High Level Design

Based on the analysis, the High Level Design (HLD) is carried out below for Classes, Interfaces, Constants, Statics, Exceptions, and overall design considerations.

1 Design Principles:

The following design principles may be adhered to in the HLD:

• Flexible & Extensible Design

The design should be flexible. That is, it should be easy to change the changeable parameters (like base rate, load factor etc.) easily from the Application space. This should not need re-building of the library of classes.

The design should be extensible. That is, it should be easy to add new behaviour (classes) wherever indicated in the specification (like Booking Classes, Booking, Passenger, etc.). This should not require a re-coding of the existing applications.

• Minimal Design

Only the stated models and behaviour should be coded. No extra class or method should be coded.

Less code, less error principle to be followed.

• Reliable & Safe Design

Reliability should be a priority. Everything should work as designed and coded.

Data members, methods and objects should be made constant wherever possible.

Parameters should be appropriately defaulted wherever possible.

The system should never be allowed to go into an inconsistent state.

All possible errors of data and processing must be appropriately thrown and caught handled.

• Testable Design

Every class should support the output streaming operator for checking intermittent output if needed.

Every class should be tested with an appropriate test application for its unit functionality.

Test Applications and regression test suites should be designed for testing the application on (at least) the common scenarios of use.

2 Classes:

• Class **Station** HAS-A name.

• Class **Railways** is a singleton called IndianRailways. It has a collection of the Stations and their mutual distances. IndianRailways is a constant object.

• Class **Date** is a simple class.

•Class **Gender** is a simple abstract class having two derived class Male and Female following parameterized Polymorphism.

• Class **BookingClasses** HAS-A loadFactor,luxuryStatus and ReservationCharge. Remaining attributes may be encoded on the methods in the hierarchy classes. Inclusion-Parametric polymorphic hierarchy is followed.

•Class **BookingCategory** has 6 types of booking category. Flat single level hierarchy is followed.

• Class **Booking** has parallel polymorphic hierarchy as of BookingCategory.

• Make data members as const (or const reference) wherever possible.

•Class **Consession** follows an ad-hoc polymorphism.

•Class **Disability** stores all types of disability and stores concession rates according to several BookingClasses. It also follows a Inclusion-parametric Polymorphism.

2.1 Modeling Sub-Types

During the analysis, we see that there are number of classes and sub-classes including Gender, BookingClass, BookingCategory, and Divyaang where static sub-typing exists as the sub-classes mostly have the same set of data members and methods.

Note: we have already decided to use a static hierarchy with ad-hoc polymorphism for Concessions. Because all of the sub-classes are nearly unique and have nothing to do with one another. Also there is common method, so inheriting a virtual function is useless here.

There are own limitations for both inclusion polymorphism and static typecasting. So to get the best of both approaches, we may opt for a inclusion polymorphism of one level and have parametric polymorphism for the alike leaf classes.

So we would use the above inclusion-parametric polymorphism for Gender, BookingClass, , Divyaang. And we use ad-hoc polymorphism for Concessions and Exceptions.

BookingCategory and Booking class is using flat one level inclusion hierarchy for ease of use.

Finally, Date, Station, Railways, and Passenger are simple classes,hence no hierarchy.

2.2 Virtual Construction Idiom

This concept will be used to implement the parrel construction mentioned for Booking Class. We know that constructor for a class is static and it cannot be virtual. But conceptually, here the situations is to choose between a set of Booking sub-classes as there are BookingCategorys - one for each. Now, we need to create an appropriate object of another (Booking) hierarchy. Notionally, we need to virtualize the construction process. A naive solution would be to explicitly check the type of BookingCategory object, and create corresponding Booking class object which suffers from the usual evils of being type unsafe.

So we would use explicit type-switch fuction inside Booking Category Class to construct correspong Booking Class sub-classes. This hierarchy design is followed.

3 Interfaces

• Constructors / Destructors: Proper constructor and destructor for every class

• Copy Functions: Provide user_defined Copy Constructor and / or Copy Assignment Operator for a class if used in the design (should not be needed). Otherwise, block them.

• Provide output streaming operator for every class to help output process as well as debugging

• Class **Station** to have *GetName*() for accessing its name and *GetDistance*(.) to get distance to another station.

• Class **Railways** to have *GetDistance(.,.)* to get distance between a pair of stations. It should also have proper interface for making it a singleton IndianRailways

• Class **BookingClasses** to have GetLoadFactor(), GetName(),GetReservationCharge() to get access to various BookingClasses properties. Depending on the polymorphic hierarchy, these methods may be non-polymorphic and / or polymorphic (and in some case pure) in BookingClasses and its various derived classes. Consider making them const methods.

•**BookingCategories** will have CreateBooking() to create booking type parallel to its type.
• Class **Booking** to have ComputeFare() to implement the fare computation logic and ReserveBooking() to create type parallel to **BookingCategories.**

4 Constants

This should be constant(not necessarily static):
•Station Names
•Distances b/w Stations
•concession for Divyang Category
The following should be static constants in appropriate classes:
• Load Factors of various BookingClasses
• Base Fare Rate: Rs. 0.50 / km
• Reservation Charges of various BookingClasses
• Tatkaal contraints of various BookingClasses
• Reservation Charges of various BookingClasses
• Concessions for Senior Citizen

5 Statics

• Class Gender have gender names
• Class Date to have month names.
• Class Disability has name and all concession rates corresponding to BookingClasses.
• Class BookingClasses to have load factors,name,reservation charge and all other data members(due to parametrized polymorphism)
• Class Booking to have sBaseFarePer, sBookings (list of bookings done), sBookingPNRSerial (next available PNR.

6 Errors & Exceptions

• Date validations should include :
    All dates should be valid. For example, 29/02/2021 or 31/04/2020 should be declared invalid.
    Month should be valid(between 1 to 12). //13/2021 should be declared invalid
    Range of valid years would be 1900 to 2099
• Station validations should include (may have more):
    name cannot be empty.
• Railways validations should include (may have more):
    No duplicate Station name would be allowed
    Distance must be defined between every pair of Stations. The definition is considered symmetric -so only one direction should given.
    No duplicate distance definition is allowed
    Distance between two same Stations is not allowed
• Passenger validations should include (may have more):
    At least one of first name and last name must be non-empty. middle name may be empty.
    dateOfBirth must precede dateOfReservation and must be a valid date
    gender must be male or female. It must be valid by input. That is, it should not be possible to input a wrong gender.
    aadhaar # is 12 digit. It should be validated for absence of non-digit and length.

mobile # is 10 digit. It should be validated, if provided, for absence of non-digit and length.
disability type must be valid by input.
disabilityID #: Number of the divyangjan ID (optional)

• All Booking requests are taken to be correct. That is, the Staions as mentioned - do exist, the Date is valid (in future), and no invalid BookingClass is requested

• Booking validations should include (may have more):
fromStation and toStation must be valid (pre-existing). The distance between them must be pre-set.
dateOfBooking must be later than dateOfReservation and within one year from it.
bookingClass must be valid by input. That is, it should not be possible to input a wrong bookingClass to the request.
bookingCategory must be valid by input. That is, it should not be possible to input a wrong bookingCategory to the request.
passenger data must be consistent with the bookingCategory.
All valid booking requests can be served.

• BookingClass, BookingCategory, Concessions, and Divyaang are constructed from static data and can be assumed to be free of errors.

• If the construction of an object of a class has possibility of exception due to erroneous inputs, the same should be checked in a separate static function before invoking the constructor. This helps follow the guideline that no exception would be thrown from a constructor.

• The following principle may be followed in error management:
Every error must be properly handled and meaningfully reported.
If there are more than one validation failures, the system should attempt to report as many of them as possible in a single run.
All validations and reporting should be based on exceptional design clearly separating the normal ow from the exception ow.
An appropriate hierarchy of exception classes may be designed for the error management.

• In no case, the system may be allowed to go to an inconsistent state and / or crash.

Low-Level Design

Based on the High Level Design (HLD), the Low Level Design (LLD) is performed. LLD makes use of the specific constructs and idioms of C++.

1 DESIGN PRINCIPLE:

The following design principles may be adhered to in the LLD:

• Encapsulation
Maximized encapsulation for every class
Use of private access specifer for all data members that are not needed by derived classes,if any. Use of protected otherwise.
Use of public access specifer for interface methods and static constants and friend functions only.

• STL Containers
Use STL containers (like vector, map, hashmap, list, etc.) and their iterators. Do not use arrays.
Use iterators for STL containers. Do not use bare for loops.

• Pointers & References

Minimize the use of pointers. Use pointers only if you need null-able entities
If you use pointer for dynamically allocated objects (should be minimized), remember to delete at an appropriate position.
Use const reference wherever possible.

• Use CamelCase for naming variables, classes, types and functions
• Every name should be indicative of its semantics
• Start every variable with a lower case letter
• Start every function and class with an upper case letter
• Use a trailing underscore (_) for every non-static data member
• Use a leading 's' for every static data member
• Do not use any global variable or function (except main(), and friends)
• Every polymorphic hierarchy must provide a virtual destructor in the base class
• *Constructors and destructors should never throw
• *Virtual functions should not be called in constructors of base classes
• Prefer C++ style casting (like static cast<int>(x) over C Style casting (like (int))

2 DESIGN of Classes, Data Members & Methods:

Design of Classes:

(1)Station:

This is a simple class use as data member by railways and booking class with *unique* object for each station.
The specific data member(name of the station as string variable) should neither be needed nor changed once objects are made, so they private and const.
**Constructors: can be created passing the station name (string) or copy constructor.
**copy assignment operator is blocked(made private) as there are are only five unique stations (at in this assignment) and they should not be duplicated.
**GetName function returns the name of the station for furute use. As this returns a constant data member(const refernce to string) the function is also declared const so that it would not change the name.
**GetDistance fuction returns the distance between this station and a other station. this is invoked as member function of station class and calls the member function of Railways class,
**Getdistance as the distance are already stored in Railways data member.
**Destructor is trivial.
**Output operator << overloaded using friend function. prints only the name of the station in the format : stationName

(a)Data members : Private data members
st_name - name of the station
(b) Member functions: All are public
Station()-constructor
~Station()-destructor
GetName()- Returns the name of the station.

(2)Railway Class:

This is a singleton class named IndianRailways

**Constructor is private (singleton class) to ensure only one object creation. static data members sStations,sDistStations are build(initiated before it) inside it. As these datas are constant for this assignment(as per assumption) they are hared-coded here.
**Copy constructor is blocked
**Copy assignment operator is blocked
**Destructor is private for the same reason. As we had array of pointers we need to explicitely detele then by iterating to ensure no memory leakage.
**sStations ,sDistStations are utility data members to easily find a station from given list of stations, and find distance between a pair of station respectively. Map data structure used for effective search of station/distance by their name/pair of station. They are constructed in a way that there is unique object for each station and other use pointer to that;
**IndianRailways is singletone instance of the Railways class that is constant and static for the whole program. When inveked local object will be created and refernce to it will be returned.
**GetDistance function,given two stations, returns the distance between them. Declared const as it does not change any data member of the class.
**GetStation function , given a name of a station, returns the pointer to the station stored in sStations. Declared const as it does not change any data member of the class.
**Output operator << overloaded using friend function. prints the list of the stations in the format : (Stations: stationName1 stationName2.........), name of the sations seperated by spaces. prints the distances between a pair of stationS in the format:

```
(Distance b/w pair of Stations:
stationName1 stationName2 : distance
stationName1 stationName2 : distance
...........
)
```

(a)Data Members: Protected data members

- sStations: station names
- sDistStations: all pairs of distances stored symmetrically

(b) Member Functions: All are public

- Railways(): constructor
- ~Railways():Destructor
- GetDistance()
- GetStation()

(3)Booking:
**sBookingPNRSerial static data member that computes available PNR(basically tracks no. of Booking object created)
**sBaseFarePerKM,sACSurcharge,sLuxuryTaxPercent are static to be easily chaged at intantiation point and available throughout the program, const as it may not be changed during runtime, protected as need not be accessed from outside the class
**fromStation_, toStation_,date_,bookingClass_,PNR_ are protected to be accessed by subclass (future), const as they should not be changed once a booking is constructed.
**fare_,bookingStatus_ , are no-const if they need to be changed in future implementation.

**sBookings is public data member that stores all the bookings done. will be accessed from main.
**copy assignment operator is blocked as booking is unique and can not be duplicated
**ComputeFare function calculates the fare for the current booking following the bussiness logic.
This function is declared vitual for future extension(can be overriden in derived classes)
**output operator << overloaded using friend function. prints in the format :

```
(bookingMessage_
PNR Number = PNR_
From Station = fromStation_
To Station = toStation_
Travel Date = date_
Travel Class = bookingClass_
Fare = fare_)
```

**Destructor is purely virtual to support polymorphism and eliminate slicing. As we had array of pointers we need to explicitly delete them by iterating to ensure no memory leakage.

Data Members :
<u>protected</u>
- fstation, tostation
- name
- Fare :computed from ComputeFare()
- DateOfBooking: date of travelling
- DateOfRservation : current date(from day())
- bookingClass : Booking class
- sPNRserial: to assing PNR to new bookings
- sBaseFare: base fare rat per KM
- PNR, IsAc, IsLuxury, isSitting,Tiers,Name

<u>Public</u>
- passanger

(b) Member Functions: All are public
- Booking()-Constructor
- ~Booking()-Destructor
- ComputeFare()- Calculates the amount of ticket
- ReserveBooking: takes bookingcategory from constructor as input and constructs corresponding booking sub-class, with the help of CreateBooking() from Booking Category.
- day() - Compute day from date using time.h

(c)Hierarchy:
Has a parallel hierarchy to that of BookingCategory.

(4)Date:
This is a simple class use as data member by other classes.

The specific data members should neither be needed nor changed once a date objects are made so they private. Months are to output as month name in letter, so enum is used for that.
Constructors: date object can be made by either passing three integers (dd,mm,yy) or a date class instance. So constructor and copy constructor are provided.
Copy assignment operator is also defined for future need.
Destructor is trivial.
Output operator << overloaded using friend function. prints in the format : (dd/MMM/yy)

(a) Data Members :All are private
- enum to store months
- date_ ,month_ ,year_

(b) Member Functions: All are public
- Date()-Constructor
- ~Date()-Destructor
- validDate()- Check validity (for day, month and year)

(5) Passenger:
Passanger is a simple class. Although all data members are private gender and disability is deliberately made public because they will be needed while calculating concession on fare.
(a) Data Members :
Private:
- firstname , middlename , lastname, aadhar,MObNo
- disabilityID
- dob- date of birth

Public:
- gender
- disabilitytype

(b) Member Functions: All are public
- Passenger()- Constructor
- ~Passenger- Destructor
- GetAge():return age of the passanger

(6)BookingClass:

This is a class hierarchy following the following polymorphic design:
Asingle level hierarchy. Subclasses derived from its base class.

```
                              BookingClass
                              (abstract class)
                                         |
          (concrete singletone class)              ----------------------------------------
          1.ACChairCar
          2.SecondSitting
          3.ACFirstClass
          4.AC3Tier
          5.AC2Tier
          6.Sleeper
          7.FirstClass
          8.ExecutiveChairCar
```

[Design principles of these classes]

**BookingClass

As loadFactor,name,reservation charge are commonly used and needed to be returned for other function and classes they are specified in the Base Abstract class itself.
They are protected so that can be acessed by subclasses but not from outside.
But the luxury status,ac status, sitting status and no. of tiers along with data members of the abstract base class are given a static reference in derived class BookingClassTypes.
The constructor of the base class will be used by subclasses.
GetLoadFactor and GetName function are defined at the base class as they will be They are constant function so no data member will be modified.
Destructor is purely virtual to ensure there is no 'slicing'. It is instatiated after the class definition.
Output operator << overloaded using friend function. prints the list of the stations
in the format :

```
          (MOde:Sitting/Sleeping
           Comfort:AC/Non-AC
           Bunks: tiers_
           Luxury:Yes/No)
```

(a) Data Members :All are protected
- Name: name of the station
- loadFactor
- reservationCharge

(b) Member Functions: All are public
- GetLoadFactor : return the load factor
- Get reservationCharge: return reservationCharge
- GetName() : return the Name

It has derived class BookingClassTypes(data members: all static) that is statically typecasted into 6 types AC3Tier, ACFirstClass, AC2Tier, FirstClass, ACChairCar, Sleeper, SecondSitting,ExecutiveChairCar

(7)Booking Category:
This is base abstract Class having pure virtual destrucor to ptrevent slicing.
(a) Data Members :All are protected
- Name

(b) Member Functions: All are public
- CreateBooking(): returns corresponding subclasses of Booking.

The sub-classes singleton are General, Ladies, SeniorCitizen, Divyang, Tatkal, premTatkal

(8)Gender Class:
(a)Data member:private
- Name

(b) Member Functions: protected
- getName()
- IsMale() : return true if male false if female

(c) Hierarchy:
It has derived class genderTypes(static datea member sName) templating two tyoes: Male,Female.

(9) Concession Class:
This is a base class having no attribute of Operation.
The subclasses are:
- NoConcession:
  Method:
  getConcession() :always returns 0
- LadiesConcession:
  Method:
  getConcession() :not yet configured
- SeniorConcession:
  Method:
  getConcession(Passager) : returns concession based on gender of Passanger
- DivyangConcession:
  Method:
  getConcession(BCname,Passanger) : returns concession based on disability of Passanger and name of the BookingClass

(10) Disability Class:

This is also a simple parameterized polymorphic hierarchy class. Very similar to that of Gender Class.
All parametrs(name and bookingClass specific concession rates) of DisablilityTypes derived class are static.
Here the4 types are: Blind,Cancer,TB,OrthoPed.