**********

**Name: Hritaban Ghosh**
**Roll Number: 19CS30053**

**********

# Low-Level Design

## Design Principles:

### 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.

By Hritaban Ghosh 19CS30053

# Design of Classes, Data Members and Methods:

## Class Station

This class is used to encapsulate a station. It stores it's name.

**Friend Functions:**
1) ostream& operator<<(ostream&, const Station&)
- The insertion operator is overloaded as a friend function and it prints out the name of the Station.
- It takes a output streaming operator by reference and returns the same by reference in order to support chaining. It also takes a Station object by const reference in order to access it's data members and print to the console.

**Data members**

**data fields:**
(All data fields are private and are only accessible to the class. They are not needed to be accessible to other classes and encapsulation is intensified.)
1) string name_ (stores the name of the station), for example "Mumbai".

**Methods**

**private member methods:**
1) Constructor:
- The Constructor of this class takes one argument which is the name of the Station and initializes name_ in the initializer list of the Constructor.
- **The body of the Constructor is empty.**

**public member methods:**
1) static createStation(std::string) :
- This method takes one argument which is the name of the Station and checks whether the string is empty or not, before using the constructor to construct the Station Object. By default the customer cannot select a non-existent Station.
- This is a static method which checks for validity of the input before calling the constructor of the Station Class for object creation.
- This method will be used to throw exceptions if the input is invalid. This is done to avoid throwing exceptions from the constructor which leads to undefined state.
- This is a static method and belongs to the class.

2) Destructor
- The Destructor of Station Class is a default destructor.

3) string GetName() const:
- This member function returns the name of the station.
- This function is qualified as constant and is not allowed to change any data members of the object.

P.T.O.

4) double GetDistance(const Station&) const:
- This method returns the distance between two Stations as per the Distance Matrix initialized in Railways.
- This method takes one of the Station as an argument passed by const reference and the other is implicitly passed to the method as this pointer.
- This method is qualified as constant and is not allowed to change any data members of the object.
- It uses the double Railways::GetDistance(const Station&, const Station&) const to compute the distance between the two stations and return the same.

5) bool operator==(const Station&) const:
- The equality operator is overloaded as a member function-Returns true if the names of the Stations are the same else false
- This function is qualified as constant and is not allowed to change any data members of the object.
- This method takes a Station object by const reference as an argument and the current object is implicitly passed as this pointer.

6) bool operator!=(const Station&) const:
- The inequality operator is overloaded as a member function-Returns true if the names of the Stations are different else false
- This method takes a Station object by const reference as an argument and the current object is implicitly passed as this pointer.
- This function is qualified as constant and is not allowed to change any data members of the object.

## Class Railways

This class is a singleton class and is used to encapsulate the list of stations and the distance matrix of the Railways system.

**Friend Functions:**

1) ostream& operator<<(ostream&, const Railways&)
- The insertion operator is overloaded as a friend function and it prints out the all the information of the Indian Railways System to the console. (The List of Stations and the Distance Matrix)
- It takes a output streaming operator by reference and returns the same by reference in order to support chaining. It also takes a Railways object by const reference in order to access it's data and print to the console.

**Static Data Members:**

1) static const std::vector<Station> sStations
- Collection of Stations
- It is qualified as const as none of the data can be changed after initialization.
- This static constant is public. It is made public because testing applications and other units of the program must be able to view the Stations accessible through this Railways system.

2) static const std::vector<std::vector <double>> sDistances
- Distance Matrix to get distance between any two stations
- It is qualified as const as none of the data can be changed after initialization.

- This static constant is private and is only accessible to the class. They are not needed to be accessible to other classes and encapsulation is intensified.
- Information from this matrix can be obtained through the GetDistance public member method.

**Methods**

**Static member methods:**
1) static const Railways& IndianRailways()
- Singleton instance of the class Railways called IndianRailways
- This method is public and is used to create an instance of the class statically and return the same. This means that it creates a single instance of the class which is used throughout the program.
- The constructor is made private and thus the only way to access an instance of this class is through this method.

**private member methods:**
1) Constructor :
- The Constructor of this class is declared private in order to make it inaccessible. An instance of this class can only be created using the static member method IndianRailways.
- The body of the Constructor is empty.

2) Destructor
- The Destructor of Railways Class is a default destructor.

**public member methods:**
1) double GetDistance(const Station&, const Station&) const
- It returns the distance between two stations passed to it by const reference. It returns a double value.
- It uses the list of stations and the matrix to access the required distance and return the same.
- This function is qualified as constant and is not allowed to inadvertently change any data members of this singleton object.

## Class Date

This class is used to encapsulate date (dd/MMM/yy). It stores the date, month and year of the the date.

**Friend Functions:**
1) ostream& operator<<(ostream&, const Date&)
- The insertion operator is overloaded as a friend function and it prints out the date to the console in a dd/MMM/yyy format.
- It takes a output streaming operator by reference and returns the same by reference in order to support chaining. It also takes a Date object by const reference in order to access it's data members and print to the console in  dd/MMM/yyy format.

**Static Data Members:**

(All static constants are private and are only accessible to the class. They are not needed to be accessible to other classes and encapsulation is intensified.)
1) const vector of strings of monthNames (vector containing names of the months in a year)
2) const vector of strings of dayNames (vector containing names of the days in a week)
3) const int MAX_VALID_YR = 2099 (const value of year used to check validity of the date)
4) const int MIN_VALID_YR = 1900 (const value of year used to check validity of the date)

**Data members**

**data fields:**
(All data fields are private and are only accessible to the class. They are not needed to be accessible to other classes and encapsulation is intensified.)
1) enum type of Month (to enumerate all the 12 months and indexed starting from 1)
2) enum type of Day (to enumerate all the 7 days in a week and indexed starting from 0)
3) unsigned int date_ (to store the dd part of the Date)
4) Month month_ (to store MMM part of the Date)
5) unsigned int year_ (to store yy part of the Date)

**Static Methods:**
**private static methods:**
1) static bool isLeap(int) :
- This method takes an integer value as argument (passed by value) and returns a boolean value (return by value).
- It returns true(1) if the integer passed to it is a leap year and returns false(0) otherwise.
- This method is a utility function i.e. it is to be used by the class alone for its internal computation and should not be exposed to the user and thus is declared as private static.

2) static bool validate(unsigned int d,unsigned int m,unsigned int y):
- This member function returns true if the date formed using d,m and y is valid or not. It considers the Gregorian Calendar between the years 1900 and 2099 both inclusive.

**public static methods:**
1) static Date GetCurrentDate():
- This is a static method which returns the current system date as a Date Class Objetct.
- This is a static  method and belongs to the class.

**Methods**

**private member methods:**
Constructor :
- The Constructor of this class takes three arguments d, m and y as unsigned integers and initializes the date_, month_ and year_ respectively.
- The body of the Constructor is empty.

**public member methods:**
1) static createDate( unsigned int d, unsigned int m, unsigned int y):

- This method takes three arguments d, m and y as unsigned integers and checks whether the date foremed using these is valid or not before using the constructor to create a date object.
- This is a static method which checks for validity of the input before calling the constructor of the Date Class for object creation.
- This method will be used to throw exceptions if the input is invalid. This is done to avoid throwing exceptions from the constructor which leads to undefined state.
- This is a static  method and belongs to the class.

2) Destructor
- The Destructor of Date Class is a default destructor.

3) void print() const
- This member function prints out the date in dd/MMM/yy format.
- This function is qualified as constant and is not allowed to change any data members of the object.

4) bool validDate() const
- This member function returns true if the date stored in the object is valid or not. It considers the Gregorian Calendar between the years 1900 and 2099 both inclusive.
- This function is qualified as constant and is not allowed to change any data members of the object.

5) std:: string day() const
- This member function returns the day of the week corresponding to the date. It uses system calendar for it’s operation.
- This function is qualified as constant and is not allowed to change any data members of the object.

6) unsigned int DiffOfYears(const Date&) const:
- This member function returns the difference in the number of years between the “this” object and the Date object taken as argument.
- This function is qualified as constant and is not allowed to change any data members of the object.

7) bool operator==(const Date&) const
- The equality operator is overloaded as a member function-Returns true if the dates are the same else false.
- This method takes a Date object by const reference as an argument and the current object is implicitly passed as this pointer.
- This function is qualified as constant and is not allowed to change any data members of the object.

8) bool operator!=(const Date&) const
- The inequality operator is overloaded as a member function-Returns true if the dates are different else false.
- This method takes a Date object by const reference as an argument and the current object is implicitly passed as this pointer.
- This function is qualified as constant and is not allowed to change any data members of the object.

By Hritaban Ghosh 19CS30053

# Class Passenger

This class is used to encapsulate Passenger Information. It stores the date, month and year of the the date.

**Data members**

**data fields:**
(All data fields are private and are only accessible to the class. They are not needed to be accessible to other classes and encapsulation is intensified.)
1) Name name
- Stores the name of the Passenger using Name Class.

2) Date dateOfBirth
- Stores the DOB of the Passenger using Date Class.

3) const Gender& gender
- Stores the Gender of the Passenger using Gender Class.

4) std::string aadhaarNumber
- Stores the 12 digit aadhaarNumber.

5) std::string mobileNumber
- Stores the 10 digit Mobile Number of the Passenger or “” if not provided.

6) const Divyaang* const disabilityType
- Stores the disability type of the Passenger if any.

7) const std::string disabilityID
- Stores the disability ID of the Passenger if any.

**Methods**

**private member methods:**

1)Constructor :
- The Constructor of this class takes seven arguments name, dateOfBirth, gender, aadhaarNumber, mobileNumber with mobileNumber being defaulted to “”, disabilityType defaulted to 0, and disabilityID defaulted to “”. These are initialized in the initializer list.
- The body of the Constructor is empty.

**public member methods:**
1) static Passenger createPassenger(…...):
- This is a static method which checks for validity of the input before calling the constructor of the Passenger Class for object creation.
- This method will be used to throw exceptions if the input is invalid. This is done to avoid throwing exceptions from the constructor which leads to undefined state.
- It takes 11 arguments, 3 of which are defaulted.
- The arguments are firstName, middleName, lastName, day, month, year (of Birth), gender, aadhaarNumber, mobileNumber defaulted to “”, disabilityType defaulted to 0 and disabilityID defaulted to “”.

- This is a static  method and belongs to the class.

2) Copy Constructor
   - The copy constructor takes a Passenger object as argument by const reference and copies each data member of the argument into the now being created object via the initializer list.
   - The body of the Copy Constructor is empty.
3) virtual Destructor
   - The Destructor of Date Class is a default destructor.
   - The body of the Destructor is empty.
4) Name GetName() const:
   - This member function returns the name of the Passenger.
   - This function is qualified as constant and is not allowed to change any data members of the object.
5) Date GetDOB() const:
   - This member function returns the date of birth of the Passenger.
   - This function is qualified as constant and is not allowed to change any data members of the object.
6) Gender GetGender() const:
   - This member function returns the gender of the Passenger.
   - This function is qualified as constant and is not allowed to change any data members of the object.
7) std::string GetAadhaar() const:
   - This member function returns the aadhaar number of the Passenger.
   - This function is qualified as constant and is not allowed to change any data members of the object.
8) std::string GetMobile() const:
   - This member function returns the mobile number of the Passenger if it had been provided else returns “”.
   - This function is qualified as constant and is not allowed to change any data members of the object.
9) const Divyaang* const GetDisability() const:
   - This member function returns the type of disability of the Passenger if any else returns 0.
   - This function is qualified as constant and is not allowed to change any data members of the object.
10) const std::string GetDisabilityID() const:
   - This member function returns the Disability ID of the Passenger if any else returns “”.
   - This function is qualified as constant and is not allowed to change any data members of the object.
11) unsigned int GetAge() const:
   - This member function returns the age of the Passenger.
   - It takes help of the Date class.
   - This function is qualified as constant and is not allowed to change any data members of the object.

By Hritaban Ghosh 19CS30053

# Class BookingClass

This class is an abstract base class and is used as an interface for the BookingClass Hierarchy.
The entire hierarchy is implemented using inclusion and parametric polymorphism.

**Methods**

**protected member methods**

1) Constructor :
   - The Constructor of this class is the default constructor.
   - The body of the Constructor is empty.
2) virtual Destructor
   - The Destructor of this Class is a default destructor.
   - It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
   - The body of the Destructor is empty.

**public member methods:**

1) virtual double GetLoadFactor() const = 0
   - It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
   - It's role is to return the Load Factor of the Booking Class
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.
2) virtual std::string GetName() const = 0
   - It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
   - It's role is to return the Name of the Booking Class
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.
3) virtual bool IsSitting() const = 0
   - It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
   - It's role is to return true if the Booking Class has Seat else false
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.
4) virtual bool IsAC() const = 0
   - It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
   - It's role is to return true if the Booking Class has AC else false
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

5) virtual int GetNumberOfTiers() const = 0
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the Number Of Tiers of the Booking Class
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

6) virtual bool IsLuxury() const = 0
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return true if the Booking Class is Luxury else false
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

7) virtual double GetReservationCharge() const = 0
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the Reservation Charge of the Booking Class
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

8) virtual double GetMinimumTatkalCharges() const =0:
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the Minimum Tatkal Charges of the Booking Class
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

9) virtual double GetMaximumTatkalCharges() const =0:
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the Maximum Tatkal Charges of the Booking Class
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

10) virtual double GetMinimumTatkalDistance() const =0:
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the Minimum Tatkal Distance of the Booking Class
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

11) virtual double GetTatkalLoadFactor() const =0:

- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the Tatkal Load Factor of the Booking Class
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

## Templatised Class with template<typename T>

## Class BookingClassTypes: Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass. It has common information which is to be stored across various Booking Class Types.
This results in formation of singleton concrete classes as all virtual functions of the Base Class have been implemented.

**Static Data Members:**

1) static const double sloadFactor
    1. It stores the load factor that is to be levied in the computation of fare.
    2. It is qualified as const as none of the data can be changed after initialization.
    3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetLoadFactor method which is const qualified.
2) static const std::string sName
    1. It stores the Name of the Booking Class.
    2. It is qualified as const as none of the data can be changed after initialization.
    3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetName method which is const qualified.
3) static const bool sIsSitting
    1. It stores true if the Booking Class has Seat else false.
    2. It is qualified as const as none of the data can be changed after initialization.
    3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the IsSitting method which is const qualified.
4) static const bool sIsAC
    1. It stores true if the Booking Class has AC else false .
    2. It is qualified as const as none of the data can be changed after initialization.
    3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the IsAC method which is const qualified.
5) static const int sNumberOfTiers
    1. It stores the Number Of Tiers of the Booking Class.
    2. It is qualified as const as none of the data can be changed after initialization.
    3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetNumberOfTiers method which is const qualified.
6) static const bool sIsLuxury

1. It stores true if the Booking Class is Luxury else false.
2. It is qualified as const as none of the data can be changed after initialization.
3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the IsLuxury method which is const qualified.

7) static const double sReservationCharge
   1. It stores the Reservation Charge of the Booking Class.
   2. It is qualified as const as none of the data can be changed after initialization.
   3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetReservationCharge method which is const qualified.
8) static const double sMinimumTatkalCharges
   1. It stores the Minimum Tatkal Charges of the Booking Class.
   2. It is qualified as const as none of the data can be changed after initialization.
   3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetMinimumTatkalCharges method which is const qualified.
9) static const double sMaximumTatkalCharges
   1. It stores the Maximum Tatkal Charges of the Booking Class.
   2. It is qualified as const as none of the data can be changed after initialization.
   3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetMaximumTatkalCharges method which is const qualified.
10) static const double sMinimumTatkalDistance
   1. It stores the Minimum Tatkal Distance of the Booking Class.
   2. It is qualified as const as none of the data can be changed after initialization.
   3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetMinimumTatkalDistance method which is const qualified.
11) static const double sTatkalLoadFactor
   1. It stores the Tatkal Load Factor of the Booking Class.
   2. It is qualified as const as none of the data can be changed after initialization.
   3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetTatkalLoadFactor method which is const qualified.

**Methods**

**Static member methods:**

1) static const BookingClassTypes<T>& Type()

- Singleton instance of the class BookingClassTypes<T> can be obtained by calling BookingClassTypes<T>::Type().
- This method is public and is used to create an instance of the class statically and return the same. This means that it creates a single instance of the class which is used throughout the program.
- The constructor is made private and thus the only way to access an instance of this class is through this method.

By Hritaban Ghosh 19CS30053

**private member methods:**

1) Constructor :
   - The Constructor of this class is the default constructor.
   - The body of the Constructor is empty.
2) virtual Destructor
   - The Destructor of this Class is a default destructor.
   - It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
   - The body of the Destructor is empty.

**public member methods:**

1) virtual double GetLoadFactor() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the static constant sloadFactor by value.
   - This function is qualified as constant and is not allowed to change any data members of the object.
2) virtual std::string GetName() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the name of the Booking Class which is sName.
   - This function is qualified as constant and is not allowed to change any data members of the object.
3) virtual bool IsSitting() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the static constant sIsSitting.
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.
4) virtual bool IsAC() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the static constant sIsAC.
   - This function is qualified as constant and is not allowed to change any data members of the object.
5) virtual int GetNumberOfTiers() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the static constant sNumberOfTiers.
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.
6) virtual bool IsLuxury() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the static constant sIsLuxury.
   - This function is qualified as constant and is not allowed to change any data members of the object.
7) virtual double GetReservationCharge() const
   - It is a **virtual** function and has been implemented for this class.
   - It returns the static constant sReservationCharge.

- This function is qualified as constant and is not allowed to change any data members of the object.

8) virtual double GetMinimumTatkalCharges() const
- It is a **virtual** function and has been implemented for this class.
- It returns the static constant sMinimumTatkalCharges.
- This function is qualified as constant and is not allowed to change any data members of the object.

9) virtual double GetMaximumTatkalCharges() const
- It is a **virtual** function and has been implemented for this class.
- It returns the static constant sMaximumTatkalCharges.
- This function is qualified as constant and is not allowed to change any data members of the object.

10) virtual double GetReservationCharge() const
- It is a **virtual** function and has been implemented for this class.
- It returns the static constant sMinimumTatkalDistance.
- This function is qualified as constant and is not allowed to change any data members of the object.

11) virtual double GetReservationCharge() const
- It is a **virtual** function and has been implemented for this class.
- It returns the static constant sTatkalLoadFactor.
- This function is qualified as constant and is not allowed to change any data members of the object.

**Note**: The **public static unit test function** is described in the test plan.

- BookingClassTypes<ACFirstClassType> ACFirstClass
- BookingClassTypes<ExecutiveChairCarType> ExecutiveChairCar
- BookingClassTypes<AC2TierType> AC2Tier
- BookingClassTypes<FirstClassType> FirstClass
- BookingClassTypes<AC3TierType> AC3Tier
- BookingClassTypes<ACChairCarType> ACChairCar
- BookingClassTypes<SleeperType> Sleeper
- BookingClassTypes<SecondSittingType> SecondSitting

By Hritaban Ghosh 19CS30053

## Class ExecutiveChairCar : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass via parameterized polymorphism . It stores information for the Booking Class Category “Executive Chair Car”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

For example:

Information stored in this template class:

```
const double BookingClass::ExecutiveChairCar::sLoadFactor = 5.00
const std::string BookingClass::ExecutiveChairCar::sName = "Executive Chair Car"
const bool BookingClass::ExecutiveChairCar::sIsSitting = true
const bool BookingClass::ExecutiveChairCar::sIsAC = true
const int BookingClass::ExecutiveChairCar::sNumberOfTiers = 0
const bool BookingClass::ExecutiveChairCar::sIsLuxury = true
const double BookingClass::ExecutiveChairCar::sReservationCharge = 60.0
const double BookingClass::ExecutiveChairCar::sMinimumTatkalCharges = 400.0
const double BookingClass::ExecutiveChairCar::sMaximumTatkalCharges = 500.0
const double BookingClass::ExecutiveChairCar::sMinimumTatkalDistance = 250
const double BookingClass::ExecutiveChairCar::sTatkalLoadFactor = 0.3
```

## Class ACChairCar : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “AC Chair Car”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class SecondSitting: Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “Second Sitting”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class ACFirstClass : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “AC First Class”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class AC2Tier : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “AC 2 Tier”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

By Hritaban Ghosh 19CS30053

## Class FirstClass : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “First Class”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class AC3Tier : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “AC 3 Tier”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class Sleeper : Specialization of BookingClass

This class is a Specialization of the Abstract Base Class BookingClass implemented via parameterized polymorphism . It stores information for the Booking Class Category “Sleeper”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class and Hierarchy of Divyaang Class

This class is an abstract base class and is used as an interface for the Divyaang Class Hierarchy.
The entire hierarchy is implemented using inclusion and parametric polymorphism.

**Methods**

**protected member methods**
1) Constructor :
- The Constructor of this class is the default constructor.
- The body of the Constructor is empty.

2) virtual Destructor
- The Destructor of this Class is a default destructor.
- It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
- The body of the Destructor is empty.

**public member methods:**
1) virtual const double GetConcession(const BookingClass&) const = 0
- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It’s semantic meaning is given in the next line.
- It’s role is to return the concession offered for a particular disability type based on the Booking class.
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

P.T.O.

By Hritaban Ghosh 19CS30053

## Templatised Class with template<typename T>

## Class DivyaangTypes: Specialization of Divyaang

This class is a Specialization of the Abstract Base Class Divyaang. It has common information which is to be stored across various Divyaang Class Types.
This results in formation of singleton concrete classes as all virtual functions of the Base Class have been implemented.

**Static Data Members:**
1) static const std::map <std::string,double> ConcessionFactor
1. It stores the concession factor that is to be levied in the computation of fare based on the name of the Booking Class.
2. It is qualified as const as none of the data can be changed after initialization.
3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetConcessionFactor method which is const qualified.

**Methods**

**Static member methods:**
1) static const DivyaangTypes<T>& Type()
- Singleton instance of the class DivyaangTypes<T> can be obtained by calling DivyaangTypes<T>::Type().
- This method is public and is used to create an instance of the class statically and return the same. This means that it creates a single instance of the class which is used throughout the program.
- The constructor is made private and thus the only way to access an instance of this class is through this method.

**Private Member Methods:**
1) Constructor :
- The Constructor of this class is the default constructor.
- The body of the Constructor is empty.

2) virtual Destructor
- The Destructor of this Class is a default destructor.
- It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
- The body of the Destructor is empty.

**Public Member Methods:**
1)virtual double GetConcessionFactor(const BookingClass&) const
- It is a **virtual** function and has been implemented for this class.
- It returns the concession factor based on the Booking Class.
- This function is qualified as constant and is not allowed to change any data members of the object.

- DivyaangTypes<BlindType> Blind
- DivyaangTypes<OrthopaedicallyHandicappedType> Orthopaedically_Handicapped
- DivyaangTypes<CancerPatientsType> Cancer_Patient
- DivyaangTypes<TBPatientsType> TB_Patient

## Class Blind : Specialization of Divyaang

This class is a Specialization of the Abstract Base Class Divyaang implemented via parameterized polymorphism . It stores information for the Divyaang Class Category “Blind”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

For example:

Information stored in this template class:

```
const std::map <std::string,double> Divyaang::Blind::ConcessionFactor =
{
        {"AC First Class",0.50},
        {"Executive Chair Car",0.75},
        {"AC 2 Tier",0.50},
        {"First Class",0.75},
        {"AC 3 Tier",0.75},
        {"AC Chair Car",0.75},
        {"Sleeper",0.75},
        {"Second Sitting",0.75}
}
```

## Class Orthopaedically_Handicapped : Specialization of Divyaang

This class is a Specialization of the Abstract Base Class Divyaang implemented via parameterized polymorphism . It stores information for the Divyaang Class Category “Orthopaedically_Handicapped”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class Cancer_Patient : Specialization of Divyaang

This class is a Specialization of the Abstract Base Class Divyaang implemented via parameterized polymorphism . It stores information for the Divyaang Class Category “Cancer_Patient”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

## Class TB_Patient : Specialization of Divyaang

This class is a Specialization of the Abstract Base Class Divyaang implemented via parameterized polymorphism . It stores information for the Divyaang Class Category “TB_Patient”.
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

By Hritaban Ghosh 19CS30053

## Class and Hierarchy of Concessions Class

This class is a base class and is used as an interface for the Concessions Class Hierarchy.
The entire hierarchy is implemented using ad-hoc polymorphism.

**Methods**

**public member methods:**
1) Constructor :
   - The Constructor of this class is the default constructor.
   - The body of the Constructor is empty.
2) Destructor
   - The Destructor of this Class is a default destructor.
   - The body of the Destructor is empty.
3) double GetConcessionFactor() const
   - It's role is to return the concession offered for a particular Booking Category.
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

## Class General_Concession: Specialization of Concessions

This class is a Specialization of the Base Class Concessions.
**Methods**

**Public Member Methods:**
1) Constructor :
   - The Constructor of this class is the default constructor.
   - The body of the Constructor is empty.
2) Destructor
   - The Destructor of this Class is a default destructor.
   - The body of the Destructor is empty.
3) double GetConcessionFactor() const
   - It's role is to return the concession offered by General Booking Category which is 0.
   - This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

## Class Ladies_Concession: Specialization of Concessions

This class is a Specialization of the Base Class Concessions.
**Methods**

**Public Member Methods:**
1) Constructor :
   - The Constructor of this class is the default constructor.
   - The body of the Constructor is empty.
2) Destructor
   - The Destructor of this Class is a default destructor.

- The body of the Destructor is empty.

3) double GetConcessionFactor(const Passenger&) const
- It's role is to return the concession offered by Ladies Booking Category which is for now 0.
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

## Class SeniorCitizen_Concession: Specialization of Concessions

This class is a Specialization of the Base Class Concessions.
**Methods**

**Public Member Methods:**
1) Constructor :
- The Constructor of this class is the default constructor.
- The body of the Constructor is empty.

2) Destructor
- The Destructor of this Class is a default destructor.
- The body of the Destructor is empty.

3) double GetConcessionFactor(const Passenger&) const
- It's role is to return the concession offered by Senior Citizen Booking Category which is for now 0.4 for 60+ Male and 0.5 for 58+ Female.
- It uses Passenger gender in order to determine the concession offered.
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

## Class Divyaang_Concession: Specialization of Concessions

This class is a Specialization of the Base Class Concessions.
**Methods**

**Public Member Methods:**
1) Constructor :
- The Constructor of this class is the default constructor.
- The body of the Constructor is empty.

2) Destructor
- The Destructor of this Class is a default destructor.
- The body of the Destructor is empty.

3) double GetConcessionFactor(const BookingClass&, const Passenger&) const
- It's role is to return the concession offered by Divyaang Category based on the Passenger's disability and the Booking Class.
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.

By Hritaban Ghosh 19CS30053

## Class and Hierarchy of BookingCategory Class

This class is an abstract base class and is used as an interface for the Booking Category Class Hierarchy.
The entire hierarchy is implemented using inclusion and parametric polymorphism.

**Methods**

**protected member methods**
1) Constructor :
- The Constructor of this class is the default constructor.
- The body of the Constructor is empty.

2) virtual Destructor
- The Destructor of this Class is a default destructor.
- It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
- The body of the Destructor is empty.

## Templatised Class with template<typename T>

## Class BookingCategoryTypes: Specialization of BookingCategory

This class is a Specialization of the Abstract Base Class Divyaang. It has common information which is to be stored across various Divyaang Class Types.
This results in formation of singleton concrete classes as all virtual functions of the Base Class have been implemented.

**Static Data Members:**
1) static const std::string sName
1. It stores the name of the Booking Category.
2. It is qualified as const as none of the data can be changed after initialization.
3. This static constant is private. It is made private so that no one can inadvertently change it's value. It can be obtained through the GetName method which is const qualified.

**Methods**

**Static member methods:**
1) static const Booking CategoryTypes<T>& Type()
- Singleton instance of the class Booking CategoryTypes<T> can be obtained by calling Booking CategoryTypes<T>::Type().
- This method is public and is used to create an instance of the class statically and return the same. This means that it creates a single instance of the class which is used throughout the program.
- The constructor is made private and thus the only way to access an instance of this class is through this method.

**Private Member Methods:**

1) Constructor :
- The Constructor of this class takes one argument which is the name of the class and is defaulted with the static constant sName.
- The body of the Constructor is empty.

2) virtual Destructor
- The Destructor of this Class is a default destructor.
- It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
- The body of the Destructor is empty.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)
- It returns whether the passenger is eligible to choose this booking category or not.
- This is a static method and belongs to the class.

- BookingCategoryTypes<GeneralType> General
- BookingCategoryTypes<LadiesType> Ladies
- BookingCategoryTypes<SeniorCitizenType> SeniorCitizen
- BookingCategoryTypes<DivyaangType> Divyaang
- BookingCategoryTypes<TatkalType> Tatkal
- BookingCategoryTypes<PremiumTatkalType> PremiumTatkal

## Class General: Specialization of BookingCategory

This class is a Specialization of the Abstract Base Class BookingCategory implemented via parameterized polymorphism . It stores information for the BookingCategory Class "General". This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)
- It returns true for all Passengers.
- This is a static method and belongs to the class.

## Class Ladies: Specialization of BookingCategory

This class is a Specialization of the Abstract Base Class BookingCategory implemented via parameterized polymorphism . It stores information for the BookingCategory Class "Ladies". This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)

- It returns true if the Passenger is a female.
- This is a static  method and belongs to the class.

## Class SeniorCitizen: Specialization of  BookingCategory

This class is a Specialization of the Abstract Base Class BookingCategory implemented via parameterized polymorphism . It stores information for the BookingCategory Class  “SeniorCitizen”. This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)

- It returns true if the passenger is male and 60+ or if the passenger is female and 58+.
- This is a static  method and belongs to the class.

## Class Divyaang: Specialization of  BookingCategory

This class is a Specialization of the Abstract Base Class BookingCategory implemented via parameterized polymorphism . It stores information for the BookingCategory Class  “Divyaang”. This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)

- It returns true if the passenger has a valid disability.
- This is a static  method and belongs to the class.

## Class Tatkal: Specialization of  BookingCategory

This class is a Specialization of the Abstract Base Class BookingCategory implemented via parameterized polymorphism . It stores information for the BookingCategory Class  “Tatkal”. This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)

- It returns true for all Passengers.
- This is a static  method and belongs to the class.

## Class PremiumTatkal: Specialization of  BookingCategory

This class is a Specialization of the Abstract Base Class BookingCategory implemented via parameterized polymorphism . It stores information for the BookingCategory Class  “PremiumTatkal”. This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**

1)static bool isEligible(const Passenger&)

- It returns true for all Passengers.
- This is a static  method and belongs to the class.

By Hritaban Ghosh 19CS30053

# Class and Hierarchy Booking Class

This class is an abstract base class and is used as an interface for the Booking Class Hierarchy.
The entire hierarchy is implemented using inclusion and parametric polymorphism.

**Friend Functions:**

1) ostream& operator<<(ostream&, const Date&)
   - The insertion operator is overloaded as a friend function and it prints out all the Booking Information to the console.
   - It takes a output streaming operator by reference and returns the same by reference in order to support chaining. It also takes a Booking object by const reference in order to access it's data members and print to the console.

Example:

BOOKING SUCCEEDED:
Name of the Passenger: Mr. Harold Peterson
DOB: 12/Feb/1999
Gender: Male
PNR Number : 1
From Station : Mumbai
To Station : Delhi
Travel Date : 15/Feb/2021
Travel Class : AC First Class
Booking Category: General
--> Mode : Sleeping
--> Comfort : AC
--> Bunks : 2
--> Luxury : Yes
Fare : 4763

**Static Data Members:**

1) static const double sBareFarePerKM
   - Stores the Bare Fare per Kilometer that is to levied in the computation of the fare.
   - It is qualified as const as none of the data can be changed after initialization.
   - This static constant is private. It is made private so that no one can inadvertently change it's value.
2) static int sBookingPNRSerial
   - Stores the current PNR Serial starting from 1. It is unique to a Booking Object. It is incremented after every construction of a new Booking object.
   - This static variable is private. It is made private so that no one can inadvertently change it's value.
3) static std::vector<Booking*> sBookings
   - Stores pointers to Bookings made in the Railway Booking System. This is to keep in track all the Bookings that have been made.

P.T.O.

- This static variable is public. It is made public so that it can be accessed in the testing applications and in other parts of the program.

**Data members**

**data fields:**
(All data fields are private and are only accessible to the class. They are not needed to be accessible to other classes and encapsulation is intensified.)
1) Station fromStation
   - Stores the Station of Departure.
2) Station toStation
   - Stores the Station of Arrival.
3) Date dateOfBooking
   - Stores the Date of the Journey.
4) const BookingClass& bookingClass
   - Stores the reference to the Booking Class.
   - It is qualified as const as none of the data can be changed after initialization.
5) const BookingCategory& bookingCategory
   - Stores the reference to the Booking Category.
   - It is qualified as const as none of the data can be changed after initialization.
6) int fare
   - Stores the fare required to pay for the ticket. This is computed using member function ComputeFare.
7) int PNR
   - Stores the unique PNR Number of the ticket.
   - Uses the sBookingPNRSerial to initialize it’s value and after that the sBookingPNRSerial is incremented.
8) bool bookingStatus = true
   - Stores the Booking Status.
   - By default the booking status value is “true”.
8) std::string bookingMessage = "BOOKING SUCCEEDED"
   - Stores the Booking Message to be displayed when Booking is done.
   - By default the booking message is “ BOOKING SUCCEEDED”.
9) const Passenger&  passenger
   - Stores Passenger Information.
   - It is qualified as const as none of the data can be changed after initialization.
10) Date dateOfReservation
   - Stores the Date of Booking

**Methods**

**protected member methods:**
1) Constructor :

- The Constructor of this class takes six arguments fromStation, toStation, date, bookingClass, bookingCategory and passenger and initializes the data members using the initializer list.
- The body of the Constructor is empty.

2) virtual Destructor

- The destructor deletes the pointer to the object in the vector sBookings.
- The destructor also deletes the passenger object that is associated with the booking.
- It is qualified as virtual to provide for further futuristic polymorphic hierarchy.

**public member methods:**

3) virtual int ComputeFare() const = 0

- It is a **pure virtual** function and has not been implemented for this class. It is to be implemented in subsequent specializations of this class. It's semantic meaning is given in the next line.
- It's role is to return the fare computed based on the passenger, booking class and booking category.
- This function is qualified as constant and is not allowed to change any data members of the object even in the specializations of the class.
- It computes the fare based on the data provided in the data members of the Booking object and the Business Logic and returns the fare after rounding it to the nearest integer.

4) bool operator==(const Booking&) const

- The equality operator is overloaded as a member function-Returns true if the PNR Numbers of the Bookings are the same else false.
- This method takes a Booking object by const reference as an argument and the current object is implicitly passed as this pointer.
- This function is qualified as constant and is not allowed to change any data members of the object.

5) bool operator!=(const Booking&) const

- The inequality operator is overloaded as a member function-Returns true if the PNR Numbers of the Bookings are different else false.
- This method takes a Booking object by const reference as an argument and the current object is implicitly passed as this pointer.
- This function is qualified as constant and is not allowed to change any data members of the object.

## Templatised Class with template<typename T>

## Class BookingTypes: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking. It has common information which is to be stored across various Divyaang Class Types.
This results in formation of singleton concrete classes as all virtual functions of the Base Class have been implemented.

By Hritaban Ghosh 19CS30053

**Methods**

**Public Member Methods:**
1) Constructor :
- The Constructor of this class takes six arguments namely fromStation, toStation, date, bookingClass, bookingCategory and passenger and initializes the data members of the parent class using the initializer list.
- The body of the Constructor is used to compute the fare and then initialise the fare data field of the parent class as the computation itself depends on other data fields.

2) virtual Destructor
- The Destructor of this Class is a default destructor.
- It is qualified as virtual to handle the deletion of objects in the polymorphic hierarchy.
- The body of the Destructor is empty.

- BookingTypes<GeneralType> GeneralBooking
- BookingTypes<LadiesType> LadiesBooking
- BookingTypes<SeniorCitizenType> SeniorCitizenBooking
- BookingTypes<DivyaangType> DivyaangBooking
- BookingTypes<TatkalType> TatkalBooking
- BookingTypes<PremiumTatkalType> PremiumTatkalBooking

## Class GeneralBooking: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking implemented via parameterized polymorphism . It stores information for the Booking Class "GeneralBooking".
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**
1) int ComputeFare() const
- Computes and returns the fare based on the Business logic of the General Booking Category.

## Class LadiesBooking: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking implemented via parameterized polymorphism . It stores information for the Booking Class "LadiesBooking".
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**
1) int ComputeFare() const
- Computes and returns the fare based on the Business logic of the Ladies Booking Category.

## Class SeniorCitizenBooking: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking implemented via parameterized polymorphism . It stores information for the Booking Class "SeniorCitizenBooking".
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**
1) int ComputeFare() const
- Computes and returns the fare based on the Business logic of the SeniorCitizen Booking Category.

## Class DivyaangBooking: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking implemented via parameterized polymorphism . It stores information for the Booking Class "DivyaangBooking".
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**
1) int ComputeFare() const
- Computes and returns the fare based on the Business logic of the Divyaang Booking Category.

## Class TatkalBooking: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking implemented via parameterized polymorphism . It stores information for the Booking Class "TatkalBooking".
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**
1) int ComputeFare() const
- Computes and returns the fare based on the Business logic of the Tatkal Booking Category.

## Class PremiumTatkalBooking: Specialization of Booking

This class is a Specialization of the Abstract Base Class Booking implemented via parameterized polymorphism . It stores information for the Booking Class "PremiumTatkaBookingl".
This is a singleton concrete class as all virtual functions of the Base Class have been implemented.

**Public Member Methods:**
1) int ComputeFare() const
- Computes and returns the fare based on the Business logic of the PremiumTatkal Booking Category.

By Hritaban Ghosh 19CS30053

# Class and Hierarchy of Exceptions

## Class Bad_Date : Specialization of std::exception

The objects of this class is used to throw exceptions whenever a date which is invalid is being created, for example 29/02/2021 or 31/04/2020 . It is also used in erroneous situations where the date of birth of a passenger is in the future or the date of Reservation is in the past. It may show a massage like “Date is invalid or you have an inconsistency in the data inputted”.

## Class Bad_Railways : Specialization of std::exception

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the master data of the railways booking system.

Example: Duplicated station name or negative distance.

## Class Bad_Station : Specialization of Bad_Railways

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the master data of the stations list of the railways booking system.

Example: Duplicated station name. Or when user enters both fromStation and to Station to be the same.

## Class Bad_Distance : Specialization of Bad_Railways

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the master data of the distance matrix of the railways booking system.

Example: Negative distance or 0 distance between two different station

## Class Bad_Booking : Specialization of std::exception

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the

data of the Booking object. For example, when the date of Booking is in the past or the Booking Category Type does not match with the Booking Class Type.

## Class Bad_Passenger : Specialization of std::exception

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the data of a passenger object. For example the passenger’s date of birth is in the future or the passenger’s age is negative, aadhaar number has letters, spaces and more than or less than 12 digits etc.

### Class Bad_Name : Specialization of Bad_Passenger

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the data of the Name object. For example the passenger has not entered both the first name and the last name, or the passenger has left all the data fields of the name blank.

### Class Bad_Age : Specialization of Bad_Passenger

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the data of the dateOfBirth of a passenger. For example, if the date Of Birth is in the future, or age comes out to be negative or inifinity.

### Class Bad_Aadhaar : Specialization of Bad_Passenger

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the aadhaar Number of the Passenger. For example, aadhaar number has letters, spaces and more than or less than 12 digits etc.

### Class Bad_Mobile : Specialization of Bad_Passenger

The objects of this class is used to throw exceptions whenever an error or inconsistency is found in the mobile Number of the Passenger. For example, mobile number has letters, spaces and more than or less than 10 digits etc.

## Coding Principles and Guidelines

- 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

By Hritaban Ghosh 19CS30053

- Prefer C++ style casting (like static_cast<cast>(x) over C style casting (like (int))
- The project should compile without any compiler warning
- Indent code properly
- Comment the code liberally and meaningfully
- Keep the Code Simple
- Design code with scalability and reuse in mind.
- Correct errors as they occur.
- Indicate a brief description of what a variable is for (reference to commenting)
- Try to define different sections of the code by segmenting blocks of code into a paragraph
- The code should be such that one should be able to understand it even after returning to it after some time gap, without that person having to look at every line of it
- The language functions that are complex or the structure that is difficult to be comprehended should be avoided
- Avoid Commenting on Obvious Things
- Deep nesting structure should be avoided
  Too many nesting structures make it difficult to understand the code.
- Folllow the Principle of DRY which stands for Don’t Repeat Yourself, also known as DIE (duplication is evil).
- It is more important to be correct than to be fast. It is more important to be maintainable than to be fast.