Computer Science & Engineering Department
I. I. T. Kharagpur

Software Engineering: CS20006

Assignment – 4: UML Diagrams: Analysis & Design Phases *Marks: 100*
Assignment – 5: Test Plan, Implementation & Test Report *Marks: 100*

Assign Date: *18th March, 2021* [**Assignment 4**] Submit Date: *23:55, 28th March, 2021*
Assign Date: *18th March, 2021* [**Assignment 5**] Submit Date: *23:55, 4th April, 2021*

Important Notes

These assignments develop over Assignment 3 in the following ways:

1. Extends the specification by introducing information of passenger and booking category
2. Extends the requirements for an exception-safe design
3. Expands the design principles by mixing parametric polymorphism (templates for static time polymorphism) with polymorphic hierarchy (virtual functions for run-time polymorphism) for better type-safety, efficiency, flexibility, and code re-use.
4. Expects extensive use of STL for better quality, efficiency, and code re-use
5. Expects a proper multi-file code organization (with `.h` and `.cpp` files) for better code maintenance
6. Expects a more formal development process supported by UML, HLD, LLD, Test Plan, and Test Reprots

Much of the document is re-used from Assignment 3. New additions (to existing sections) or new sections are marked with * or highlight. Deletions are struck out as ~~deleted~~. These are for ease of reading and understanding. However, the entire document should be carefully studied for fine details.

Explanatory Appendix B for Unit Testing has been omitted for brevity.

We need to develop a rudimentary railway reservation / booking system (somewhat like IRCTC Train Ticket Booking, but extremely scaled down in features). We present various stages of this development process leading finally to the specific tasks of the assignment.

1 Specification

This is the outline specification that has been acquired from the client.

1.1 Requirement Statement

The entities involved in the booking system design include:

- Station (Section 1.4.1): Every Station is identified by its name. Booking is done between any two Stations.
- Railways: It is the Indian railways. It has a collection of Stations with pairwise distance between Stations known a priori. Naturally, there can be only one Railways, called IndianRailways, in the system.
- *Date: Any valid date in dd/MMM/yyyy or dd/mm/yyyy format[1] is allowed. No same day booking is allowed. Hence the date of travel must be later than the date of booking (current date in the system). Booking for up to one year in advance is allowed.
- BookingClass (Section 1.4.2): There are several BookingClasses for travel (as in Indian Railways fare classes explained). Each BookingClass has the following attributes:
  - *Name*: Name of the BookingClass

[1] Choice may be made between the two formats – both need not be supported

  - *Fare Load Factor*: The factor by which the fare for travel by this BookingClass would be loaded over the base fare. This may change from time to time.
  - *Seat / Berth*: Whether the BookingClass provides sleeping berths or just seats. This will not change in future.
  - *AC / Non-AC*: Whether BookingClass is air-conditioned or otherwise. This will not change in future.
  - *# of Tiers*: How many tiers exist in the coach for this BookingClass. This will not change in future.
  - *Luxury / Ordinary*: Whether this BookingClass is considered luxurious by the Government. This may change from time to time.
  - **Reservation Charge*: The reservation or booking charge as levied for the BookingClass. This may change from time to time.

  Further, some BookingCategorys allow for priority booking (called *Tatkal*[2]) on a higher *Tatkal* fare depending on the BookingClass of travel and fare as given in the *Talkal Charges Matrix* in Section 1.4.3.

  New booking classes may be added in future.

- *BookingCategory (Section 1.4.3): There are several BookingCategory for travel (as in IRCTC Book Ticket). Each BookingCategory has the following attributes:
  - *Name*: Name of the BookingCategory
  - *Eligibility*: Eligibility criteria / conditions for the BookingCategory – typically dependent on the Passenger

  Further,
  - Some BookingCategorys allow for *Concessional* fare based on the BookingClass and the *eligibility* of the Passenger as given in the *Booking Category Matrix* and the *Disability Concession Factor Matrix* in Section 1.4.3.
  - Some BookingCategorys allow for priority (*Tatkal*) booking on a higher *Tatkal* fare depending on the BookingClass of travel and fare as given in the *Talkal Charges Matrix* in Section 1.4.3.

  New booking categories may be added in future.

- Booking (Section 1.3): A Booking is requested with the following information:
  - *fromStation*: Station from which the travel starts for the Booking. This is given by the name of the Station
  - *toStation*: Station at which the travel ends for the Booking. This is given by the name of the Station
  - **dateOfBooking*: Date of travel for the Booking. This must be greater than *dateOfReservation* and within one year of it.
  - **bookingClass*: BookingClass for the Booking. This is given from a set of available options (as if a drop-down menu, if the application were build with a GUI).
  - **bookingCategory*: BookingCategory for the Booking. This is given from a set of available options (as if a drop-down menu, if the application were build with a GUI).
  - *passenger*: Details of the passenger including name, date of birth, gender, aadhaar number, mobile number, and disability id. ~~category of the passenger. *This is for future extension and optional for now.*~~
  - **dateOfReservation*: Date on which the Booking is done.

  On request of a Booking, the same is processed and fare is computed based on the business logic given in Section 1.3. The Booking is then confirmed with *PNR* and other details on the output. *PNR* is serially allocated starting with 1.

- Passenger: A Passenger may have the following details:
  - *name*: Name of the passenger comprising (input as three separate strings):
    - **firstName*: Optional if *lastName* is present
    - **middleName*: Optional

[2] *Tatkal* means *immediately*. It is a ticket issued by the Indian Railways to provide reservation to passengers who have to undertake a train journey at short notice

        * **lastName*: Optional if *firstName* is present
    - *dateOfBirth*: Date of birth to be used for verification of age and decisions about eligibility for a BookingCategory
    - *gender*: Gender of the passenger: *male* or *female* – to be used for verification of identity and decisions about eligibility for a BookingCategory
    - *aadhaar #*: 12-digit Aadhaar Number to be used as a unique ID and input as a string
    - *mobile #*: 10-digit Mobile number (optional) and input as a string
    - **disability type*: Type of disability (optional)
    - **disabilityID #*: Number of the divyangjan ID (optional) and input as a string. This is used to check eligibility for *Divyaang* BookingCategory booking.
    - ~~*category*: One of *General*, *Ladies*, *Senior Citizen*, *Divyaang*, *Tatkal*, *Premium Tatkal*~~

1.2 Assumptions

The following **assumptions** are made for the design:

- IndianRailways has a given set of Stations with distances known a priori. The list of Stations and distances between them are given as Master Data in Section 1.4. No new station can be added to the IndianRailways and distance between pair of stations do not change.
- *A Booking, as requested, is always available, **if valid** - between any pair of Stations, on any Date, for any BookingClass, in any BookingCategory, and for any Passenger
- ~~No passenger information is considered for the Booking~~
- ***The booking system would always be in a consistent, well-defined state**. All input data (of settable masters like Stations and distance, or Booking inputs etc.), should be validated for format, type, and consistency. All kinds of errors and exceptions in business logic and processing algorithm should also be handled properly.

1.3 *Business Logic

The fare between a pair of stations for a booking class is determined through the following steps:

- *Base Fare Rate*: The base fare for every KM of travel = Rs. 0.5. This may change from time to time.
- *Base Fare*: The base fare between two stations is computed by multiplying the distance between the stations with the base fare for every KM of travel. The base fare applies to the *Sleeper* booking class.
- *Loaded Fare*: For booking classes other the *Sleeper*, the fare is loaded by a factor with respect to the *Sleeper* booking class fare as shown in the *Booking Class Matrix* (Section 1.4.2). The load factor may change from time to time.
- ~~*AC Surcharge*: Further, for air-conditioned classes, AC surcharge of Rs. 50 will be charged on the *loaded fare*. This may change from time to time.~~
- ~~*Luxury Tax*: Finally, there is a 25% luxury tax to be imposed for all luxury class bookings on the fare computed with surcharge. This may change from time to time. The luxury classification as well as taxation rate may change from time to time.~~
- Depending on the BookingCategory, a passenger may get some percentage of concession on the loaded fare according to the tables given in Section 1.4.3. Currently concessions are allowed in *SeniorCitizen* and *Divayaang* categories only. However, in future, concession may be allowed in *Ladies* category as well.
- No concession is allowed for *Tatkal* booking (in *Tatkal* and *PremiumTatkal* categories). For a *Tatkal* booking, a premium is charged on the base fare (as shown in Section 1.4.3) capped by minimum and maximum amounts and minimum distance of travel. After adding this premium, the loading for the booking class is applied as before (Section 1.4.2).

  The premium is double of *Tatkal* for every *PremiumTatkal* booking.
- Finally, a reservation charge is added to the fare depending on the class of travel (Section 1.4.2).
- Final fare is rounded to the nearest integer.
- *dateOfBooking* has no effect on the fare.
- ~~*Passenger* has no effect on the fare as it is being ignored for now.~~

1.3.1 Example: Booking Category = General

For a booking from **Delhi** to **Mumbai**:

By *AC3Tier*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *AC3Tier* = Rs. 723.50 * 2.50 = Rs. 1808.75
- After adding the reservation charge, we get Rs. 1808.75 + Rs. 40.00 = Rs. 1848.75 ≈ Rs. 1849/= (rounded)

By *ACFirstClass*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *ACFirstClass* = Rs. 723.50 * 6.50 = Rs. 4702.75
- After adding the reservation charge, we get Rs. 4702.75 + Rs. 60.00 = Rs. 4762.75 ≈ Rs. 4763/= (rounded)

1.3.2 Example: Booking Category = Senior Citizen

For a booking from **Delhi** to **Mumbai**:

By *AC3Tier* for *Male*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *AC3Tier* = Rs. 723.50 * 2.50 = Rs. 1808.75
- Concession fare = Rs. 1808.75 * (1.00 - 0.40) = Rs. 1085.25
- After adding the reservation charge, we get Rs. 1085.25 + Rs. 40.00 = Rs. 1125.25 ≈ Rs. 1125/= (rounded)

By *ACFirstClass* for *Female*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *ACFirstClass* = Rs. 723.50 * 6.50 = Rs. 4702.75
- Concession fare = Rs. 4702.75 * (1.00 - 0.50) = Rs. 2351.375
- After adding the reservation charge, we get Rs. 2351.375 + Rs. 60.00 = Rs. 2411.375 ≈ Rs. 2411/= (rounded)

1.3.3 Example: Booking Category = Divyaang

For a booking from **Delhi** to **Mumbai**:

By *AC3Tier* for *Blind*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *AC3Tier* = Rs. 723.50 * 2.50 = Rs. 1808.75
- Concession fare = Rs. 1808.75 * (1.00 - 0.75) = Rs. 452.1875

- After adding the reservation charge, we get Rs. 452.1875 + Rs. 40.00 = Rs. 492.1875 ≈ Rs. 492/= (rounded)

By *ACFirstClass* for *Cancer Patient*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *ACFirstClass* = Rs. 723.50 * 6.50 = Rs. 4702.75
- Concession fare = Rs. 4702.75 * (1.00 - 0.50) = Rs. 2351.375
- After adding the reservation charge, we get Rs. 2351.375 + Rs. 60.00 = Rs. 2411.375 ≈ Rs. 2411/= (rounded)

1.3.4 Example: Booking Category = Tatkal

For a booking from **Delhi** to **Mumbai**:

By *AC3Tier*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km = Rs. 723.50
- Loaded fare for *AC3Tier* = Rs. 723.50 * 2.50 = Rs. 1808.75. This is the basic fare to be used computing tatkal charge
- 30% of basic fare = Rs. 1808.75 * 0.30 = Rs. 542.625
- Tatkal charge = Rs. 400.00 (by maximum cap)
- After adding the reservation charge, we get Rs. 1808.75 + Rs. 400.00 Rs. 40.00 = Rs. 2248.75 ≈ Rs. 2249/= (rounded)

For a booking from **Chennai** to **Bangalore**:

By *ACFirstClass*:

- Distance from Chennai to Bangalore = 350km
- Base fare = 350km * Rs. 0.5 / km = Rs. 175.00
- Loaded fare for *ACFirstClass* = Rs. 175.00 * 6.50 = Rs. 1137.50
- 30% of basic fare = Rs. 1137.50 * 0.30 = Rs. 341.25
- Tatkal charge = Rs. 0.00 (by minimum distance cap)
- After adding the reservation charge, we get Rs. 1137.50 + Rs. 0.00 + Rs. 60.00 = Rs. 1197.50 ≈ Rs. 1198/= (rounded)

1.4 Master Data

While it will be nice to read the master data from master files at the start of the system run, it will be fine to hard-code these data for this assignment. However, the hard-coding should be done in limited, well-documented areas of the code so that it will be easy to change them as needed. These should be hard-code inside the implementation of functions / methods.

1.4.1 Stations

IndianRailways has *five* Stations, namely: *Mumbai*, *Delhi*, *Bangalore*, *Kolkata*, and *Chennai*. The distances between the stations are given below:

Station Distance Matrix

From Station	**To Station**				
	Mumbai	*Delhi*	*Bangalore*	*Kolkata*	*Chennai*
	Distance in KM				
Mumbai	X	1447			
Mumbai			981		
Mumbai				2014	
Mumbai					1338
Delhi		X	2150		
Delhi				1472	
Delhi					2180
Bangalore			X	1871	
Bangalore					350
Kolkata				X	1659

Distance between a pair of stations is symmetric

1.4.2 Booking Classes

IndianRailways has *eight* booking classes as follows - shown with their respective attributes:

****Booking Class Matrix***

Booking Class	*Name*	**Fare Load Factor*	*Seat / Berth*	*AC*	*# of Tiers*	*Luxury / Ordinary*	**Reservation Charge (in Rs.)*
ACFirstClass (1A)	**AC First Class**	6.50	Berth	Yes	2	Luxury	60.00
**ExecutiveChairCar*	**Executive Chair Car**	5.00	Seat	Yes	0	Luxury	60.00
AC2Tier (2A)	**AC 2 Tier**	4.00	Berth	Yes	2	Ordinary	50.00
FirstClass (FC)	**First Class**	3.00	Berth	No	2	Luxury	50.00
AC3Tier (3A)	**AC 3 Tier**	2.50	Berth	Yes	3	Ordinary	40.00
ACChairCar (CC)	**AC Chair Car**	2.00	Seat	Yes	0	Ordinary	40.00
Sleeper (SL)	**Sleeper**	1.00	Berth	No	3	Ordinary	20.00
SecondSitting (2S)	**Second Sitting**	0.60	Seat	No	0	Ordinary	15.00

- *New booking classes may be added in future*
- *Fare load factors may change from time to time*
- *Reservation charges may change from time to time*
- *Luxury / Ordinary categorization may change according to tax rules*
- *Seat / Berth & AC / non-AC classification, and # of tiers will not change in future*
- *IRCTC Book Ticket*
- *Indian Railways fare classes explained*

1.4.3 *Booking Categories

Tickets can be booked in the IndianRailways in one of *six* categories have the respective attributes:

Booking Category Matrix

Booking Category	*Name*	*Concession Factor*	*Remarks*
General	**General**	0.00	General booking available to all
Ladies	**Ladies**	0.00	Special booking for ladies and 12− years male Applies for berth priority. No fare concession
SeniorCitizen	**Senior Citizen**	0.40 0.50	Special booking for 60+ years male Special booking for 58+ years female
Divyaang	**Divyaang**	-	Special booking for the disabled as charged by ***Disability Concession Matrix***
Tatkal	**Tatkal**	0.00	Priority booking 1 day before travel as charged by ***Tatkal Charges Matrix***
PremiumTatkal	**Premium Tatkal**	0.00	Priority booking 1 day before travel as charged by ***Tatkal Charges Matrix***

- *New booking categories may be added in future*
- *Fare load factors may change from time to time*
- *Processing fees may change from time to time*
- *Rail Fare Concession for Senior Citizens*
- *Rail Fare Concession for Disabled Persons*

Disability Concession Factor Matrix

Booking Class	**Type of Disability**			
	Blind	**Orthopaedically Handicapped**	**Cancer Patients**	**TB Patients**
ACFirstClass	0.50	0.50	0.50	0.00
ExecutiveChairCar	0.75	0.75	0.75	0.00
AC2Tier	0.50	0.50	0.50	0.00
FirstClass	0.75	0.75	0.75	0.75
AC3Tier	0.75	0.75	1.00	0.00
ACChairCar	0.75	0.75	1.00	0.00
Sleeper	0.75	0.75	1.00	0.75
SecondSitting	0.75	0.75	1.00	0.75

- *Rail Fare Concession for Disabled Persons*
- *Concession Rules*

Talkal Charges Matrix

Booking Class	**Minimum Tatkal Charges (in Rs.)**	**Maximum Tatkal Charges (in Rs.)**	**Minimum Distance for charge (in Km)**
ACFirstClass	400.00	500.00	500
ExecutiveChairCar	400.00	500.00	250
AC2Tier	400.00	500.00	500
FirstClass	400.00	500.00	500
AC3Tier	300.00	400.00	500
ACChairCar	125.00	225.00	250
Sleeper	100.00	200.00	500
SecondSitting	10.00	15.00	100

- ***The Tatkal Charges have been fixed as a percentage of fare at the rate of 10% of basic fare for second class (SecondSitting) and 30% of basic fare for all other classes subject to minimum and maximum as given above***
- ***A Premium Tatkal ticket has same charge rules as above but is charged at double of Tatkal***
- ***No tatkal charge is levied for travel below the minimum distance for charge***
- *Indian Railways Tatkal Scheme*

2 *Analysis of Specification

As discussed in SDLC and UML, the first target in analysis is to extract the Use Case and Class diagrams for the system. Then build the other diagrams – Sequence, Communication, Activity, and State Machine.

2.1 *Use Case Diagram

We first analyze the specifications to identify the actors, use-cases, and the relationships in the problem. We also try to extract possible constraints on the design. Identification of the actors and use-cases for Use-Case Diagrams is left as a part of assignment exercise.

2.2 *Class Diagram

Next we need to identify the classes, attributes, methods, hierarchy, associations, relationships etc. to prepare the Class Diagram. While the overall and detailed Class Diagram will be left as an exercise in the assignment, we analyze to identify the classes and discuss various aspects of the classes to facilitate the process of design.

Our implementation language is pre-decided to be C++. So during analysis, we leave appropriate pointers for HLD and LLD in the context of C++. This is help in the translation of the Class Diagram into the C++ classes.

2.2.1 Summary of Polymorphisms in C++

Before delving into the actual task we present a summary of polymorphisms available in C++ (as was also discussed in the class). We shall make regular references to these.

C++ support the following four kinds of polymorphism.

- ***Ad-hoc Polymorphism***: *Static* polymorphism by *overloading* of methods. Available globally or on a non-virtual hierarchy (inheritance without virtual functions).
- ***Inclusion Polymorphism***: *Dynamic* polymorphism by *overriding* of methods on a polymorphic hierarchy (inheritance with virtual functions)
- ***Parametric Polymorphism***: *Static* polymorphism by *templates* where type is used as parameter. This is using *template meta-programming*.

  This is often combined with Inclusion Polymorphism.
- ***Coercion Polymorphism***: *Type casting*. This would be grossly avoided.

2.2.2 Classes in Booking Software

We identify entities from the specification as classes. Also, we extract some abstract concepts as classes as we factor and normalize for our design.

- ***Class Station**

  Station is a simple data class with unique *name*.
- ***Class Date**

  The Date class needs to validate date from string format and check for equality, leap year etc. It also needs to support computation of age and a span of a year.
- ***Class and Hierarchy of Gender**

  Gender is a simple type with *Male* and *Female* type constants or sub-types. For uniformilty of design, we can model it by inclusion and parametric polymorphism with Gender being an abstract base class.
- ***Class Railways**

  Class Railways should be a singleton and should contain the master data of stations and distances. The singleton should be constant as no station can be added and distances cannot be changed. Further, it needs to exclude duplication of Station and Station to Station distances.

- ***Class and Hierarchy of BookingClasses**

  *BookingClass needs redesign based on our experience in Assignment 3. It can be made more compact and reuseable.

  ~~Different Booking Classes should be a polymorphic hierarchy rooted at BookingClasses which may be an abstract base class. Instead of making it a flat hierarchy, it would be good to make it a multi-level hierarchy. This would need identification of abstract base sub-classes that are aligned with one or more properties of the Booking Classes.~~

  ~~If multiple properties are used in organizing the hierarchy, then the model would need multiple inheritance. However, we do not want to use multiple inheritance for the associated complications and inefficiency. Rather, we would use single inheritance on the strongest property and use the rest as HAS-A with polymorphic value based on the leaf class.~~

  ~~Naturally, there can be two candidates for this as *Fare Load Factor*, *# of Tiers*, and *Luxury / Ordinary* are more like pure attributes and clearly not useful for hierarchy:~~

  – ~~*AC or Non-AC*: Air-condition leads to comfort level, and is not fundamental to travel. So this is a weak candidate.~~
  – ~~*Seat or Berth*: This is fundamental property for a rail travel. So this is a strong candidate.~~

  ~~So we may introduce several intermediate abstract base classes on the strong property and its closest associated attribute, viz. the number of tiers.~~

  *In Assignment 3, we made a multi-level hierarchy of BookingClasseses introducing intermediate abstract classes like *Seat* or *Berth* etc. using the strongest property while modeling the rest of the attributes by HAS-A. Clarifications sought during the assignment and post-implementation analyses tell us that there is no specific semantic interpretation or role for such abstract classes in terms of the business logic involved. So it may be more appropriate the treat all the properties as HAS-A and just model using a single level flat hierarchy rooted at BookingClass which may be an abstract base class. This is then turns out to be more of a *static sub-typing* situation and we can trade off pure inclusion polymorphism with inclusion and parametric polymorphism.

  Further we may note that every concrete booking class has all fixed properties and there should be no need to construct more than one object for any of them. So there may be a singleton constant object for each which, kind of, will stand for its polymorphic type.

  ~~The hierarchy should be extensible in future as new booking classes are added.~~

  *Additionally, as is discussed and explained below for the BookingCategory, the following attributes need to be maintained for a BookingClass for handling the ***Priority Booking*** as given in *Talkal Charges Matrix* in Section 1.4.3. As stated, these may change from time to time.

  – *Tatkal Load Factor*: The factor by which premium is charged for the BookingClass.
  – *Minimum (Maximum) Tatkal Charge*: Minimum (Maximum) *Tatkal* charge for the BookingClass.
  – *Minimum Tatkal Distance*: Minimum distance of travel to levy *Tatkal* charge for the BookingClass.

- ***Class and Hierarchy of BookingCategory**

  Like BookingClasseses, different sub-categories of BookingCategory may be represented by a flat single level hierarchy or *static sub-typing* by inclusion and parametric polymorphism. Every leaf class, however, will need to implement an *Eligibility* policy as a polymorphic behaviour.

  Here we come across an interesting issue. Regarding the role of BookingCategory in terms of determination of *fare*. How should we handle the general, concessional, and priority booking categories?

  – ***General Booking*** has neither any concession nor any premium charge
  – ***Concessional Booking*** has concessions based on a mix factors from BookingCategory, Booking-Class, and *gender*, *age* & *ability type* (*divyaangjan*) of the Passenger
  – ***Priority Booking*** attracts *Tatkal* charge depending on BookingClass and *distance*.

  It will be quite interesting to depict this information through Associations and Relationships in the UML Class Diagram and is left as an exercise.

  While ***General Booking*** does not need any additional support, both concessional and priority booking would need further information representation and polymporphic computation.

- ***Concessional Booking***: Concession is a *ternary* relationship between BookingCategory, BookingClass, and Passenger. This actually gets to be a *quaternary* relationship when we consider concessions due to disability which depends on the specific type of disability too.

  So we can clearly see that we need to acknowledge Concessions and Divyaang (Disability) as key abstract concepts with appropriate polymorphic behavior to complete the modeling of BookingCategory.
- ***Priority Booking***: In contrast to above, *Tatkal* charges depend on the BookingClass and can be subsumed in it by normalization[3]. Hence, we simply add the data members in BookingClass above.

- ***Class and Hierarchy of Divyaang (Disabled)**

  Disability Concession Factor Matrix in Section 1.4.3 tell us that four types of disability are to be considered for *Divyaang* BookingCategory. So different sub-categories of Divyaang may be represented by a flat single level hierarchy or *static sub-typing* by inclusion and parametric polymorphism.

- ***Class and Hierarchy of Concessions**

  For keeping the information of (*selective*) concessions for ***General*** as well as ***Concessional Booking*** according to respective BookingCategorys, a Concessions hierarchy may be created. For this, we note that concession is dependent on BookingCategory, BookingClass, and *gender*, *age* & *ability type* (*divyaangjan*) of the Passenger as given in *Booking Category Matrix* and *Disability Concession Factor Matrix* in Section 1.4.3. Let us summarize the dependence of these information in the table below:

Concessions class based on BookingCategory	**Dependency on class**			**Remarks**
	BookingClasses	**Passenger**	**Divyaang**	
General	No	No	No	No concession
Ladies	No	Yes	No	No concession for now
SeniorCitizen	No	Yes	No	Concession based on gender
Divyaang	Yes	Yes	Yes	Concession as in matrix

Tatkal categories are not considered as there is no concession

  Clearly, no common interface can compute (extract) the concession across different Concessions classes in the hierarchy as specialized from a base class Concessions. So we need to model it by flat single level ad-hoc polymorphism and there is no logical scope to use inclusion or parametric polymorphism. Consequently, the base class cannot be abstract either.

- ***Class and Hierarchy of Booking**

  ~~Booking may be treated as a simple concrete class with the parameters mentioned in the specification. We may keep Passenger as a null-able default for future extension.~~

  ~~Booking may be also be modeled as a polymorphic base class as with the introduction of Passenger in future it is likely to lead to a booking hierarchy.~~

  *In assignment 3, Booking could be a simple concrete class because there was only one algorithm (business logic) to compute fare that used some attribute values of the respective BookingClass. The model now has to improve, because the business logic of Booking depends on the BookingCategory.

  So Booking can be an abstract base class rooting an inclusion and parametric polymorphic hierarchy that parallels the hierarchy of the BookingCategory.

  Though it looks like a cool solution, it get may get tricky in the details. Note that we need to *create* an object of the appropriate Booking sub-class based on its BookingCategory. This means we need to *virtualize* the construction process which is not possible. The inclusion polymorphism of the Booking hierarchy can be used to invoke the appropriate fare computation business logic only after we have the right Booking class object for the BookingCategory.

- ***Class and Hierarchy of Passenger**

  ~~Class Passenger may be an empty abstract base class. Since we are not going to use it, we would not need to make objects for the same. However, it would be good to have it as a polymorphic base for future extension, especially since the specification talks of various categories of passengers.~~

[3]Normalization is the process of organizing the data in the database. Normalization is used to minimize the redundancy from a relation or set of relations. The normal form is used to reduce redundancy from the database table. You will learn about these in depth in your DBMS course.

*The Passenger class is a simple data class that needs to keep data member values as specified. Of course, it will need a number of validation methods for the constraints specified for its data members like *name*, *age*, *aadhaar* (syntax only), etc.

- ***Class and Hierarchy of Exceptions**

  Since erroneous inputs and conditions are allowed now, we need to design a hierarchy of Exceptions classes derived from `std::exception`.

  Fundamentally, we can identify the following top level Exceptions types:

  - Bad_Date: Format or date validity errors.
    This is used by several classes.
  - Bad_Railways: Errors in master data of Bad_Railways. This may be specialized into a number specific sub-classes for the types of error like Station *name* and duplication, distance duplication etc.
    This should be caught in the application.
  - Bad_Passenger: Errors in data of Passenger. This may be specialized into a number specific sub-classes for the types of error like *name*, *age*, *aadhaar*, *mobile*, etc.
    This should be caught in the application.
  - Bad_Booking: Errors in data of Booking. This may be specialized into a number specific sub-classes for the types of error like *date of booking*, *ladies*, *senior citizen*, etc.
    This should be caught in the application.

2.3 *Sequence, Communication, Activity & State Machine Diagrams

Left as a part of assignment exercise.

3 High Level Design

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

3.1 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 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 (Section 6.1).
  - Test Applications (Section 6.2) and regression test suites should be designed for testing the application on (at least) the common scenarios of use.

3.2 *Classes

*The classes and hierarchy as outlined in Requirement Specification (Section 1.1) and Analysis (Section 2), can be put in HLD.

- ~~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 discussed in the lecture modules.~~
- ~~Class BookingClasses HAS-A loadFactor. Remaining attributes may be encoded on the methods in the hierarchy classes.~~
- ~~Class Booking HAS-A fromStation, toStation date, and bookingClass from the booking request where every station name, date and booking class are assumed to have been given correctly. Further it HAS-A fare computed and PNR allocated. Optionally, it may HAS-A bookingStatus (which would be true for this assignment always) and bookingMessage (which may be "BOOKING SUCCEEDED" for this assignment always).~~

  ~~Booking should support Passenger as a null-able parameter for future extension.~~
- *Make data members as `const` (or `const` reference) wherever possible.
- You may add any class, any data member to a class, or any hierarchy as you need for implementation. Justify your design choice for them.

3.2.1 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. Even the Booking can be modeled with this. Specifically, we observe the following:

1. Data members are constants at many places and takes by static values (specific to the sub-class).
2. Methods are mostly identical in *algorithm* and differ in *static data*.
3. In a number of cases, methods need to be invoked by *dynamic dispatch* to support a uniform type interface in the application.
4. Conceptually, in most cases, a *single-level flat hierarchy* with an abstract base class and concrete sub-classes suffice the representation.
5. Most of the classes also represent *static concepts*. Hence, it is desirable that only a *single constant object* of the class should be constructed that can represent the type and be used as a placeholder everywhere for type consistency.

This leads to the question of which form/s of polymorphism in C++ should we use to model the hierarchy. Note that we have already decided to use a static hiearchy with *ad-hoc polymorphism* for Concessions.

If we use *inclusion polymorphism*, we have a greater flexibility for hierarchy along with dynamic dispatch based on the sub-class type, but the code bloats. This is good for (3), (4) & (5), but not (1) & (2).

If we use *parametric polymorphism*, it is relatively difficult to have flexible hierarchy or have dynamic dispatch, but we can have a more compact code with better reuse (and less code to actually write). This is good for (1), (2) & (5), but not (3) & (4).

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. To understand the approaches better, let us work out the example for the design of Gender concept which has two sub-types *Male* and *Female*. We first model by inclusion polymorphism and then by parametric polymorphism and for both cases check the way to write the applications.

By Inclusion Polymorphism

We code Gender as follows:

Header File

```
// Gender.h
#ifndef __GENDER_H
#define __GENDER_H

#include <string>
using namespace std;

// Abstract Base Class - Concept of Gender
class Gender {
    const string& name_; // Name of the gender
protected:
    Gender(const string& name) : name_(name) { }
    virtual ~Gender() { }
public:
    const string& GetName() const { return name_; }

    virtual const string GetTitle() const = 0; // Salutation specific to gender

    static bool IsMale(const Gender&); // Checking and matching gender
};

// Male class - specialized gender
class Male : public Gender {
    Male() : Gender(Male::sName) {}
    static const string sName; // Name "Male" for this gender sub-type

public:
    static const Gender& Type() { // Singleton of Male that represents the type Male
        static const Male theObj; // May be non-const if the type has changeable behavior

        return theObj;
    }

    const string GetTitle() const   // Dynamic dispatch
    { return "Mr."; }               // Salutation is hard-coded - may be taken out as static
};

// Female class - specialized gender
class Female : public Gender {
    Female() : Gender(Female::sName) { }
    static const string sName; // Name "Female" for this gender sub-type

public:
    static const Gender& Type() { // Singleton of Female that represents the type Female
        static const Female theObj; // May be non-const if the type has changeable behavior

        return theObj;
    }

    const string GetTitle() const   // Dynamic dispatch
    { return "Ms."; }               // Salutation is hard-coded - may be taken out as static
};

inline bool Gender::IsMale(const Gender& g) { return &g == &Male::Type(); }
#endif // __GENDER_H
```

Note that there is significant duplication of code between `Male` *and* `Female` *class codes.*

Source File

```
// Gender.cpp
#include <string>
```

```
using namespace std;

#include "Gender.h"

// Names defined as static constants
const string Male::sName = "Male";
const string Female::sName = "Female";
```

We are now ready to use the above classes.

Application File

```
// Gender_App.cpp
#include <string>
using namespace std;

#include "Gender.h"

class Person {
    const string name_;
    const Gender& gender_;
public:
    Person(
        const string& name,
        const Gender& gender) : // Singleton constant Gender sub-class object
        name_(name), gender_(gender) {}

    friend ostream& operator<<(ostream& os, const Person& p) {
        os << p.gender_.GetTitle() << " "   // Dynamic dispatch based on gender type
           << p.name_ << " is a "           // Name set for the Person
           << p.gender_.GetName()           // Static  dispatch on Gender to get the
                                            // name of the gender
           << endl;
        return os;
    }
};

int main() {
    Person p1("Ramen Bag",
        Male::Type());      // Type-safe expression of Male
    Person p2("Elisa Tang",
        Female::Type());    // Type-safe expression of Female

    cout << p1;
    cout << p2;

    return 0;
}
```

Output:

```
Mr. Ramen Bag is a Male
Ms. Elisa Tang is a Female
```

By Parametric Polymorphism

Now we code Gender as follows using parametric polymorphism with inclusion polymorphism:

Header File

```
#ifndef __GENDER_H
#define __GENDER_H
```

```
#include <string>
using namespace std;

// Forward declaration of templatized class
template<typename T>
class GenderTypes;  // Generic Gender type to generate specific genders

// Generic gender type
class Gender { // Abstract Base Class
    const string& name_; // Name of the Gender

    // Tag types - to instantiate the template
    // Note that these names are placeholders only and are not exposed outside the class
    // Also they are put inside the class for not cluttering the global namespace
    struct MaleType {};
    struct FemaleType {};

protected:
    Gender(const string& name) : name_(name) {}
    virtual ~Gender() { }

public:
    const string& GetName() const { return name_; }

    virtual const string GetTitle() const = 0; // Salutation specific to gender

    static bool IsMale(const Gender&); // Checking and matching gender

    // Enumerated types - the target sub-types
    typedef GenderTypes<MaleType> Male;
    typedef GenderTypes<FemaleType> Female;
};

// Specific gender  types
template<typename T>
class GenderTypes : public Gender {
    static const string sName;          // Respective name of the gender
    static const string sSalutation;    // Respective salutation for the gender

    GenderTypes(const string& name = GenderTypes<T>::sName) : Gender(name) { }
    ~GenderTypes() { }

public:
    // Singleton object - placeholder for the respective type
    static const GenderTypes<T>& Type() {
        static const GenderTypes<T> theObject;  // May be non-const for changeable behavior

        return theObject;
    }

    const string GetTitle() const           // Dynamic dispatch
    { return GenderTypes<T>::sSalutation; } // Salutation parametrized by static
};

inline
bool Gender::IsMale(const Gender& g) { return &g == &Gender::Male::Type(); }
#endif // __GENDER_H
#endif // __GENDER_H
```

Note that the earlier duplication of code between `Male` *and* `Female` *class codes are now removed and refactored into the template code. It improves code reuse. In this small example, however, the reduction in LoC is not* *visible (actually it bloats). When we have more sub-classes, like for BookingClass, we shall have significant code reduction and reuse.*

Source File

```
// Gender.cpp

#include <string>
using namespace std;

#include "Gender.h"

// Names defined as static constants
const string Gender::Male::sName = "Male";
const string Gender::Female::sName = "Female";

// Salutations defined as static constants
const string Gender::Male::sSalutation = "Mr.";
const string Gender::Female::sSalutation = "Ms.";
```

Application File

Note that only change in the application is in the scoping of the sub-types as `Male` *(*`Female`*) becomes* `Gender::Male` *(*`Gender::Female`*). This is even more type-safe as the global namespace is not cluttered.*

```
// Gender_App.cpp
#include <iostream>
#include <string>
using namespace std;

#include "Gender.h"

class Person {
    const string name_;
    const Gender& gender_;
public:
    Person(
        const string& name,
        const Gender& gender) : // Singleton constant Gender sub-class object
        name_(name), gender_(gender) {}

    friend ostream& operator<<(ostream& os, const Person& p) {
        os << p.gender_.GetTitle() << " "   // Dynamic dispatch based on gender type
           << p.name_ << " is a "           // Name set for the Person
           << p.gender_.GetName()           // Static  dispatch on Gender to get the
                                            // name of the gender
           << endl;
        return os;
    }
};

int main() {
    Person p1("Ramen Bag",
        Gender::Male::Type());   // Type-safe expression of Male - note the change in scoping
    Person p2("Elisa Tang",
        Gender::Female::Type()); // Type-safe expression of Female - note the change in scoping

    cout << p1;
    cout << p2;

    return 0;
}
```

Output:

```
Mr. Ramen Bag is a Male
Ms. Elisa Tang is a Female
```

So we may use the above inclusion-parametric polymorphism for Gender, BookingClass, BookingCategory, Divyaang, and Booking. And we use ad-hoc polymorphism for Concessions and Exceptions. Finally, Date, Station, Railways, and Passenger will have no hierarchy.

3.2.2 Virtual Construction Idiom

We know that constructor for a class is static and it cannot be virtual. But conceptually, we come across such situations often when we need to choose between a set of sub-classes based on the information of some other types. For example, there are as many Booking sub-classes as there are BookingCategorys - one for each. Now given a BookingCategory in the input, how do we invoke the constructor of the right sub-class of Booking. The situation is again that of a type-switch, except here based on the type of one hierarchy (BookingCategory), 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.

Let us consider a tiny example to understand the problem and the solution. Consider an application for a Swimming Pool Slot booking where separate slots (and pools) based on gender. So we have a PoolSlot abstract class with MalePoolSlot FemalePoolSlot as respective specializations. We use our earlier design of Gender and produce the following code using an explicit type-switch in `PoolSlot::ReservePoolSlot()` function:

```
#include <iostream>
#include <string>
using namespace std;

class Gender {
    const string& name_;
protected:
    Gender(const string& name) : name_(name) { }
    virtual ~Gender() { }
public:
    const string& GetName() const { return name_; }
    static bool IsMale(const Gender&);  // In a good OOP design we must not have /
                                        // need such an interface!
};
class Male : public Gender {
    Male() : Gender(Male::sName) {}
    static const string sName;

public:
    static const Gender& Type() {
        static const Male theObj;
        return theObj;
    }
};
class Female : public Gender {
    Female() : Gender(Female::sName) { }
    static const string sName;

public:
    static const Gender& Type() {
        static const Female theObj;
        return theObj;
    }
};

// Explicit checking of type
bool Gender::IsMale(const Gender& g) { return &g == &Male::Type(); }
```

```
// Names defined as static constants
const string Male::sName = "Male";
const string Female::sName = "Female";

class PoolSlot {
protected:
    const string name_;
    PoolSlot(const string& name) : name_(name) { }
public:
    ~PoolSlot() {}
    static PoolSlot* ReservePoolSlot(const string& name, const Gender& g);
};

class MalePoolSlot : public PoolSlot {
public:
    MalePoolSlot(const string& name) : PoolSlot(name) {
        cout << "MalePoolSlot created for " << name_ << endl;
    }
};

class FemalePoolSlot : public PoolSlot {
public:
    FemalePoolSlot(const string& name) : PoolSlot(name) {
        cout << "FemalePoolSlot created for " << name_ << endl;
    }
};

PoolSlot* PoolSlot::ReservePoolSlot(const string& name, const Gender& g) {
    PoolSlot* p = 0;

    // This is the type-switch that we must avoid
    // This is error-prone, not scalable, and type-unsafe
    if (Gender::IsMale(g))
        p = new MalePoolSlot(name);
    else
        p = new FemalePoolSlot(name);

    return p;
}

int main() {
    PoolSlot* p1 = PoolSlot::ReservePoolSlot("Ramen Bag", Male::Type());
    PoolSlot* p2 = PoolSlot::ReservePoolSlot("Elisa Tang", Female::Type());

    delete p1;
    delete p2;

    return 0;
}
```

Now we refine the design:

1. Drop the explicit type-checking function `Gender::IsMale()`.
2. Introduce a virtual function `Gender::CreatePoolSlot()` for dynamically switching type based on gender
3. Replace explicit type-switch in `PoolSlot::ReservePoolSlot()` by dynamic dispatch on gender type
4. Construct appropriate type of PoolSlot object on overridden versions of `Male::CreatePoolSlot()` and `Female::CreatePoolSlot()` respectively.

```
#include <iostream>
```

```
#include <string>
using namespace std;

class PoolSlot;

class Gender {
    const string& name_;
protected:
    Gender(const string& name) : name_(name) { }
    virtual ~Gender() { }
public:
    const string& GetName() const { return name_; }
    //static bool IsMale(const Gender&);
    virtual PoolSlot* CreatePooSlot(const string& name) const = 0;
};
class Male : public Gender {
    Male() : Gender(Male::sName) {}
    static const string sName;

public:
    static const Gender& Type() {
        static const Male theObj;
        return theObj;
    }

    PoolSlot* CreatePooSlot(const string& name) const;
};

class Female : public Gender {
    Female() : Gender(Female::sName) { }
    static const string sName;

public:
    static const Gender& Type() {
        static const Female theObj;
        return theObj;
    }

    PoolSlot* CreatePooSlot(const string& name) const ;
};

//bool Gender::IsMale(const Gender& g) { return &g == &Male::Type(); }

// Names defined as static constants
const string Male::sName = "Male";
const string Female::sName = "Female";

class PoolSlot {
protected:
    const string name_;
    PoolSlot(const string& name) : name_(name) { }
public:
    ~PoolSlot() {}
    static PoolSlot* ReservePoolSlot(const string& name, const Gender& g);
};

class MalePoolSlot : public PoolSlot {
public:
    MalePoolSlot(const string& name) : PoolSlot(name) {
        cout << "MalePoolSlot created for " << name_ << endl;
    }
};

// Creates MalePoolSlot object
PoolSlot* Male::CreatePooSlot(const string& name) const {
    return new MalePoolSlot(name);
}

class FemalePoolSlot : public PoolSlot {
public:
    FemalePoolSlot(const string& name) : PoolSlot(name) {
        cout << "FemalePoolSlot created for " << name_ << endl;
    }
};

// Creates FemalePoolSlot object
PoolSlot* Female::CreatePooSlot(const string& name) const {
    return new FemalePoolSlot(name);
}

PoolSlot* PoolSlot::ReservePoolSlot(const string& name, const Gender& g) {
    //PoolSlot* p = 0;
    //if (Gender::IsMale(g))
    //    p = new MalePoolSlot(name);
    //else
    //    p = new FemalePoolSlot(name);

    //return p;

    // Dynamic dispatch takes care of the type switch on gender
    return g.CreatePooSlot(name);
}

int main() {
    PoolSlot* p1 = PoolSlot::ReservePoolSlot("Ramen Bag", Male::Type());
    PoolSlot* p2 = PoolSlot::ReservePoolSlot("Elisa Tang", Female::Type());

    delete p1;
    delete p2;

    return 0;
}
```

Effectively, we achieve *virtualization* in construction based on an object hierarchy. However, the cost is - the design of the Gender now gets tightly coupled with the design of PoolSlot (which should not have been). There would the work arounds for that too - but that's later.

So now we are ready to do a good design for Booking class hierarchy.

3.3 Interfaces

*The interfaces, as outlined in Requirement Specification (Section 1.1) and Analysis (Section 2), can be put in HLD.

- 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(), IsSitting(), IsAC(), GetNumberOfTiers(), and IsLuxury() 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.~~
- ~~Class Booking to have ComputeFare() to implement the fare computation logic. Should it be `virtual` (polymorphic) for future extensions?~~
- *Make methods `const` wherever possible.
- You may add any interface to a class (or `private` / `protected` methods) as you need for implementation. Justify your design choice for them.

3.4 Constants

*Various static constants as outlined in Requirement Specification (Section 1.1), Master Data (Section 1.4), and Analysis (Section 2), can be put in HLD.

~~The following should be static constants in appropriate classes:~~

- ~~*Load Factors* of various BookingClasses~~
- ~~*Base Fare Rate*: Rs. 0.50 / km~~
- ~~*AC Surcharge*: Rs. 50.00~~
- ~~*Luxury Tax*: 25% on booking amount~~

3.5 Statics

*Various static data members as outlined in Requirement Specification (Section 1.1), Master Data (Section 1.4), and Analysis (Section 2), can be put in HLD.

- ~~Class Date to have month and day names.~~
- ~~Class Railways to have sStations (list of stations) and sDistStations (distance between stations).~~
- ~~Class BookingClasses to have load factors.~~
- ~~Class Booking to have sBaseFarePerKM, sBookings (list of bookings done), sBookingPNRSerial (next available PNR), sACSurcharge, and sLuxuryTaxPercent~~
- You may add any `static` to a class as you need for implementation.

3.6 *Errors & Exceptions

The design should take care of extensive validations for data and consistency of business logic. In this regard the following points may be noted:

- Date validations should include (may have more):
  - All dates should be valid. For example, `29/02/2021` or `31/04/2020` 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*.
  - *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 *booking-Category* 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 flow from the exception flow.
  - 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**.

- ~~There is no error in input, processing, or output.~~

- ~~No error or exception handling to be incorporated in the design for this assignment. However, structure the code flow well so that they can be incorporated later with minimal changes (adhering to the need of flexibility).~~

4 Low-Level Design

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

4.1 Design Principles

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

- *Encapsulation*
  - Maximize encapsulation for every class
  - Use `private` access specifier for all data members that are not needed by derived classes, if any. Use `protected` otherwise.
  - Use `public` access specifier 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.

4.2 Design of Classes, Data Members & Methods

This is left as an exercise in the assignment. Design based on the HLD and the principles and document well.

5 Implementation

After completing the LLD, we perform the coding (implementation). In this we adhered to a set of basic guidelines and code organization.

5.1 Basic Coding Guidelines

An indicative set of guidelines are listed in Section A. You may add more on your own.

5.2 *Code Organization

~~Ideally, the definition of every class (or hierarchy) should be put in a corresponding `.h` file with the `static` definitions and method implementations in the respective `.cpp`. The application should be in `Application.cpp` file. However, for simplicity, it would be acceptable if all the codes are put in the `Application.cpp` file with the application.~~

The code should be properly organized according to the following guidelines:

1. Every major class <`classname`>, and the hierarchy related to it, should be written in header file `classname.h`. Small and frequently invoked methods should also be inlined in this file.
2. The source or implementation file `classname.cpp` will contain the static definitions and remaining methods.
3. The final application would be written in `Application.cpp` (or some such file - to be documented in README). There may be more than one application file for testing.

   Positive and negative test files may be separate.

6 Test Plan

We also need to prepare a test plan to test the implementation at different stages of development so that better quality and productivity can be ensured. Variety of test processes are common. We shall follow two of these in the current assignment.

6.1 Unit Tests

This is typically the basic test process which is engaged during development (however, it may be useful for future testing and debugging as well). In this, we test every class as it is implemented. We test all non-static & static member functions and friend functions. For a class hierarchy, the unit test is done typically at both concrete classes and the overall hierarchy levels specifically checking the polymorphic methods.

~~For the purpose of understanding, in Section **??** we illustrate the test plan and test function for a few unit cases for the `Fraction` class we have developed in Assignment 2.~~

6.2 Application Test

After the units have been tested, we integrate them into the application and test various scenarios for the application. ~~A sample test application was provided for the `Fraction` class in Assignment 2. However, since it was just a single class application, the application code looked pretty much like the unit test application code with the exception of the comparison with golden data.~~

Like the units, we again need to enumerate scenarios for the application in the test plan and write the application test.

~~In addition, a sample test application for booking is given in Section **??** with the expected output in Section **??**. Your codes should pass this test application too.~~

7 Tasks

7.1 Assignment 4: UML Diagrams: Analysis & Design Phases

The following tasks are to be completed for the assignment:

1. **UML Diagrams**: Study the specifications of the booking system, and the analyses & design as discussed above to prepare the final analysis cum design document `UML.pdf` with the following *Diagrams*:
   - *Use Case Diagram*: Identify the actors, use-cases & relationships.
   - *Class Diagram*: Show the classes (with properties & operations), relationships & associations.
   - *Sequence Diagram*: Depict the lifelines, messages & interaction fragments.
   - *Communication Diagram*: Depict the frames, lifelines & messages.
   - *Activity Diagram*: Model the activities, edges, controls, objects & actions.
   - *State Chart Diagram*: Appropriately define the state machine for the problem.
   - *Note*:
     - *Use annotations liberally in the diagrams as needed. Add more notes in the document to explain the diagrams as appropriate.*
     - *Some UML tools like Visual Paradigm Online may used to generate the diagrams*
     - *You need to model handling of errors wherever needed, however, do not model exceptions as these are specific to C++.*
2. **Analysis & Design**: Complete the HLD and the LLD in C++. Document the salient points from your design in `Design.txt` / `Design.pdf`. Follow the quality guidelines and design principles outlined above.
   - *Note*:
     - *Necessary parts of HLD and all of LLD should be for C++.*
     - *Design of every class (and hierarchy) should detail the non-static & static data members, non-static & static methods with const-ness and polymorphism as applicable.*
     - *Details of methods (algorithm) may be skipped - only interface would suffice.*
     - *Exceptions should be modeled.*
3. **Bundle and Submissions**: Name and bundle your files as given in Section 8 and submit to Moodle.

7.2 Assignment 5: Test Plan, Implementation & Test Report

The following tasks are to be completed for the assignment:

1. **Implementation**: Implement the LLD in C++ following the basic coding guidelines (Section A).
2. **Test Planning**: Write a unit test and application test plan in `Testplan.txt` covering all scenarios. Also, write the test suite with a couple of test cases for every scenario and golden output. **Note that all wrong input or erroneous data situations are to be handled. Hence, the test plan and suite must cover positive and negative tests both.**
3. **Testing & Test Report**: Implement unit test and application test codes and perform testing. Based on the test plan and suite, generate the PASS/FAIL report.
4. **Bundle and Submissions**: Name and bundle your files as given in Section 8 and submit to Moodle.

8 Submission of Files

8.1 Assignment 4: UML Diagrams: Analysis & Design Phases

The following files must be submitted as a single ZIP file:

1. `UML.pdf`: Use Case, Class and Sequence, Communication, Activity & State Machine Diagrams, annotations, notes.
2. `Design.txt` / `Design.pdf`: The design document stating the design details (especially LLD) with principles and guidelines followed

Every file must have your name and roll number.

8.2 Assignment 5: Test Plan, Implementation & Test Report

The following files must be submitted as a single ZIP file:

1. `Documents.zip`
   (a) `Testplan.txt` / `Testplan.pdf`: The test plan document stating scenarios for unit tests and of the test application, and test cases (with golden output).
   (b) `Testreport.txt` / `Testreport.pdf`: The test report document based on the test plan and suite showing PASS / FAIL of test cases.
2. `Source.zip`:
   (a) Source (`.cpp`) and header (`.h`) files for classes.
   (b) Source (`.cpp`) and header (`.h`) files for test applications.
   (c) `README` file that describes the contents of every file in the `Source.zip`. Also, mention the compiler (with version, and compiler options, if any) that you have used.
3. `Outputs.zip`
   (a) Output from the your test application developed from the test plans
      - The output file can be generated by redirecting the output to a text file or by copy-paste from the console in a text file.
      - There is no need to include the `a.out` file.

   (b) Both positive as well as negative outputs must be shown.

Every file (with the exception of program output) must have your name and roll number.

9 Marks

9.1 Assignment 4: UML Diagrams: Analysis & Design Phases

The marks are distributed as follows:

UML		**[60]**
Breakup		
Use Case Diagram	*[5]*	
Class Diagram	*[20]*	
Sequence Diagram	*[10]*	
Communication Diagram	*[5]*	
Activity Diagram	*[15]*	
State Machine Diagram	*[5]*	
Design		**[40]**
Breakup		
Design of Station & Railways Classes	*[4]*	
Design of Date Class	*[4]*	
Design of BookingClass Class & Hierarchy	*[5]*	
Design of Divyaang Class & Hierarchy	*[3]*	
Design of Concessions Class & Hierarchy	*[3]*	
Design of BookingCategory Class & Hierarchy	*[5]*	
Design of Passenger Class & Hierarchy	*[5]*	
Design of Booking Class & Hierarchy	*[8]*	
Design of Exceptions Class & Hierarchy	*[3]*	

9.2 Assignment 5: Test Plan, Implementation & Test Report

The marks are distributed as follows:

Implementation		**[40]**
Breakup		
Happy Paths	*[25]*	
Exceptional Paths	*[15]*	
Test Planning		**[30]**
Breakup		
Unit Test Scenarios	*[10]*	
Application Test Scenarios	*[5]*	
Test Cases & Goldens	*[15]*	
Testing & Test Report		**[10]**
Breakup		
Unit Test Report	*[7]*	
Application Test Report	*[3]*	
Quality of Design & Implementation		**[20]**
Breakup		
Adherence to Design Protocols		
Singletons	*[2]*	
const-ness	*[2]*	
Sub-Typing	*[6]*	
Coding Guidelines	*[5]*	
Code Comments	*[5]*	

A Coding Guidelines

It is advised to follow the guidelines below while coding:

- 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`)
- No constant value should be written within the code - should be put in the application as static
- Prefer to pass parameters by value for build-in type and by const reference for UDT
- 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)`)
- The project should compile without any compiler warning
- Indent code properly
- Comment the code liberally and meaningfully
- Adopt more guidelines as you prefer. Try to document them