EISTI

Projet Voie Ferrée

Livrable n°2 : Analyse dynamique et algorithmes

MVE ASSEKO Simplice – TYGREAT Thomas

TAING Paul – COUDERT Thibault

SOMMAIRE

I. Rappel du cahier des charges 3

II. Les Types Abstraits 4

a. **Les Types Abstraits Sommet** 4

1. Le Type Abstrait Sommet 4

2. Le Type Abstrait Produit 5

3. Le Type Abstrait Ville 6

b. **Les Types Abstraits Arête** 7

1. Le Type Abstrait Arête 7

2. Le Type Abstrait Liaison 8

c. **Les Types Abstraits Graphe** 9

1. Le Type Abstrait Graphe 9

2. Le Type Abstrait ReseauTransport 12

3. Le Type Abstrait ListeProduit 15

d. **Le Type Abstrait Wagon** 18

e. **Le Type Abstrait Train** 19

III. L'analyse dynamique 20

a. **Use Case du client** 21

b. **Use Case du PDG** 22

c. **Le diagramme de séquence du calcul de la composition** 24

d. **Le diagramme de séquence du calcul du chemin** 26

e. **Quelques exemples de diagramme d'état-transition** 28

I. Rappel du cahier des charges

Il s’agit de réaliser un logiciel permettant à une entreprise de gérer les livraisons de produits à ses clients par voie ferrée.

Il faut d’une part déterminer la composition du train et donc de chaque wagon, sachant qu’il y a des produits incompatibles qui ne doivent pas se trouver dans le même wagon. On devra optimiser cette répartition afin d’utiliser un nombre raisonnable de wagons. Ceci se fait grâce à l’algorithme de coloriage de graphes vu en théorie de graphes.

D’autre part, l’application doit calculer le plus court chemin entre le siège de l’entreprise et la ville de livraison afin d’optimiser le parcours du train. Le réseau ferroviaire devra être lu à partir d’un fichier fourni. On pourra représenter ce réseau sous forme de graphe et ainsi appliquer l’algorithme de Dijkstra pour trouver le plus court chemin.

On déduit ainsi la composition des wagons ainsi que l’itinéraire du convoi. Le programme doit comprendre :

- Une application client qui permet la saisie des produits ainsi que de leurs incompatibilités. Le client choisit également la ville de livraison.
- Une application Fournisseur qui permet de récupérer les données concernant la commande (stockées dans un fichier binaire) et d’afficher l’itinéraire correspondant ainsi que la composition des wagons.

Pour ce second livrable, nous intéressons à l’analyse dynamique du système, c'est-à-dire l’ensemble des comportements des objets, mais aussi aux différents cas d’utilisation du système. De plus dans cette seconde partie, nous devons compléter les types abstraits définis dans le premier livrable.

II. Les Types Abstraits

a. Les Types Abstraits Sommet

1. Le Type Abstrait Sommet

Type abstrait Sommet

Concept : Un sommet possède un nom qui correspond au libellé dans le cas du produit et au nom de la ville dans le cas du Réseau de transport.

Opérations de base :

Constructeur creerSommet(chaine nom) : Sommet

Observateur recNom() : Chaine

Transformateur affNom(chaine nom) : Sommet

Axiomes : Sommet s

Pré-conditions :

Post-conditions :

// On gère l'observateur recNom

creerSommet(nom).recNom().estEgal(a)
Sommet s => s.affNom(nom2).recNom().estEgal(nom2)

Fin Sommet

2. Le Type Abstrait Produit

Type abstrait Produit **hérite de** Sommet :

Concept : Sommet qui possède un numéro de wagon. Permet d'appliquer l'algorithme de coloration pour répartir les produits selon leurs incompatibilités. Le numéro de wagon tient lieu de couleur. On remarque qu'un Produit hérite d'un Sommet : il n'est pas nécessaire de réécrire les observateurs ou transformateurs déjà définis dans le Type Abstrait Sommet.

Opérations de base :

Constructeur creerProduit(chaine libellé) : Produit

Observateur recNumWagon() : Entier
Observateur estAffecte() : Booleen

Transformateur affNumWagon(Entier num) : Produit

Axiomes : Produit p

Pré-conditions :

définie(p.recNumWagon()) <=> p.estAffecte()

Post-conditions :

// On gère l'observateur estAffecte

NON creerProduit(nom).estAffecte()
p.affNumWagon(i).estAffecte()

// On gère l'observateur recNumWagon

p.affNumWagon(i).recNumWagon().estEgal(i)

Fin Produit

3. Le Type Abstrait Ville

Type abstrait Ville **Hérite de** Sommet :

Concept : Sommet qui peut être marqué, afin de réaliser l'algorithme de Dijkstra dans le cas du Réseau de transport. Une ville possède un indice, allant de 1 à nbVilles. On remarque qu'un Produit hérite d'un Sommet, ainsi le constructeur creerVille surcharge creerSommet : il n'est pas nécessaire de réécrire les observateurs ou transformateurs déjà définis dans le Type Abstrait Sommet.

Opérations de base :

Constructeur creerVille(chaine nom, Indice n) : Ville

Observateur estMarque() : Booleen
Observateur recIndice() : Entier

Transformateur marquer() : Ville
Transformateur demarquer() : Ville

Axiomes : Produit p

Pré-conditions :

Post-conditions :

// On gère l'observateur estMarque

NON creerVille(nom,n).estMarque()
v.marquer().estMarque()
NON v.demarquer().estMarque()

// On gère l'observateur recIndice

creerVille(nom,n).recIndice().estEgal(n)

Fin Ville

b. Les Types Abstraits Arête

1. Le Type Abstrait Arête

Type abstrait Arete

Concept : Ensemble de deux sommets. Nous travaillons avec des arêtes non orientées, donc pas de notion d'origine ou de destination.

Opérations de base :

Constructeur creerArete(Sommet sA, Sommet sB) : Arete

Observateur recS1() : Sommet
Observateur recS2() : Sommet

Axiomes : Sommet sa, sb

Pré-conditions :

définie(creerArete(sA,sB)) <=> NON sA.estEgal(sB)

Post-conditions :

// On gère l'observateur recS1

creerArete(sA,sB).recS1().estEgal(sA)

// On gère l'observateur recS2

creerArete(sA,sB).recS2().estEgal(sB)

Fin Arete

2. Le Type Abstrait Liaison

Type abstrait Liaison **Hérite de** Arete

Concept : Arete valuée servant pour l'algorithme du plus court chemin. La valuation correspond au temps de parcours entre les deux villes. On remarque qu'une Liaison hérite d'une Arete, ainsi le constructeur creerLiaison surcharge creerSommet : il n'est pas nécessaire de réécrire les observateurs ou transformateurs déjà définis dans le Type Abstrait Arete.

Opérations de base :

Constructeur creerLiaison(Sommet sa, Sommet sb, Entier temps) : Liaison

Observateur recTemps() : Entier

Axiomes : Sommet sa, sb ; Entier tps

Pré-conditions :

définie(creerLiaison(sa, sb, tps)) ==> tps >= 0

Post-conditions :

// On gère l'observateur recS1

creerArete(sA,sB).recS1().estEgal(sA)

// On gère l'observateur recS2

creerArete(sA,sB).recS2().estEgal(sB)

Fin Liaison

c. Les Types Abstraits Graphe

1. Le Type Abstrait Graphe

Type abstrait Graphe

Concept : Un graphe est un ensemble de sommets et d'arêtes. Il est générique : les graphes que nous allons utiliser pour le programme héritent de ce graphe. On ne peut pas ajouter un sommet qui appartient déjà au graphe. De même, on ne peut pas ajouter une arête qui appartient déjà au graphe et il faut que ses deux sommets appartiennent au graphe.

Opérations de base :

Constructeur creerGraphe() : Graphe

Observateur areteAppartient(Arete a) : Booleen
Observateur sommetAppartient(Sommet s) : Booleen
Observateur recSommets() : Vecteur
Observateur recAretes() : Vecteur
Observateur recNombreSommets() : Entier
Observateur recNombreAretes() : Entier

Transformateur ajouterSommet(Sommet s) : Graphe
Transformateur ajouterArete(Arete a) : Graphe

Operations d'extension :

//Permet de récupérer tous les sommets liés au sommet 'S' par une arête
Observateur recSommetsVoisins(Sommet S) : Graphe

Axiomes : Graphe g,g1 ; Sommet s,s1,s2 ; Arete a,a1,a2

Pré-conditions :

définie(g.ajouterSommet(s)) => NON (g.sommetAppartient(s))
définie(g.ajouterArete(a)) => NON (g.areteAppartient(a)) ET
g.sommetAppartient(a.recS1()) ET g.sommetAppartient(a.recS2())
définie(g.recSommetsVoisins(s)) => g.sommetAppartient(s)

Post-conditions :

// On gère l'observateur areteAppartient

NON creerGraphe().areteAppartient(a)
a1.estEgal(a2) => g.ajouterArete(a1).areteAppartient(a2)
NON a1.estEgal(a2) =>g.ajouterArete(a1).areteAppartient(a2) = g.areteAppartient(a2)

// On gère l'observateur sommetAppartient

NON creerGraphe().sommetAppartient(s)
s1.estEgal(s2) => g.ajouterSommet(s1).sommetAppartient(s2)
NON s1.estEgal(s2) =>g.ajouterSommet(s1).sommetAppartient(s2) = g.sommetAppartient(s2)

// On gère l'obervateur recNombreSommets

creerGraphe().recNombreSommets() = 0
g.ajouterSommet(s).recNombreSommets() = 1 + g.recNombreSommets()

// On gère l'observateur recNombreAretes

creerGraphe().recNombreAretes() = 0
g.ajouterArete(a).recNombreAretes() = 1 + g.recNombreAretes()

// On gère l'observateur recSommets

g.recSommets().borneInf() = 1
g.recSommets().borneSup() = g.recNombreSommets()
g1.egalA(g.ajouterSommet(s)) =>
g1.recSommets().recVal(g1.recNombreSommets()).estEgal(s)

// On gère l'observateur recAretes

g.recAretes().borneInf() = 1
g.recAretes().borneSup() = g.recNombreAretes()
g1.egalA(g.ajouterArete(a)) => g1.recAretes().recVal(g1.recNombreAretes()).estEgal(a)

Opérations d'extension : recSommetsVoisins(Sommet s) : vecteur de sommets

Références locales

```
                vectArete,vectSommetsVoisins : Graphe
                nbVoisins : Entier
DEBUT
        vectArete <-- g.recAretes()
        nbVoisins <-- 0
        Pour i <-- vectArete.borneInf() à vectArete.borneSup() pas 1 Faire
                Si (vectArete.recVal(i).recS1().estEgal(s)) OU
                                        (vectArete.recVal(i).recS2().estEgal(s)) Alors
                        nbVoisins <-- nbVoisins + 1
                FinSi
        FinPour
        vectSommetsVoisins <-- creerVecteur(1,nbVoisins)
        nbVoisins <-- 1
        Pour i <-- vectArete.borneInf() à vectArete.borneSup() pas 1 Faire
                // Si le premier sommet est s
                Si (vectArete.recVal(i).recS1().estEgal(s)) Alors
                        // Alors le second sommet est son voisin
                        vectSommetsVoisins.affVal(nbVoisins ,
                                                vect.Arete.recVal(i).recS2())
                        nbVoisins <-- nbVoisins + 1
                Sinon
                        // Si le second sommet est s
                        Si (vectArete.recVal(i).recS2().estEgal(s)) Alors
                                // Alors le premier sommet est son voisin
                                vectSommetsVoisins.affVal(nbVoisins ,
                                                vect.Arete.recVal(i).recS1())
                                nbVoisins <-- nbVoisins + 1
                        FinSi
                FinSi
        FinPour
        retourner vectSommetsVoisins
FIN
```

Fin Graphe

2. Le Type Abstrait ReseauTransport

Type abstrait ReseauTransport **Hérite de** Graphe

Concept : Ensemble de Villes et d'Arêtes Valuées permettant de modéliser le Réseau de transport. On doit implémenter l'algorithme de Dijkstra. Celui-ci retourne un vecteur qui est la liste des villes par lesquelles le train doit passer. On remarque qu'un ReseauTransport hérite d'un Graphe, ainsi le constructeur creerReseauTransport surcharge creerGraphe : il n'est pas nécessaire de réécrire les observateurs ou transformateurs déjà définis dans le Type Abstrait Graphe. Les transformateurs ajouterVille et ajouterLiaison surchargent respectivement les transformateurs ajouterSommet et ajouterArete du Type Abstrait Graphe.

Opérations de base :

Constructeur creerReseauTransport() : ReseauTransport

Transformateur ajouterVille(Ville v) : ReseauTransport
Transformateur ajouterLiaison(Liaison l) : ReseauTransport

Opérations d'extension :

//algorithme de Dijkstra pour trouver le plus court chemin entre 2 villes
Observateur calculerChemin(Ville origine, Ville destination) : Vecteur

Axiomes : ReseauTransport res ; Ville v1, v2

Pré-conditions :

// L'indice d'une ville ajoutée correspond à son classement dans l'ordre d'ajout, et correspondra donc à sa position dans le vecteur
definie(res.ajouterVille(v1)) => v1.recIndice() .estEgal(res.recNombreSommets()+1)
définie(res.calculerChemin(v1, v2)
=> res.sommetAppartient(v1) ET res.sommetAppartient(v2)

Opérations d'extension : calculerChemin(Ville origine, Ville Destination)

```
        Références locales :
                vectSommet : vecteur
                vectDist : vecteur
                vectPred : vecteur
                TousLesSommetsMarques : boolean
DEBUT
        // Création des vecteurs
        vectSommet ← rt.recSommets()
        vectDist ← creerVecteur( 1 , rt.recNombreSommets() )
        vectPred ← creerVecteur( 1 , rt.recNombreSommets() )

        // Initialisation des distances
        Pour i ← vectDist.recBorneInf() à vectDist.recBorneSup() pas 1 Faire
                vectDist.affVal(i,-1)
        FinPour
        vectDist.affVal( origine.recIndice() , 0 )

        // RAPPEL : l'indice d'une ville est sa position dans le vecteur

        TousLesSommetsMarques ← FAUX
        TantQue NON TousLesSommetsMarques Faire
                // On récupère le sommet non marqué le moins cher
                pos_min ←  -1
                Pour i ← vectSommet.recBorneInf() à vectSommet.recBorneSup() Faire
                        Si  NON vectSommet.recVal(i).estMarque Alors
                                Si pos_min = -1 Alors
                                        pos_min ← i
                                Sinon
                                        Si  vectDist.recVal(i) < vectDist.recVal(pos_min) Alors
                                                        pos_min ← i
                                        FinSi
                                FinSi
                        FinSi
                FinPour
                // On stocke le nouveau pivot ( sommetTMP )
                sommetTmp .EgalA(vectSommet.recVal(pos_min))
                // On récupère les voisins du sommet en question
                vectTmp ← rt.recSommetsVoisins(sommetTmp)
                // On fait les MAJ éventuelles
                Pour i ← vectTmp.recBorneInf() à vectTmp.recBorneSup() pas 1 Faire
                        Si NON vectTmp.recVal(i).estMarque() Alors
                                // On récupère l'arete concernée
                                Pour vectArete.recBorneInf() à vectArete.recBorneSup()Faire
                                        Si  ( ( vectArete.recVal(i).recS1().estEgal(sommetTmp)
ET  vectArete.recVal(i).recS2().estEgal(vectTmp.recVal(i)) )
                                                OU
(vectArete.recVal(i).recS1(vectTmp.recVal(i)).estEgal() ET
vectArete.recVal(i).recS2().estEgal(sommetTmp) ) ) Alors
                                                areteTmp ← vectArete.recVal(i)
                                        FinSi
                                FinPour


// On verifie que le nouveau chemin est plus court
                                Si vectDist( vectTmp.recVal(i).recIndice() ) >
vectDist(vectTmp.recIndice()) + areteTmp.recTemps() Alors
                                                vectDist( vectTmp.recVal(i).recIndice() ) ←
vectDist(vectTmp.recIndice()) + areteTmp.recTemps()
```

```
                                        vectPre ( vectTmp.recVal(i).recIndice() ) ←
sommetTmp
                    FinSi
               FinSi
          FinPour
          // On marque le sommet courant
          sommetTmp.marquer()
          // On vérifie s'il reste des sommets non marqués
          TousLesSommetsMarques ← VRAI
          Pour i ← vectSommet.recBorneInf() à vectSommet.recBorneSup() pas 1
Faire
               Si (NON vectSommet.recVal(i).estMarque) Alors
                    TousLesSommetsMarques ← FAUX
                    Break  // On sort de la boucle pour
               FinSi
          FinPour

     FinTantQue
FIN
```

Fin ReseauTransport

3. Le Type Abstrait ListeProduit

Type abstrait ListeProduits **Hérite de** Graphe

Concept : Ensemble de Produits et d'Aretes permettant de modéliser les incompatibilités entre les produits. On doit implémenter l'algorithme de coloration de graphes afin de répartir les produits selon leurs incompatibilités. On remarque qu'une ListeProduit hérite d'un Graphe, ainsi le constructeur creerListeProduit surcharge creerGraphe : il n'est pas nécessaire de réécrire les observateurs ou transformateurs déjà définis dans le Type Abstrait Graphe. Le transformateur ajouterProduit surcharge le transformateur ajouterSommet.

Opérations de base :

Constructeur creerListeProduits() : Graphe

Transformateur ajouterProduit(Produit p) : ListeProduits

Opérations d'extension :

// Algorithme de coloration de graphes pour répartir les produits
Transformateur affecterNumerosWagon() : ListeProduits

Axiomes : ListeProduits lp

Pré-conditions :

Post-conditions :

Opérations d'extension : affecterNumeroWagon()
Références locales :
couleur : entier
vectDeg : vecteur d'entier
tmp : vecteur d'arete
cpt : entier
pos : entier

```
DEBUT
    vectDeg = creerVecteur(1,lp.recNombreSommets())
    // On initialise le degré de chaque sommet à 0
    Pour i <-- 1 à lp.recNombreSommets() pas 1 Faire
        vectDeg.affVal(i,0);
    FinPour
    // On va remplir le tableau des degrés
    tmp = lp.recAretes()
    Pour j <-- tmp.recBorneInf() à tmp.recBorneSup() pas 1 Faire
        vectDeg.affVal( tmp.recVal(i).recS1() ,vectDeg.recVal(tmp.recVal(i).recS1())
+ 1 )
        vectDeg.affVal( tmp.recVal(i).recS2() ,vectDeg.recVal(tmp.recVal(i).recS2())
+ 1 )
    FinPour

    couleur <-- 1
    cpt <-- vectBorneInf()
    TantQue ( cpt < vectDeg.recBorneSup() ) Faire
        // On récupère le sommet non marqué de degré max
        pos <-- vectDeg.recBorneInf()
        Pour i <-- vectDeg.recBorneInf() à vectDeg.recBorneSup() pas 1 Faire
            Si( vectDeg.recVal(i) > vect.recVal(pos) )Alors
                pos <-- i
            FinSi
        FinPour
        // On lui affete la nouvelle couleur
        lp.recAretes().recVal(pos).affNumWagon(couleur)
        // On incremente les compteurs
        cpt <-- cpt + 1
        // On met à jour le vectDeg
        vectDeg.affVal(pos,-1)
```

```
            // On met à jour les autres sommets
            Pour i <-- lp.recSommets().recBorneInf() à lp.recSommets().recBorneSup()
pas 1 Faire
                  Si ( NON lp.recSommets().recVal(i).estAffecte() ) Alors
                        tmp <-- lp.recSommetsVoisins( lp.recSommets().recVal(i) )
                        affectable <-- VRAI
                        Pour j <-- tmp.recBorneInf() à tmp.recBorneSup() pas 1
Faire
                              Si ( tmp.recVal(j).estAffecte() ) Alors
                                    Si (
tmp.recVal(j).recNumWagon().estEgal(couleur) ) Alors
                                          affectable <-- FAUX
                                    FinSi
                              FinSi
                        FinPour
                        Si affectable Alors
                              lp.recSommets().recVal(i).affNumWagon(couleur)
                              cpt <-- cpt + 1
                              vectDeg.affVal(i,-1)
                        FinSi
                  FinSi
            FinPour
            // On change la couleur
            couleur <-- couleur + 1
      FinTantQue
FIN
```

Fin ListeProduits

d. Le Type Abstrait Wagon

Type abstrait Wagon

Concept : Rassemble plusieurs produits compatibles. Est associé à un numéro. Fait partie d'un train. A sa construction, le wagon est vide. Les numéros de wagon vont de 1 à nbWagons.

Opérations de base :

Constructeur creerWagon(Entier numero) : Wagon

Observateur recNumero() : Entier
Observateur recProduits() : Vecteur
Observateur produitAppartient(Produit p) : Booleen
Observateur recNombreProduits() : Entier

Transformateur ajouterProduit(Produit p) : Wagon

Axiomes : Produit p ; Wagon w,w1 ; Entier n

Pré-conditions :

définie(creerWagon(n)) => n>0
définie(w.ajouterProduit(p)) => NON (w.produitAppartient(p))

Postconditions :

// On gère l'oservateur recNumero

creerWagon(n).recNumero().estEgal(n)

// On gère observateur produitAppartient

NON creerWagon(n).produitAppartient(p)
p1.estEgal(p2) => w.ajouterProduit(p1).produitAppartent(p2)
NON p1.estEgal(p2) =>
w.ajouterProduit(p1).produitAppartient(p2) = w.produitAppartient(p2)

// On gère l'observateur recNombreProduits

creerWagon(n).recNombreProduits() = 0
w.ajouterProduit(p).recNombreProduits() = 1 + w.recNombreProduits()

// On gère l'observateur recProduits

w.recProduits().borneInf() = 1
w.recProduits().borneSup() = w.recNombreProduits()
w1.egalA(w.ajouterProduit(p)) =>
w1.recProduits().recVal(w1.recNombreProduits()).estEgal(p)

Fin Wagon

e. Le Type Abstrait Train

Type abstrait Train

Concept : A chaque commande est associé un train, qui contient les wagons nécessaires à l'acheminement de tous les produits de la commande.

Opérations de base :

Constructeur creerTrain() : Train

Observateur recWagons() : Vecteur
Observateur wagonAppartient(Entier num) : booleen
Observateur recNombreWagons() : Entier

Transformateur ajouterWagon(Wagon w) : Train

Axiomes : Wagon w ; Train t

Pré-conditions :

définie(t.ajouterWagon(w)) => NON (t.wagonAppartient(w.recNumero()))

Postconditions :

// On gère l'observateur wagonAppartient

NON creerWagon().wagonAppartient(w)
w1.estEgal(w2) => t.ajouterWagon(w1).produitWagon(w2)
NON w1.estEgal(w2) =>
t.ajouterWagon(w1).wagonAppartient(w2) = t.wagonAppartient(w2)

// On gère l'observateur recNombreWagons

creerTrain().recNombreWagons() = 0
t.ajouterWagon(w).recNombreWagons() = 1 + g.recNombreWagons()

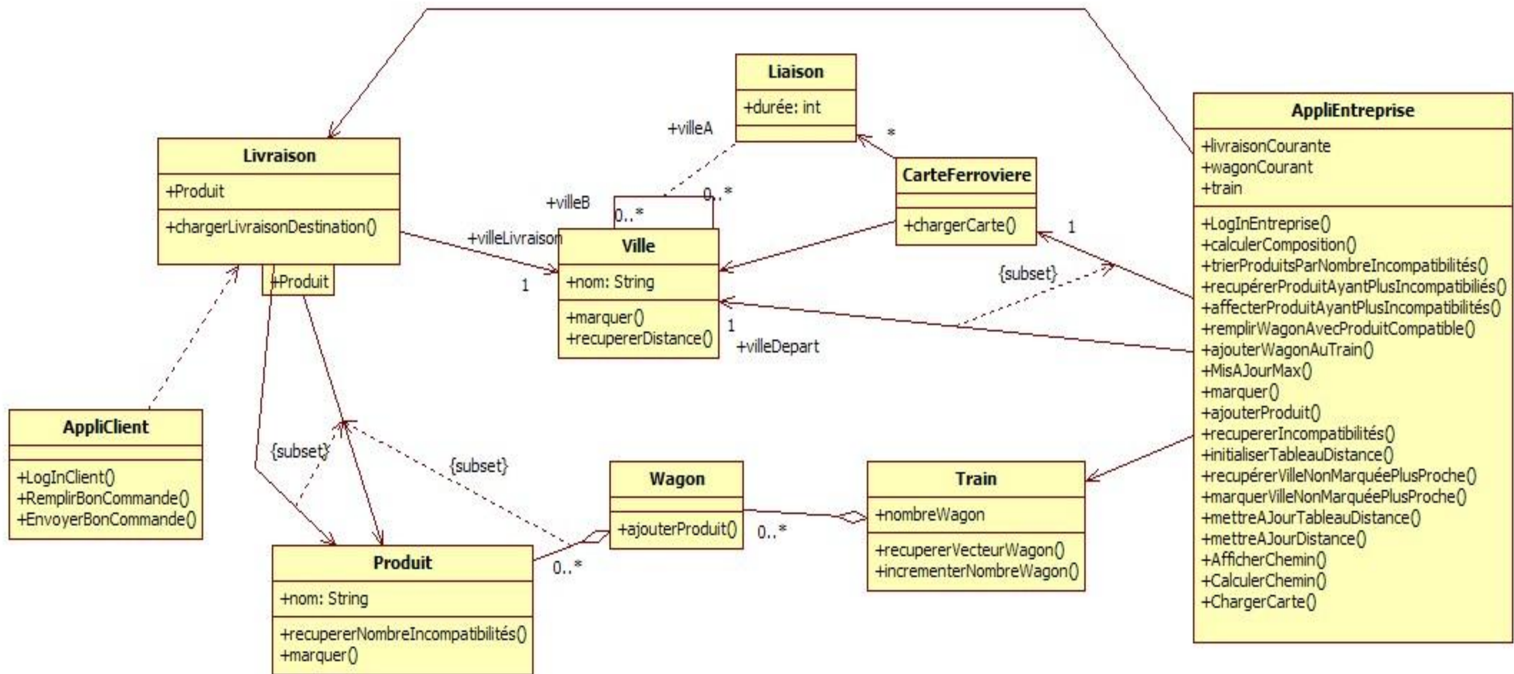
// On gère l'observateur recWagons

t.recWagons().borneInf() = 1
t.recWagons().borneSup() = w.recNombreWagons()
t1.egalA(t.ajouterWagon(w)) =>
t1.recWagons().recVal(t1.recNombreWagons()).estEgal(w)

FIN Train

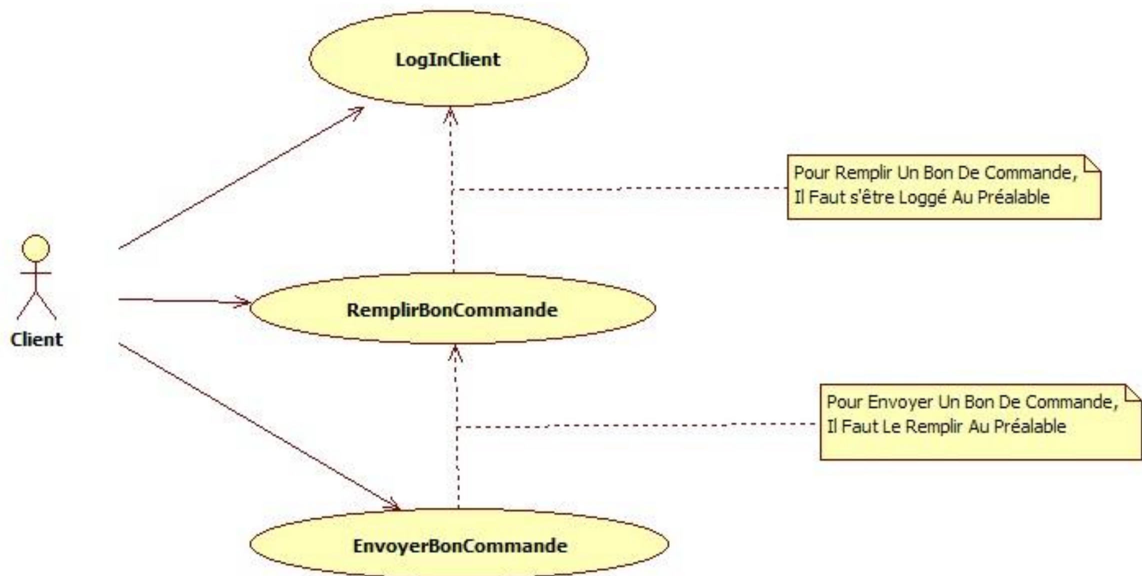
III. L’analyse dynamique

Pour l’analyse dynamique du modèle Voie Ferrée, nous avons décidé, par souci de commodité, de nous baser sur le troisième corrigé proposé. Pour rappel, voici le diagramme de classe corrigé (avec les modifications apportées pour nos besoins) :



a. Use Case du client

Le premier cas d'utilisation que nous traitons est celui du client. Nous avons considéré qu'un client pouvait faire les actions suivantes : se connecter à l'application client (LogInClient), remplir une commande (remplirBonCommande) et enfin valider une commande, ce qui équivaut à l'enregistrer (EnvoyerCommande). Ces trois actions sont indépendantes dans le sens où dans l'action remplirCommande il n'y a pas d'appel à la fonction LogInClient, cependant certaines actions nécessitent qu'une autre est effectuée auparavant.



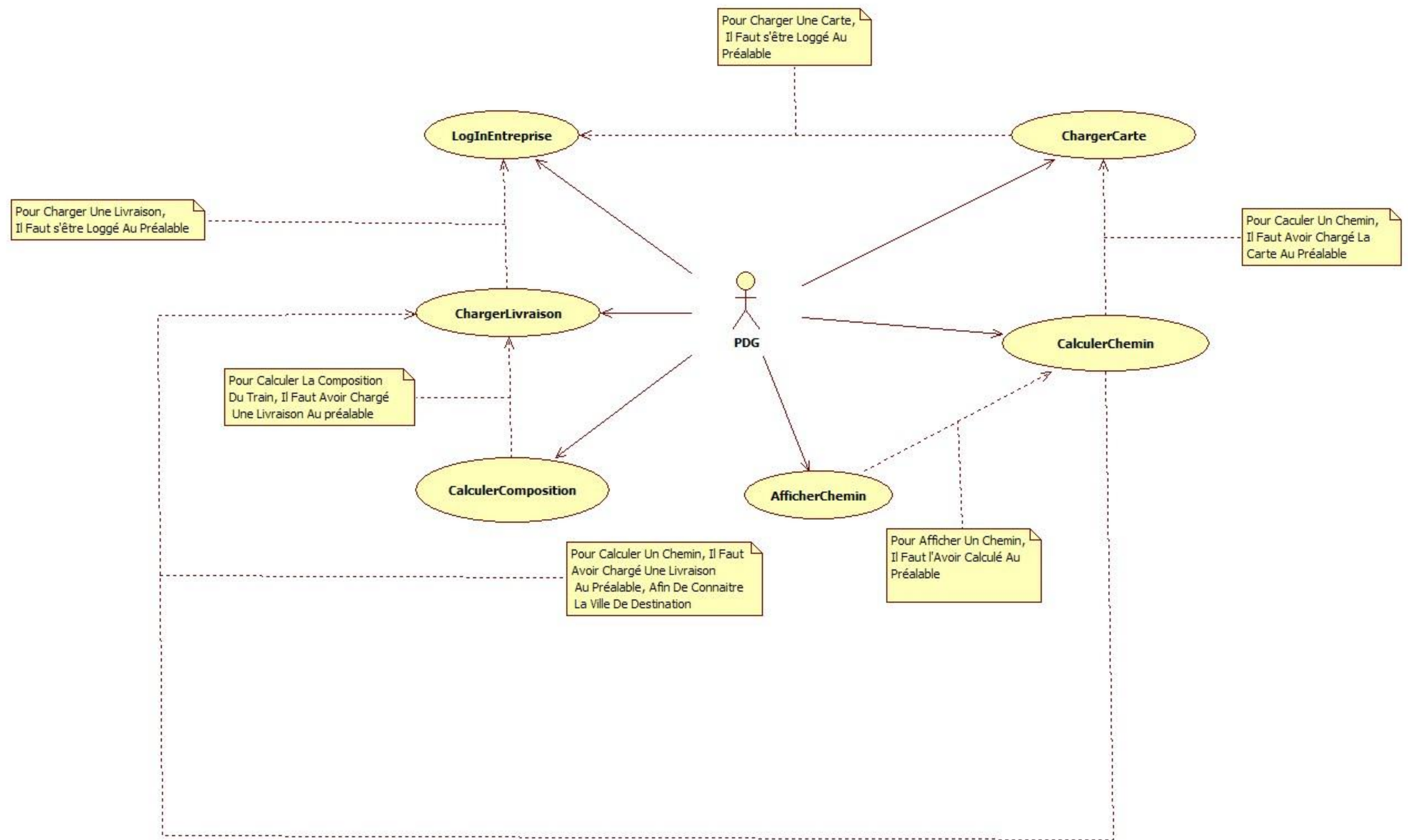
b. Use Case du PDG

Le second cas d'utilisation que nous traitons est celui d'un utilisateur de l'application client. L'acteur sera nommé PDG. Voici la liste des actions possibles pour le PDG :

- Se connecter à l'application entreprise (LogInEntreprise)
- Charger une carte à partir d'un fichier externe (ChargerCarte)
- Charger une livraison à partir d'un fichier externe (ChargerLivraison)
- Calculer la composition d'un train (CalculerComposition)
- Calculer le plus court chemin entre deux villes (CalculerChemin)
- Afficher le plus court chemin entre deux villes (AfficherChemin)

Toutes ces actions sont réalisables directement par l'utilisateur, et sont indépendantes du même point de vue que précédemment. Il est clair que pour pouvoir AfficherChemin, il faut d'abord avoir pu CalculerChemin. Mais pour CalculerChemin, on a besoin de ChargerCarte pour connaitre le réseau ferrovière et de ChargerLivraison pour connaitre la ville de destination. Pour effectuer chacun de ces chargements, il faut s'être connecté à l'application au préalable (LogInEntreprise).

Voici donc le diagramme d'utilisation pour un PDG.



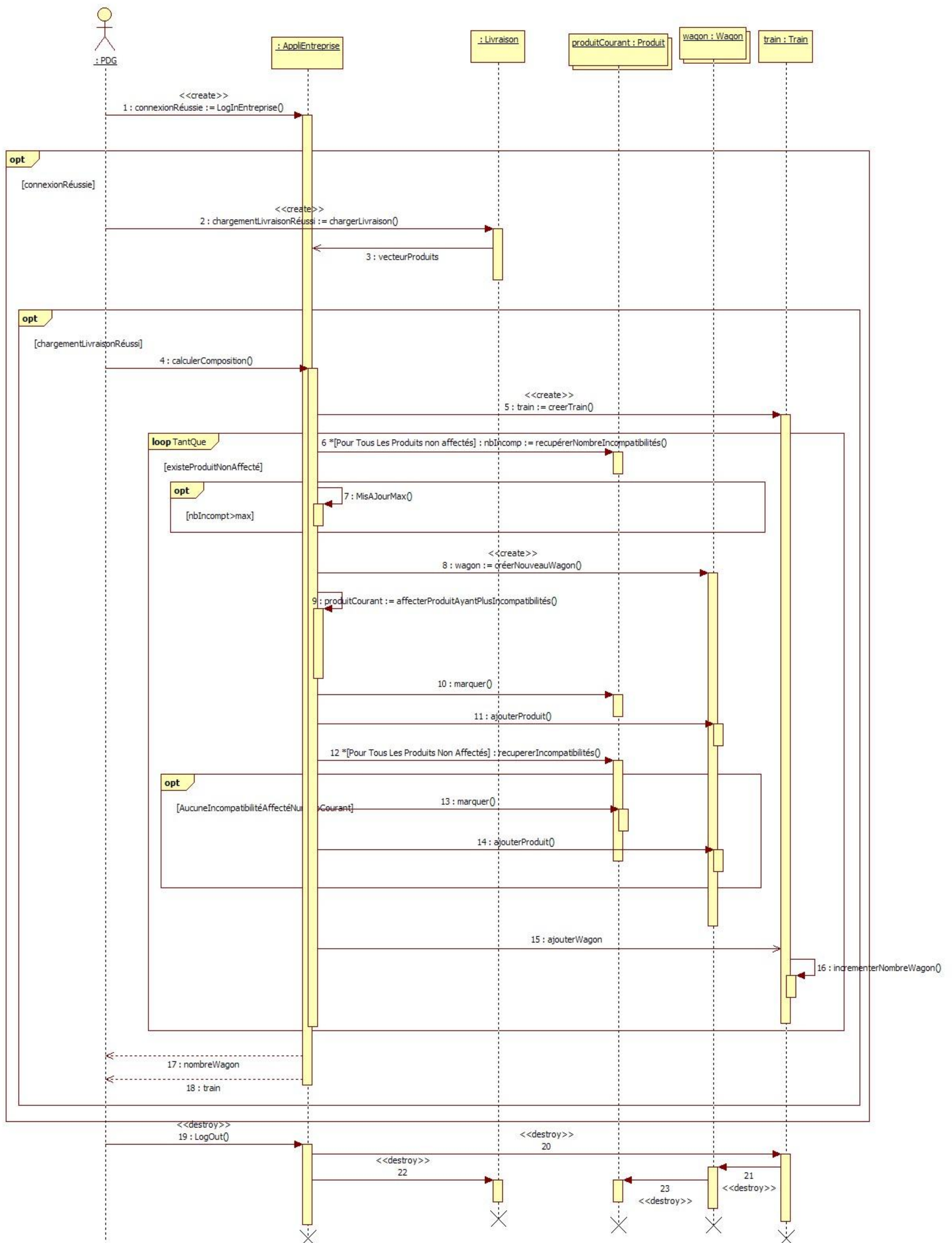
c. Le diagramme de séquence du calcul de la composition

Le diagramme de séquence du calcul de la composition du train, permet de déterminer le nombre de wagons nécessaires pour satisfaire les contraintes d'incompatibilité des produits. Il s'agit en réalité d'une application de l'algorithme glouton de coloriage d'un graphe.

Comme nous l'avons expliqué dans le diagramme de cas d'utilisation, certaines actions ne peuvent être effectuées à condition qu'une autre action ait eu lieu au préalable. Ainsi, pour plus de clarté, nous présentons dans notre diagramme de séquence les autres actions qui doivent avoir été faites auparavant : dans le cas du calcul de la composition du train, il faut d'abord se logger, puis charger une livraison avant de pouvoir calculer sa composition.

Si ces deux opérations ont réussi avec succès, alors l'utilisateur peut calculer la composition du train. La première étape consiste à créer un nouvel objet de type Train. Ensuite, tant qu'il reste un produit non affecté, on va récupérer le produit non affecté qui possède le plus grand nombre d'incompatibilités. On crée un nouveau wagon, on affecte le produit à ce wagon, puis on complète le wagon avec des produits non affectés en veillant à ne pas avoir d'incompatibilité dans un même wagon. Lorsque le wagon est rempli, on l'ajoute au train, et on recommence l'opération jusqu'à épuisement des produits.

Une dernière remarque sur le diagramme. Les signatures de certaines opérations ne sont pas affichées sur le schéma suivant. Les opérations en question sont ajouterProduit(Produit p) et ajouterWagon(Wagon w).



: PDG
: AppliEntreprise
: Livraison
produitCourant : Produit
wagon : Wagon
train : Train
<<create>>
1 : connexionRéussie := LogInEntreprise()
opt
[connexionRéussie]
<<create>>
2 : chargementLivraisonRéussi := chargerLivraison()
3 : vecteurProduits
opt
[chargementLivraisonRéussi]
4 : calculerComposition()
<<create>>
5 : train := creerTrain()
loop TantQue
[existeProduitNonAffecté]
6 *[Pour Tous Les Produits non affectés] : nbIncomp := recupérerNombreIncompatibilités()
opt
[nbIncompt>max]
7 : MisAJourMax()
<<create>>
8 : wagon := créerNouveauWagon()
9 : produitCourant := affecterProduitAyantPlusIncompatibilités()
10 : marquer()
11 : ajouterProduit()
12 *[Pour Tous Les Produits Non Affectés] : recupererIncompatibilités()
opt
[AucuneIncompatibilitéAffectéNu Courant]
13 : marquer()
14 : ajouterProduit()
15 : ajouterWagon
16 : incrementerNombreWagon()
17 : nombreWagon
18 : train
<<destroy>>
19 : LogOut()
<<destroy>>
20
<<destroy>>
22
21
<<destroy>>
23
<<destroy>>

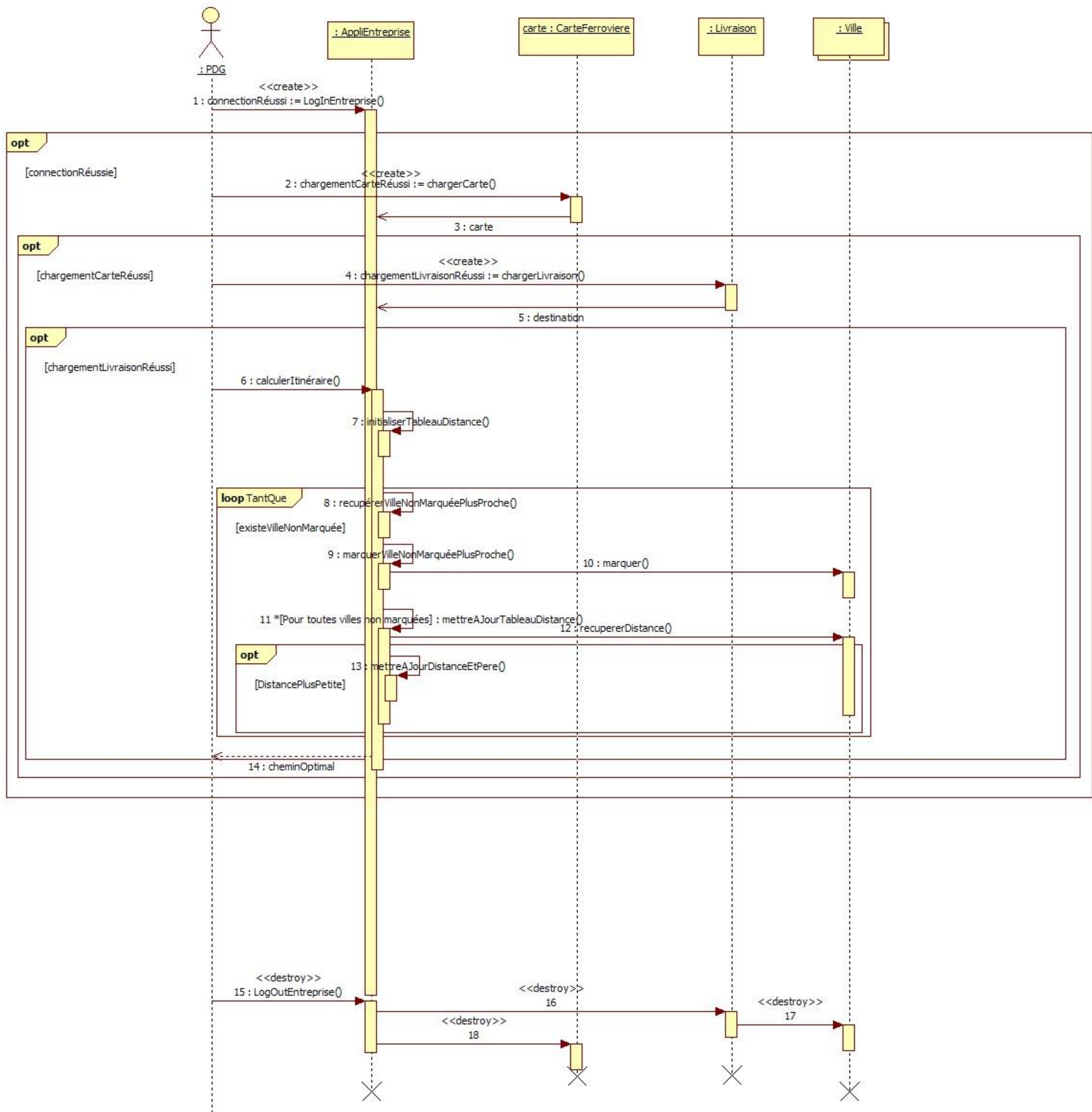
d. Le diagramme de séquence du calcul du chemin

Le diagramme de séquence du calcul du chemin, permet de déterminer le meilleur chemin permettant de rejoindre une ville de départ à une ville de destination. Dans notre application, la ville de départ est toujours la même, celle dans laquelle se trouve le siège de la société. Il s'agit en réalité d'un application de l'algorithme de Dijkstra à notre modèle de voie ferrée.

Comme nous l'avons expliqué dans le diagramme de cas d'utilisation, certaines actions ne peuvent être effectuées à condition qu'une autre action ait eu lieu au préalable. Ainsi, pour plus de clarté, nous présentons dans notre diagramme de séquence les autres actions qui doivent avoir été faites auparavant : dans le cas du calcul du chemin, il faut d'abord se logger, charger une livraison (pour connaitre la ville de destination) puis charger la carte ferroviaire avant de pouvoir calculer le plus court chemin.

Si ces trois opérations ont réussi avec succès, alors l'utilisateur peut calculer le chemin à emprunter. La première étape de l'algorithme consiste à initialiser le tableau des distances (0 pour la ville de départ et l'infini pour toutes les autres villes). Ensuite, tant qu'une ville n'est pas marquée, on récupère la ville non marquée qui a la plus petite valeur dans le tableau des distances (on l'appelle pivot et on marque la ville en question), et on met à jour le tableau des distances : pour tous les sommets voisins de la ville pivot, si le nouveau chemin est plus court (coût jusqu'au pivot + temps de la liaison entre pivot et son voisin < coût actuel du voisin) alors on met à jour la distance et le sommet précédent du sommet voisin. Lorsque toutes les villes sont marquées, on connait le père de chacun, et à partir de chaque sommet on peut remonter jusqu'à la ville de départ.

Une dernière remarque sur le diagramme. La signature d'une certaine opération n'est pas affichée sur le schéma suivant. L'opération en question est récupérerDistance(Ville v).


: PDG
: AppliEntreprise
carte : CarteFerroviere
: Livraison
: Ville
<<create>>
1 : connectionRéussi := LogInEntreprise()
opt
[connectionRéussie]
<<create>>
2 : chargementCarteRéussi := chargerCarte()
3 : carte
opt
[chargementCarteRéussi]
<<create>>
4 : chargementLivraisonRéussi := chargerLivraison()
5 : destination
opt
[chargementLivraisonRéussi]
6 : calculerItinéraire()
7 : initialiserTableauDistance()
loop TantQue
[existeVilleNonMarquée]
8 : recupérerVilleNonMarquéePlusProche()
9 : marquerVilleNonMarquéePlusProche()
10 : marquer()
11 *[Pour toutes villes non marquées] : mettreAJourTableauDistance()
12 : recupererDistance()
opt
[DistancePlusPetite]
13 : mettreAJourDistanceEtPere()
14 : cheminOptimal
<<destroy>>
15 : LogOutEntreprise()
<<destroy>>
16
<<destroy>>
17
<<destroy>>
18

e. Quelques exemples de diagramme d'état-transition

Dans notre diagramme de classe, aucune classe ne se prête véritablement à la présentation d'un diagramme d'état transition. En effet, les états des objets sont soit unique soit binaire. Néanmoins, nous allons donner quelques exemples simples de diagramme d'état transition que nous pourrions utiliser ici.

Diagramme d'état transition d'un produit :

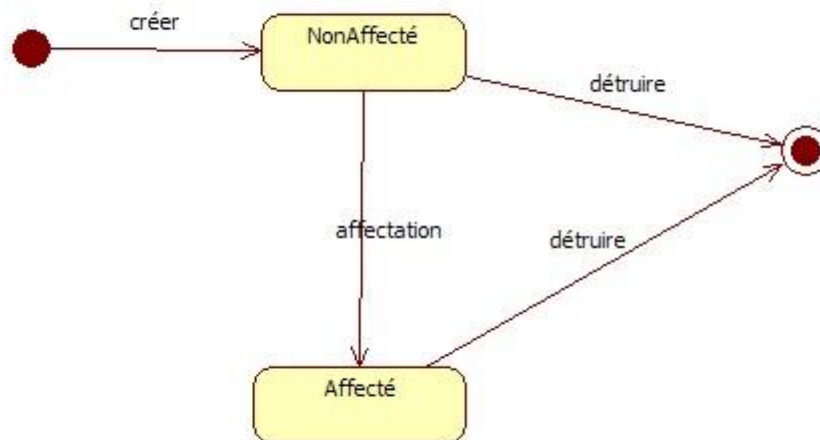


Diagramme d'état transition d'une ville