Every title in (Notes ||Constraints ||Additions) is a
title of an entity in the UML. If a blue box appears close to an entity
or cardinality in the UML, saying "see notes" it means there are
notes in the section NOTES concerning that entity/cardinality.

####################################**NOTES**##################################
Station:
If a station can be reached by several lines, it means a station has also several lines, thus I used a many-to-many relationship.

Line:
Line has an integer that keeps track of the number of segments, this will be useful to create a trigger in the database, to check if two different stations A and B have at least two segments.

Direction:
Several segments can have the same direction, thus I did a class
just for the directions, using x,y coordinates. Direction has two primary
keys the x and y coordinates, hence two Direction instances with the
same x and y coordinates are considered equal. I was not able to implement
a bitwise operation with (x&y) in postgressSQL to form a unique PK, thus
I decided to add them both as PK.

Driving_Time :
Allows manager to know which train
is currently being driven by what Driver, furthermore
this information is kept in history, as Driving_Time is an entity.

StartTime:
Saves when a train first started at the first station of a line.

Maintenance:
A Train can have several maintenances, so does a Coach.
But only one Maintenance takes care of a (Train || Coach),
that is why I used one-to-many relationship.

Problem:
Instead of having a list of problems I created an entity
so all the problems are saved, when a train is maintained, or
was maintained.

Train && Coach:
After what I understood from the Enoncé, a coach and a train are different.
Thus, I thought that a coach, or several coaches can be carried by a train,
which is the vehicle that has the engine to carry all the coaches, and the
vehicle the Driver can drive. A Coach can also be in Maintenance. A coach can
be carried by only one train at a given time.

Train && Driver Cardinality:
A train can be driven by several drivers, and a driver can drive several trains
thus, I used a many-to-many relationship. It is true that a Driver can only drive
a given train at a time, and a train can only be driven by one drive at a time, hence a one-to-one relation might also be correct.
One can always know which Driver is driving which train and vice-versa (like the administrator requested) because of the Driving_Time entity.

Customer:
A Customer can buy a PrePaid card, and also login to see his balance, but
he does not need to register in order to login, it means that Login Credentials
must be given to the Customer when he buys a PrePaid card, that is why
I used LoginCredentials entity. Both RegisteredCustomer, and Customer have a flag that
Defines if they are logged in or not. Obviously, if a Customer does not have a PrePaidCard,
and consequently no LoginCredentials, his flag islogged is always going to be False.

WorkingTime:
A Driver can start working at given WorkingTime, in this case the
variable start_time must be true, so the Administrator knows a Driver
started working at that time. If the variable start_time is true, the
variable finish_time must be false, and the other way around as well.
The constraints are in the UML itself. A Driver can have 0..* WorkingTimes,
for instance, in only one week he can 5 or 6 different schedules.

HomeAddress:
HomeAddress and Driver have a 1..1 cardinality. I put a
HomeAddress ID foreign key in the Driver entity, and a Driver ID in the
HomeAddress entity as well.

ID_card && PrePaidIDCard:
When a customer gets in a coach, the respective cards record that coach,
along with the time of trip (TripTime Entity). A card may have several coaches
recorded. Note that I have more than one entity that deals with time, but
all of them have different variables and purposes. For instance :
TripTime != StartTime != Average_Time != etc.. Also I did not use inheritance for the time entities, because most of them do not have common attributes.
################################**Notes END**################################

################################**Constraints**################################
Context Station:

```
//"This implicates that for two different stations A and B where you can go from A
//to B and from B to A there must be (at least) two segments between these two
//stations"
self.start_station.id != self.final_station.id implies
        self.has.Line.number_of_segments >= 2
```

```
Context Customer:
//A customer that wants to register must input
//his credit card info, and the credit card name
//must be equal to his name

 inv: self.registers.credit_card != NULL
 inv: self.register.credit_card.name = self.name


Context RegisteredCustomer:

        //In order to a RegisteredCustomer to be logged in
        //his username and password, must match some RegisteredCustomer
        //A (Customer || RegisteredCustomer) can only have one login session
        //open, that is why I used a cardinality of 0..1 for the logins.


        if self.RegisteredCustomer->iterate(c |
                if(c.username == self.login.LoginForm.username) &&
                if(c.password == self.login.LoginForm.password)
                endif

        )then
                c.islogged = True
        endif

        //if islogged var is true, user is successfully logged in
        //and thus may verify is balance
        self.checks_balance():
                if self.islogged then
                        return self.ID_card.balance
                endif

        //If the customer is logged in, returns all records, where records is saved
        //by the Card of the customer, containing the coaches where the customer
        //was, along with the time of trip, using the TripTime entity.
        self.rides_history():
                if self.islogged then
                        return self.RegisteredCustomer.ID_card.records
                endif

Context Customer:
        //Similar explanations to the RegisteredCustomer
        //go for the Customer as well.
        //The only difference is that the login credentials
```

```
    //of a Customer are saved in a different entity called
    //LoginCredentials, whereas a RegisteredCustomer has its own credentials

    if self.Customer->iterate(c |
        if(c.LoginCredentials.username == self.login.LoginForm.username) &&
        if(c.LoginCredentials.password == self.login.LoginForm.password)
        endif
    )then
        c.islogged  = True
    endif

    //function
    self.checks_balance():
        if self.islogged == True then
            return self.PrePaidCard.balance
        endif
```

##############################**Constraints END**##############################

##################################**Additions**################################

Addition 1:
Added a SSLCertificate entity so the Customers can securely
(register && login), plus evil twin websites attacks become more obvious
for the end user (because of the httpS, etc).
The SSLCertificate must be signed by a Certificate Authority, and its domain name, and ip address, must match those of the website.

```
Context Website:
    inv: self.has.SSLCertificate.domain = self.domain
    inv: self.has.SSLCertificate.ip_address = self.ip_address
```

Addition 2:
The exercise specifies that credit card information must be stored, thus
no external company should take care of that. I decided to implement my own security measures. The encryption used is AES 254bit, and each RegisteredCustomer
has a key. The key cannot be used unless the RegisteredCustomer is logged in.
I tried to check the security approaches and encryption used by most agencies
that secure bank transfers and store securely bank information here[1], and in some other places but I did not find any reference to the kind of encryption used.

[1]:https://www.pcisecuritystandards.org/pci_security/

```
Context EncryptionKey:
    inv: self.encrypts implies RegisteredCustomer.islogged = True
    inv: self.decrypts implies RegisteredCustomer.islogged = True
```