Baptiste SEUX
Paul SVENSON
Kevin CALLET
Kacem JEDOUI
Victor BERTIN

Mars - Avril - Mai 2020

Verbiage Voiture

Projet Base de données
Documentation

Avec le soutien de K. Altisen et C. Bobineau

Introduction

Ce projet intègre les notions vues dans le cours de Principes des systèmes de gestion de bases de données ainsi que le cours d'analyse et conception objet de logiciels embarqués.

L'objectif était de développer une application de covoiturage. Le travail comprend la conception d'une base de donnée permettant de définir les relations nécessaires ainsi que la programmation en JAVA de l'application. L'utilisation de l'API jdbc permet de réaliser l'interface entre la base de donnée et l'application.

Ce rapport détaillera les résultats aboutis, les limites de notre application et un bilan qui soulignera la partie gestion de projet.

Plan

I - Analyse statique

L'objectif de cette partie sera de constituer la base pour la conception de notre base de donnée.
L'étude du cahier de charges nous a permis d'identifier les propriétés élémentaires ainsi que les dépendances fonctionnelles, contraintes de valeurs, contraintes de multiplicités et autres contraintes.

Propriétés élémentaires : {idemail, nom,prenom,villeresidence, mdp, solde, idimmatriculation, Marque, Modele, Energie, Puissance, nbplace, idtrajet, datetrajet, nbplacedepart, nbtronconrestants ,idtroncon, ville_départ,ville_arrivée, longitudedepart,logitudearrive,latitudedepart,latitudearrive, distanceparcourue, tempsparcours, tempsattente, lieudepart, lieuarrivee, idparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours, suivipassager}.

A partir de ces propriétés on peut établir les dépendances et contraintes,

1) Dépendances et contraintes

Dépendances fonctionnelles	Contraintes de valeur
idemail → nom, prenom, villeresidence, mdp, solde	solde >0
idimmatriculation → Marque, Modele, Energie, Puissance, nbplace	idimmatriculation ∈ {'AB_123_CD'} Energie ∈ {essence, diesel, électrique} nbplace > 0 Puissance > 0
idtrajet → nomtrajet, datetrajet, nbplacedepart, idemail, idimmatriculation	

idtroncon → ville_départ, ville_arrivée, longitudedépart, logitudearrivé, latituredépart, latitudearrivé distanceparcourue, tempsparcours, lieudepart, lieuarrivee idparcours → dateparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours idparcours, idtroncon → suivipassager	distanceparcourue > 1km tempsparcours > 0min suivipassager ∈ { ‘Debut’, ‘Fin’ }
Contraintes de multiplicité	Autres contraintes (contextuelles)
idemail -|->> idimmatriculation idemail -|->> idparcours idemail -|->> idtrajet idimmatriculation - ->> idemail idimmatriculation -|->> idtrajet idtrajet - ->> idtroncon idtronçon -->> idparcours idtroncon -|-> tempsattente idparcours -- >> idtroncon	- Un utilisateur ne peut pas faire partie de plus de deux trajets lors d’un parcours - Le temps d’attente ne doit pas dépasser 1h entre les deux trajets - La distance ne doit excéder 1 km entre le point d’arrivée du premier trajet et de départ du deuxième trajet

A partir de ce tableau on peut élaborer le schéma Entités Associations correspondant.

2) Schéma E/A

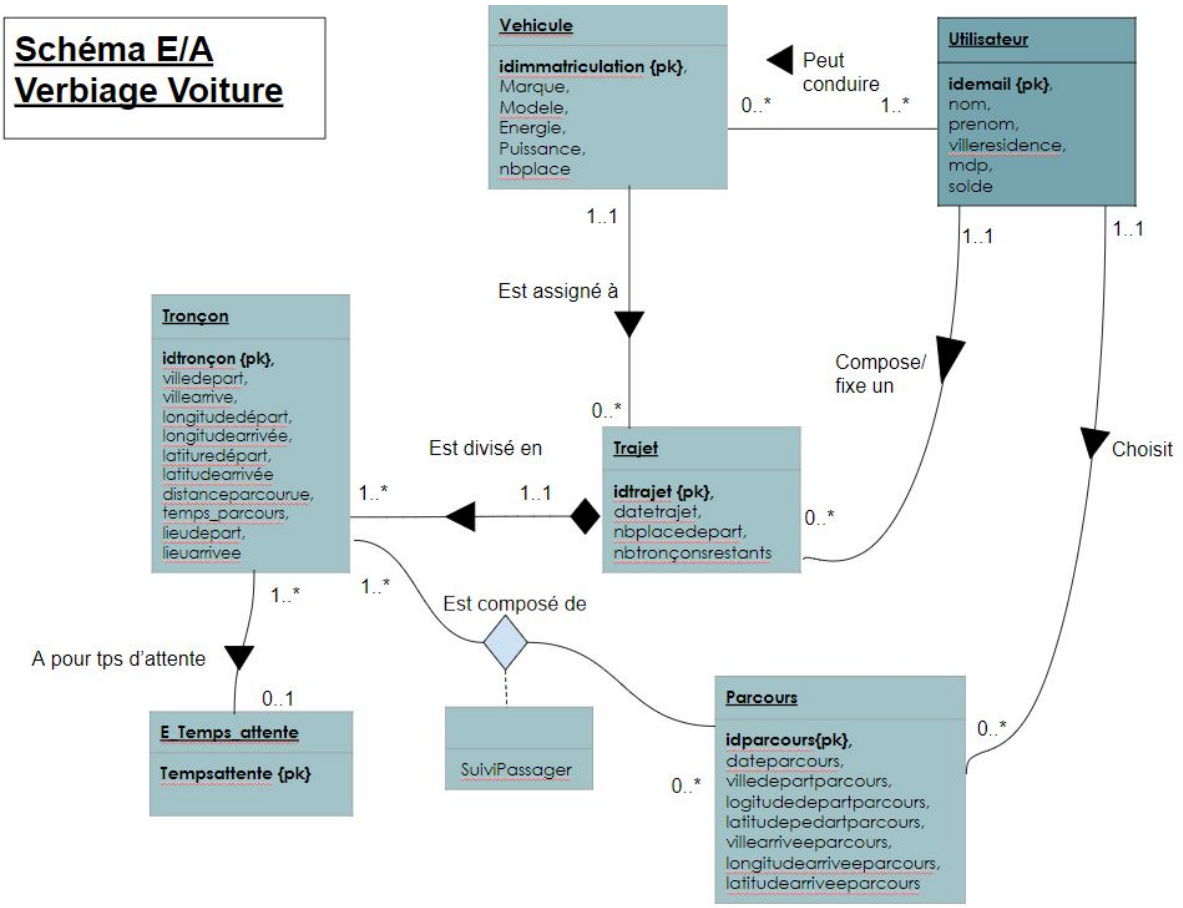


Figure : schéma E/A.

3) Contexte

La création du voyage commence quand un utilisateur,appelé ici conducteur, crée le trajet où il choisit une voiture et le nombre de places pour ce trajet. Il crée alors les tronçons associés sachant qu'un trajet doit avoir au moins un tronçon.

Pour le côté passager, un ou plusieurs utilisateurs, à la limite du nombre de places disponibles dans les trajets, choisissent les tronçons qui correspondent à leurs critères de recherches et qui peuvent appartenir à 1 ou 2 trajets max (cas de la correspondance), le voyage est ainsi composé.

Les passagers marquent le début et la fin du tronçon avec l'attribut SuiviPassager qui correspond à la montée et descente de

la voiture et ce pour chaque tronçon.

Lorsque les passagers finissent les tronçons effectués, c'est à dire tous les tronçons sont marqués 'Fin', le voyage est considéré comme fini, le coût du trajet est déduit du solde du passager et affecté au solde du conducteur.

On peut ainsi passer aux étapes relatives au passage au schéma relationnel ainsi qu'à la construction de la base de donnés qu'en détaillera dans la section III.

II - Analyse dynamique

L'analyse dynamique est une analyse préliminaire du cahier des charges client pour élaborer une ébauche d'architecture de l'application sous forme de diagrammes et de machines à état.

1) Analyse des cas d'utilisation

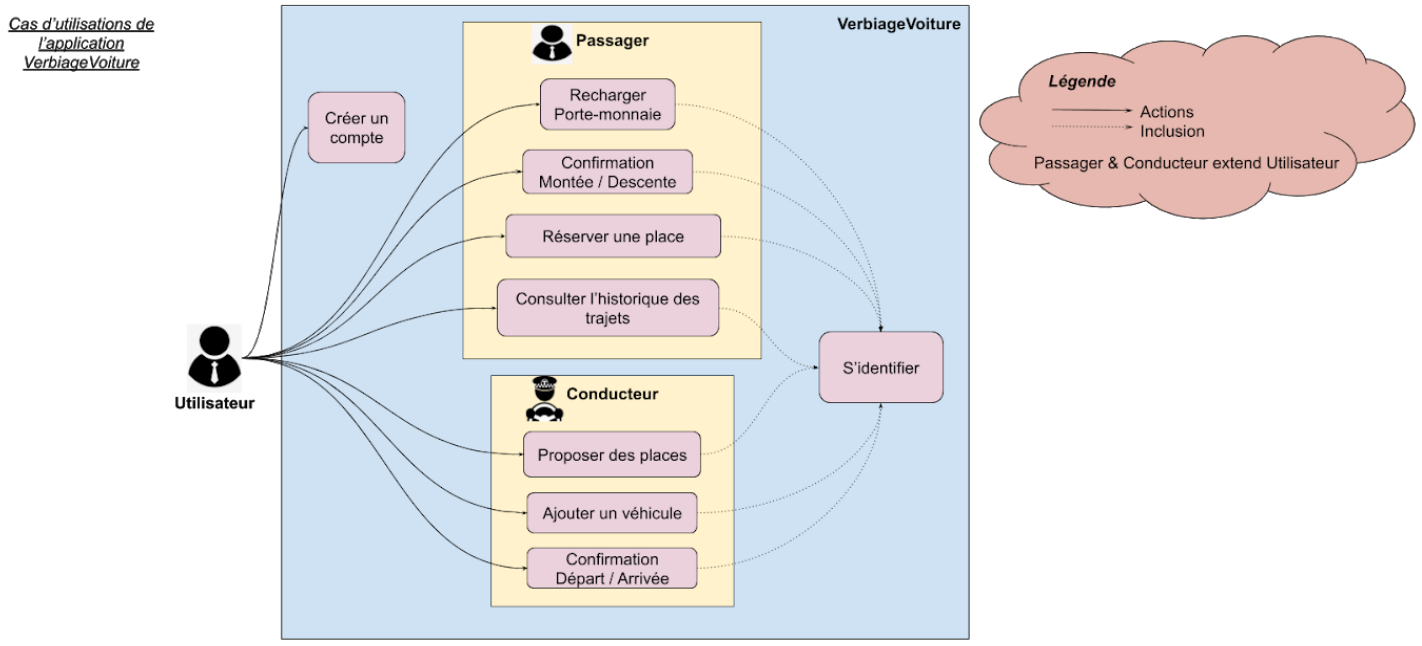


Figure : diagramme des cas d'utilisation.

D'abord, nous avons réfléchi aux différents cas d'utilisation de notre application.
La première distinction qui nous a paru essentielle est la séparation des utilisateurs : un passager n'utilisera pas l'application dans le même but qu'un conducteur. Le premier recherche des parcours et le règle une fois qu'il a été effectué, tandis que le second propose des trajets et a la possibilité de rentrer ses différents véhicules.
En fonction de la nature de l'utilisateur, on peut alors définir les différentes utilisations de l'application et tout cela est résumé dans le diagramme ci-dessus.

Ce diagramme a constitué une réelle roadmap pour une gestion de projet par feature : nous avons cherché à implémenter les features les unes après les autres, la création d'une première nous donnant des acquis techniques (par rapport au package JDBC, au

fonctionnement de le stack des vues, etc) pour développer la suivante.

2) Précision des 3 cas d'utilisation majeurs grâce à des diagrammes États - Transition

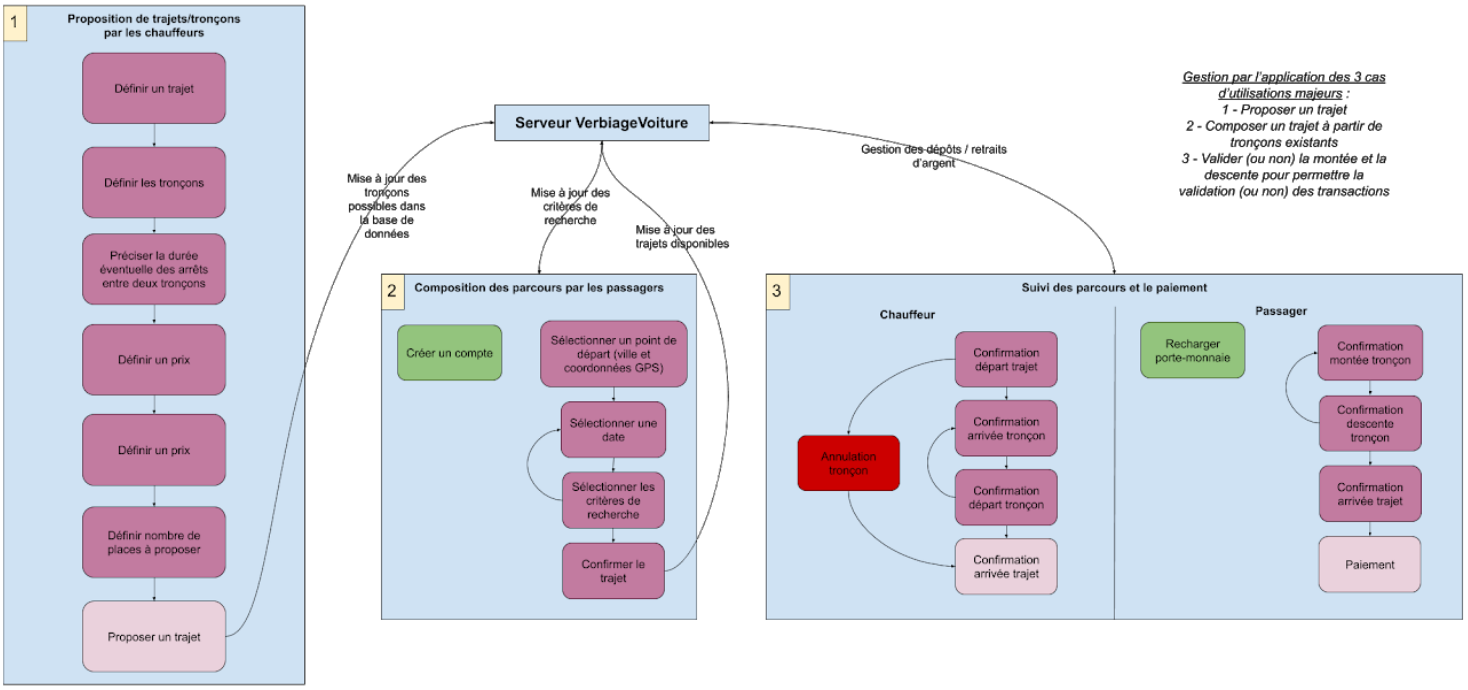


Figure : Diagrammes États - Transition

Nous avons ensuite cherché à préciser le cheminement de l'utilisateur dans les 3 cas d'utilisation majeurs de l'application:

- la proposition d'un trajet par un conducteur ;
- la recherche d'un parcours par un passager ;
- le suivi des trajets côté passager / côté conducteur.

Ces utilisations nous ont paru être les plus importantes car l'application est avant tout une application de covoiturage.

Sur les diagrammes ci-dessus, chaque bloc correspond à un état, ie à un choix que va se retrouver à effectuer l'utilisateur. Ces blocs sont reliés par des flèches, et cela permet de comprendre la chronologie des actions utilisateur.

Finalement nous avons poussé la réflexion un peu plus loin en créant une entité "Serveur Verbiage Voiture" (en faite, cela correspond à notre base de données) et en imaginant l'impact d'une action utilisateur sur ce serveur et la réponse de ce dernier sur l'application. Des flèches externes permettent d'illustrer ces interactions serveur / application.

Ce diagramme est en faite une matérialisation des algorithmes à implémenter : ils ont été d'une grande aide car l'implémentation s'est alors résumée à retranscrire sous forme de codes les enchaînements et états décrits sur ce diagramme.

3) Illustration des interactions Utilisateur - Utilisateur et Utilisateur - Système via un diagramme de séquence

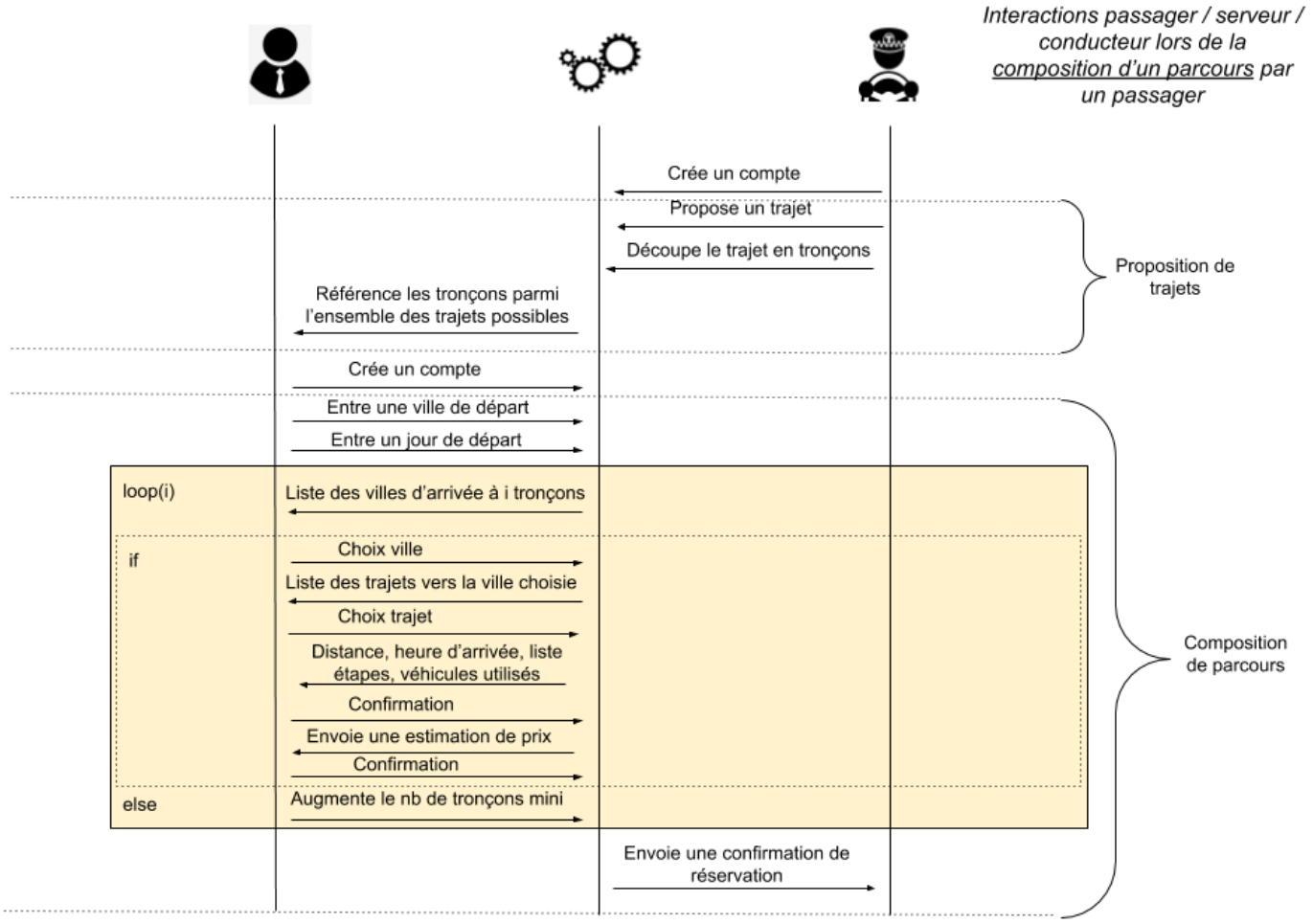


Figure : Diagramme de séquence de la composition de parcours passager.

Après avoir réfléchi aux actions Utilisateur, il faut réfléchir à comment les utilisateurs peuvent interagir entre eux et en quoi le système (ici, l'application et la base de données Oracle) est l'intermédiaire entre les utilisateurs.

Dans ce diagramme précis, nous avons choisi de détailler la recherche d'un parcours passager car c'est la contrainte qui nous a paru la plus difficile à implémenter. En effet, si il n'existe pas de trajet direct, il faut établir une correspondance qui soit la plus avantageuse possible pour le voyageur.

Une méthode possible pour répondre à cette problématique est de proposer la liste des villes d'arrivée à 1 tronçon à partir de la ville de départ indiquée, puis si ça ne convient pas la liste des villes d'arrivée à 2 tronçons, etc.

Finalement ce n'est pas la méthode que nous avons choisie mais le schéma a l'avantage d'illustrer les actions du système sur l'interface homme - machine lors d'un choix effectué par l'utilisateur.

III - Conception de la base de données

1) Passage au relationnel

Dans cette partie, on prendra les informations tirées du Schéma E/A pour le traduire en schéma relationnel étape par étape.

a) Analyse

i) Les types d'entités simples

Vehicule(<u>idimmatriculation</u>, marque, modele, energie, puissance, nbplace).

Utilisateur(<u>idemail</u>, nom, prenom, villeresidence, mdp, solde).

Trajet(<u>idtrajet</u>, datetrajet, nbplacedepart, nbtronconsrestants).

Parcours(<u>idparcours</u>,dateparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours).

E_Temps_attente(tempsattente).

ii) Types d'entités faibles

Tronçon(<u>idtroncon</u>, #(1) <u>idtrajet</u>, villedepart,villearrivee, longitudedepart, longitudearrivee, latitudedepart, latitudearrivee, distanceparcourue, tempsparcours, lieudepart, lieuarrivee).

Vérifier que tout trajet ait au moins un tronçon

iii) Relations avec cardinalité 0..1

D'après le *schéma E/A*, on a la relation :

- E_tps_attente(tempsattente) qui correspend avec cette relation.

On a cependant choisi d'injecter l'attribut tempsattente directement dans Troncon.

- le '#' sert à marquer la clé étrangère

iv) Relation avec cardinalité 1..1

Trajet(<u>id_trajet</u>, date_trajet, heure_trajet, nb_place_depart, id_immatriculation, #id_email)

Parcours(<u>id_parcours</u>, date_parcours, heure_parcours, nb_tronçons_restants, #id_email)

v) Relations avec cardinalité x..n

PeutConduire(<u>id_email</u>, <u>id_immatriculation</u>)

Vérifier que toute voiture ait au moins un conducteur

EstComposéDe(<u>id_parcours</u>, <u>id_tronçon</u>, SuiviPassager)

Vérifier que tout parcours ait au moins un tronçon

En conclusion voici les relations utilisées dans la base de données de notre application:

2) Synthèse du schéma relationnel

- Vehicule(<u>id_immatriculation</u>, marque, modele, energie, puissance, nb_place)
  Contrainte: Energie ∈ {essence, diesel, électrique}

- Utilisateur(<u>id_email</u>, nom, prenom, ville_residence, mdp, solde)
  Contraintes: id_immatriculation référence Vehicule
  id_email référence Utilisateur

- Trajet(<u>idtrajet</u>, datetrajet, nbplacedepart, nbtronconsrestants, #<u>idimmatriculation</u>, #<u>idemail</u>)
  Contrainte: id_email référence Utilisateur

- Tronçon(<u>idtronçon</u>, #<u>idtrajet</u>, villedepart,villearrivee, longitudedepart, longitudearrivee, latitudedepart, latitudearrivee, distanceparcourue, tempsparcours, lieudepart,lieuarrivee).
  Contrainte: id_trajet référence Trajet

- Parcours (<u>idparcours</u>, dateparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours,#<u>idemail</u>).
  Contraintes: id_email référence Utilisateur
  id_immatriculation référence Véhicule

- PeutConduire(#<u>idemail</u>, #<u>idimmatriculation</u>).
  Contrainte: id_email référence Utilisateur
  id_immatriculation référence Véhicule

- EstComposeDe(#<u>idparcours</u>, #<u>idtronçon</u>, SuiviPassager)
  Contrainte: id_parcours référence Parcours
  id_tronçon référence Tronçon
  SuiviPassager ∈ {'Debut', 'Fin'}

<u>Contraintes non représentées:</u>

Vérifier que tout trajet ait au moins un tronçon
Vérifier que toute voiture ait au moins un conducteur
Vérifier que tout parcours ait au moins un tronçon

3) Normalisation des relations :

a) Rappel sur la normalisation :

La normalisation est un processus qui permet de partir d'une table générale, composés de tous les attributs, sur laquelle on applique l'algorithme de normalisation, pour avoir plusieurs tables correspondants.
Les formes normales c'est les étapes qui assurent la bonne définition de ces tables et il permet de vérifier la qualité de la conception en évitant la redondance des attributs.

b) Etude des formes normales :

Tables	Formes normales	Optimisations
Utilisateur(<u>idemail</u>, nom, prenom, villeresidence, mdp, solde).	les attributs dépendent directement de la clé <u>idemail</u> il s'agit donc d'une **3FN**	∅
Vehicule(<u>idimmatriculation</u>, marque, modele, energie, puissance, nbplace).	les attributs dépendent directement de la clé <u>idimmatriculation</u> il s'agit donc d'une **3FN**	∅
Trajet(<u>idtrajet</u>, datetrajet, nbplacedepart, nbtronconsrestants, #<u>idimmatriculation</u>, #<u>idemail</u>)	Les attributs datetrajet, nbplacedepart et nbtroncorestants ne dépendent que de la clé idtrajet. Il s'agit d'une **1FN**	Trajet(<u>idtrajet</u>, #<u>idimmatriculation</u>, #<u>idemail</u>) • **3FN** TrajetInfos(#<u>idtrajet</u>, datetrajet, nbplacedepart, nbtronconsrestants) • **3FN**

Parcours (idparcours, dateparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours, #idemail)	Les attributs ici présents dépendent juste de idparcours D'où c'est une **1FN**	Parcours(idparcours, #idemail) • **3FN** ParcoursInfos(#idparcours, dateparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours) • **3FN**
Troncon(idtroncon, #idtrajet, villedepart, lieudepart,villearrivee,lieuarrivee, longitudedepart, longitudearrivee, latitudedepart, latitudearrivee, distanceparcourue, tempsparcours)	Les attributs ici présents dépendent juste de idtronçon D'où c'est une **1FN**	Troncon (idtroncon, #idtrajet) • **3FN** TronconInfos(#idtroncon, villedepart, lieudepart,villearrivee,lieuarrivee, longitudedepart, longitudearrivee, latitudedepart, latitudearrivee, distanceparcourue, tempsparcours • **3FN**
PeutConduire(#idemail, #idimmatriculation)	il s'agit d'une **3FN**	∅
EstComposeDe(#idparcours, #idtronçon,SuiviPassager)	SuiviPassager depend de idtroncon et de idparcours, il s'agit d'une **3FN**	∅

Notons que ces optimisations n'ont pas été implémentés dans la base de donnés, on détaillera cette particularité dans la partie Bilan.

Passons maintenant à la conception de la base de données.

4) Passage à la conception :

Vue générale des choix effectués :

La conception de notre base de donnés va être divisée en 7 tableaux, on détaillera les choix effectués ci dessus.

Utilisateur : Cette relation nous permettra de créer un User avec les informations de bases associés (idemail, nom, prenom, villeresidence, mdp, solde), la clé ici c'est le idemail qui est unique pour chaque utilisateur.
Au niveau de la base de donnés, la seule contrainte ici c'est le solde > 0, il nous permettra de faciliter les manipulation du paiement dans la suite.

Vehicule : Cette relation permet d'ajouter un véhicule, les informations ici c'est (idimmatriculation, marque, modele, energie, puissance, nbplace) avec idimmatriculation pour clé.
La contrainte ici est l'energie ∈ {essence, diesel, électrique}

PeutConduire : Cette relation permet d'associer un véhicule ou plusieurs véhicules à un utilisateur, les véhicules peuvent etre partagés par plusieurs utilisateurs. Les informations nécessaires ici sont les clés étrangères idemail et idimmatriculation.

Trajet : La relation permet de créer un trajet, on parle ici d'un utilisateur conducteur, les informations nécessaires sont (idtrajet, datetrajet, nbplacedepart, nbtronconsrestants, #idimmatriculation, #idemail), on a idtrajet comme clé primaire, idimmatriculation et idemail sont deux clés étrangères.

Troncon : Lorsqu'on crée un Trajet, le conducteur doit ensuite créer au moins un tronçon,les tronçons caractérisent les points d'arrêt dans le trajet, exemple pour un trajet (Grenoble - Lyon - Paris), on a deux tronçons (Grenoble - Lyon) et (Grenoble - Paris).
les informations nécessaires ici sont (idtroncon, #idtrajet, villedepart,villearrivee, longitudedepart, longitudearrivee, latitudedepart, latitudearrivee, distanceparcourue, tempsparcours, lieudepart, lieuarrivee), où idtroncon est la clé primaire et idtrajet la clé étrangère.

NB: *Les deux attributs lieudepart et lieuarrivee ont été rajouté dans la table au cours du développement de l'application car ils permettent la dénomination du lieu désigné par les coordonnées GPS de départ et d'arrivée. Sachant qu'un tronçon peut commencer et finir*

dans une même ville, cette dénomination permet d'éviter à l'utilisateur d'avoir à choisir entre plusieurs tronçons de départ au sein d'une même ville à partir des coordonnées GPS de ces tronçons.

Parcours : La relation permet de créer un parcours, il s'agit ici d'un utilisateur passager, les informations nécessaires sont (idparcours, dateparcours, villedepartparcours, logitudedepartparcours, latitudepedartparcours, villearriveeparcours, longitudearriveeparcours, latitudearriveeparcours, #idemail), la clé primaire est ici est idparcours, idemail est une clé étrangère.

NB : *En théorie la création du parcours n'est faite que si l'algorithme de recherche du parcours aboutit à un résultat, c'est à dire s'il trouve 1 ou 2 trajets qui correspondent aux critères de recherches entrées par le passager.*

NB2 : *En pratique, à partir de critères de recherche non obligatoires que sont la ville de départ/le lieu de départ et/ou la ville d'arrivée/le lieu d'arrivée, la création de parcours dans l'application de ne fera qu'à partir de tronçons n'appartenant qu'à un seul trajet (cf V Etat d'avancement pour plus de détails).*

EstComposeDe : Cette relation permet d'effectuer le suivi du parcours du passager après la création du parcours, les informations necessaires sont (#idparcours, #idtronçon,SuiviPassager), avec idparcours et idtroncon deux clés étrangères.
La contrainte ici est SuiviPassager ∈ {Debut, Fin}.

5) Implémentation de la base de donnés

L'implémentation de la base de données élaborée dans les sections précédentes peut-être consultée dans les trois fichiers joints avec l'archive du projet dans le dossier BDD:

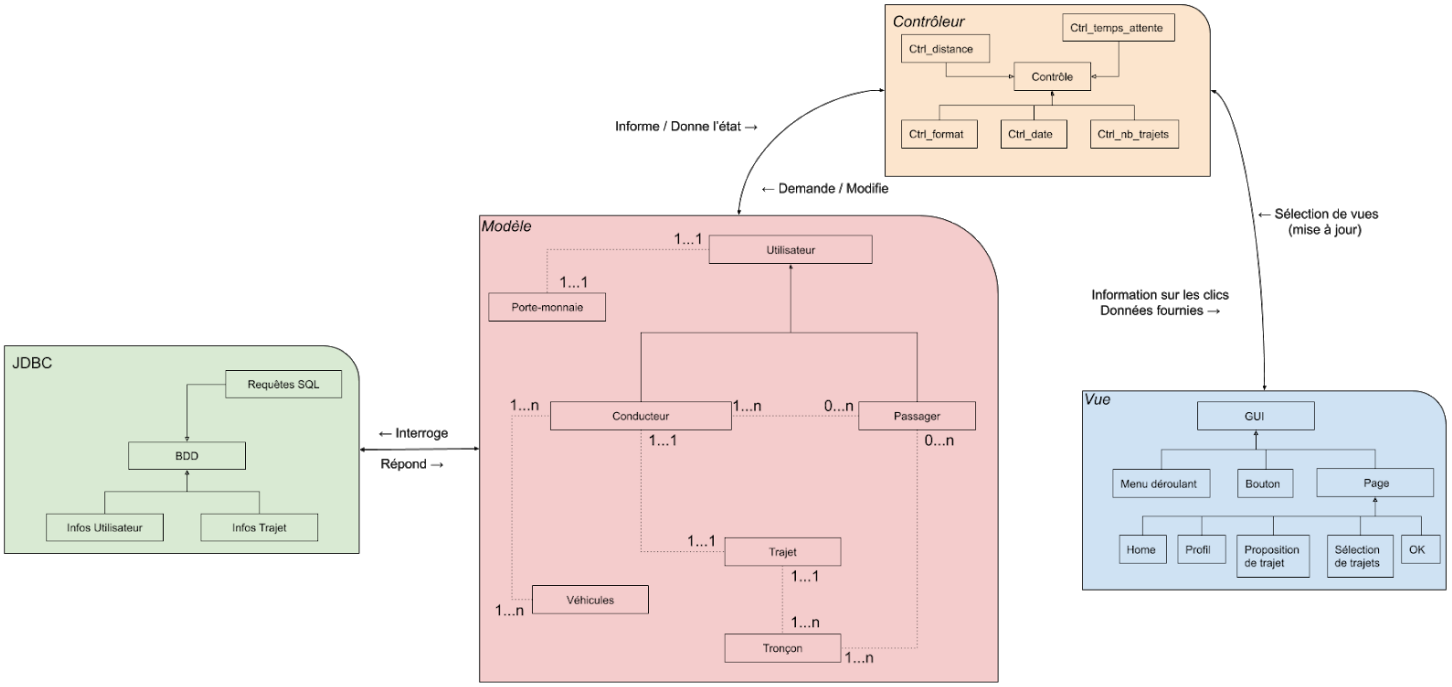
- init.sql: Ce fichier contient toutes les requêtes d'initialisation de la base de données, à savoir la création de toutes les tables de la base avec les attributs et contraintes correspondantes au schéma relationnel donné en 3)
- remplissage.sql: Ce fichier contient les requêtes nécessaires afin de remplir la base nouvellement créé avec des données pour pouvoir tester l'application
- requetes.sql: Ce fichier contient les requêtes SQL nécessaires au fonctionnement de l'application.

IV - Architecture de l'application

L'architecture de l'application s'articule autour du modèle MVC. Ainsi, le package global *fr.ensimag* contient trois sous-packages: *controler*, *model* et *view*. Chaque sous-package contient une des classes principales, en l'occurence Controler, Model et View.

La méthode *main()* de la classe Main à l'extérieur du package global est exécutée au démarrage de l'application. Elle est très simple, elle instancie un objet de chaque classe principale évoquée précédemment. Ces classes sont expliquées dans les parties suivantes.

1) Patron MVC (Modèle - Vue - Contrôleur)



❏ Architecture MVC

On retrouve donc sur le schéma précédent le modèle MVC associé à un bloc nommé JDBC permettant de réaliser l'interface avec la base de données en ligne.

La vue réalise l'interface graphique destinée à l'utilisateur. Le contrôleur possède des méthodes permettant de contrôler par exemple le format des données que la vue lui fournit avant d'appeler des méthodes adéquates pour les insérer en base de données.
La modèle instancie les différents objets et notamment le profil utilisateur courant.

❏ Constructeur des classes

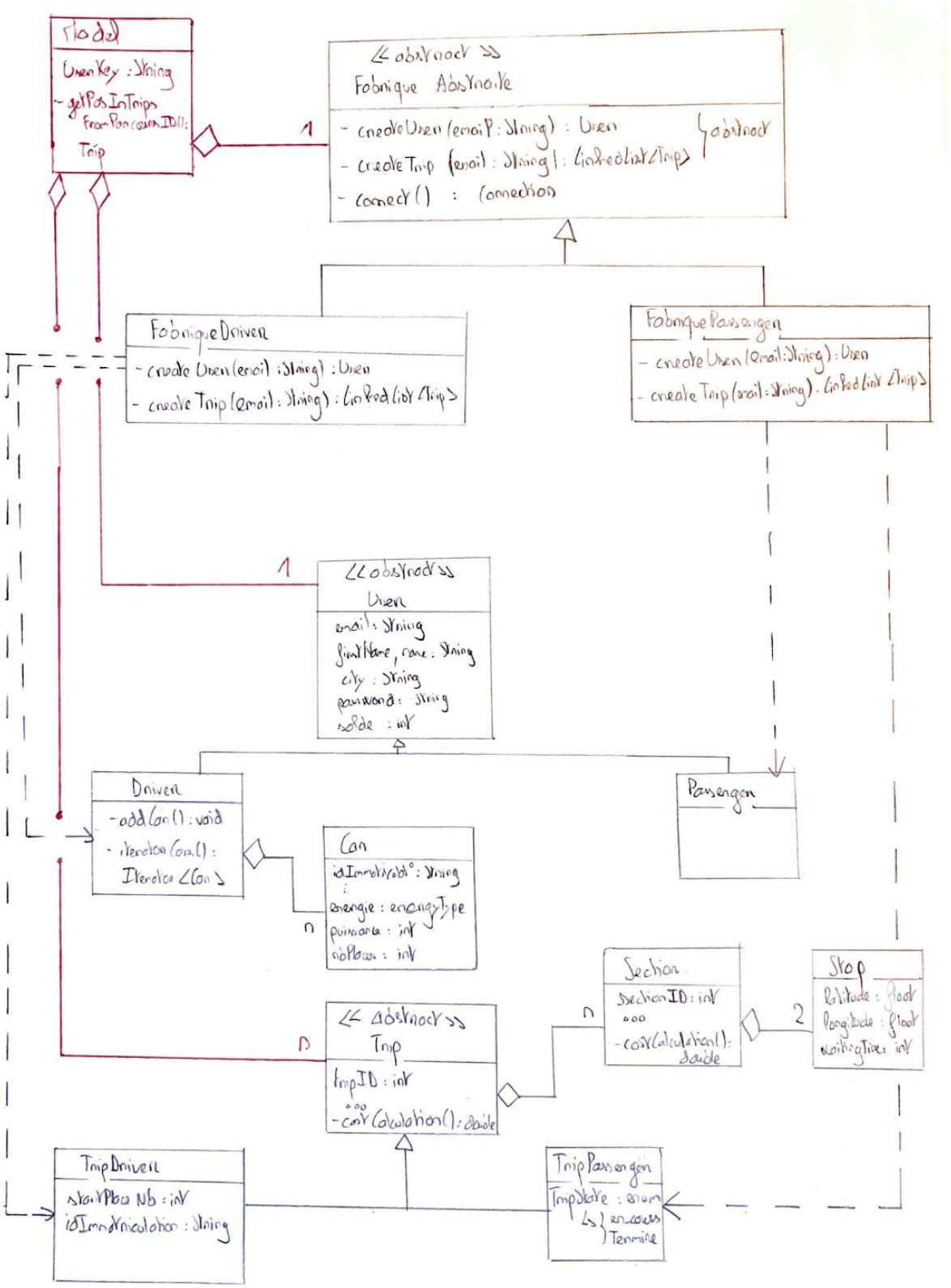
Nous avons pris le parti de ne pas faire communiquer la vue et le modèle (en tous cas pour le moment, car en finissant d'implémenter correctement le patron observateur et le Listener cela serait possible). Dès lors, la vue communique avec le contrôleur, qui lui même communique avec le modèle, d'où les constructeurs suivants :

- View(Controler c) ;
- Controler(Model m) ;
- Model().

Dans le diagramme ci-dessus, on voit que c'est le modèle qui communique directement avec la base de données via le package JDBC. Finalement, ce n'est pas ce que nous avons implémenté : la classe InterventionOnDb gère toutes les modifications de la base à effectuer et les requêtes d'information.

2) Modèle

a) Architecture du modèle



Le modèle est le lieu où les données utilisateurs sont instanciées en dur. On distingue 2 types différents de données utilisateurs :

- les données personnelles, que l'utilisateur complète lorsqu'il s'inscrit;
- les données propres à l'application, c'est-à-dire les voyages qu'il va proposer en tant que conducteur ou auxquels il va s'inscrire en tant que passager.

Lorsque l'utilisateur s'authentifie, on conserve dans un String spécial (UserKey) son email qu'on réutilisera pour l'instancier dans l'application. L'instanciation de l'utilisateur a lieu lors du choix Passager / Utilisateur: selon le choix effectué, on instancie plutôt un Driver ou un Passenger grâce à la méthode createUser() du patron Fabrique Abstraite.

Lorsque l'utilisateur clique sur "My trips", on instancie ses données voyages ; on crée donc une liste de TripPassenger ou de TripDriver, selon le choix effectué au préalable. Cette création se fait via la méthode createTrip() du patron Fabrique Abstraite.

b) Patron Fabrique Abstraite

Le patron Fabrique Abstraite est le coeur de l'application et permet la création d'objets dont on ne connaît pas au départ la future classe. Typiquement, on ne sait pas l'utilisateur va se connecter en tant que passager ou en tant que conducteur. Ce patron permet de gérer facilement les 2 possibilités.

La dualité Passager / Conducteur est essentiel dans notre projet :

- dès le début dans notre diagramme des cas d'utilisation, nous avons séparé ces 2 entités en listant des cas d'utilisation propre à chaque une ;
- dans la création de la base de données cette donnée était une des contraintes principales et il a fallu alors créer une table TRAJET, correspondant aux trajets conducteurs, et une table PARCOURS, correspondant aux trajets passagers ;
- il était alors logique de prendre cette distinction en compte dans l'architecture de l'application.

c) Classes annexes

- Section

Un trajet est découpé en tronçon par le conducteur ; lors de son choix de parcours, un passager peut décider de choisir un parcours composé de 2 tronçons appartenant chacun à un trajet différent.
Pour gérer cette contrainte, on crée une classe Section qui correspond physiquement à un tronçon.

- Stop

Une des contraintes à prendre en compte était que la distance entre l'arrivée du premier tronçon et le départ du tronçon suivant soit de moins de 1km.
Pour gérer cela, on crée une classe Stop qui correspond à un point sur une carte : il possède des attributs latitude et longitude, ainsi que le nom de la ville et le nom du lieu dit correspondant aux coordonnées. On peut alors adjoindre à cette méthode distanceEuclidienne(Stop) qui permet de calculer la distance euclidienne entre 2 Stop.

Cette approximation convient bien parce qu'à priori les 2 points arrivée / départ sont proches (car dans la même ville) ; elle ne convient plus pour calculer la distance entre 2 Stop correspondant à 2 villes différentes. Pour cela, on implémente une méthode distanceVolOiseau(Stop) qui permet de calculer la distance à vol d'oiseau entre 2 points éloignés (on prend alors en compte la courbure de la terre et son rayon).

Grâce à cette méthode, on peut calculer la distance correspondant à une Section et même le temps de parcours. La méthode static tempsParcours(int) permet alors de calculer un temps de parcours à adjoindre à la Section.

- Car

Un conducteur possède des voitures différents ; lorsqu'il est instancié dans l'application, on lui joint un liste de voitures qu'il est apte à conduire.
L'objet voiture ne possède pas de méthode particulière mais contient tous les attributs de la table VEHICULE correspondant à l'IDIMMATRICULATION indiqué ; on peut imaginer que dans une future amélioration de l'application, le choix de véhicule lors de la création d'un trajet serait plus précis et on aurait alors besoin de plus d'informations que simplement l'immatriculation (implémentation actuelle).

3) Contrôleur

a) Les classes Control

La partie contrôle du modèle MVC est tout aussi importante que les autres car elle permet de contrôler tout ce qui sort mais surtout rentre dans base de données. Par exemple, pour son inscription, l'utilisateur remplit différents champs, il faut alors vérifier de format de chacun d'entre eux (notamment grâce à des expressions régulières) mais aussi vérifier que l'email utilisée n'existe pas déjà en base de données. Pour l'authentification, il faut contrôler en base de données s'il un utilisateur existant match ou non avec l'email et le mot de passe renseignés.

Chaque page de l'application a donc une classe de contrôle associée (situées dans le package fr.ensimag.controler)

b) La classe InterventionOnDB

La classe InterventionOnDB est une classe particulière car elle est dans le package *Controler* mais n'y appartient pas vraiment, par contre c'est bien les méthodes de ce package qui l'utilise.

Elle gère toutes les interactions avec la base de données qui correspondent à des modifications : "modifie le solde de tel utilisateur", "ajoute tel trajet avec n tronçons", etc.

Cette implémentation a été choisie par souci de simplicité : dès qu'on a besoin d'interagir avec la base de données, on crée un objet InterventionOnDB ce qui a pour effet d'instancier la connexion et de mettre le autocommit sur off, puis on applique la méthode correspondante et on referme la transaction avec closeTransaction() qui permet de valider le commit et de fermer la connexion.
On notera aussi la présence d'une méthode doRollback() qui permet d'annuler les changements s'il y a eu un souci avant qu'on puisse commit.

4) Vue

La vue joue de rôle de l'IHM, elle correspond au frontend de l'application. On y retrouve donc différentes pages contenant différents champs, listes déroulantes et bien d'autres éléments permettant à l'utilisateur de s'y retrouver.

Remarque: par manque de temps l'UI et l'UX n'ont pas pu être réellement travaillées en profondeur.

a) Conception

L'objet view de classe View instancié dans la méthode *main()* possède un attribut window de classe Window. Cet attribut correspond au coeur de l'interface graphique étant donné qu'elle extends la classe JFrame de la library javax.swing (bibliothèque graphique). Après la configuration de certains paramètres, l'objet window est rendu visible, ceci a pour effet de faire apparaître la fenêtre de l'application.

La fenêtre contient un objet principal nommé, en l'occurrence, *content* qui extends JPanel, autre classe de la librairie javax.swing. L'organisation des composants au sein d'un JPanel est gérée par un LayoutManager. Il en existe plusieurs, mais le choix s'est tourné vers un CardLayout pour *content*. Ceci permet d'organiser les sous JPanels, jouant le rôle des pages de l'application, sous forme de pile de JPanel, seul celui sur le dessus est visible. Les changement de page sont majoritairement réalisés suite à un clic sur un bouton et/ou après avoir obtenu un résultat positif d'une méthode de contrôle.

On instancie donc dans *content* de *window* toutes nos pages (JPanels).
Le layoutManager choisit pour ces JPanels est le GridBagLayout, plus complexe que la plupart des autres mais du coup plus flexible et personnalisable.

La composition de ces JPanels s'est en parti faite grâce à l'outil visuel proposé sous Intellij, générant des fichiers .form (basé sur du xml).

b) Les pages

Voici une liste des classes correspondant à aux pages de l'application avec une description pour chacune d'entre elle:

- LoginForm: page d'accueil de l'application, elle propose à l'utilisateur de s'identifier avec son adresse email et un mot de passe. Dirige vers une page de classe RoleChoicePage en cas de connexion réussie.

  Si l'utilisateur n'a pas déjà un compte, un bouton le redirigeant vers la page d'inscription est présent. On y

retrouve aussi un bouton en cas de mot de passe oublié (non implémenté).

- RegisterForm: page permettant à l'utilisateur de s'inscrire sur l'application après avoir renseigné les champs nécessaires.

- RoleChoicePage: page suivant la page de login (après une connexion réussie) permettant à l'utilisateur de choisir son rôle: passager ou conducteur.

- PassengerPage: cette page permet à l'utilisateur plusieurs actions à l'aide de boutons:
  - atteindre la page lui permettant de consulter son profil
  - atteindre la page lui permettant de rechercher et réserver un trajet
  - atteindre la page lui permettant de visualiser les trajets qu'il a réservé
  - atteindre la page précédente
  - se déconnecter (redirection vers la page d'identification)

- ProfilePage: page permettant à l'utilisateur de consulter et mettre à jour ses données (nom, prénom, ville de résidence, changer son mot de passe) mais aussi de pouvoir recharger son porte feuille.

- PassengerResearchRoute: page permettant à l'utilisateur de construire son itinéraire et de réserver sa place pour celui de son choix.

- PassengerTripsPage: page de visualisation des trajets auxquels l'utilisateur a réservé une place. Cette page permet aussi de valider les sections lorsqu'elles ont été effectuées et enfin de valider chaque trajet pour procéder au paiement.

- DriverPage: cette page permet à l'utilisateur plusieurs actions à l'aide de boutons:
  - atteindre la page lui permettant de consulter son profil
  - atteindre la page lui permettant de proposer un trajet
  - atteindre la page lui permettant de visualiser les trajets qu'il a proposé
  - atteindre la page permettant d'ajouter un nouveau véhicule
  - atteindre la page précédente

    - se déconnecter (redirection vers la page d'identification)

- DriverBuildTrip: page permettant à l'utilisateur de construire et proposer son itinéraire. Pour cela il devra choisir une date, une heure de rendez-vous, un lieu de départ et d'arrivée, le véhicule qu'il utilisera ainsi que le nombre de sièges disponibles. Il peut ensuite si nécessaire ajouter des villes étapes à son parcours.

Remarque: par manque de temps pour le développement, le nombre de villes étapes est limité à 3 maximum.

- DriverTripsPage: page récapitulant les trajets en cours ou futurs proposés par l'utilisateur. Elle permet aussi de valider chaque section de trajet puis les trajets.

- AddVehiclePage: page permettant à l'utilsateur d'ajouter un véhicule à la liste de ses véhicules.

c) Interaction Modèle - Vue via le patron Observateur

Dans la théorie du modèle MVC, les interactions Modèle - Vue sont régis par le patron observateur. En d'autres termes, la vue (l'observateur) scrute le modèle (le sujet) à la recherche de modifications et le sujet va donc notifier l'observateur quand il y a effectivement un changement.Pour cela, nous avions implémenté le patron observateur dans le package fr.ensimag.observer avec toutes les méthodes abstraites à implémenter (d'où l'existence d'un abstractModel dont nous ne servons pas).

Dans notre cas, cela aurait permis typiquement de détecter des changements dans la base de données quand un passager est en train de créer un parcours mais qu'un autre passager réserve le même tronçon ; ainsi, le premier passager aurait pu être notifié directement et ainsi changer de choix.

Nous avons préféré nous concentrer sur d'autres fonctionnalités de l'application à défaut de développer cette feature, qui serait néanmoins utile si l'application était utilisée par une population très large.

V - Etat de l'application à l'issue du projet

1) Fonctionnalités implémentés

On fera ici le parallèle avec le cahier de charges.

a) Proposition de trajets

1) La création d'un compte sur VV. 2) L'ajout de véhicule(s) pour le compte. 3) La proposition d'un trajet et du ou des tronçons le composant, en précisant la durée éventuelle des arrêts entre deux tronçons.	Les 3 fonctionnalités citées dans cette partie ont été implémentés et testées. NB: le contrôle de validité des champs renseignés par l'utilisateur n'est pas implémenté entièrement. Il est donc parfois nécessaire de renseigner les informations sous le bon format (notamment pour l'ajout de véhicule et de trajet)

b) Composition de parcours

1) La création d'un compte sur VV. 2) La recherche d'un parcours à partir d'un point de départ, et d'un point d'arrivée avec pour contrainte: La proposition d'un parcours qui correspond à un trajet ou 2 trajets maximums,les propositions ne pourront comprendre que des tronçons pour lesquels il reste de la place dans le véhicule.	La recherche d'un parcours n'est pas complète: nous avons réussi à afficher les tronçons résultats qui correspondent à un même trajet sans la vérification de la date ni du nombre de places restantes, mais la correspondance n'a pas été implémenté coté jdbc. NB : *l'algorithme marche au niveau des tests côté BD mais on n'a pas eu le temps de l'implémenter coté java*

c) Suivi des parcours et paiement

Pour les passagers : 1) Rechargement du porte-monnaie (simulée). 2) Confirmation d'arrivée de chaque tronçon composant un trajet au fur et à mesure. 3) Paiement des tronçons effectués en fin de parcours par le passager.	Les 3 fonctionnalités ont été implémentées. NB : - *L'implémentation proposée pour continuer à afficher les trajets déjà effectués dans MyTrips provoque des latences au clic sur le bouton.* - La gestion des erreurs n'est pas toujours parfaite ; on aurait aimé afficher une fenêtre pop-up affichant la cause de l'erreur à chaque fois mais par manque de temps nous n'avons pas pu l'implémenter.

2) Limites

a) Au niveau de la base de donnés

Comme cité dans la partie Etude des FN, notre base de données n'est pas optimisée pour toutes les tables au niveau de la conception. En effet, nous avons été à court de temps et nous avons donc choisi de ne pas effectuer de changements tardifs qui pourraient poser des problèmes d'intégrations avec la partie jdbc développée.

Nous nous sommes rendu compte tardivement que pour la gestion du suivi du nombre de places restantes par tronçon, il aurait fallu ajouter un autre attribut "nbrePlaceRestTronc" dans la table Troncon. Celui ci prendrait la valeur de l'attribut "nombreplacedepart" de la table Trajet, permettant de voir le nombre de passagers pour chaque tronçon et ainsi valider ainsi le parcours si le tronçon n'est pas complet.

Par manque de temps, nous n'avons pas pu implémenter l'annulation d'un trajet par un conducteur. Une façon d'intégrer cela serait d'ajouter un attribut suiviConducteur{Depart, Fin, Annule} dans la table Troncon.

b) Au niveau de l'application

Ci-après quelques points que nous n'avons pas eu le temps de développer:

- améliorer la validation de trajet (faire le lien entre validation passager et conducteur) ;
- sauvegarder l'état de chaque trajet ;
- implémenter le contrôle de format de la plaque d'immatriculation lors de l'ajout d'un véhicule ;
- gestion de l'annulation d'un trajet ;
- Implémentation de l'algorithme de correspondance ;
- changement de mot de passe et mot de passe oublié à la connexion ;
- meilleures gestions des exceptions pour à chaque fois indiquer à l'utilisateur ce qu'il s'est passé ;
- le listener : modifications en direct de la vue quand un utilisateur A effectue une action sur son application qui va influer les choix que va pouvoir effectuer l'utilisateur B sur sa propre application.

Un point sur l'algorithme de correspondance.

En effet, les requêtes SQL nécessaires pour la recherche des trajets et tronçons qui constituent le parcours avec correspondance ont été faites et testés, il manque juste l'implémentation en jdbc de cette partie, voilà un exemple simplifié d'exécution de la requête (sans prendre compte de la distance entre les tronçons, et le temps d'attente < 1h) :

Un passager choisit le parcours (Grenoble - Lille).

Dans notre base, on n'a pas de trajet direct de Grenoble à Lille, il faudra donc passer par 2 trajets :

- trajet avant correspondance : idtrajet = 6
  - troncon 1.1 : idtroncon = 8 (Grenoble - Lyon)
  - troncon 1.2 : idtroncon = 9 (Lyon - Paris)
- trajet après correspondance : idtrajet = 14
  - troncon 2.1 = idtroncon = 17 (Paris - Lille)

On divisera la recherche ici en deux étapes :

1. Trouver le troncon correspondance.
2. Afficher l'ensemble du parcours.

```
WITH tronc_depart AS (
    select idtroncon, idtrajet, villedepart, villearrivee from
troncon where idtrajet in (select idtrajet from troncon where
villedepart ='Grenoble')
),
     tronc_arrivee AS (
         select idtroncon, idtrajet, villedepart, villearrivee
from troncon where idtrajet in (select idtrajet from troncon where
villearrivee ='Lille')
     )
SELECT tronc_depart.*, tronc_arrivee.*
FROM tronc_depart
         inner join tronc_arrivee on tronc_depart.villearrivee =
tronc_arrivee.villedepart;

Resultat
=> 9,6,Lyon,Paris,17,14,Paris,Lille

Nous affichons ici le dernier tronçon avant correspendance
(idtroncon = 9), correspendant au idtrajet=6, qui est composé de
Lyon - Paris et le premier tronçon après correspendance (idtroncon
= 17), correspendant au idtrajet=14, qui est composé de Paris -
Lille.
```

Il reste donc de récupérer tout le trajet.

```
-- itinéraire
select * from troncon where idtrajet = 6 and idtroncon between
(select idtroncon from Troncon where villedepart = 'Grenoble' and
idtrajet = 6) AND 9

union

select * from troncon where idtrajet = 14 and idtroncon between 17
and (select idtroncon from Troncon where villearrivee= 'Lille' and
idtrajet = 14)
```

Resultat =>

```
8,6,Grenoble,Lyon
9,6,Lyon,Paris
17,14,Paris,Lille
```

Nous affichons ici les idtroncons (8,9,17) appartenant aux trajets 6 et 14, où notre parcours se décompose en :
Grenoble - Lyon && Lyon - Paris && Paris - Lille.

VI - Bilan

1) Répartition des tâches

La répartition a été basée sur la section 3 du sujet fourni. Dans un premier temps, on a divisé le travail entre :
Analyse statique, schéma E/A, conception de la base de données : Paul Svensson, Kacem Jedoui, Kevin Callet.
Analyse fonctionnelle et architecture de l'application: Victor Bertin, Baptiste Seux.

Dans un deuxième temps, nous avons reconstitué les équipes selon l'avancement de chaque partie.
Base de Données : Paul Svensson, Kacem Jedoui, Kevin Callet
JDBC : Paul Svensson, Kevin Callet, Victor Bertin
Interface Graphique : Baptiste Seux, Victor Bertin

Cette répartition des tâches a montré ses limites lorsque nous avons dû tous nous mettre sur la partie Implémentation : malgré les diagrammes, nous avons eu mal à expliquer les différents choix structuraux effectués. Le fait d'être à distance n'a bien sûr pas aidé.
Sa force s'est par contre manifestée par le fait que certains étaient très au point sur les manipulations de la base de données et pouvaient alors formuler des requêtes SQL poussées, sans avoir à passer par des intermédiaires dans le code, tandis que certains étaient très à l'aise avec l'aspect "code" et pouvaient ainsi proposer des prototypes de nouvelles idées très rapidement.

2) Retour sur ce projet

a) Points forts à garder

Après discussion, nous sommes tous d'accord pour dire que ce projet nous a beaucoup apporté. En commençant par la mise en application directe du cours (base de données + conception d'architecture logicielle) en développant une application "simple" mais réaliste (basée sur l'exemple de Blablacar qui obtient un réel succès ces dernières années).

Nous avons en plus pu découvrir le fonctionnement d'une interface graphique en java, couplée à un backend pour le contrôle et la communication avec une base de données en ligne.

Ainsi nous avons trouvé le projet intéressant mais avons recensé quelques améliorations que nous aurions apprécié lors de cette période de conception et développement.

Remarque: nous avons vraiment apprécié la réactivité des profs à répondre sur la discussion Riot.

b) Améliorations à considérer

Au niveau de l'application :

- amélioration du contrôle du format d'adresse mail dans la page d'inscription
- implémentation d'un contrôle de la force du mot de passe choisi
- validation d'inscription par mail
- gestion de la partie mot de passe oublié
- améliorer l'UI et l'UX de l'application
- améliorer la procédure de rechargement de portefeuille
- implémenter une web view de Google Maps pour sélectionner plus intuitivement les différents lieux de départ, tronçons, arrivée (pour éviter de devoir rentrer les coordonnées gps à la main)

Au niveau de la gestion du projet :

- En première remarque, le temps. Nous sommes frustrés en cette fin de projet notamment par le manque de temps. Nous nous sommes réellement investis dans ce projet et sommes un peu déçus, avec une semaine supplémentaire, l'application aurait été bien meilleure.

- Pourquoi à l'avenir ne pas remplacer le partiel de base de données pour nous laisser plus de temps sur le développement de cette application. Nous pensons que ceci est bien plus formateur et utile qu'un partiel avec des requêtes théoriques. Nous pourrions présenter ce genre de projets lors d'entretiens d'embauche, c'est pourquoi c'est toujours mieux d'avoir un rendu propre et fiable.