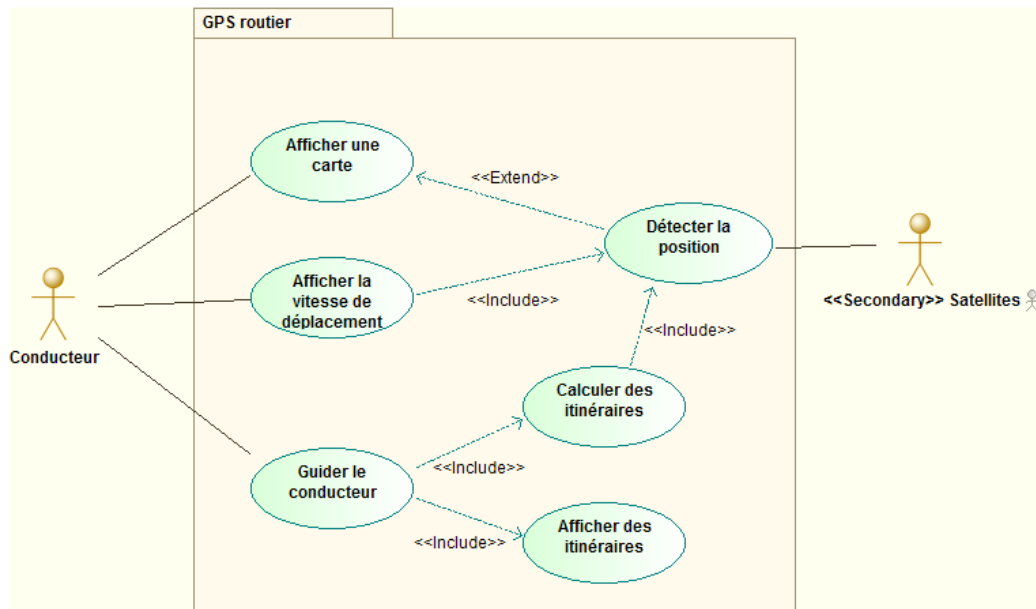


FIGURE 14 – Diagramme de cas d'utilisation - GPS routier

Lors de la réalisation de l'activité *Etre guidé*, le conducteur commence par allumer le GPS. Celui-ci cherche alors à détecter les satellites pour pouvoir calculer la position du véhicule. Le conducteur peut alors indiquer sa destination pour que l'application puisse calculer un itinéraire. Lorsque l'itinéraire est calculé, il s'affiche dans l'application et plusieurs cas sont alors possibles :

- soit le conducteur suit correctement la route indiquée : l'application lui indique alors simplement la route à suivre jusqu'à arriver à destination
- soit le conducteur ne respecte pas l'itinéraire indiqué : l'application recalcule alors un nouvel itinéraire en tenant compte de la nouvelle position et met à jour l'affichage de la route à suivre.

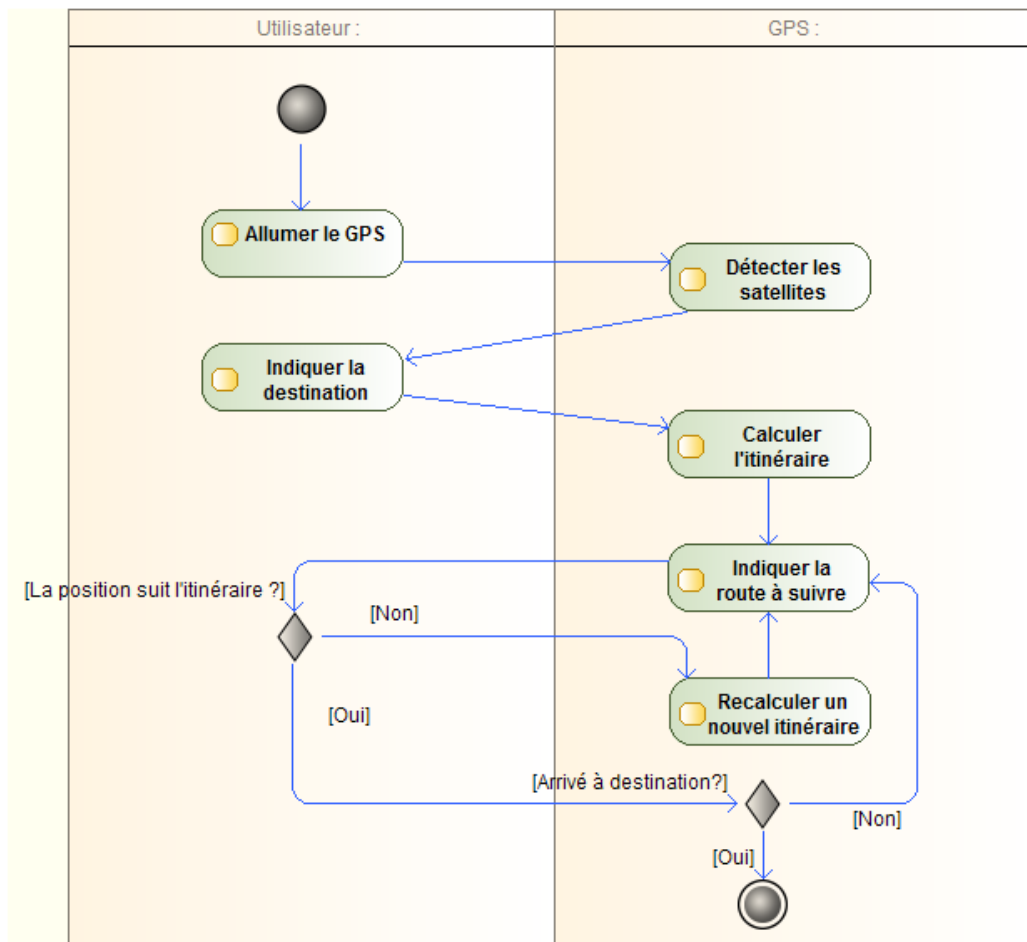


FIGURE 15 – Diagramme de l'activité *être guidé* - GPS routier

Notre analyse du système nous conduit pour commencer à identifier trois structures d'objets contribuant à son fonctionnement : l'adresse de destination, l'itinéraire et la position GPS de la voiture.

L'adresse de destination est traduite en position (coordonnées x,y) pour être utilisée dans le système. Un itinéraire est quand à lui composé d'une suite ordonnée de positions. Nous ajoutons donc cette classe à notre modèle et définissons la position GPS de la voiture comme un type particulier de position.

Enfin, nous ajoutons une classe affichage qui utilise la position GPS pour centrer la carte dessus et l'itinéraire pour le représenter sur le fond de carte et guider le conducteur.

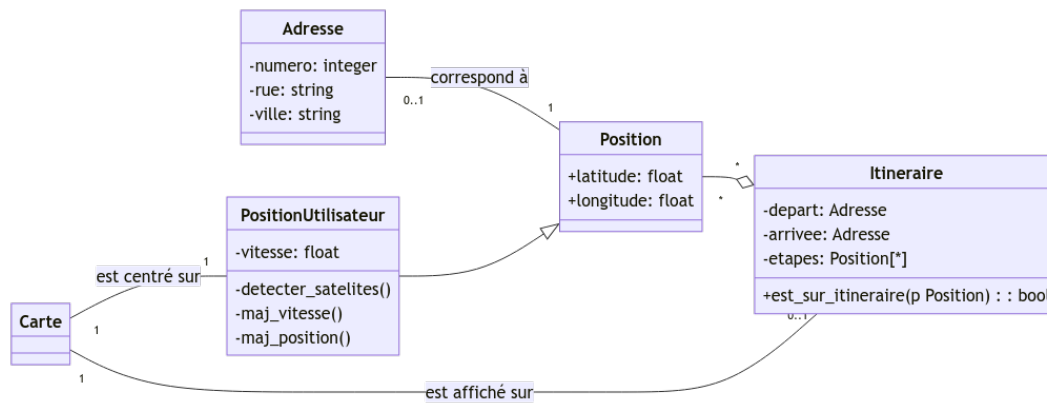


FIGURE 16 – Diagramme de classes - GPS routier

14.8 Réveille-matin

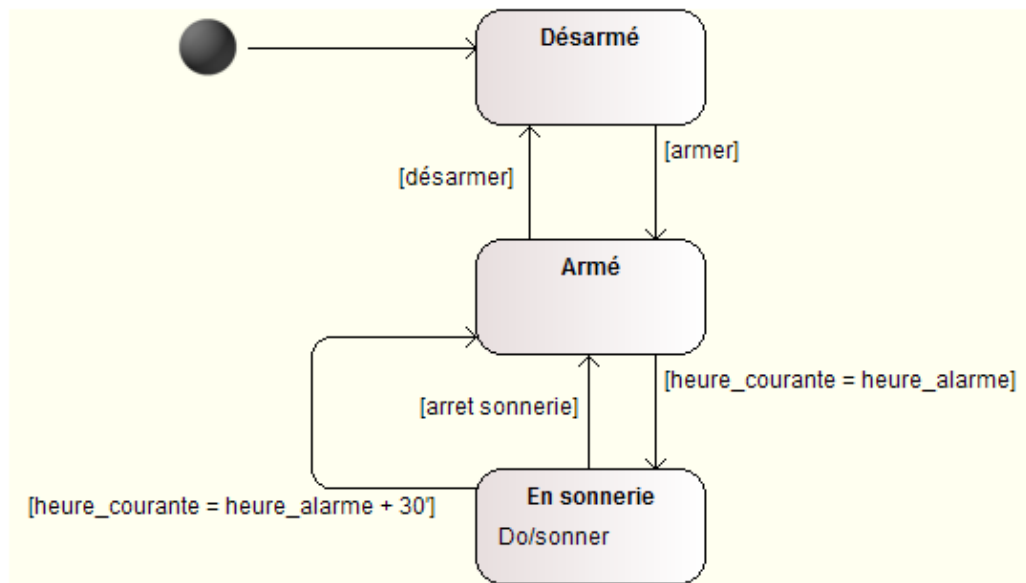


FIGURE 17 – Diagramme d'états-transitions

14.9 Cohérence de diagrammes de classes et de séquence

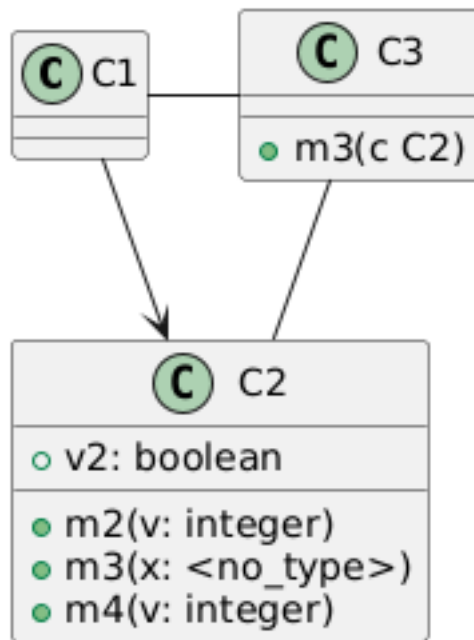


FIGURE 18 – Diagramme de séquence

14.10 Démineur

Nous identifions deux fonctionnalités principales : configurer une partie et jouer une partie. *Configurer une partie* signifie choisir le nombre de cases et/ou le nombre de mines. Quant au cas d'utilisation *Jouer une partie*, il comprend les sous-fonctionnalités *poser un drapeau* et *découvrir une case*. Comme il n'est pas obligatoire de poser des drapeaux pour pouvoir jouer cette sous-fonctionnalité est optionnelle. En revanche il est obligatoire de découvrir au moins une case à un moment de la partie (sinon elle ne se termine jamais) : cette fonctionnalité est donc obligatoire.

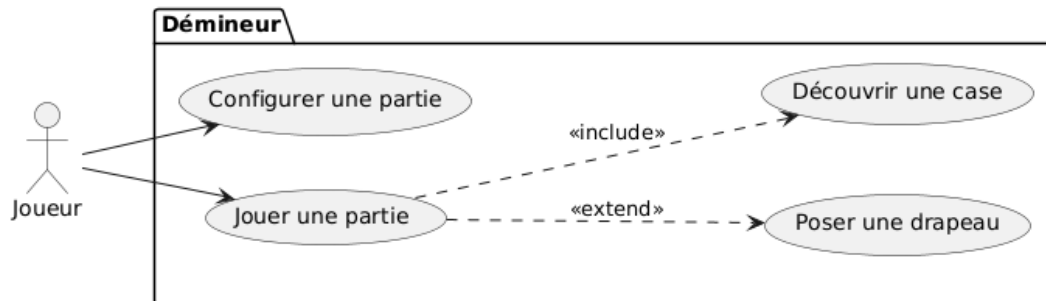


FIGURE 19 – Diagramme de cas d'utilisation du démineur

Les trois grandes classes que nous identifions sont **Partie**, **Grille** et **Case**. Une partie se joue sur une grille qui est composée de cases. Dans la classe **Partie**, nous prévoyons des attributs `nb_mines_initiales` contenant le nombre de mines que le joueur souhaite inclure dans sa grille et `taille_grille` qui renseignera la taille de la grille utilisée. Nous ajoutons également un attribut dérivé `nb_mines_restantes` qui nous permettra de déterminer si la partie est gagnée. Enfin la classe **Partie** contient des méthodes permettant de jouer (marquer et découvrir une case).

Le plateau est composé de cases qui sont positionnées à l'aide de leur numéro de ligne et de colonne. Les cases peuvent être de trois types différents : minées, vides ou numérotées (lorsqu'elles sont à côté d'une case minée). Le type de la case influe sur le comportement de la méthode `decouvrir()` : respectivement partie perdue, démasquage des cases voisines et démasquage de la seule case. Le comportement des méthodes de la classe **Case** dépend également de l'état de celle-ci : il ne se passe pas la même chose lorsque l'on essaye de marquer une case déjà marquée ou une case découverte par exemple.

Le diagramme de classe que nous avons établi est le suivant :

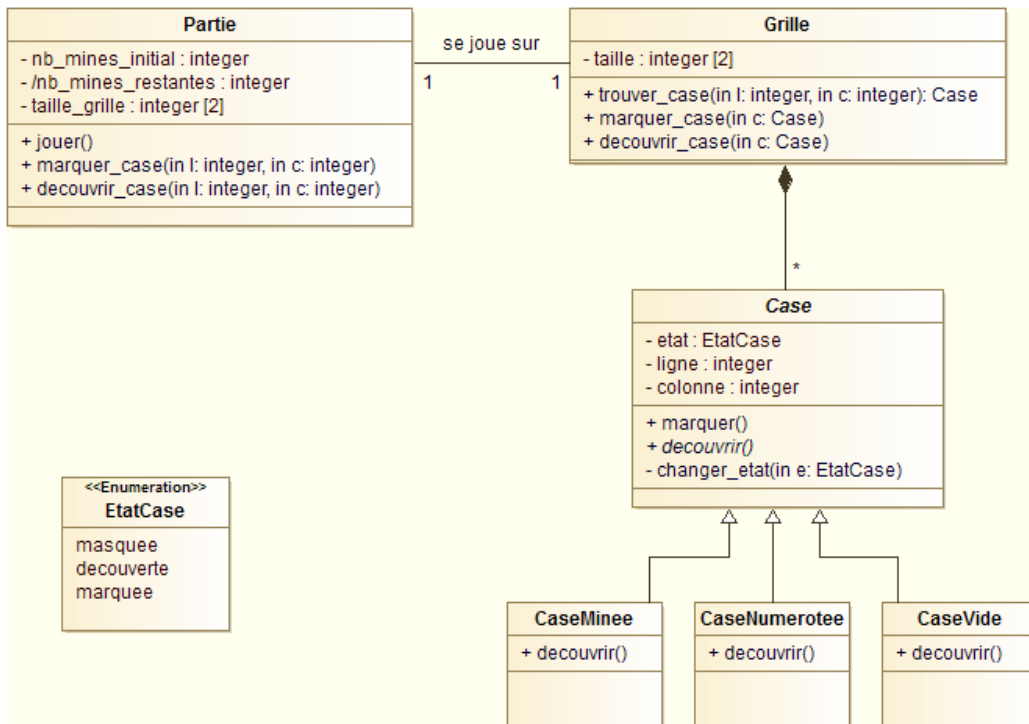


FIGURE 20 – Diagramme de classes du démineur

Pour compléter la modélisation, nous décrivons ensuite le cas d'utilisation *Découvrir une case* à l'aide d'un diagramme de séquence. Ce diagramme est intéressant ici car les objets interagissent fortement entre eux.

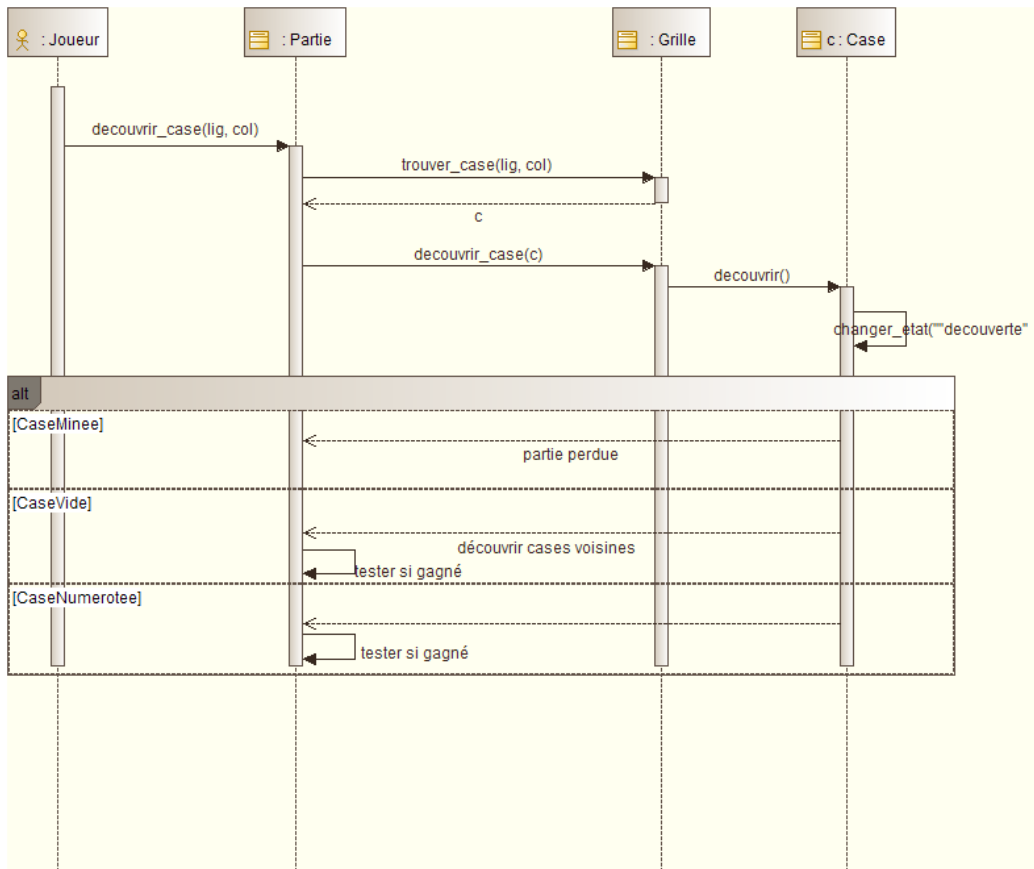


FIGURE 21 – Diagramme de séquence - Découvrir une case

De même, nous utilisons un diagramme de séquence pour décrire le cas d'utilisation *Marquer une case*.

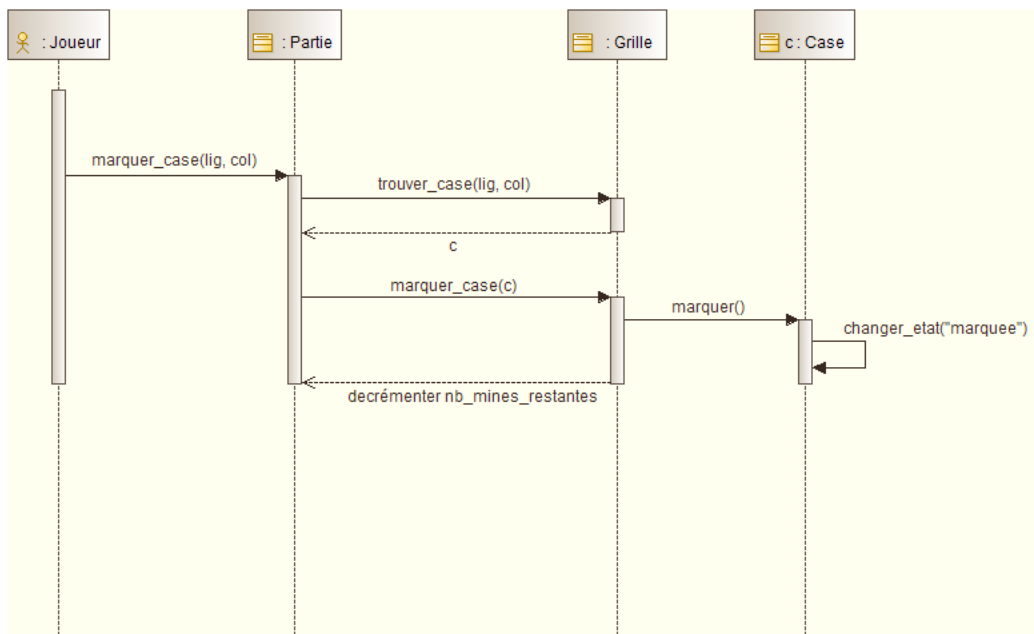


FIGURE 22 – Diagramme de séquence - Marquer une case

14.11 VéliDescartes

Nous identifions deux acteurs principaux utilisant le système : le client qui veut prendre des abonnements, emprunter&déposer des vélos et signaler des incidents, et l'administrateur du système qui souhaite visualiser l'état du système.

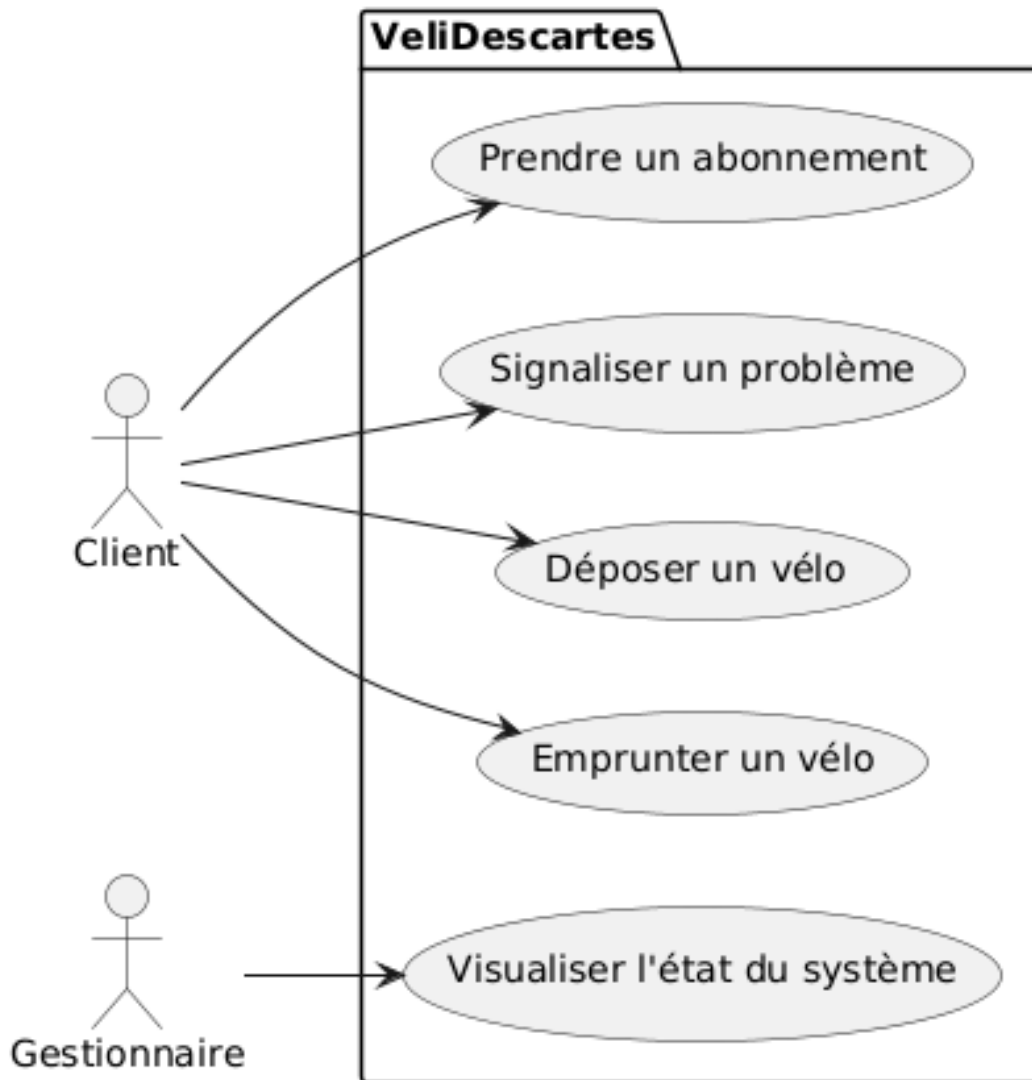


FIGURE 23 – VéliDescartes - Diagramme de cas d'utilisation

Pour élaborer la structure de l'application, nous identifions plusieurs entités. Tout d'abord le client de VéliDescartes. Un client peut posséder plusieurs abonnements, mais un seul valide pour une date donnée (dates de début et durées de validité permettent de s'en assurer ; nous illustrerons cela plus loin dans un diagramme d'activité). Ce client emprunte des vélos qui sont déposés dans des stations. Ici encore un client peut emprunter plusieurs vélos, mais un seul à la fois. Les dates de début et fin de location enregistrées dans la classe d'association `Location` nous permettent de gérer cette contrainte (un diagramme de séquence détaillera l'implémentation de cette contrainte). L'utilisation d'une classe d'association permet de stocker les informations ne pouvant être attachées directement ni aux clients, ni aux vélos. Les stations comme les vélos sont localisés avec leur position GPS. Cette géolocalisation nous permet de savoir quels vélos sont présents dans une station.

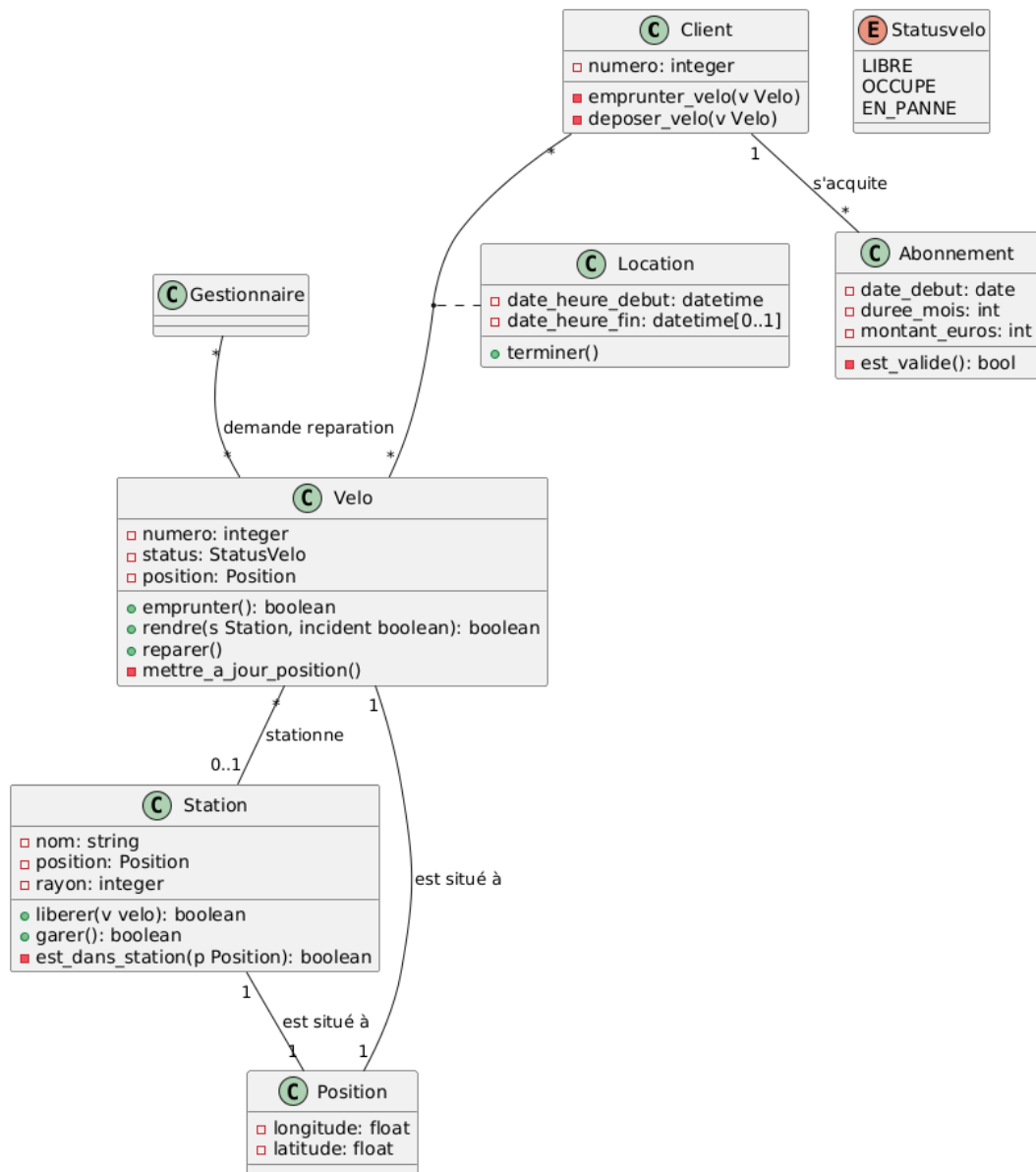


FIGURE 24 – VéliDescartes - Diagramme de classes

Afin d'illustrer notre diagramme de classe, et notamment rendre explicite les cardinalités **Client-Abonnement** et **Client-Velo**, nous réalisons un diagramme d'objets. Dans ce diagramme, qui fige l'état du système au 9 février 2024 à 8h40, le client bob a déjà souscrit à deux abonnements, dont un est actuellement valide. Et il a effectué trois locations de vélo : deux sont terminées et une troisième est en cours (avec un vélo déjà utilisé lors d'une location précédente).

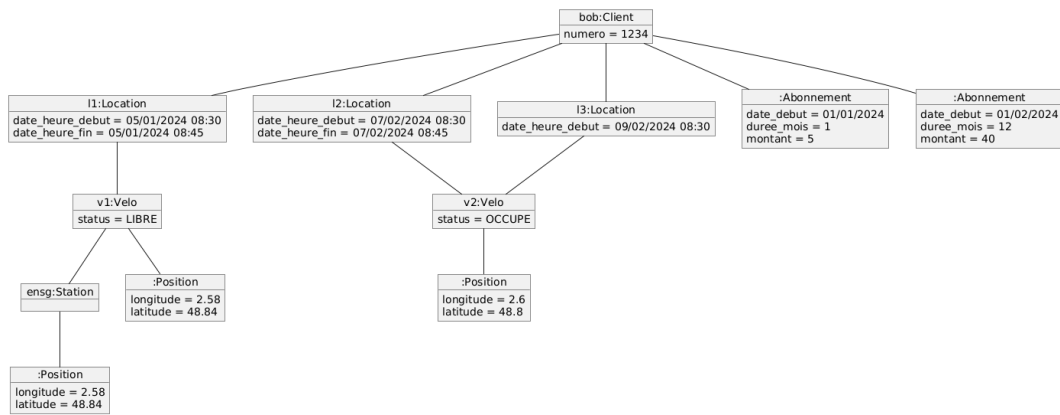


FIGURE 25 – VéliDescartes - Diagramme d'objets

Toujours pur illustrer le diagramme de classe, nous représentons les différents états des vélos dans un diagramme d'états-transitions :

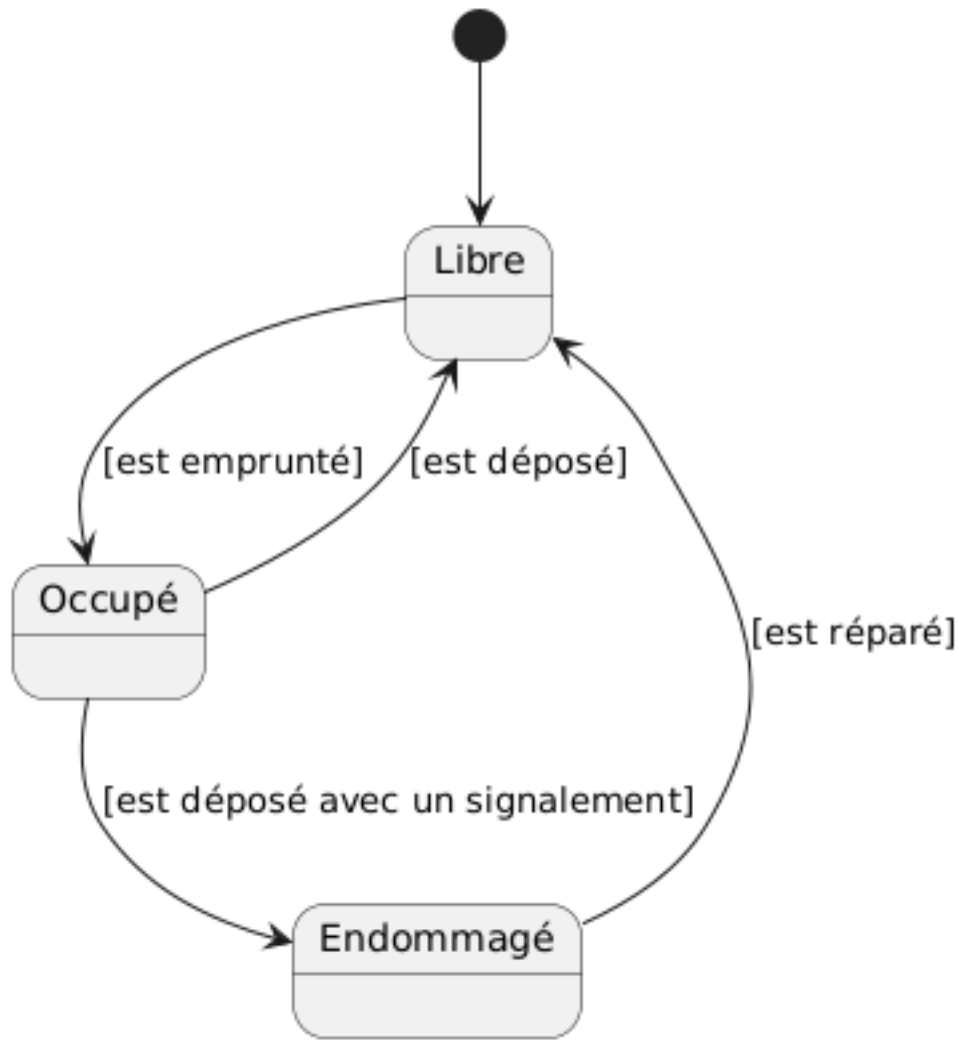


FIGURE 26 – VéliDescartes - Diagramme de'états-transitions d'un vélo

Afin d'expliciter le fait qu'un client ne peut posséder qu'un seul abonnement valide à la fois, nous réalisons un diagramme d'activité. Nous pouvons voir que si un abonnement existe déjà pour la date de début, l'activité se termine sans créer de nouvel abonnement :

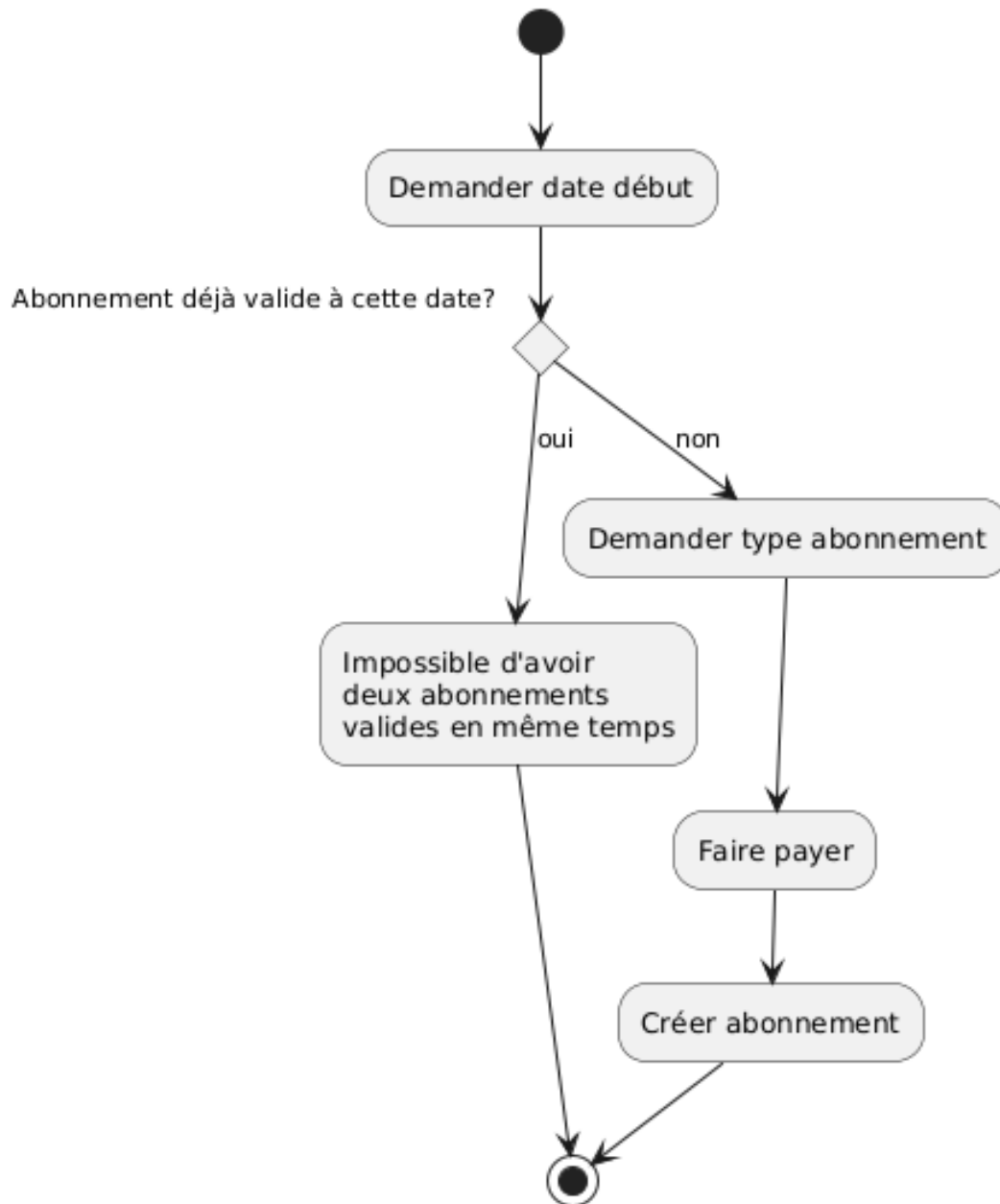


FIGURE 27 – VéliDescartes - Diagramme d'activité *prendre abonnement*

Enfin nous réalisons deux diagrammes de séquence pour illustrer deux des fonctionnalités principales de l'application : *emprunter un vélo* et *déposer un vélo*. Cela nous permet de mettre en évidence les interactions entre les objets, de montrer à quel moment intervient la création d'un objet `Location`, ou encore d'expliciter les différentes conditions permettant de réaliser ces actions (ex : le vélo doit être situé dans la stations pour pouvoir le déposer, on ne peut pas emprunter un vélo s'il n'est pas libre).

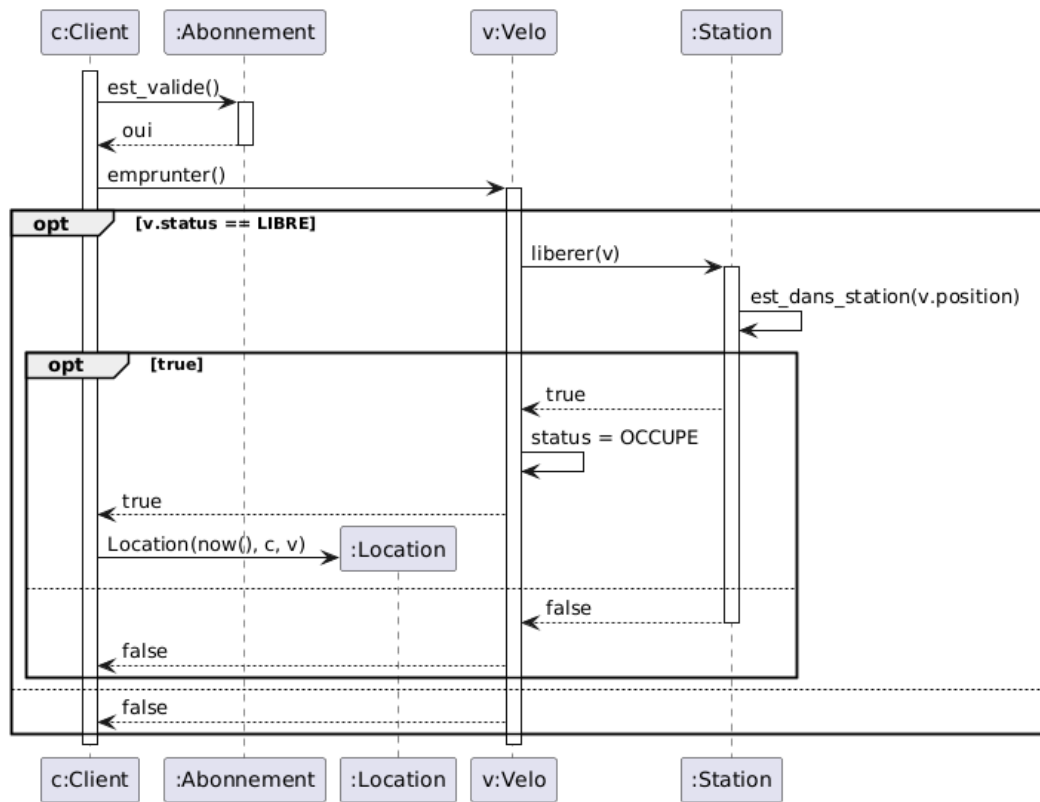


FIGURE 28 – VéliDescartes - Diagramme d'activité *emprunter un vélo*

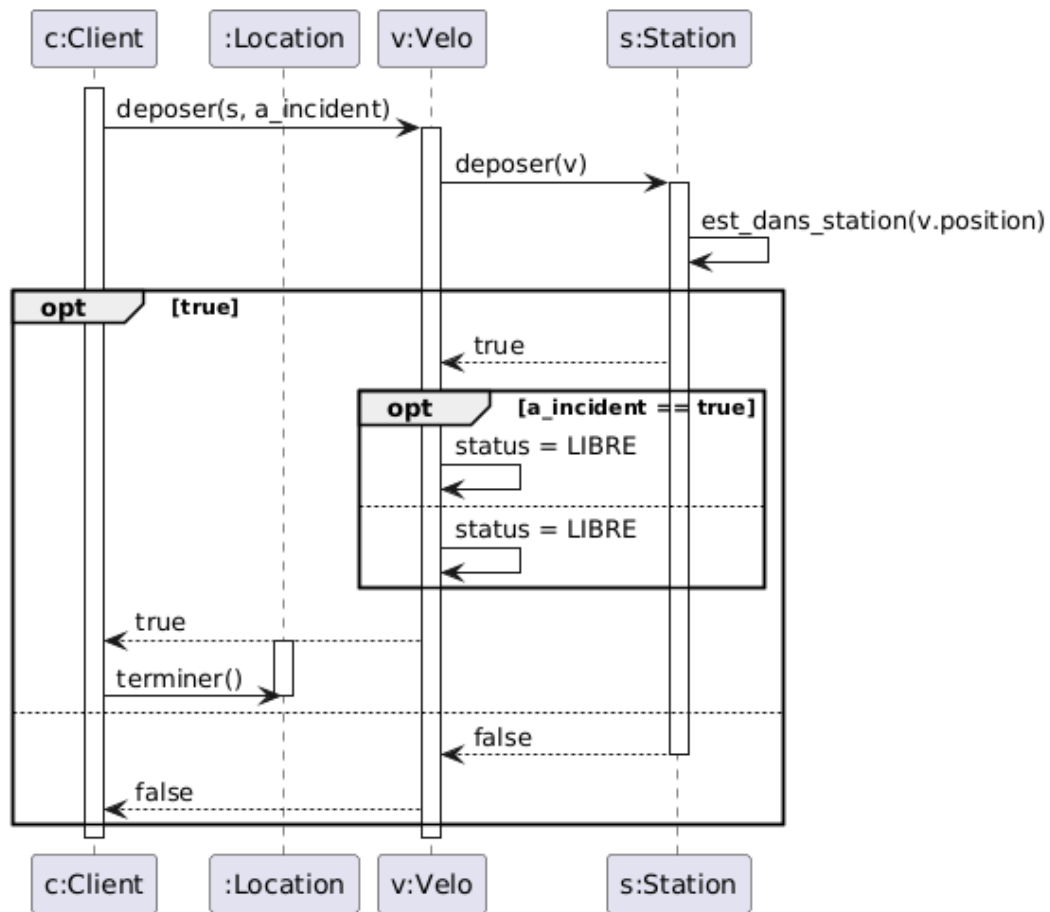


FIGURE 29 – VéliDescartes - Diagramme d'activité *déposer une vélo*