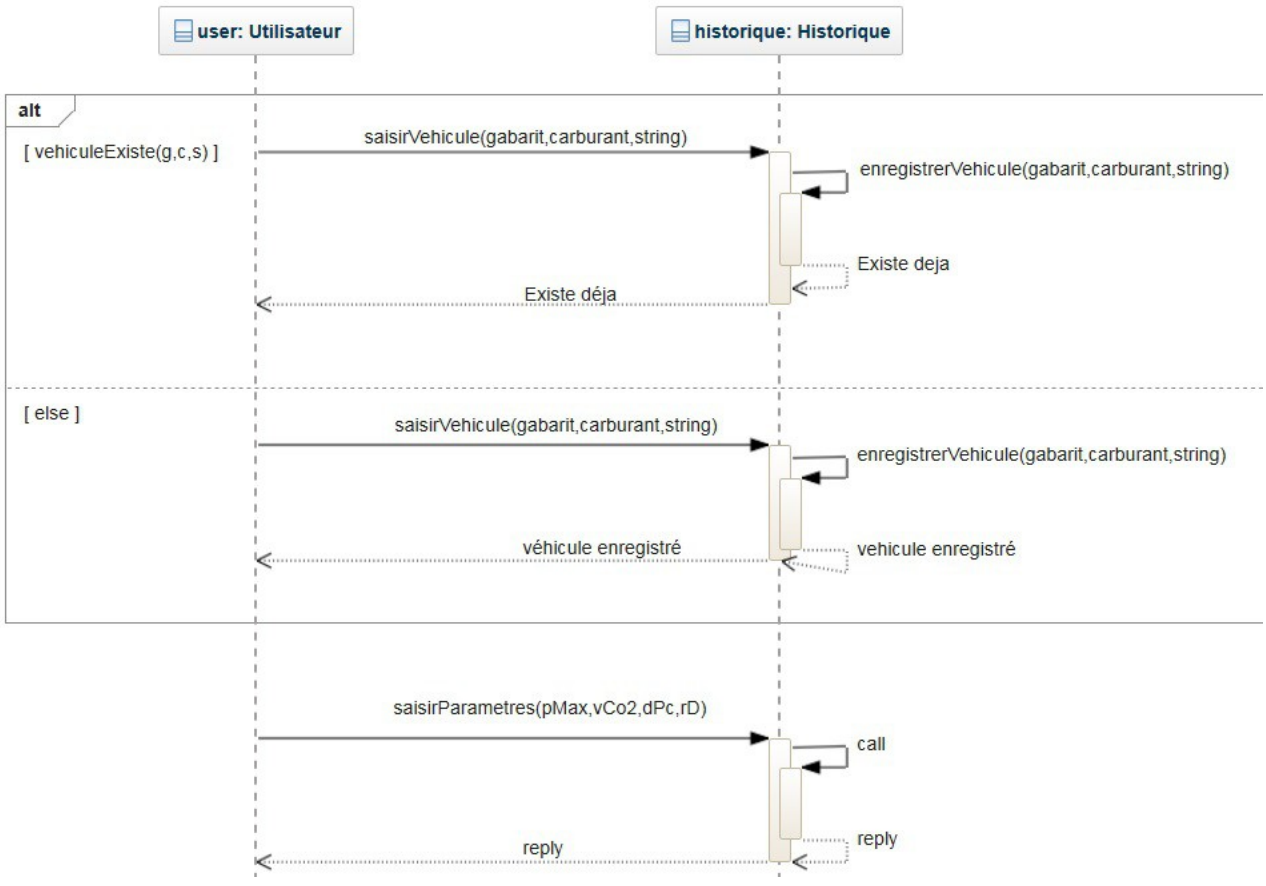
Modification de la classe route : séparation avec la classe tronçon pour respecter les contraintes des fichiers XML.

Invariants : Un invariant est une propriété qui est tellement évidente qu'elle ne peut pas être modifiée.

Quelque invariants de notre application :

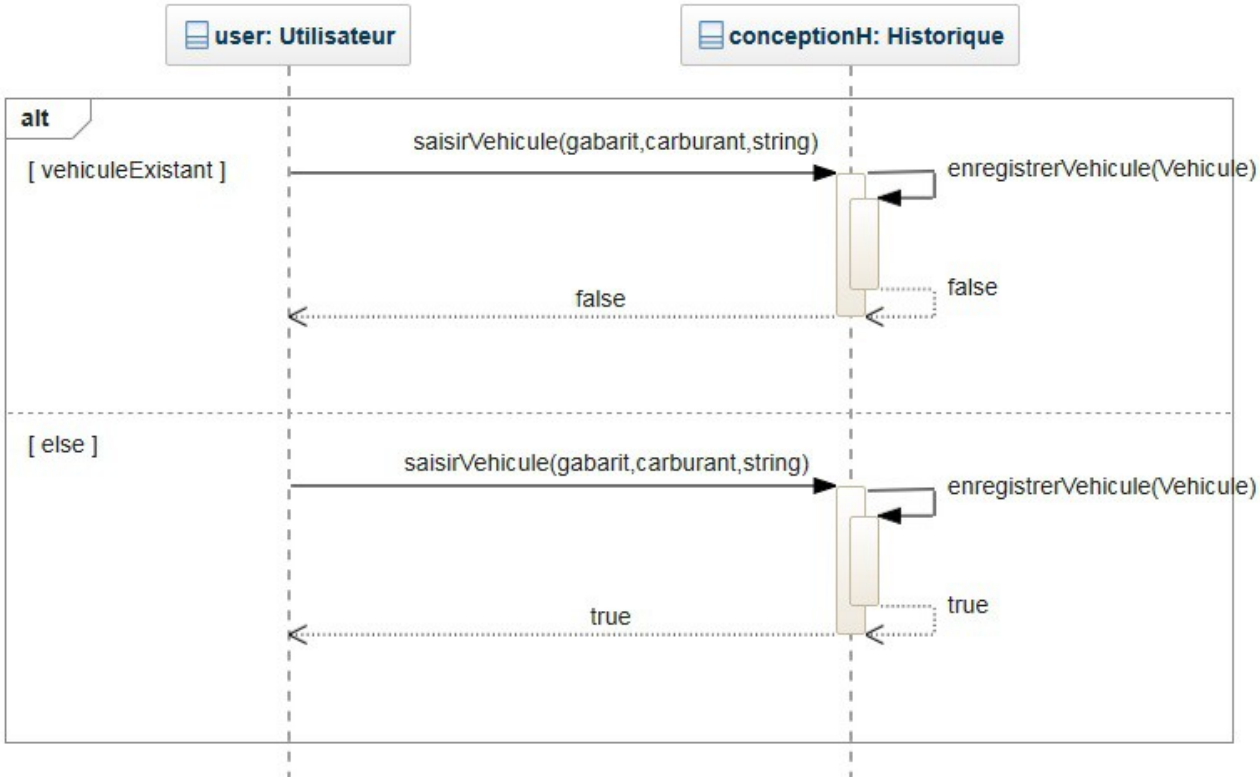
- Un utilisateur a un seul historique
- La ville de départ dans un trajet est différente de la ville d'arrivée
- Les paramètres PrixMax, volumeCo2 sont positifs
- La distance d'un trajet est positive
- Un historique doit exister pour pouvoir être supprimé ou modifié
- Un historique peut comporter plusieurs trajets

Scenario d'utilisation:
C'est un scenario d'utilisations de plusieurs fonctions s'enchainant dans un ordre logique, dans le cadre d'une utilisation normale de l'application.



Fonctions de la classe Utilisateur:

Utilisateur saisirVehicule:
Fonction qui permet la saisie de véhicules.

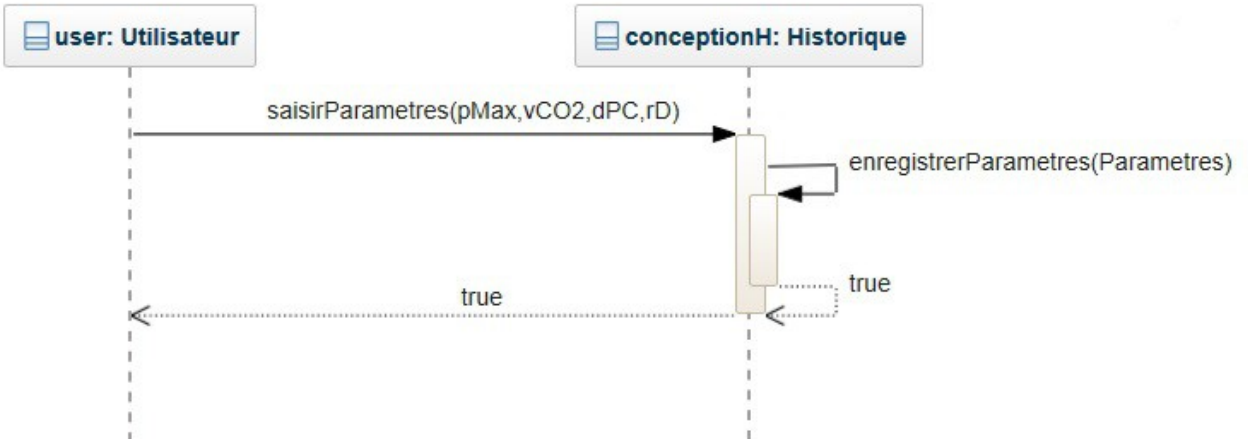


pré conditions:
-gabarit != null ∧essence != null ∧nom != null

déroulement:
L'utilisateur souhaitant enregistrer un nouveau véhicule dans le système clique sur le bouton enregistrer nouveau véhicule puis renseigne les caractéristiques de celui-ci. Si le véhicule existe déjà il en est notifié et invité à recommencer la saisie ou à quitter

post condition:
-Renvoie v, un nouveau véhicule, s'il n'existe pas dans l'Historique, null sinon. Ne modifie rien dans les autres classes.

Utilisateur saisirParametres:
Fonction qui permet la saisie de parametres.

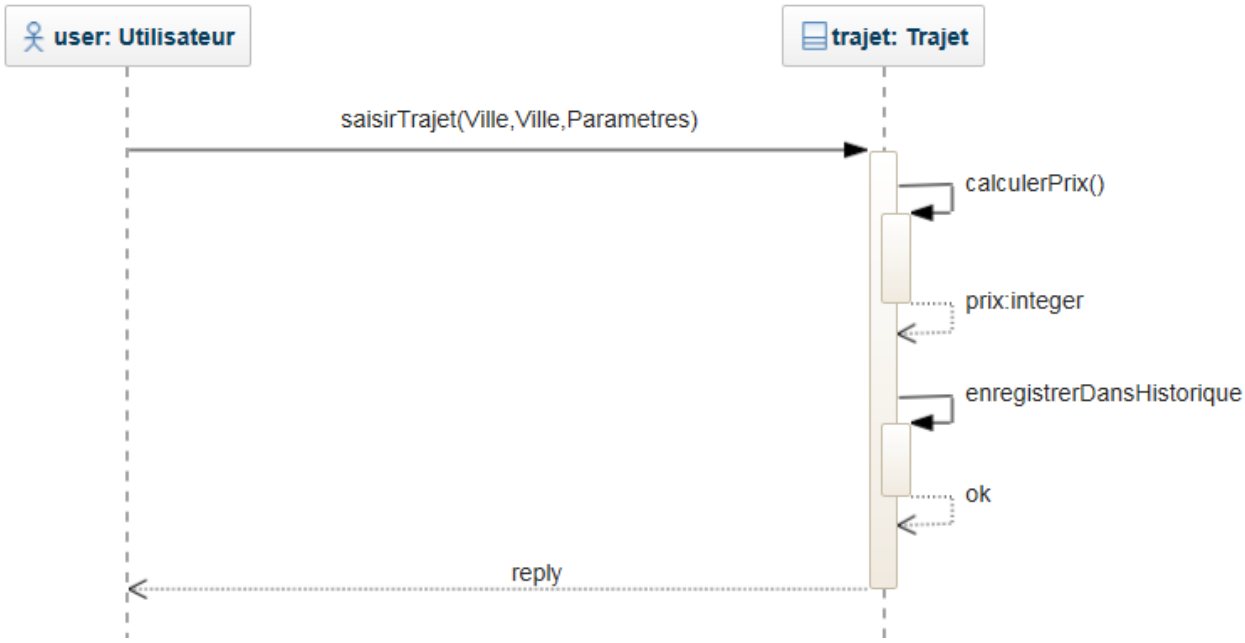


Pré conditions:
-prixMax >= 0 ∧volumeCO2 >= 0 ∧(rapideDuree == faux ∧disancePluscourte == vrai) ∨ (rapideDuree == vrai ∧disancePluscourte == faux))

Déroulement:
Cree de nouveaux parametres si ceux-ci n'existent déjà pas dans l'Historique.

Post conditions:
-Renvoie p, un nouveau parametres, s'il n'existe pas dans l'historique, null sinon. Ne modifie rien dans les autres classes.

Utilisateur saisirTrajet:
Fonction qui permet de saisir le trajet, à partir de parametres, d'une ville de départ et d'arrivée.

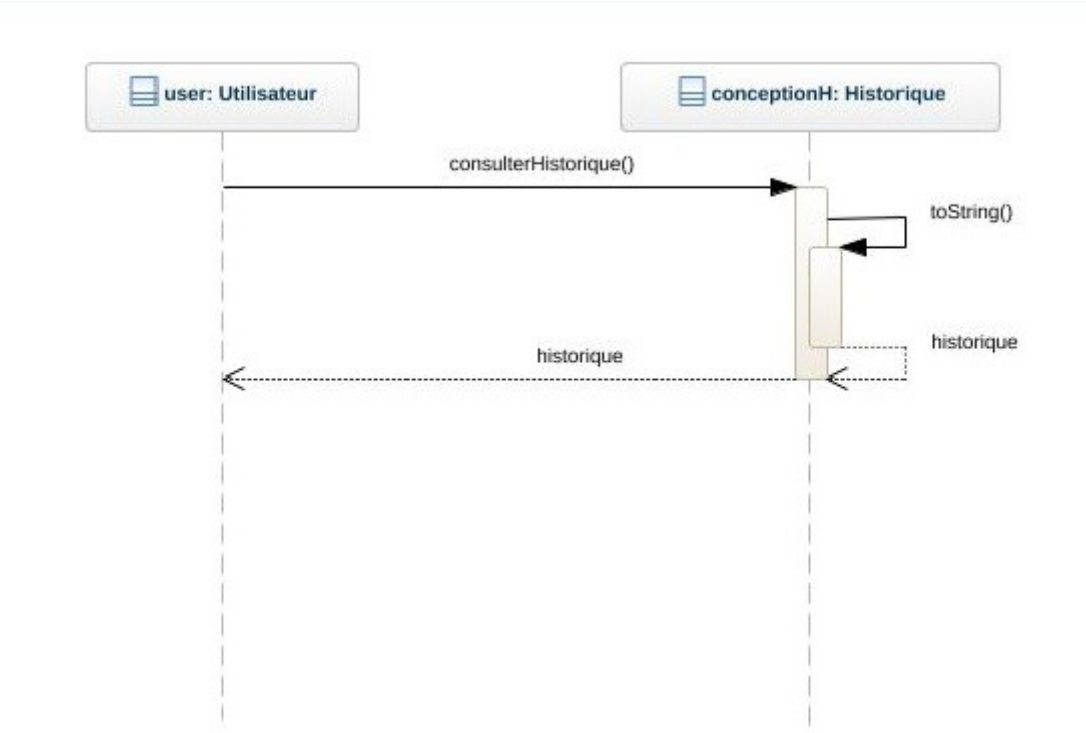


pré condition:
-villeDeDepart.exists() ∧villeDArrivee.exists() ∧!Utilisateur.Historique.contains(Parametres)

déroulement:
Crée un trajet suivant les paramètres de la ville de départ vers la ville d'arrivee.
Si le trajet n'existe pas l'ajoute dans l'Historique. Sinon, appelle la méthode retracer().

Post condition:
-if(Utilisateur.Historique.contains(Trajet)) then Trajet.retracer else
Utilisateur.Historique.Add(Trajet)

Utilisateur consulterHistorique:
Fonction qui permet à l’utilisateur de consulter son historique.

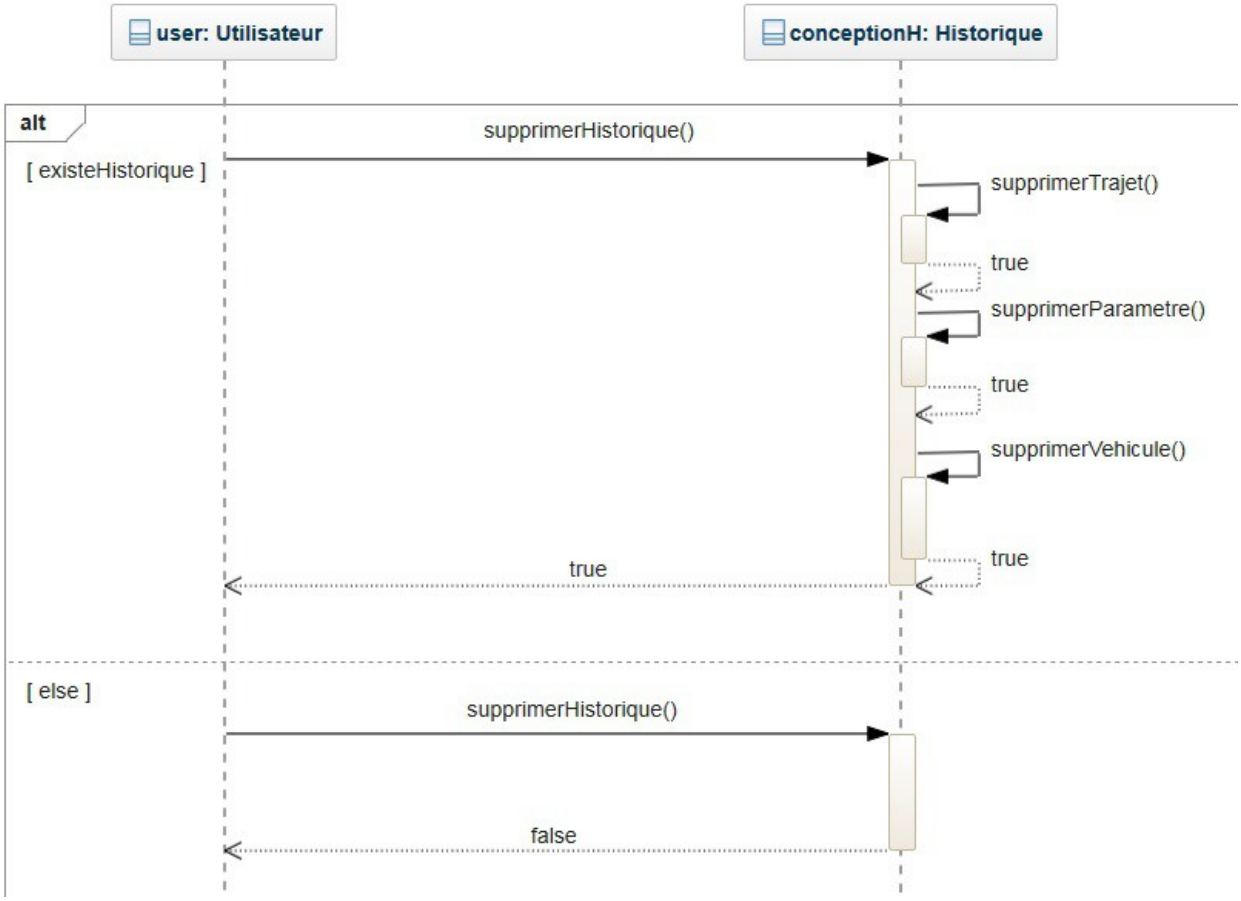


pré condition:
-Utilisateur.Historique.lenght() >= 0 L'Historique peut être vide. La taille ne peut pas être négative.

déroulement:
Pour chaque élément contenu dans l'Historique, l'affiche. Aucun élémént n'est modifié.

Post condition:
-La méthode s'arrête une fois l'Historique entièrement parcouru. Rien n'est modifié.

Utilisateur supprimerHistorique:
Fonction qui permet à l'utilisateur de supprimer une partie de son historique, ou entièrement.



pré condition:
-Utilisateur.Historique.length() > 0. L'Historique doit contenir au moins un élément.

déroulement:
L'utilisateur clique sur supprimer l'historique puis le système lui demande de confirmer son choix. Il pourra confirmer ou annuler

Post condition:
-Utilisateur.Historique.length()--. Le reste n'est pas modifié.
Si un historique existe déjà il est supprimé

Utilisateur modifierHistorique:
Fonction qui permet à l’utilisateur de modifier son historique.

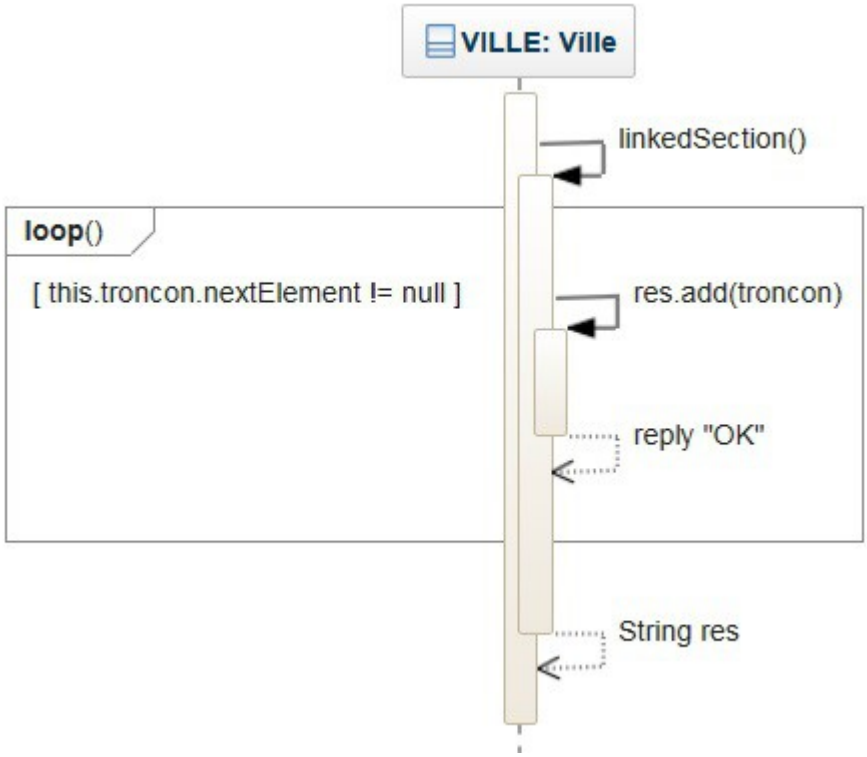
pré condition:
-Utilisateur.Historique.length() > 0

déroulement:
L'utilisateur sélectionne un élément de l'Historique et le modifie.

Post condition:
-L'élément est bien modifié, le reste de l'Historique n'est pas modifié.

Fonctions de la classe Ville:

Ville linkedSections:
Il s'agit tout simplement d'un geter, pour faciliter l'implémentation.



Pré conditions:
-La ville existe dans la base de données $\wedge$ la ville a au moins un tronçon de route qui y passe.

Déroulement:
Pour chaque troncon de route qui passe par la ville, l'ajoute au résultat, sans modifier ni les routes, ni les troncons, ni la ville.

Post conditions:
-res = Liste des troncons de la ville.
-La ville, les routes et les troncons ne sont pas modifiées.

Ville estEvitable:
Fonction qui détermine si elle est présente ou non sur l'un des trajets possibles.
On suppose que MAP est le graphe du système.

pré condition:
-Map.contains(Ville)

déroulement:
Vérifie parmis tous les trajets possibles que la ville n'apparaît pas une fois.

Post condition:
-For(Trajets t : Trajets[] possibles){
  if(t.contains(Ville)) then true
 } else false

Fonctions de la classe Troncon:

Troncon estEvitable:
Fonction qui détermine si l'utilisateur n'a pas à passer par ce troncon, I.E n'apparait dans aucun des trajets possibles.
On suppose que MAP est le graphe du système.

Pré conditions:
-Map.contains(Troncon)

Déroulement:
Vérifie parmis tous les trajets possibles que le troncon n'apparaît pas une fois.

post condition:
-For(Trajets t : Trajets[] possibles){
  if(t.contains(Troncon)) then true
 } else false

Troncon prixAPayer:
Il s'agit tout simplement d'un geter, pour faciliter l'implémentation.

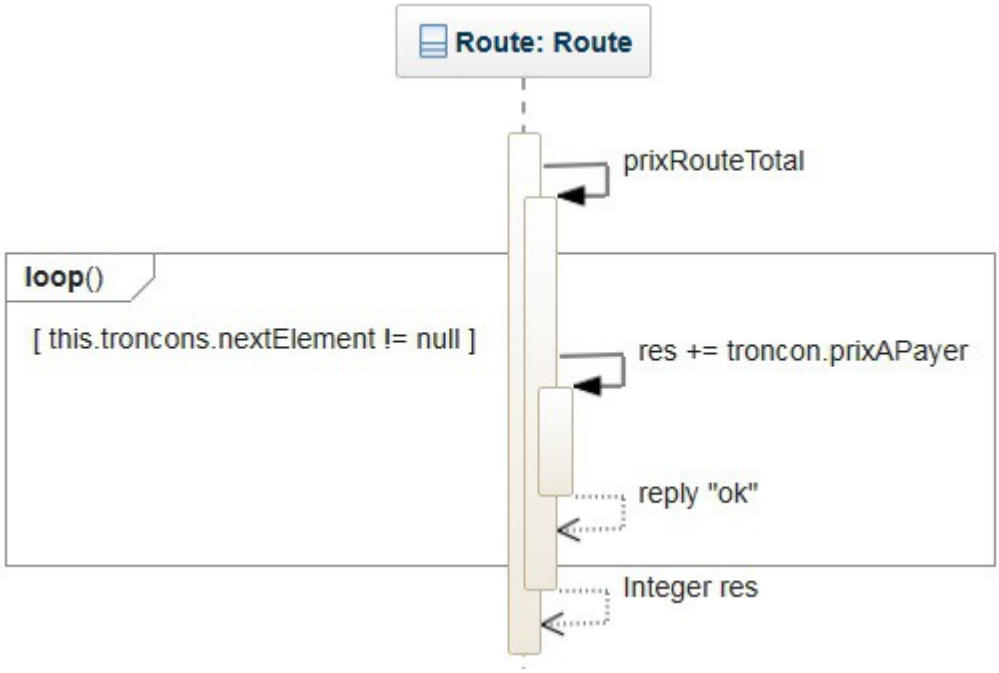
pré conditions:
-Le troncon doit etre valide.
-prix indiqué >= 0.

Déroulement:
Renvoie le prix à payer du troncon.

post conditions:
res = prix a payer du troncon.

Fonction de la classe Route:

Route prixTotal:
Dans le cas où le trajet ne passe que par une route, pour faciliter les calculs.
On favorise cependant plus les calculs sur les troncons.



pré conditions:
-La route a au moins un troncon.

Déroulement:
Pour chaque troncon de la route, on ajoute au résultat le prix du troncon, sans modifier la route et le troncon.

post conditions:
-Res = ∑ des prix troncons de la route.
-La route ainsi que ses troncons ne sont pas modifiés.

Fonctions de la classe Trajet:

Trajet calculerDistance:
Fonction qui calcule la distance totale du trajet.

Pré conditions:
-Trajet.troncons.length > 0 Le trajet doit contenir au moins un troncon

Déroulement:
Pour chaque troncon du trajet, on ajoute au résultat la longueur du troncon, sans modifier la route et le troncon.

Post condition:
-Res = ∑ des distances des troncons de la route.

Trajet calculerTemps:
Fonction qui calcule la duree totale du trajet estimée.

Pré condition:
-Trajet.troncons.length > 0 Le trajet doit contenir au moins un troncon

Déroulement:
Pour chaque troncon du trajet, on ajoute au résultat la longueur du troncon multipliée par sa vitesse, sans modifier la route et le troncon.

Post condition:
-Res = ∑ des distances des troncons de la route * Troncon.vitesse

Trajet calculerPrix:
Fonction qui calcule le prix total du trajet.

Pré condition:
-Trajet.troncons.length() > 0

déroulement:
Pour chaque troncon de la route, ajoute le prix au resultat

post condition:
-Res = ∑ du prix des troncons de la route ∧res >= 0

Trajet calculerVolumeCO2:
Fonction qui calcule le volume de CO2 du trajet.

Pré condition:

Déroulement:

Post condition:

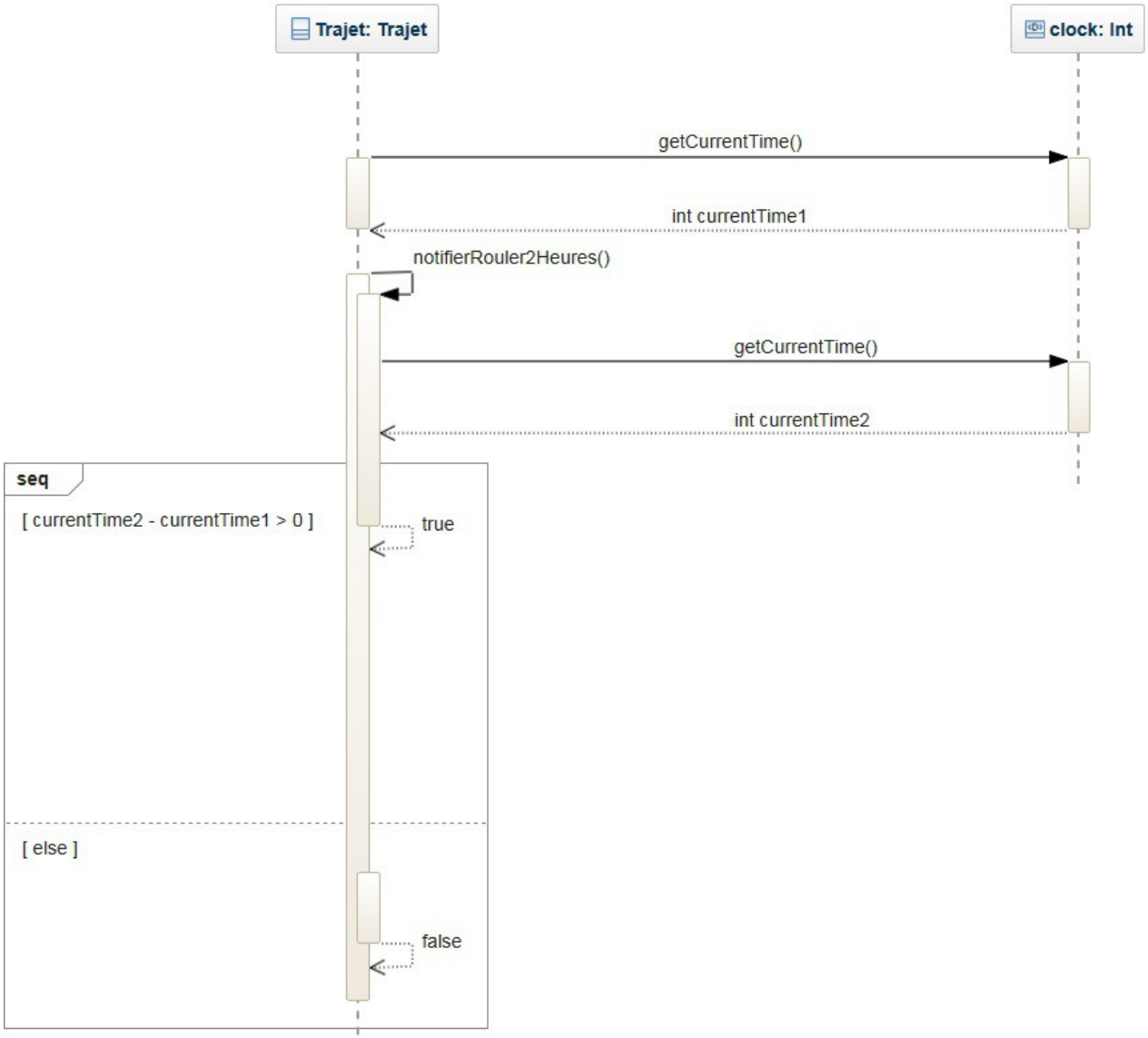
Trajet notifierParametresNonRespectés:
Comme indiqué dans le cahier des charges, il est possible que les parametres choisis par l'utilisateur ne soient pas valides dès le départ, I.e impossible de trouver un chemin respectant tous les parametres ou lorsqu'il se trompe de trajet.

Pré condition:
-L'un des parametres n'est plus respecté

déroulement:
Si l'un des parametres n'est plus respecté, alors affiche un message à l'écran. Ne fait rien sinon.

post condition:
-Affiche un message : "Le parametre : " + parametre " n'est plus respecté"

Trajet notifierRouler2Heures:
Comme indiqué dans le cahier des charges, l'application se doit de notifier l'utilisateur si celui-ci a roulé plus de 2 Heures, pour qu'il aille se reposer. Cela est basé sur les recommandations officielles des ministères.

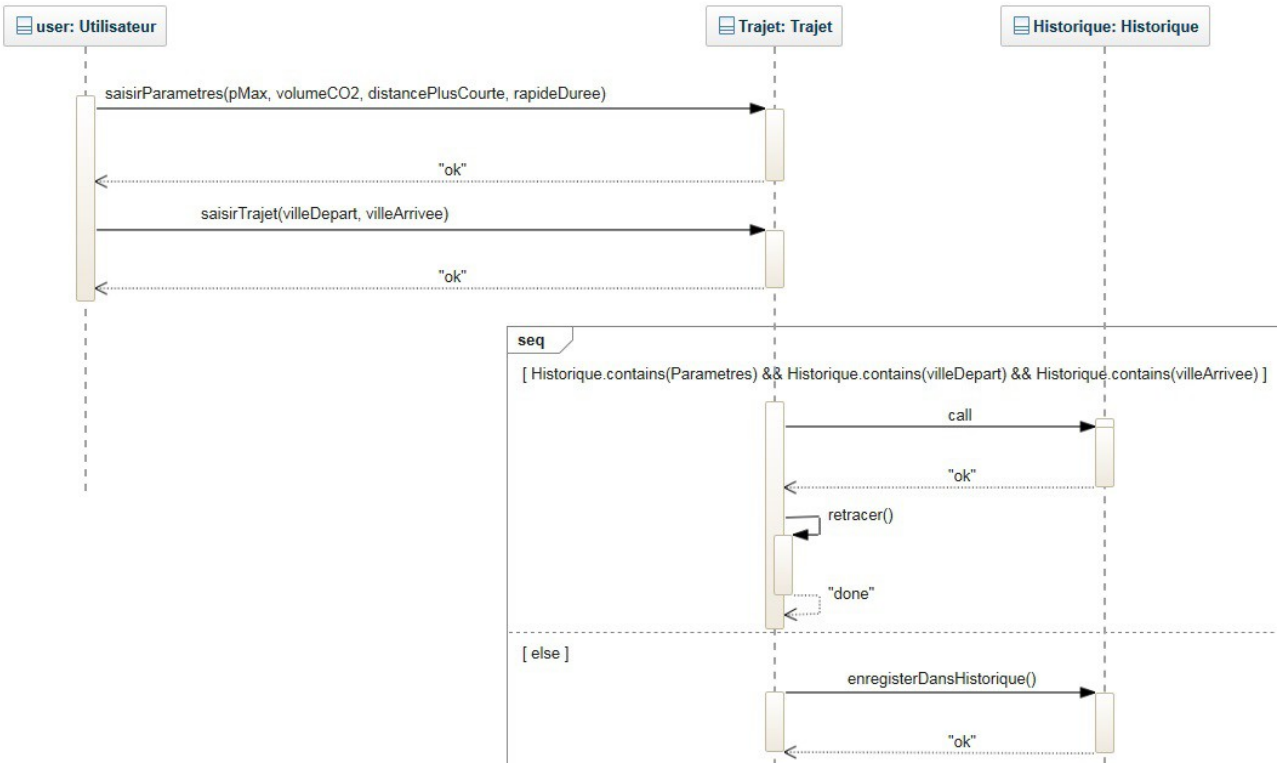


pré conditions:
-Le trajet doit etre en cours.

Déroulement:
L'appel à cette fonction permet de seulement calculer la durée du trajet en cours. Le trajet ne doit pas etre modifié.

post condition:
-Res = vrai si l'utilisateur a bien roulé 2 Heures, sinon res = faux. Rien n'est modifié.

Trajet retracer:
Cette fonction a pour but d'économiser du calcul en évitant de devoir reparcourir la base de données pour effectuer à nouveau ce trajet.

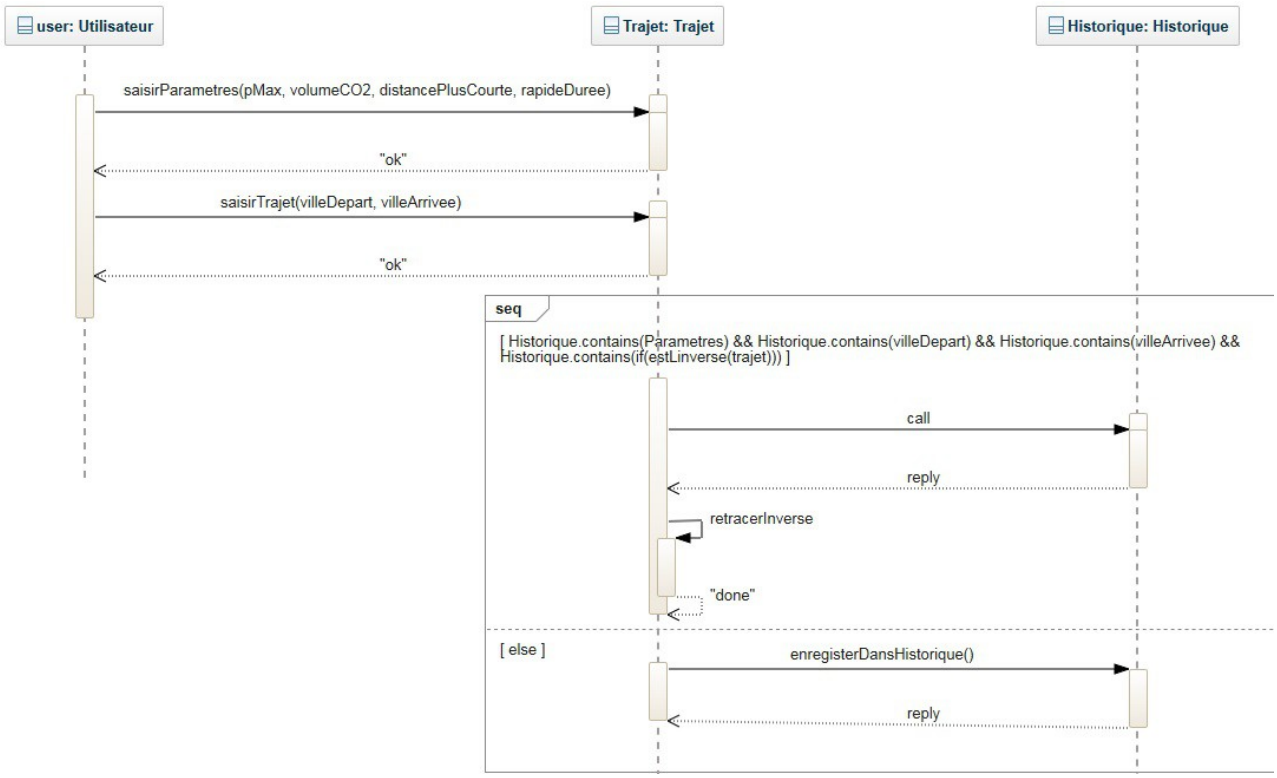


pré condition:
-Le trajet doit déjà exister dans l'historique de l'utilisateur (On suppose que le trajet saisi est valide)
-Le véhicule ainsi que les parametres utilises doivent etre les memes.

déroulement:
L'application retrace le trajet, en utilisant lse memes parametres, memes routes, troncons et villes parcourus.

post condition:
-L'application retrace le trajet, sans modifier l'historique, le trajet demandé ou les parametres.

Trajet retracerInverse:
Comme pour la fonction retracer, le but de cette fonction est d'économiser du calcul et de la mémoire en récupérant les données d'un trajet qui a déjà été effectué.

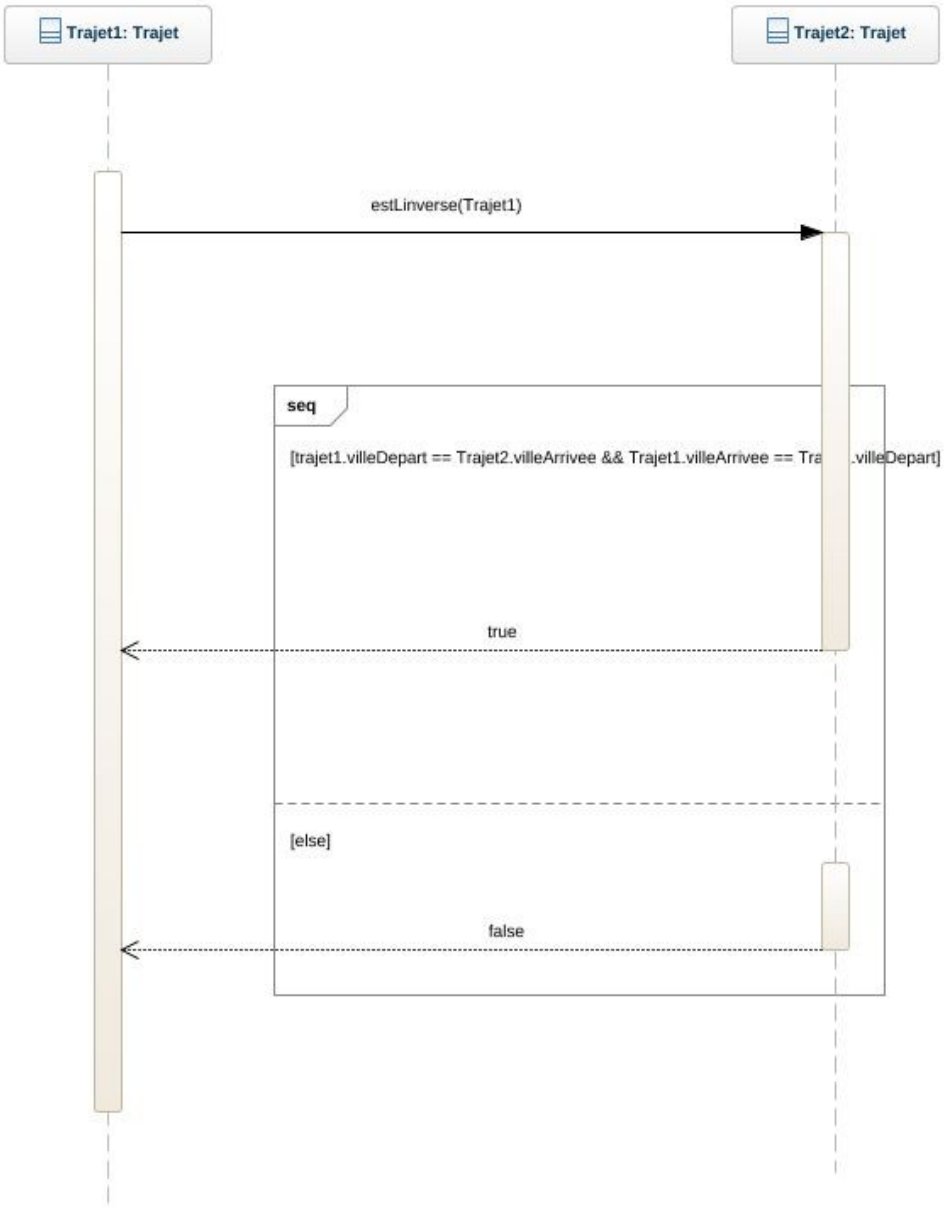


pré condition:
-Le trajet saisi est valide.
-Le trajet est l'inverse d'un trajet déjà effectué et enregistré dans l'Historique.

Déroulement:
Le programme retrace le trajet inverse, sans modifier l'Historique.

post condition:
Le trajet inverse est tracé. L'historique n'est pas modifié.

Trajet estLInverse:
Cette fonction sert à faciliter l'implémentation de la fonction retracerInverse.



pré conditions:
-Les 2 trajets sont valides.

Déroulement:
Le programme vérifie que les 2 trajets ont bien le memes parametres, ainsi que des villes de depart et d'arrivée inversés.

post conditions:
Renvoie vrai si les 2 trajets ont bien les memes parametres et si les villes de départ et d'arrivée sont inversés, faux sinon.

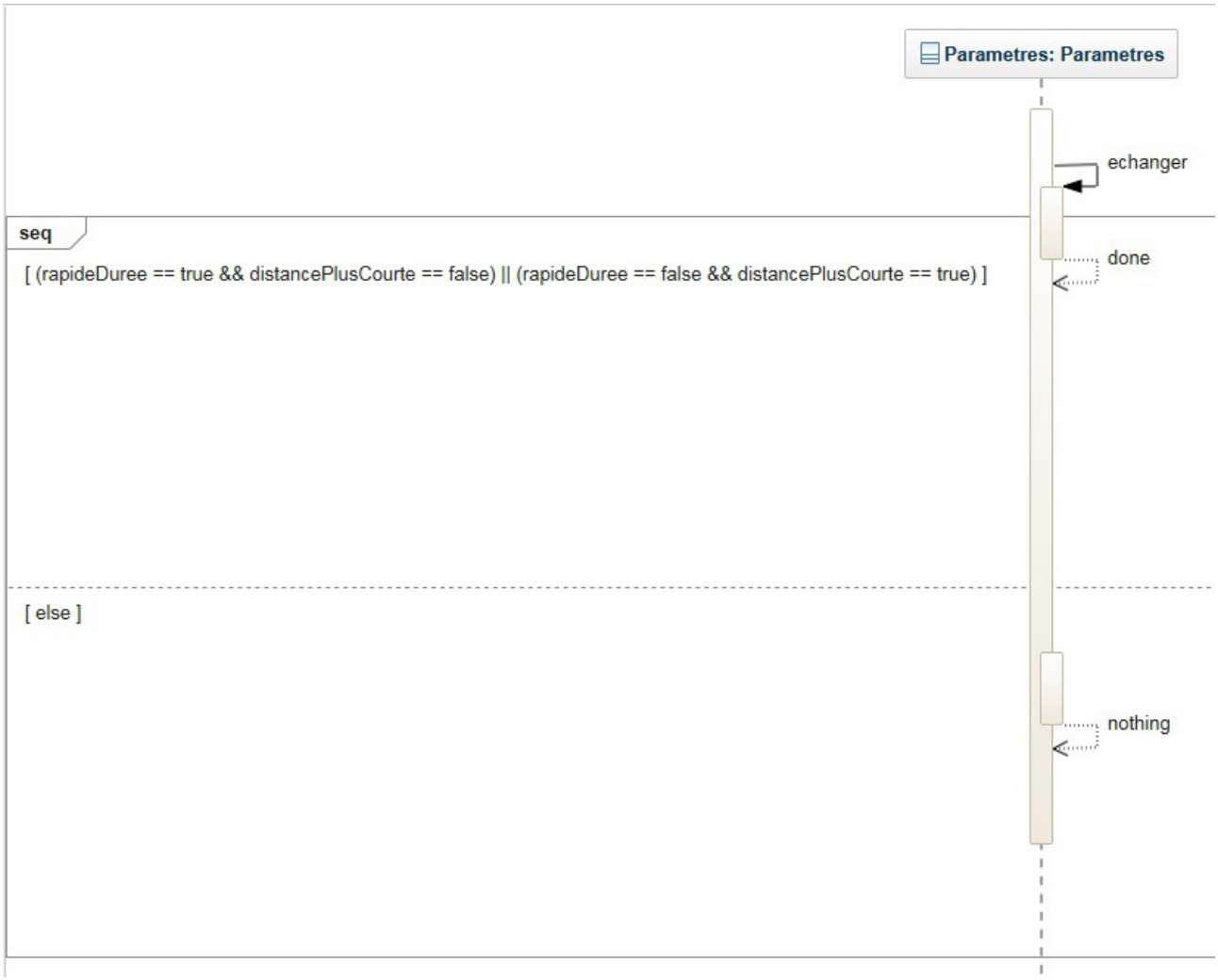
Trajet enregistrerDansHistorique:
Fonction qui permet d’enregistrer le trajet dans l’Historique.

Pré condition:
-Trajet.exists() ∧Trajet.isOver()

Déroulement:
Lorsque le trajet est terminé, l'ajoute dans l'Historique s'il n'existe pas déjà dans l'historique.

post condition:
-Utilisateur.Historique.add(Trajet);

Fonctions de la classe parametres:
Parametres echanger:
Fonction qui permet d'échanger les paramètres booléens distancePlusCourte et rapideDuree, puisque les 2 ne peuvent pas etre vraies en meme temps. Cela est utile dans le cas où l'utilisateur se trompe de chemin, ce qui fait que l'un ou l'autre des parametres ne peut plus etre respecté.



Pré condition:
-(1)rapideDuree = false $\wedge$distancePlusCourte = true $\vee$(2)rapideDuree = true et distancePlusCourte = false

Déroulement:
L'attribut qui est faux devient vrai, alors que celui qui était vrai devient faux.

Post condition:
-Si cas (1), alors rapideDuree = vrai $\wedge$distancePlusCourte = faux.
-Si cas (2), alors rapideDuree = faux $\wedge$distancePlusCourte = vrai. Les autres attributs de la classe parametres, ainsi que le trajet, ne sont pas modifiés.

Parametres toString: Pas de détails supplémentaires à fournir pour ce genre de fonction.
Fonctions qui facilite l'affichage.

Fonctions de la classe véhicule:

Vehicule toString: Pas de détails supplémentaires à fournir pour ce genre de fonction.
Fonctions qui facilite l'affichage.

Vehicule modifierGabarit:
Fonction qui permet de modifier le gabarit d'un véhicule.

pré condition:
-Vehicule.exists() $\wedge$ g = Vehicule.garabarit.

déroulement:
Modifie le gabarit du véhicule.

Post condition:
-Vechicule.gabarit != g

Vehicule modifierCarburant:
Fonction qui permet de modifier le type de carburant d'un véhicule.

pré condition:
-Vehicule.exists() $\wedge$ c = Vehicule.carburant

déroulement:
Modifie le type de carburant du véhicule.

Post condition:
-Vechicule.carburant != c

Vehicule modifierNom:
Fonction qui permet de modifier le nom d'un véhicule.

pré condition:
-Vehicule.exists() $\wedge$ n = Vehicule.nom

déroulement:
Modifie le nom du véhicule.

Post condition:
-Vechicule.nom != n

Fonctions de la classe Historique:

Historique enregistrerParametres:
Fonction qui permet d'enregistrer un parametres enregistré dans l'historique.

pré condition:
-Parametres.exists() ∧!Historique.contains(Parametres)

déroulement:
Enregistre les parametres dans l'Historique.

Post condition:
-Historique.add(Parametres)

Historique enregistrerVehicule:
Fonction qui permet d'enregistrer un véhicule enregistré dans l'historique.

pré condition:
-Vehicule.exists() ∧!Historique.contains(Vehicule)

déroulement:
Enregistre le véhicule dans l'Historique.

Post condition:
-Historique.add(Vehicule)

Historique supprimerParametres:
Fonction qui permet de supprimer un parametres enregistré dans l'historique.

pré condition:
-Historique.contains(Parametres)

déroulement:
Supprime le parametres passé en parametres de la fonction

Post condition:
-Historique.remove(Parametres)

Historique supprimerVehicule:
Fonction qui permet de supprimer un véhicule enregistré dans l'historique.

pré condition:
-Historique.contains(Vehicule)

déroulement:
Supprime le véhicule passé en parametres de la fonction.

Post condition:
-Historique.remove(vehicule)

Historique modifierParametres:
Fonction qui permet de modifier un parametres enregistré dans l'historique.

pré condition:
-Historique.contains(Parametres) $\wedge$ p = Parametres

déroulement:
Modifie le parametres passé en parametre.

Post condition:
-Historique.Parametres != p

Historique modifierVehicule:
Fonction qui permet de modifier un véhicule enregistré dans l'historique.

pré condition:
-Historique.contains(Vehicule) $\wedge$ v = Vehicule.

Déroulement:
Modifie le véhicule.

Post condition:
-Historique.Vehicule != v

Historique toString: Pas de détails supplémentaires à fournir pour ce genre de fonction.
Fonctions qui facilite l'affichage.

Historique supprimerTrajet:
Fonction qui permet de supprimer un élément de l'historique.

Pré condition:
-Historique.contains(Trajet)

déroulement:
Supprime le trajet passé en paramètres.

Post condition:
-Historique.remove(Trajet)

Conception du graphe:
Graphe de troncons plutot que graphe des routes : plus souple pour la programmation ainsi que pour le tracé des graphes.

Structure de données :

Les informations des Villes et Routes sont stockées en 2 listes.
À partir de la liste des Routes, on construit une matrice d'adjacence qui va etre utilisée pour l'algorithme. La matrice sera construite en fonction des paramètres saisis par l'utilisateur.

Les deux listes Villes et Routes ne sont utilisées que dans la fonction Trajet retracer, ou quand on veut vérifier les attributs booléens.

L'algorithme proposé :

L'algorithme est le suivant : A* est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final. Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique.

- Description : On suppose qu'ici, une ville = un nœud.
1. Il commence à un nœud choisi (ville de départ) et l'ajoute à une liste d'attente.
2. Puis il retire le nœud avec l'évaluation heuristique la plus faible.
3. L'algorithme se termine quand la ville retirée est la destination ou quand il y a aucune ville restante dans la liste d'attente. Sinon, toutes ses villes voisines seront ajoutées à la liste. Ensuite il retourne à l'étape 2.

- Les fonctions heuristiques utilisées:
-g(x): La somme optimale depuis la ville initiale à la ville x.
-h(x): La distance aérienne entre la ville de destination et la ville x. La distance aérienne est la plus courte distance entre 2 villes, cela assure donc une heuristique admissible. h(x) est calculée par les coordonnées de la destination et de la ville x.

- Les avantages :
-On n'a pas besoin de prétraitement.
-Il ne consomme que peu de mémoire.
-Avec une bonne évaluation heuristique, l'A* est plus efficace que les autres algorithmes comme Ford-Bellman ou Dijkstra. Dans la plupart de cas, bien que l'A* soit plus lent, il donne des parcours plus favorables qu'un algorithme glouton.

- Amélioration :
-On peut remplacer la liste d'attente par un tas minimal, qui va donner la valeur minimale actuelle en O(log n) de temps, par rapport à O(n) d'une recherche linéaire.

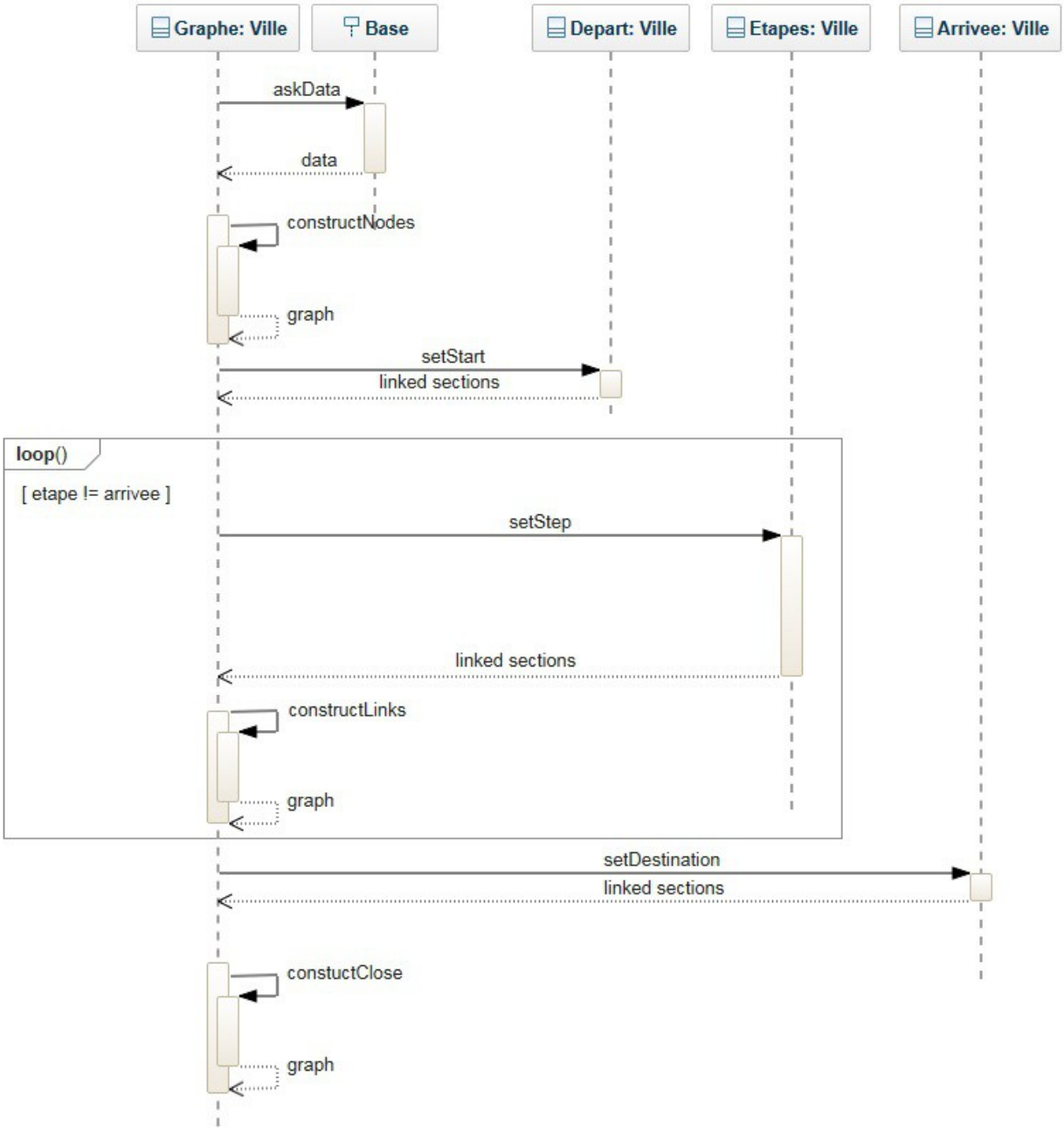
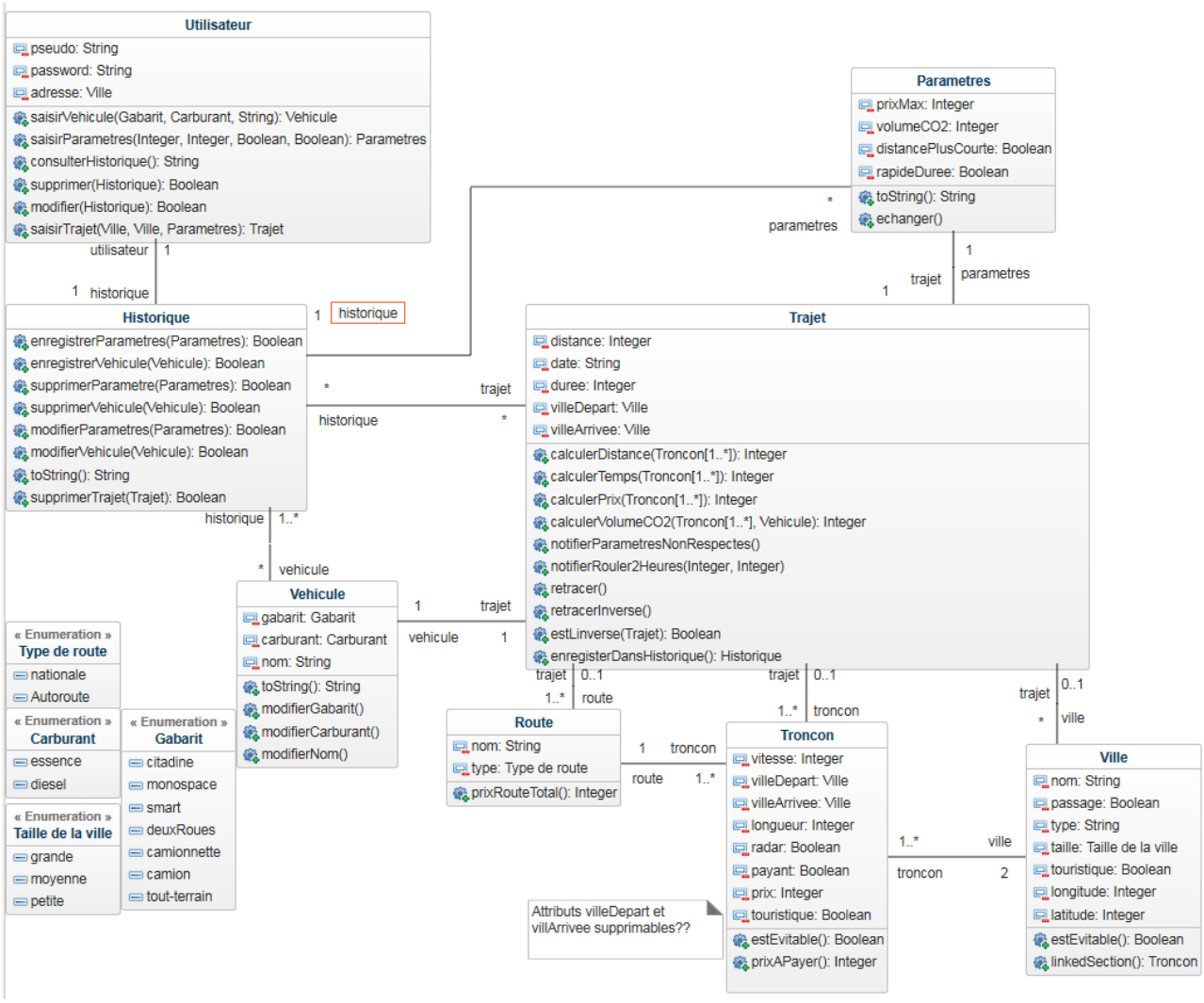

Graphe: Ville
Base
Depart: Ville
Etapes: Ville
Arrivee: Ville
askData
data
constructNodes
graph
setStart
linked sections
loop()
[etape != arrivee]
setStep
linked sections
constructLinks
graph
setDestination
linked sections
constuctClose
graph

Diagramme de conception :



Nous avons pris en compte les retours pour l'analyse.
La classe Historique est maintenue ici puisque l'historique contient la liste des véhicules et des paramètres, ainsi que les trajets effectués. C'est pour cette raison que la classe Historique est ici maintenue.