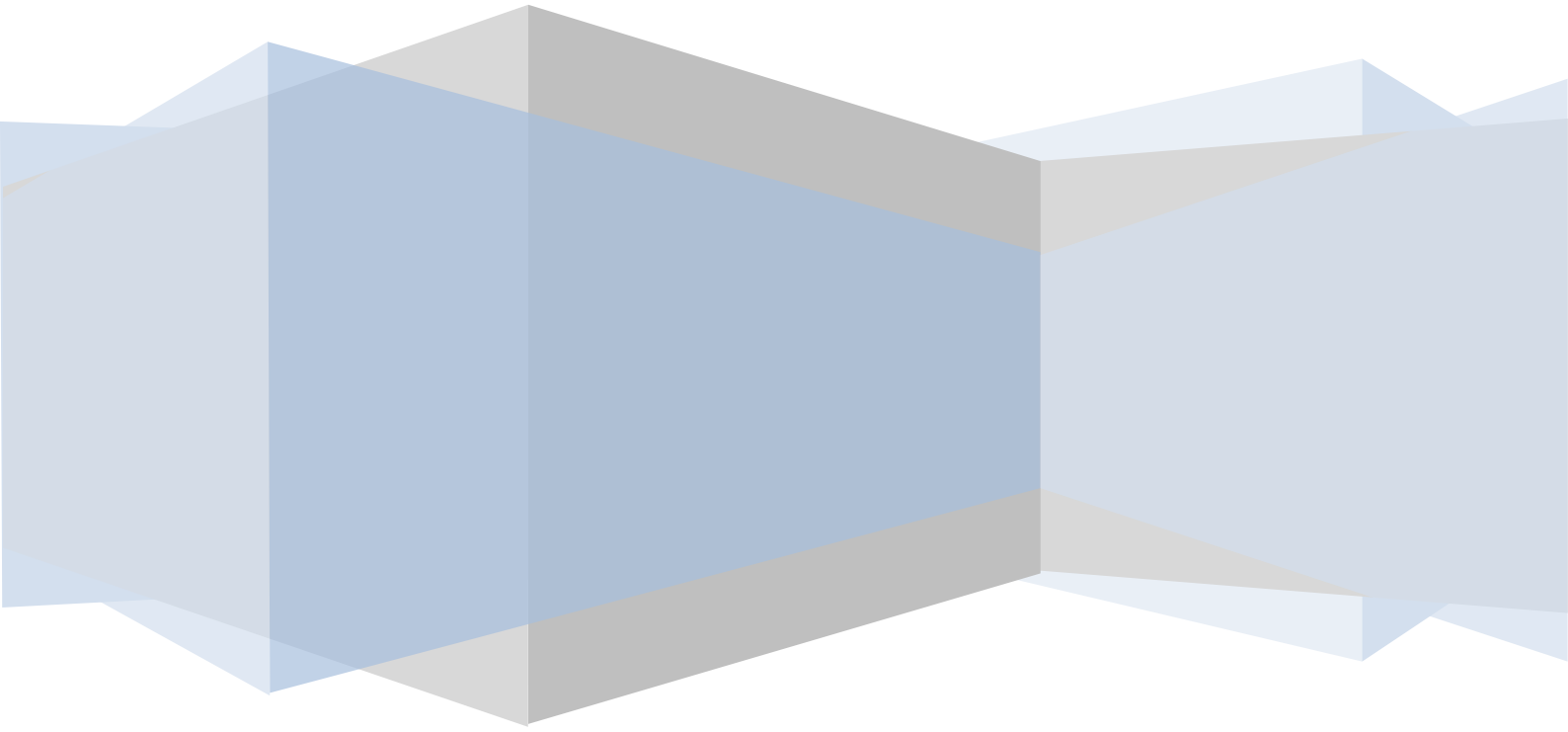
EISTI

Projet Voie Ferrée

Livrable N 1 – Analyse du modèle : TA, UML, contraintes OCL

MVE ASSEKO Simplice - Thomas TYGREAT
Thibault COUDERT – PauL TAING



Sommaire

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 Disjkstra 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 le premier livrable, nous devons avant tout modéliser le problème afin de préparer le travail du codeur, qui vient seulement par la suite. Nous devons réaliser :

- Les types abstraits permettant de modéliser les produits, les incompatibilités entre produits ainsi que la carte ferroviaire. Seules les pré-conditions sont demandées, la description des opérations sous forme axiomatique étant reportée au livrable 2.
- Les diagrammes de classe UML du programme.
- Les contraintes OCL correspondant au diagramme UML.

Types Abstraits

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

Fin Sommet

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.

Opérations de base :

Constructeur creerProduit(chaine libellé) : Produit

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

Transformateur affNumWagon(Entier num) : Produit

Fin Produit

Type abstrait Ville **Hérite de** Sommet :

Concept : Sommet qui peut être marqué, a**Fin** de réaliser l'algorithme de Disjkstra dans le cas du Reseau de transport. Une ville possède un indice, allant de 1 à nbVilles.

Opérations de base :

Constructeur creerVille(chaine nom, Indice n) : Ville

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

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

Fin Ville

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))

Fin Arete

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.

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

Fin Liaison

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 appartienent 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 ; Sommet s ; Arete a

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)

Fin Graphe

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.

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 :
définie(res.calculerChemin(v1, v2)
=> res.sommetAppartient(v1) ET res.sommetAppartient(v2)

Fin ReseauTransport

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.

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

Fin ListeProduits

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

Transformateur ajouterProduit(Produit p) : Wagon

Axiomes : Produit p ; Wagon w ; Entier n

Pré-conditions :
définie(creerWagon(n)) => n>0
définie(w.ajouterProduit(p)) => NON (w.produitAppartient(p))

Fin Wagon

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) : booleeen

Transformateur ajouterWagon(Wagon w) : Train

Axiomes : Wagon w ; Train t

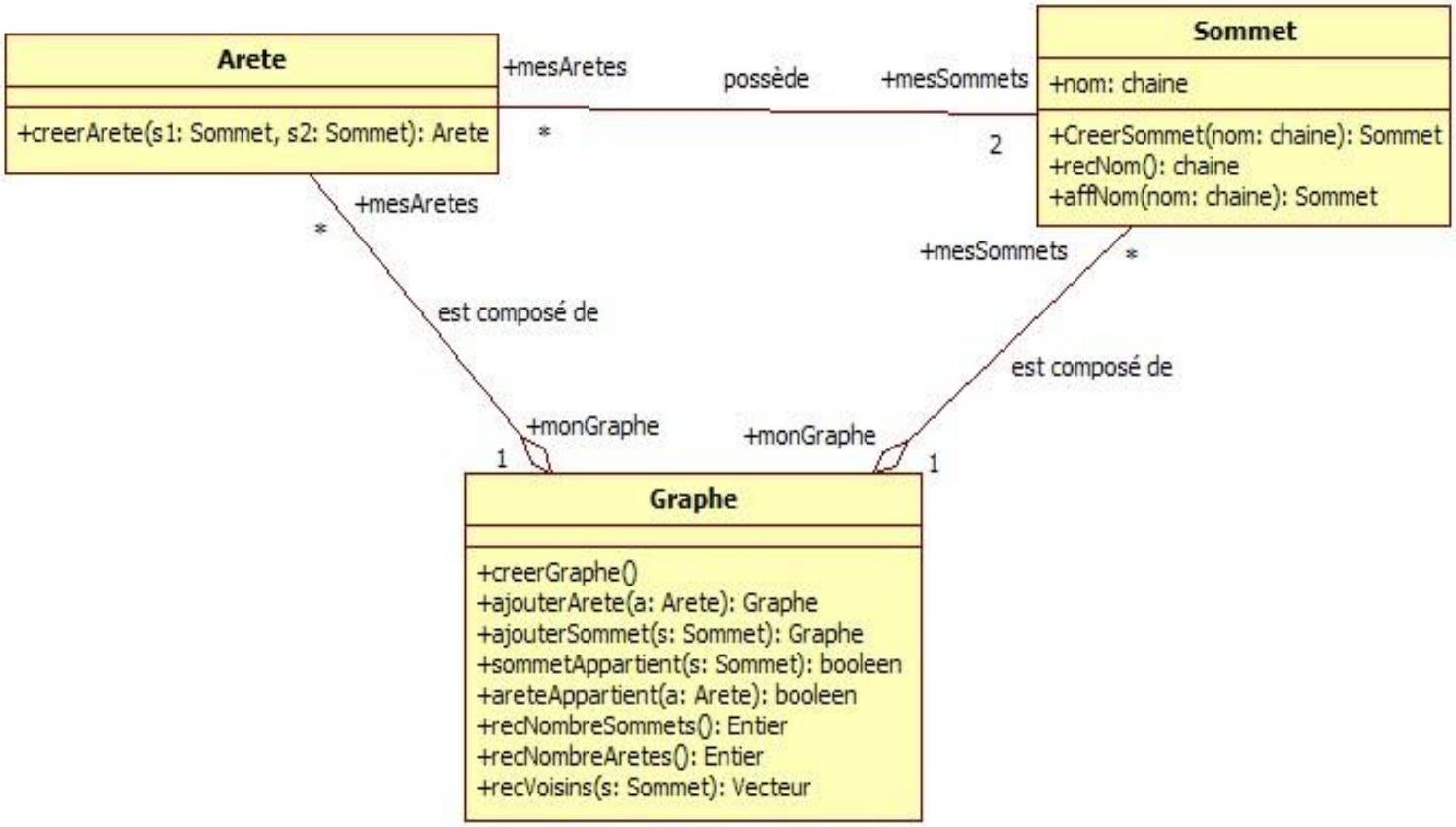
Pré-conditions :
définie(t.ajouterWagon(w)) => NON (t.wagonAppartient(w.recNumero()))

Fin Train

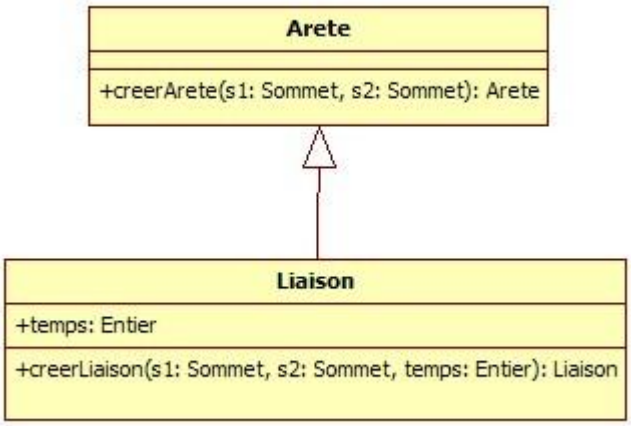
UML et contraites OCL

Etant donné que notre UML n'est pas représentable facilement sur une seule page, nous allons le construire pas à pas pour plus de clarté :

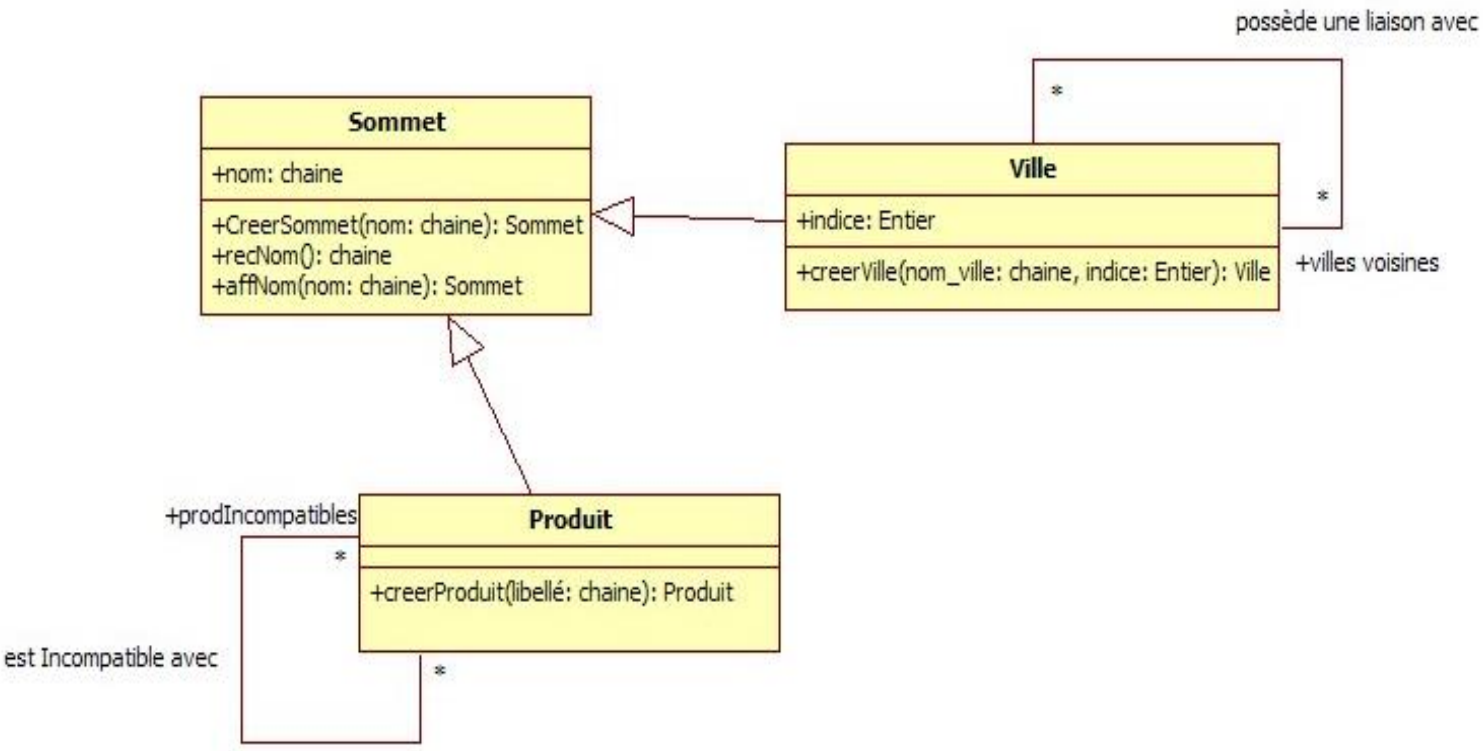
Un graphe est une agglomération de sommets et d'arêtes. Nous n'avons pas utilisé de composition car il nous semble logique qu'un Sommet ou une Arete puisse exister en dehors d'un graphe. On peut accéder aux Aretes du graphe par l'intermédiaire du rôle mesAretes et à ses sommets par l'intermédiaire du rôle mesSommets. Une arête est un ensemble de exactement 2 Sommets.

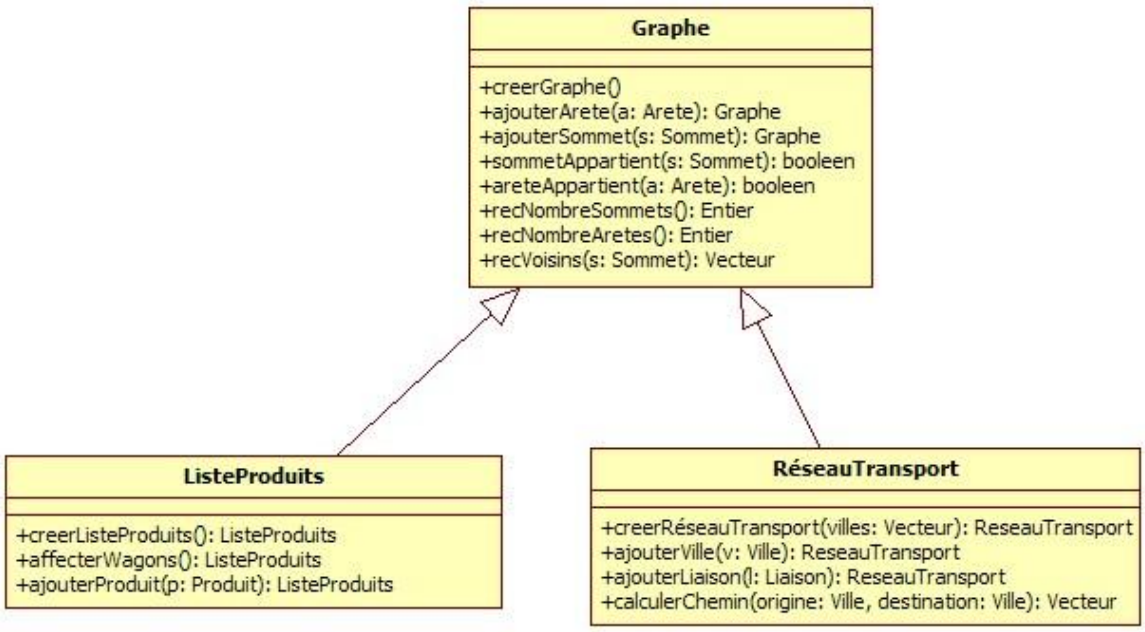


Une liaison hérite d'une arête et possède en plus un attribut 'temps' qui correspond au temps de parcours entre les deux villes :

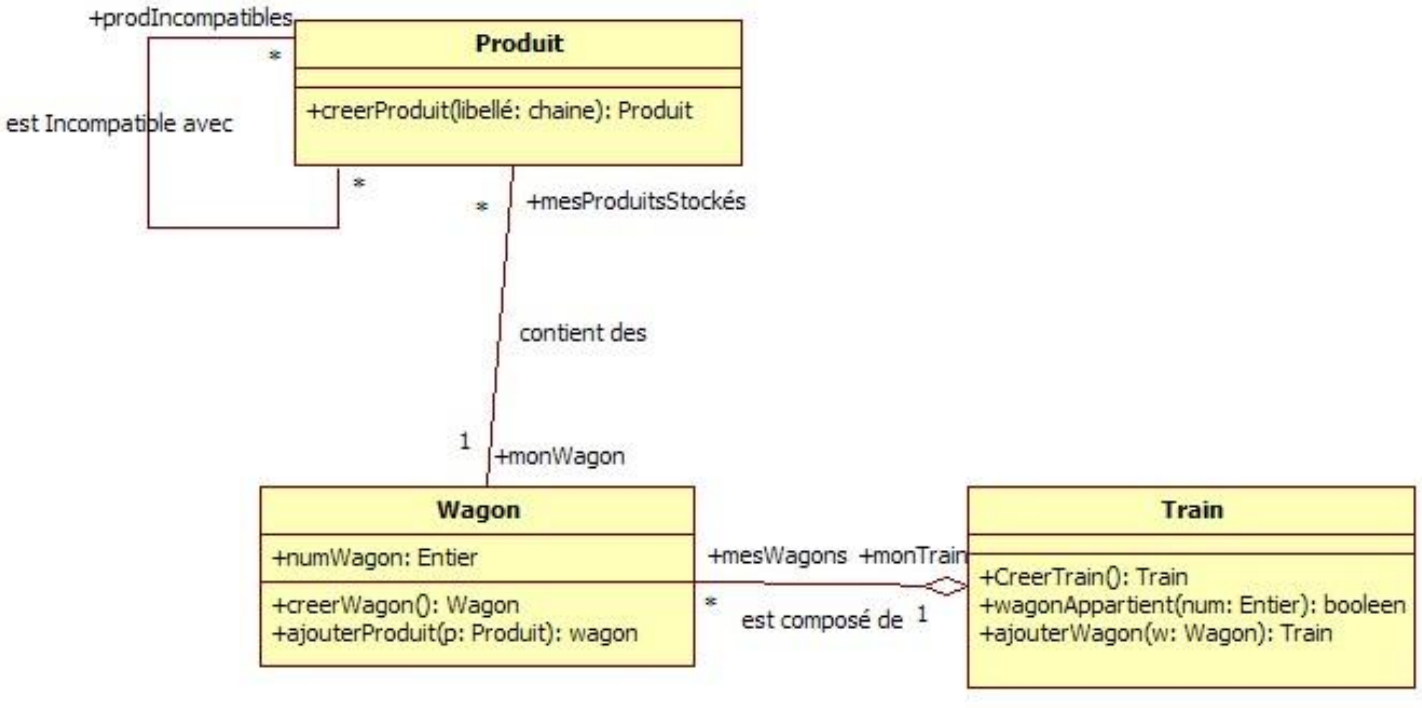


La classe Ville et la classe Produit héritent de la classe Sommet. Une Ville est réliée à plusieurs autres villes et un Produit est incompatibles avec d'autres Produits.

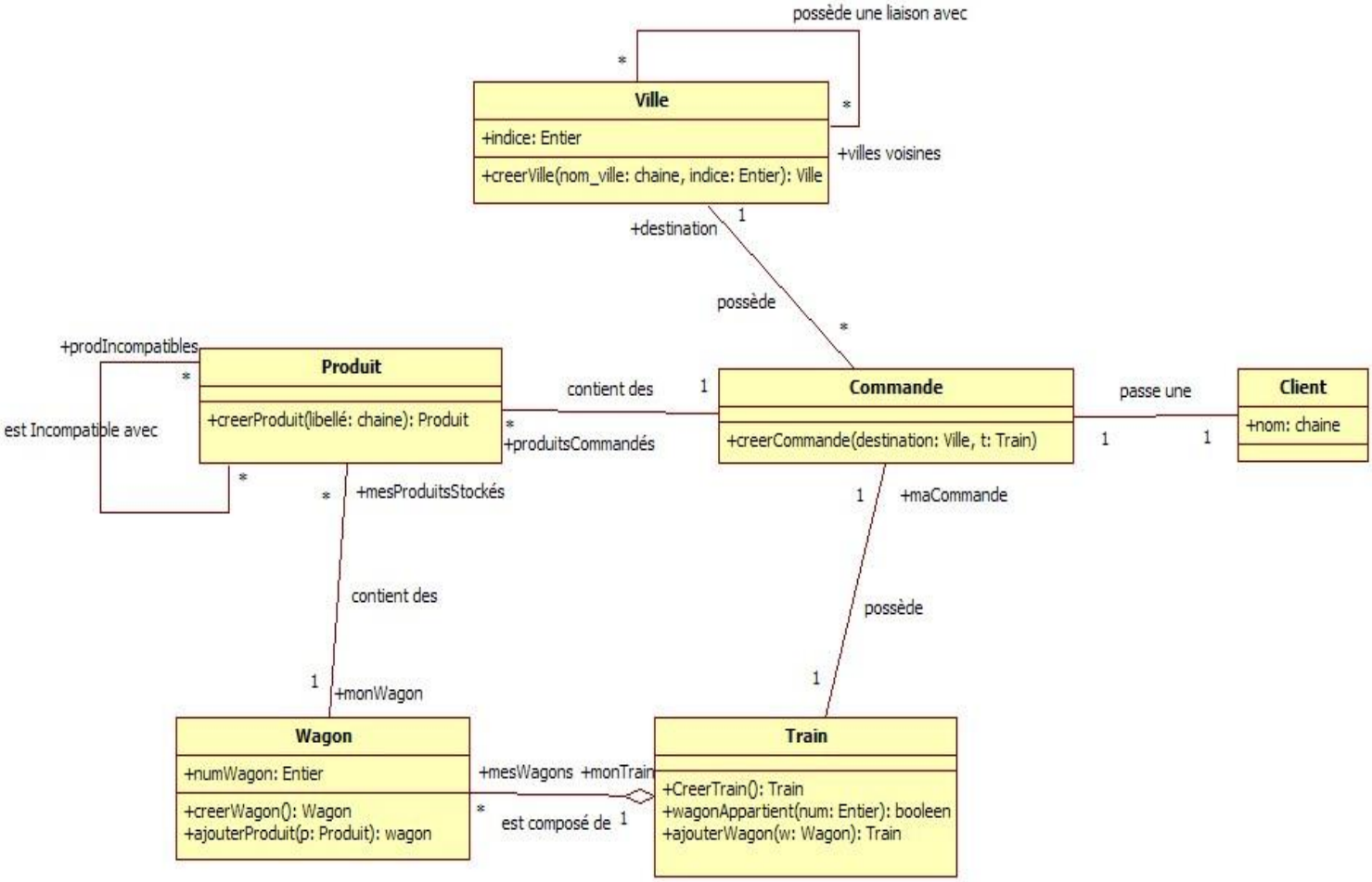




La classe ListeProduits et la classe RéseauTransport héritent de la classe Graphe. Le réseau de transport permet de calculer le plus court chemin entre 2 villes, tandis que la liste de produits permet de répartir les produits dans les wagons selon leurs incompatibilités.

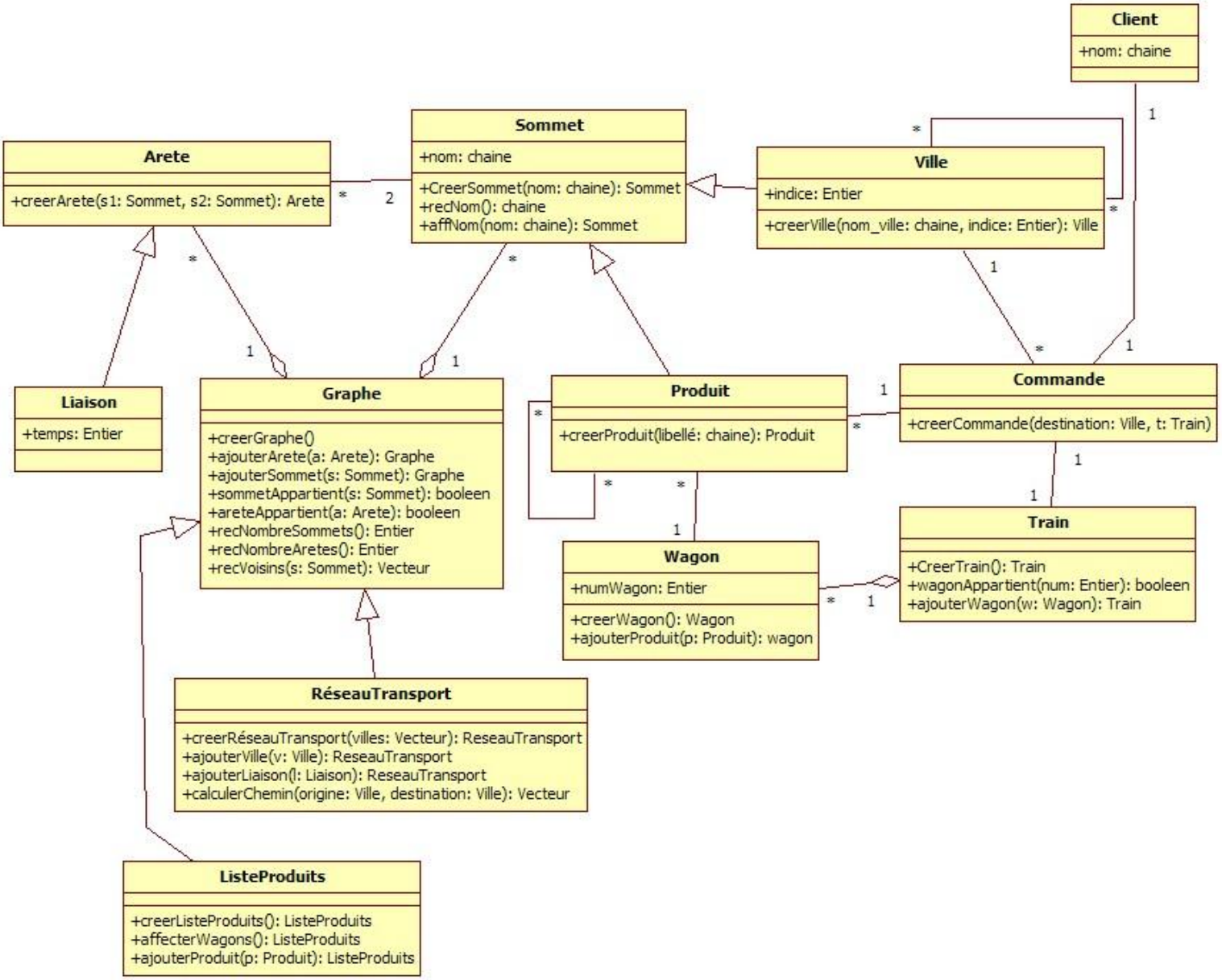


Un wagon contient des Produits. Un train est une agglomération de Wagons.



Une commande comporte un ensemble de produits, une ville qui sert de destination, un client, et un train pour connaître la répartition des produits de la commande.

Voici le schéma complet de notre diagramme UML (contraintes OCL non comprises) :



Contraintes OCL

Contrainte à exprimer : « Le temps de parcours d'une liaison doit être positif »

OCL {context Liaison inv : temps>0}

Contrainte à exprimer : « Une arête ne peut pas relier un sommet avec lui-même. En effet, un produit ne doit pas être incompatible avec lui-même et une ville n'a pas de liaison avec elle-même. »

OCL {context Arete inv : mesSommets->forAll(s1,s2 : Sommet | s1 <> s2 implies s1.nom<>s2.nom)}

Contrainte à exprimer : « On ne peut pas ajouter au graphe un sommet qui s'y trouve déjà. »

OCL { context Graphe :: ajouterSommet(Sommet s):Graphe
pre : NOT self.mesSommets->includes(s)
}

Contrainte à exprimer: « Pour ajouter une arête au graphe, il faut s'assurer qu'elle ne s'y trouve pas déjà d'une part, et que ses deux sommets sont bien présents dans le graphe d'autre part. »

OCL { context Graphe :: ajouterArete(Arete a)
pre : NOT self.mesAretes->includes(a)
AND self.mesSommets->includesAll(a.mesSommets) }

Contrainte à exprimer : « pour récupérer les sommets voisins d'un Sommet, celui-ci doit appartenir au graphe »

OCL { context Graphe :: recSommetsVoisins(Sommet s):Vecteur
pre : self.mesSommets->includes(s)
}

Contrainte à exprimer : « un produit ne peut pas être incompatible avec lui-même »

OCL {context Produit : prodIncompatibles->excludes(self)}

Contrainte à exprimer : « une ville ne peut pas être reliée à elle-même »

OCL {context Ville : villesReliées->excludes(self)}

Contrainte à exprimer : « avant d'ajouter un wagon au train, il faut s'assurer que celui-ci ne s'y trouve pas déjà et qu'il n'y a pas d'autre wagon ayant le même numéro »

OCL {contextTrain :: ajouterWagon(w :Wagon)
pre : NOT self.mesWagons->includes(w) }
AND NOT self.mesWagons->forAll(w2 : Wagon | w2.numWagon<>w.numWagon)
}

Contrainte à exprimer : « tous les produits commandés par le client doivent être stockés dans le train »

OCL {context Train inv :
self.mesWagons->mesProduitsStockés->includesAll(self.maCommande->produitsCommandes)
}

Contrainte à exprimer : « tous les produits d'un même wagon sont compatibles »

OCL {context Wagon inv :
mesProduits->forAll (p1,p2 : Produit | NOT (p1.prodIncompatibles->includes(p2)))
}

Contrainte à exprimer : « On ne peut pas ajouter un produit à un wagon si celui s'y trouve déjà »

OCL {context Wagon :: ajouterProduit(p : Produit)
Pre : NOT (self.mesProduits->includes(p))
}

Contrainte à exprimer : « Les produits placés dans un wagon sont bien des produits commandés par le client »

{context Wagon inv : self.monTrain.maCommande.prodCommandés->includesAll(self.mesProduits)}

Contrainte à exprimer : « Pour calculer le plus court chemin entre deux villes, il faut que ces villes appartienent au réseau de transport »

OCL {context ReseauTransport :: calculerChemin(Ville v1, Ville v2):Vecteur
pre : self.mesSommets->includes(v1)
AND self.mesSommets->includes(v2)
}

Contrainte à exprimer : « les sommets d'un réseau de transport sont des villes et ses arêtes sont des liaisons »

OCL {context RéseauTransport inv : mesSommets->forAll(oclIsTypeOf(Ville)) }
{context RéseauTransport inv : mesAretes->forAll(oclIsTypeOf(Liaison)) }

Contrainte à exprimer : « les sommets d'une Liste de produits sont des produits »

OCL {context ListeProduits inv : mesSommets->forAll(oclIsTypeOf(Produit)) }
