

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 Reports

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 format¹ 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**

¹Choice 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 [BookingCategories](#) allow for priority booking (called *Tatkal*²) on a higher *Tatkal* fare depending on the [BookingClass](#) of travel and fare as given in the *Tatkal 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 [BookingCategories](#) 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 [BookingCategories](#) allow for priority (*Tatkal*) booking on a higher *Tatkal* fare depending on the [BookingClass](#) of travel and fare as given in the *Tatkal 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 = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *AC3Tier* = $\text{Rs. } 723.50 * 2.50 = \text{Rs. } 1808.75$
- After adding the reservation charge, we get $\text{Rs. } 1808.75 + \text{Rs. } 40.00 = \text{Rs. } 1848.75 \approx \text{Rs. } 1849/=$ (rounded)

By *ACFirstClass*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *ACFirstClass* = $\text{Rs. } 723.50 * 6.50 = \text{Rs. } 4702.75$
- After adding the reservation charge, we get $\text{Rs. } 4702.75 + \text{Rs. } 60.00 = \text{Rs. } 4762.75 \approx \text{Rs. } 4763/=$ (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 = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *AC3Tier* = $\text{Rs. } 723.50 * 2.50 = \text{Rs. } 1808.75$
- Concession fare = $\text{Rs. } 1808.75 * (1.00 - 0.40) = \text{Rs. } 1085.25$
- After adding the reservation charge, we get $\text{Rs. } 1085.25 + \text{Rs. } 40.00 = \text{Rs. } 1125.25 \approx \text{Rs. } 1125/=$ (rounded)

By *ACFirstClass* for *Female*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *ACFirstClass* = $\text{Rs. } 723.50 * 6.50 = \text{Rs. } 4702.75$
- Concession fare = $\text{Rs. } 4702.75 * (1.00 - 0.50) = \text{Rs. } 2351.375$
- After adding the reservation charge, we get $\text{Rs. } 2351.375 + \text{Rs. } 60.00 = \text{Rs. } 2411.375 \approx \text{Rs. } 2411/=$ (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 = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *AC3Tier* = $\text{Rs. } 723.50 * 2.50 = \text{Rs. } 1808.75$
- Concession fare = $\text{Rs. } 1808.75 * (1.00 - 0.75) = \text{Rs. } 452.1875$

- After adding the reservation charge, we get $\text{Rs. } 452.1875 + \text{Rs. } 40.00 = \text{Rs. } 492.1875 \approx \text{Rs. } 492/=$ (rounded)

By *ACFirstClass* for *Cancer Patient*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *ACFirstClass* = $\text{Rs. } 723.50 * 6.50 = \text{Rs. } 4702.75$
- Concession fare = $\text{Rs. } 4702.75 * (1.00 - 0.50) = \text{Rs. } 2351.375$
- After adding the reservation charge, we get $\text{Rs. } 2351.375 + \text{Rs. } 60.00 = \text{Rs. } 2411.375 \approx \text{Rs. } 2411/=$ (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 = $1447\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 723.50$
- Loaded fare for *AC3Tier* = $\text{Rs. } 723.50 * 2.50 = \text{Rs. } 1808.75$. This is the basic fare to be used computing tatkal charge
- 30% of basic fare = $\text{Rs. } 1808.75 * 0.30 = \text{Rs. } 542.625$
- Tatkal charge = $\text{Rs. } 400.00$ (by maximum cap)
- After adding the reservation charge, we get $\text{Rs. } 1808.75 + \text{Rs. } 400.00 + \text{Rs. } 40.00 = \text{Rs. } 2248.75 \approx \text{Rs. } 2249/=$ (rounded)

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

By *ACFirstClass*:

- Distance from Chennai to Bangalore = 350km
- Base fare = $350\text{km} * \text{Rs. } 0.5 / \text{km} = \text{Rs. } 175.00$
- Loaded fare for *ACFirstClass* = $\text{Rs. } 175.00 * 6.50 = \text{Rs. } 1137.50$
- 30% of basic fare = $\text{Rs. } 1137.50 * 0.30 = \text{Rs. } 341.25$
- Tatkal charge = $\text{Rs. } 0.00$ (by minimum distance cap)
- After adding the reservation charge, we get $\text{Rs. } 1137.50 + \text{Rs. } 0.00 + \text{Rs. } 60.00 = \text{Rs. } 1197.50 \approx \text{Rs. } 1198/=$ (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 *not* 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:

<i>Station Distance Matrix</i>					
From Station	To Station				
	Mumbai	Delhi	Bangalore	Kolkata	Chennai
	<i>Distance in KM</i>				
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:

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

- 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
<i>ACFirstClass</i>	0.50	0.50	0.50	0.00
<i>ExecutiveChairCar</i>	0.75	0.75	0.75	0.00
<i>AC2Tier</i>	0.50	0.50	0.50	0.00
<i>FirstClass</i>	0.75	0.75	0.75	0.75
<i>AC3Tier</i>	0.75	0.75	1.00	0.00
<i>ACChairCar</i>	0.75	0.75	1.00	0.00
<i>Sleeper</i>	0.75	0.75	1.00	0.75
<i>SecondSitting</i>	0.75	0.75	1.00	0.75

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

Tatkal Charges Matrix

Booking Class	Minimum Tatkal Charges (in Rs.)	Maximum Tatkal Charges (in Rs.)	Minimum Distance for charge (in Km)
<i>ACFirstClass</i>	400.00	500.00	500
<i>ExecutiveChairCar</i>	400.00	500.00	250
<i>AC2Tier</i>	400.00	500.00	500
<i>FirstClass</i>	400.00	500.00	500
<i>AC3Tier</i>	300.00	400.00	500
<i>ACChairCar</i>	125.00	225.00	250
<i>Sleeper</i>	100.00	200.00	500
<i>SecondSitting</i>	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 uniformity 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 reusable.

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 **BookingClasses** 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 to 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 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 *Tatkal 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 **BookingClasses**, 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**, **BookingClass**, 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 polymorphic 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 normalization³. 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 [BookingCategories](#), a [Concessions](#) hierarchy may be created. For this, we note that concession is dependent on [BookingCategory](#), [BookingClass](#), and *gender, age & ability type (divyaang-jan)* 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 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.

³Normalization 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 hierarchy 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 templated 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 [BookingCategory](#)s - 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 be workarounds 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 **Stations** 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 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]
	<i>Breakup</i>	
Use Case Diagram		[5]
Class Diagram		[20]
Sequence Diagram		[10]
Communication Diagram		[5]
Activity Diagram		[15]
State Machine Diagram		[5]
Design		[40]
	<i>Breakup</i>	
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]
	<i>Breakup</i>	
Happy Paths		[25]
Exceptional Paths		[15]
Test Planning		[30]
	<i>Breakup</i>	
Unit Test Scenarios		[10]
Application Test Scenarios		[5]
Test Cases & Goldens		[15]
Testing & Test Report		[10]
	<i>Breakup</i>	
Unit Test Report		[7]
Application Test Report		[3]
Quality of Design & Implementation		[20]
	<i>Breakup</i>	
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

B Clarifications

It may be noted that analyses (Section 2) and designs (Section 3, and Section 4) as discussed are indicative. These are provided to get you started and to show how the specification may map to the design styles as we have discussed in C++. However, they are not binding. Many of you are coming up with alternate design proposals and asking for approval. It is okay that you make your own choice if you feel more comfortable to think in that manner. Just explain the choice clearly in the design document.

B.1 Queries

B.1.1 Deep Majumder

I have some doubts regarding the new Assignment.

1. It has been mentioned in the assignment that `BookingClass` can be reduced to a flat hierarchy. But what is the necessity of an explicit hierarchy at all? Combining anonymous namespaces and Factory patterns gives us a way to have Singletons for each booking class. Moreover, this falls in nicely with a Singleton Config manager, which contains master data read from a file. By not having a explicit (flat) hierarchy, we can get rid of a lot of code, possibly even duplicated ones. And also relieves us of explicitly initializing a lot of static constant members ($7 * 8 = 56$ in for Booking Classes). So is this an acceptable way to model `BookingClass`? Or is the hierarchy necessary?

PPD: Hierarchy is not a must, just an advise. Since this will need to be done for other class hierarchies, `BookingClass` may also be fit into the same template style.

Further, I wanted people to first master writing templates, and on hierarchy, without using other existing solution. Without a lot of discussions, it would not be easy for most to handle the solution in the way you are suggesting. However, you are free to use any style you prefer.

Finally, I have two observations on your comments - (1) With inclusion-parametric flat hierarchy, I do not think code bloats or there is need to write duplicate codes. I tried that out. Of course, I would be interested to check your solution for better options. (2) I am not sure how we can avoid the initialization of the constants. They need to be put anyway. May be I am missing something. Will look forward to what you do.

2. It has been said that the master data can be read from input files and need not be hardcoded. I plan to use some structured format like YAML, or JSON, or XML. Do I need to provide a schema as well or can I implicitly assume the data format?

PPD: This is optional. You may read it from a file if you want. You can choose any schema or your own format if you decide to do it.

3. Reading master data from a file I think introduces a problem with Singletons. In C++, all static member initialization is done before main is executed. As a result, the config file location cannot be passed from main (command line args or otherwise). So it must be hardcoded. But since C++ doesn't guarantee the order in which static variables are initialized, the filename cannot be stored in a string (or even C-string), because we do not know whether it is initialized or uninitialized (we will have an unreliable design). So the filename must be hardcoded as a string literal. To avoid repeating we have to use a macro (red-flags, I know). Is there a better way out?

PPD: Very interesting question and observation. You have rightly pointed out a tricky situation in terms of global static objects and their initialization (one of the reasons I prefer local static objects whenever possible).

*The problem of passing the file name can be addressed if we assume to read it from the `stdin` which can work even before `main()` starts (don't ask how `stdin` gets initialized and when). This reading of the file name must happen before the first initialization data is needed (and before `main()` starts). That can be done (check **Approach 2** below).*

In fact, besides the input file name, there is a deeper problem. The file is a serial structure. It will contain the initialization values in a certain order. And the global static values will be initialized in an unknown order, requiring values coming from the input file. How do we guarantee their alignment?

Here, I shall present three approaches - based on the following three philosophy:

- **Approach 1:** Do not initialize. Set to default. Set values after `main` starts.
- **Approach 2:** Have a singleton to collect all values from input file (this can be done before the first initialization). Then the accumulated values are set for initialization in whatever order they happen.
- **Approach 3:** Do not use global static data member. Use only local static data with static methods. Think, as if, every static data member is a singleton.

Of course, there could be more. And none of them is perfect. Let me now explain.

Approach 1: The singletons in the project can use the Meyer's version where they would be local static objects and hence will be constructed only on first use (naturally after `main()` starts), not before `main()`. So we do not need to be concerned about them.

The static data in this project are all of built-in type (having no constructor). So they can be defaulted to 0 (for `int`) or 0.0 (for `double`) or "" (for `string`) or `true` (for `bool`) in the code. Of course, this leads to a problem - the static objects cannot be `const` any more. That may not be a major issue because all these are `private` in the respective classes, but raises the question of how does `main()` can access and set them with the values read from the file. The solution would be to have a `InitStatic` class which does the job and can be `friend` of all necessary classes. Quite a mess.

Also note that this approach will not work if the static data member is a UDT and does not have a default constructor.

Approach 2: If you abhor the rampant use of `friend` (I do), we can follow a different scheme. We can have a singleton `SetStatic` class which reads the static values from the input file during its construction and puts in its data members. So all static initialization will get the value through this singleton object which actually reads the data from the input file. This solves the problem that static data can be initialized in arbitrary order. No matter which order it happens, the first access to static will create this singleton and read the file. A sample code files is given below.

There are four files:

- **MyClass.h:** This has an abstract base class `MyClass` with two sub-classes `MyClass::MyType1` and `MyClass::MyType2` in the emulation of a minimal inclusion-parametric polymorphism (that we are widely using in the design). Each sub-type has four static data members of `int`, `double`, `string` and `bool` types to show how to set their initial values. This is for the purpose of illustration.
- **MyClass.cpp:** This file defines the static data members of both sub-types. There is a commented code where these are initialized directly. In the other part they are emulated to have been read from the input file using the `SetStatics` class.

- **SetStatics.h:** *This is singleton class which is designed with data members as proxy of the static data members of each sub-type above. These are named explicitly for identification without class scopes (this is not good, but works - think about a better solution). The data members are non-const as they have to be set from reading the file. They are made private and endowed with corresponding Get methods (this actually is redundant as these values would not be used after the initialization - still following the usual OOP paradigm). Note that actual reading from file is not shown for simplicity.*
- **App.cpp:** *This illustrates the use of the sub-type objects with dynamic dispatch.*

```
// MyClass.h

#ifndef __MY_CLASS_H
#define __MY_CLASS_H

// ***** MyClass class & hierarchy definition
// ***** Author: P P Das
// ***** Date: 23-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

// ***** C++ Standard Library Headers
#include <iostream>
#include <string>
using namespace std;

// Forward declaration of templated class
template<class T>
class MyClassType;

// Generic BookingCategory type
class MyClass { // Abstract Base Class
    // Tag types
    struct Type1 { };
    struct Type2 { };

protected:
    MyClass() { }
    virtual ~MyClass() = 0;

public:
    virtual void Print() const = 0;

    // Enumerated types
    typedef MyClassType<Type1> MyType1;
    typedef MyClassType<Type2> MyType2;
};

inline MyClass::~~MyClass() {}

// Specific MyClass types
template<class T>
class MyClassType : public MyClass {
    static const string sName;           // Name of the person
    static const int sAge;               // Age of the person
    static const double sSalary;        // Salary of the person in Rs. Lakhs
    static const bool sIsMale;          // Gender of the person

    MyClassType() : MyClass() { }
    ~MyClassType() { }

public:
    // Singleton object - placeholder for the respective type
    static const MyClassType& Type() {
```

```

        static MyClassType theObject;

        return theObject;
    }

    void Print() const;
};

// Print methods
inline void MyClass::MyType1::Print() const {
    cout << "Printing a MyType1 Object" << endl;
    cout << "Name = " << sName << endl;
    cout << "Age = " << sAge << " years" << endl;
    cout << "Salary = Rs. " << sSalary << " lakhs" << endl;
    cout << "Gender = " << ((sIsMale)? "Male": "Female") << endl;

    return;
}

inline void MyClass::MyType2::Print() const {
    cout << "Printing a MyType2 Object" << endl;
    cout << "Name = " << sName << endl;
    cout << "Age = " << sAge << " years" << endl;
    cout << "Salary = Rs. " << sSalary << " lakhs" << endl;
    cout << "Gender = " << ((sIsMale) ? "Male" : "Female") << endl;

    return;
}
#endif // __MY_CLASS_H

// MyClass.cpp

// ***** MyClass class & hierarchy definition
// ***** Author: P P Das
// ***** Date: 23-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

// ***** C++ Standard Library Headers
#include <iostream>
#include <string>
using namespace std;

// ***** Project Headers
#include "SetStatics.h"
#include "MyClass.h"

// Model for reading from a file through a singleton
const string MyClass::MyType1::sName = SetStatics::Static().GetMyClass_MyType1_sName();
const int MyClass::MyType1::sAge = SetStatics::Static().GetMyClass_MyType1_sAge();
const double MyClass::MyType1::sSalary = SetStatics::Static().GetMyClass_MyType1_sSalary();
const bool MyClass::MyType1::sIsMale = SetStatics::Static().GetMyClass_MyType1_sIsMale();

const string MyClass::MyType2::sName = SetStatics::Static().GetMyClass_MyType2_sName();
const int MyClass::MyType2::sAge = SetStatics::Static().GetMyClass_MyType2_sAge();
const double MyClass::MyType2::sSalary = SetStatics::Static().GetMyClass_MyType2_sSalary();
const bool MyClass::MyType2::sIsMale = SetStatics::Static().GetMyClass_MyType2_sIsMale();

// Setting static values directly
//const string MyClass::MyType1::sName = "Ravi Narayan";

```

```

//const int MyClass::MyType1::sAge = 40;
//const double MyClass::MyType1::sSalary = 10.75;
//const bool MyClass::MyType1::sIsMale = true;
//
//const string MyClass::MyType2::sName = "Pratibha Kannan";
//const int MyClass::MyType2::sAge = 37;
//const double MyClass::MyType2::sSalary = 15.30;
//const bool MyClass::MyType2::sIsMale = false;

```

```

// SetStatics.h

```

```

#ifndef __SET_STATICS_H
#define __SET_STATICS_H

```

```

// ***** SetStatics class
// ***** Author: P P Das
// ***** Date: 23-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

```

```

// ***** C++ Standard Library Headers
#include <string>
using namespace std;

```

```

// ***** Project Headers
#include "MyClass.h"

```

```

class SetStatics {
    // Proxy data members for static values
    string MyClass_MyType1_sName;
    int MyClass_MyType1_sAge;
    double MyClass_MyType1_sSalary;
    bool MyClass_MyType1_sIsMale;

    string MyClass_MyType2_sName;
    int MyClass_MyType2_sAge;
    double MyClass_MyType2_sSalary;
    bool MyClass_MyType2_sIsMale;

    SetStatics() {
        // Read the input file name from stdin - not shown

        // Open the input file - not shown

        // These values are read from the input file
        // This happens only once
        MyClass_MyType1_sName = "Ravi Narayan";
        MyClass_MyType1_sAge = 40;
        MyClass_MyType1_sSalary = 10.75;
        MyClass_MyType1_sIsMale = true;

        MyClass_MyType2_sName = "Pratibha Kannan";
        MyClass_MyType2_sAge = 37;
        MyClass_MyType2_sSalary = 15.30;
        MyClass_MyType2_sIsMale = false;
    }
}

```

```

public:
    static const SetStatics& Static() {
        static SetStatics theObject;
    }
}

```

```

        return theObject;
    }

    // Get methods for the static values we need to set
    const string& GetMyClass_MyType1_sName() const { return MyClass_MyType1_sName; }
    const int GetMyClass_MyType1_sAge() const { return MyClass_MyType1_sAge; }
    const double GetMyClass_MyType1_sSalary() const { return MyClass_MyType1_sSalary; }
    const bool GetMyClass_MyType1_sIsMale() const { return MyClass_MyType1_sIsMale; }

    const string& GetMyClass_MyType2_sName() const { return MyClass_MyType2_sName; }
    const int GetMyClass_MyType2_sAge() const { return MyClass_MyType2_sAge; }
    const double GetMyClass_MyType2_sSalary() const { return MyClass_MyType2_sSalary; }
    const bool GetMyClass_MyType2_sIsMale() const { return MyClass_MyType2_sIsMale; }
};
#endif // __SET_STATICS_H

// App.cpp

// ***** MyClass Application
// ***** Author: P P Das
// ***** Date: 23-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

// ***** C++ Standard Library Headers
#include <iostream>
#include <string>
using namespace std;

// ***** Project Headers
#include "MyClass.h"

int main() {
    const MyClass& r1 = MyClass::MyType1::Type();
    const MyClass& r2 = MyClass::MyType2::Type();

    cout << endl;
    r1.Print();

    cout << endl;
    r2.Print();

    return 0;
}

```

Approach 3: Another way could be to not use global static data members at all. Instead, we could make all of them as local static data and replace them by global static methods in the respective class. Then their initialization comes under the control of the program after `main()`. So it can be managed in the order the application wants. Of course, that itself, may become a big issue as the file that provides the data is a serial structure and the order in which we want to initialize the objects may not be aligned with that order - or at least be difficult to align. The work around that would be to use separate input file for each class - but then, we would need to read the name of the file and map with the class. That too is not clean.

Now the four files will be as follows:

- **MyClass.h:** Changes static data members from global objects to static methods with local static data. This is given below.

- **MyClass.cpp:** *Definitions of static data members are omitted. This is given below.
This file may be skipped too as MyClass.h has every method as inline and will get compiled as a part of App.cpp*
- **SetStatics.h:** *Not needed.*
- **App.cpp:** *No change except excluding SetStatics.h from inclusion.*

```
// MyClass.h

#ifndef __MY_CLASS_H
#define __MY_CLASS_H

// ***** MyClass class & hierarchy definition
// ***** Author: P P Das
// ***** Date: 23-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

// ***** C++ Standard Library Headers
#include <iostream>
#include <string>
using namespace std;

// Forward declaration of templated class
template<class T>
class MyClassType;

// Generic BookingCategory type
class MyClass { // Abstract Base Class
    // Tag types
    struct Type1 { };
    struct Type2 { };

protected:
    MyClass() { }
    virtual ~MyClass() = 0;

public:
    virtual void Print() const = 0;

    // Enumerated types
    typedef MyClassType<Type1> MyType1;
    typedef MyClassType<Type2> MyType2;
};

inline MyClass::~~MyClass() {}

// Specific MyClass types
template<class T>
class MyClassType : public MyClass {
    MyClassType() : MyClass() { }
    ~MyClassType() { }
public:
    // Singleton object - placeholder for the respective type
    static const MyClassType& Type() {
        static MyClassType theObject;

        return theObject;
    }

    // Name of the person
```

```

        static const string& GetName();

        // Age of the person
        static const int GetAge();

        // Salary of the person in Rs. Lakhs
        static const double GetSalary();

        // Gender of the person
        static const bool IsMale();

        void Print() const;
};

// Name of the person
inline const string& MyClass::MyType1::GetName() {
    static const string sName = "Ravi Narayan"; // Read from file

    return sName;
}

// Age of the person
inline const int MyClass::MyType1::GetAge() {
    static const int sAge = 40; // Read from file

    return sAge;
}

// Salary of the person in Rs. Lakhs
inline const double MyClass::MyType1::GetSalary() {
    static const double sSalary = 10.75; // Read from file

    return sSalary;
}

// Gender of the person
inline const bool MyClass::MyType1::IsMale() {
    static const bool sIsMale = true; // Read from file

    return sIsMale;
}

// Print methods
inline void MyClass::MyType1::Print() const {
    cout << "Printing a MyType1 Object" << endl;
    cout << "Name = " << GetName() << endl;
    cout << "Age = " << GetAge() << " years" << endl;
    cout << "Salary = Rs. " << GetSalary() << " lakhs" << endl;
    cout << "Gender = " << ((IsMale()))? "Male": "Female" << endl;

    return;
}

// Name of the person
inline const string& MyClass::MyType2::GetName() {
    static const string sName = "Pratibha Kannan"; // Read from file

    return sName;
}

// Age of the person

```

```

inline const int MyClass::MyType2::GetAge() {
    static const int sAge = 37; // Read from file

    return sAge;
}

// Salary of the person in Rs. Lakhs
inline const double MyClass::MyType2::GetSalary() {
    static const double sSalary = 15.30; // Read from file

    return sSalary;
}

// Gender of the person
inline const bool MyClass::MyType2::IsMale() {
    static const bool sIsMale = false; // Read from file

    return sIsMale;
}

inline void MyClass::MyType2::Print() const {
    cout << "Printing a MyType2 Object" << endl;
    cout << "Name = " << GetName() << endl;
    cout << "Age = " << GetAge() << " years" << endl;
    cout << "Salary = Rs. " << GetSalary() << " lakhs" << endl;
    cout << "Gender = " << ((IsMale()) ? "Male" : "Female") << endl;

    return;
}
#endif // __MY_CLASS_H

// MyClass.cpp

// ***** Project Headers
#include "MyClass.h"

```

*So to conclude **Approach 2** seems to be the best option till we find a better solution. It is a bit of coding but is clean and less error-prone. However, I am still concerned with the fact that nice scoped names like `MyClass::MyType1::sName` have to be encoded in terms of synthetic names like `MyClass_MyType1_sName` and `GetMyClass_MyType1_sName()` - leaving a huge possibility of manual error. Let us keep looking for a better solution.*

4. Since using double-dispatch for object construction (the "virtual" constructor idiom) tightly couples classes, aren't we better off simply using the Factory pattern? The double dispatch trick works well with Visitors, but I wonder how useful it is for object construction.

***PPD:** You are right. I shall try to discuss this more if I get time to talk on various design patterns. For now, it will be too much to expect this from you all without having discussed them in the class.*

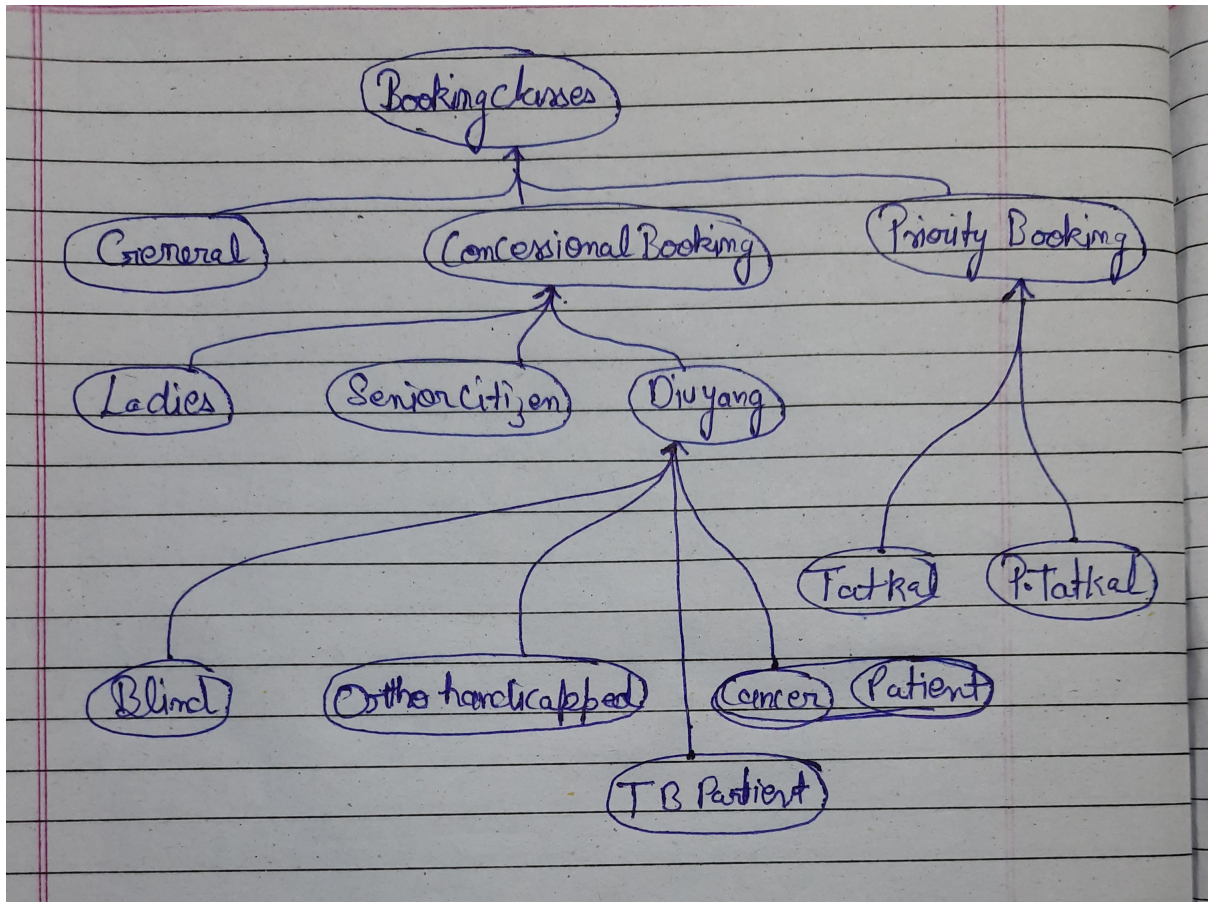
B.1.2 Kunal Singh

1. Sir in assignment 4 in the use case diagram i am only able to identify passenger as an actor and his only use case being initiation of booking which then includes / extends other dependent use cases could you help me out please ?

***PPD:** In Use-case it is not necessary that all actors be modeled in the final software. For example, we can think of the booking executive who does the booking. The same person who does intends to travel as a passenger may be the booking executive. But he is a different actor. Then the booking needs to be printed out. We need printer actor. A database is needed to keep the booking records. A booking manager is an actor who sets the static data like station names and distances. Etc. These are the use-case views. When*

you come to later diagrams in later stages, you may want to merge some of them or skip their explicit existence. Use case is about actors and actions. That must truly visualized.

2. sir I've come up with the following design of booking category actually I got confused with the clarification mail , is this design appropriate ? sorry to bother you again.



PPD: There could be multiple ways to model. And you have your choice. Yes, you may visualize in this way. However, this is the hierarchy for BookingCategory, not for BookingClass (which is information on First Class, AC Tier 2, etc.).

B.1.3 Nakul Aggarwal

1. Page No. 22 (highlighted in yellow) (important)

from station and to station 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 operational device, closely supervising the manual

Does this mean that in all the classes where there is a scope of attempting construction with wrongly formatted or invalid inputs, we should make the constructor private and use a static method at its place in the public? (like we do for singleton classes – there we need to preserve the number of calls to the constructor, here we need to preserve the validity of the inputs passed to the constructor)

PPD: *You are right. We use a static method to check the validity of the input and then construct, and finally return the object. With this, we do not need to throw from a constructor, which may leave us in an inconsistent object state.*

For example, while constructing a Passenger class object, we have to make sure various input validity like dob, aadhaar, mobile etc. hold and constraints on name can be ensured. So we can have a static method CreatePassenger that does the checking, throws appropriately on errors, and then constructor the Passenger object if everything is fine. In some case, like for Railways, this may get a little tricky because you have to decide on the interplay between the singleton static method and the value checking static method. Note that for the singleton static method, it is okay to throw an exception because it is not a constructor.

You would not do such idioms for constructing objects for classes where no input data validation is necessary, like BookingClass or Concessions. They depend on pre-validated static data only.

2. Relevance of Virtual Construction Idiom in Ad-Hoc Polymorphic Hierarchy Sir, the Concessions class that is rooted at the BookingCategory class realises ad-hoc polymorphism. In this case, how can it perform dynamic dispatch that is required in the "virtual construction"? Secondly, I am confused with the proposed model of BookingCategory hierarchy. For Concessions hierarchy ad-hoc polymorphism is used. While for Divyaang and PriorityCategories hierarchies, inclusion-parametric polymorphism is used. Both the hierarchies are rooted at BookingCategory class (that might be abstract). Does this mean that among some of the derived classes we are allowing dynamic dispatch and among others we are not? Why can't we use inclusion polymorphism at the place of ad-hoc? Will virtual construction idiom work with such a "half ad-hoc half inclusion-parametric hierarchy"?

PPD: *You are getting confused between three things - parametric polymorphism, inclusion polymorphism and virtual construction idiom. Let us look at them one by one.*

First, there are a number of hierarchies here - each rooted at its respective root class (which may be abstract). These include BookingClass, Concessions etc. They are rooted separately - NOT at BookingCategory.

Second, let us see when parametric polymorphism is useful. Using templates, we write a generic code, that instantiates to a number of specific codes based on the type. So the methods of these instantiated classes need to be "parameterizable" - by the values they use and the types they engage. Simply widely different algorithms for different overloads (overrides) of a method would not be a good candidate for parametric polymorphism. Rather, we should use inclusion polymorphism and code every algorithm for every class separately. This is what Concessions warrant because in some cases we do not give concession at all, in some we depend on gender or gender & age, and in others we use the disability information. They are not parametrically combinable. Typically, when we have a flat one level hierarchy, we can use parametric polymorphism for the leaf classes and an abstract base class to provide a uniform interface with dynamic dispatch. That is the case for many class hierarchies here.

Finally, virtual construction becomes necessary when we the particular type of object that we have to construct for a hierarchy (PoolSlot in Section 3.2.2) depends on the type of another object from a different hierarchy (Gender). In our specific context, Booking would be a hierarchy because the fare computation logic depends on the BookingCategory. How do we create an appropriate Booking object then given a specific BookingCategory (and BookingClass)? I hope this clarifies.

3. UML Diagrams Is it totally upto us to decide the level of abstraction of the diagrams? We have seen many class diagrams in the class that have no attributes/operations written in them; only generalization, aggregation etc relationships are used to design these diagrams. So is it fine if we omit some attributes/operations? I am asking this because here we have a lot of classes; this might clutter the diagram a lot if attributes and operations are written for each one of them.

PPD: *Of course, it is your choice. However, it is necessary to choose appropriate abstraction (not only limited to properties and methods, there are associations, relationships, message for Sequence etc.) so that all design considerations and details are available "only" from the diagrams. The class diagram in the solution of Assignment 3 could serve as a sample.*

We have shown several examples to handle clutter. In one diagram a class may be detailed, in others it may just come in by name. You need to make an appropriate choice.

B.1.4 Parth Jindal

1. This is in regards to the 4th assignment for CS21006. In the Error and Exceptions, It was written in the Passenger section that "gender must be male or female. It must be valid by input" Does this imply that it is assumed that the input is given by the user from a 'drop-down menu' hence there may not be any need to validate it just like for BookingClass and BookingCategory (by validation I mean if "it is out of category") or we should still validate it.

PPD: You are right. The passenger chooses the Gender from a drop-down (like she / he does for BookingClass or BookingCategory). That process can be emulated by our system by using appropriate Gender or other types in the request. Naturally, these inputs do not need validation - in a GUI system no invalid input can be given for these, and in our system the code will not compile if we provide a wrong type. So, for example, we cannot allow to input a string like "male" for Gender because it is always possible to misspell it as "meal" and the compiler will not give an error (but our runtime system will fail). Hence, it is important these as type constant as explained above.

2. Another question I had in mind was the role of disabilityID in determining the eligibility for Divyaang BookingCategory booking and its validation?

PPD: disabilityID does not need validation because it can come from various agencies and is not regulated to be in a certain format (like aadhaar is). It is more a representative for a certificate that the passenger would need to upload for a GUI based system.

The type of Divyaang is assumed to be from a drop-down in a GUI and will be emulated by type constant so that it cannot be wrong. The BookingCategory details will be validated against it and the fare will be computed by the corresponding business logic. So, Divyaang category of booking can be allowed for Divyaang type passenger only (and the fare will be based on the type of disability and the BookingClass), while General or other category of booking will be allowed for a Divyaang passenger if she / he has sought for it and qualifies.

B.1.5 Rohan Raj

1. In passenger data type, there are many optional attributes. Creating different functions for all of the cases would be time consuming and not that much important. I have an alternative. If A person don't have a first name, we can pass empty string as the argument. Same goes for all the optional arguments. Can we do that?

PPD: It would be a design choice for you.

2. While validating passenger object, do we have to consider date of reservation to calculate the age or the date extracted from the library(current date) ?

PPD: You need both - the date on which the booking is done (today from the library) and the date of travel. There are validations on both.

B.1.6 Shrinivas Khiste

1. I am having trouble understanding how to implement ad-hoc polymorphism. On the internet, it says that it involves function overloading. Then how is it defined for a class and have a hierarchy? If we use the function overloading definition then we can define different functions to calculate concessions for each type. But how do we implement it for the Exception Classes?

PPD: Ad-hoc polymorphism is function overloading. Correct. It may be between global functions, member functions of the same class, or member function of one class and its parent in a hierarchy. It is the simple static form of polymorphism and must differ in the signature between two polymorphic forms. We have discussed all these cases in the class.

While ad-hoc polymorphism is possible on an inheritance hierarchy, most often it is not interesting and / or useful. What we need there is inclusion polymorphism or function overriding where a member function of a class is redefined with a different behavior from its parent class. This gets useful specially when the inheritance is truly polymorphic (that is, uses pure or non-pure virtual functions). This has also been discussed in depth in the class. You would recall how we used inclusion polymorphism for the draw() method over a hierarchy of shapes. You could use something similar for computing concessions and / or fare in different cases.

Exception classes would usually use static hierarchy because the catch clauses catch the exception by explicit type matching with hierarchical generalization. So using static overriding is the way to go.

B.1.7 Soumita Hait

1. Should constructors and destructor be included in the class diagram?

PPD: *It is optional, because it is mandatory for all classes.*

2. Are the concession factors fixed or they can vary with time?

PPD: *They can change with time.*

3. For the Master Data, it is given in the assignment that "These should be hard-code inside the implementation of functions / methods." I cannot understand how to hard-code inside functions. Does it mean that I need to create static functions for setting/initializing the values of the various parameters?

PPD: *They should **not** be. It is a typo.*

4. It is mentioned in the assignment that "If there are more than one validation failures, the system should attempt to report as many of them as possible in a single run." Sir, if an exception arises, it is thrown and detected by the corresponding catch block, after which the program is terminated. Then, how to detect more than one error in a single run? It is also given that we need to classify the exceptions using a hierarchy of Exceptions classes.

PPD: *Good question. Please note that raising an exception does not mean termination. If that is to be done, we can simply call `exit(1)`. The whole idea of the exception is to raise-catch-handle and then proceed. That is get the system back to a consistent state. Also, if we are using exceptions, it does not mean that we cannot use other error trapping techniques, like flags, in sync with it. So an appropriate design for it is required.*

That exceptions have a hierarchy, they are layered, are for advantages of deciding which kind of exception to catch where. Like the hierarchy of exceptions, the catch clauses should also be layered on the flow.

Exceptional design is not an easy task. It is good to think about handling such issues. Of course, as a user, if there are three independent errors in the input, you would not like to run the software thrice to get them. You would like to get them all together, fix and then go ahead. We need to design keeping that in mind.

B.1.8 Suhas Jain

1. In section 1.3 Business Logic it is written that the tatkal charges are calculated on the base fare and then the load factor is multiplied to it. But in example 1.3.4 in the calculation, the tatkal charge is calculated after the base fare has been multiplied with the load factor. Which one should we follow?

PPD: *Follow the illustrative computations as shown in Sections 1.3.1, 1.3.2, 1.3.3, and 1.3.4.*

2. In the Concessions class when we talk about a flat single level ad-hoc polymorphism, how are we supposed to design this? is it like we make a Divyaang class inherit from Concessions or we make separate classes for DivyaangBlind, DivyaangCancer ...etc.?

PPD: *You can decide how you would design it. It is a matter of choice. I would rather perceive Concession and Divyaang as separate hierarchies because they are different concepts.*

Of course, Concession class would need a kind of DivyaangConcession subclass where BookingClass and Divyaang information is mapped to the concession percentage.

3. In the example of designing gender class by parametric polymorphism, why can't we model name of the gender just like we did with salutation, why are we making a separate name_ attribute in the gender base class?

PPD: *You can. The `name_` in the gender class is the name of the gender. Salutation can be mapped from this in the output handler. After all, we use male & female to talk about gender and Mr. & Ms. for salutation of the person. They are different concepts.*

B.1.9 Yashica Patodia

1. I have a doubt if multiple booking categories are allowed for the same passenger? Like if a senior citizen can avail tatkal benefits and similar multiple other intersections

PPD: *No. Not in the same booking. Only one category has to be chosen for a ticket. So it is either senior citizen or tatkal.*

Of course, the same person can do different tickets in different categories.