Railway Booking System

Software Engineering

Assignment 4

Design Document

Ashutosh Kumar Singh

19CS30008

1 General Design Principles

Firstly, there are some common points to note about each class :

- **Blocked Copy Constructor and Copy Assignment Operator** :
  Consider a class `ExampleClass`.
  The copy constructor `ExampleClass(const ExampleClass&)` and the copy assignment operator `ExampleClass& operator=(const ExampleClass&)` have been blocked by making them private data members.
  By making them private, we prevent users from using them because if anyone tries to use them outside the class, then it will give a compilation error.
  The main purpose of blocking them is to ensure that objects of UDTs are passed by reference, and to save memory.
  Note that the copy assignment operator is blocked for all the classes. The copy constructor is blocked for all classes except `Date`.

- **Passing objects and built-in types** :
  At all places, objects of UDTs are passed by reference as this helps in minimizing memory consumption, as no copy of the object is created.
  Also, if objects are passed by value, then always the copy constructor would have to be invoked and if the copy constructor is not provided explicitly by us, then the default copy constructor may mess up in a deep copy and a shallow copy.
  Also, built-in data types like int are always passed by value.

- **Singleton classes** :
  At many places, we have made classes to be singleton classes, for example, class `Railways`, all the booking classes, the booking categories, and many more, are singleton classes.
  Here, to implement them, we have used the Meyer's Singleton. Hence, for these classes, the constructor and destructor are made private.
  For each of these classes, we have a `Type()` member function which returns the singleton instance of the respective class.

- **Friend `ostream` operator function** :
  Each class has the overloaded output streaming operator implemented as a friend function : `friend ostream& operator<<(ostream&, const ExampleClass&)`.
  The `ostream` and `ExampleClass` objects are passed by reference.
  Using a friend function is the only viable option because if we wanted to keep it in the `ostream` class, then we would have to change the `iostream` library, which is not possible.
  If we wanted to keep it in the `ExampleClass` class, then the usual semantics of the `<<` operator would get changed.
  Hence the only option is to use a friend function as this achieves our purpose and also preserves the encapsulation.

- **Unit Testing Functions** :
  Each class has a member function like `static void UnitTestExampleClass()` for unit testing the respective class.
  It is made static so that we can easily call this member function without having to explicitly create an object of the class.
  In the unit testing functions, we have used `assert` statements to compare the output with the golden output.

`assert` statements are very useful as in case of a mismatch, they report the line number along with the condition that failed and also stop the execution of the program.

- **Exception Handling before object construction from constructor** :
  The construction of an object of a class has the possibility of exceptions due to erroneous inputs. Hence, we check for any errors in a separate static function like `CreateExampleClass(...)` before invoking the constructor. This helps follow the guideline that no exception should be thrown from a constructor.
  Hence, we first call the static function `CreateExampleClass(...)` from the application, which in turn calls the constructor. So, the constructors of these classes are made private to ensure that the constructor does not get directly invoked with wrong inputs.

- **Virtual Construction Idiom** :
  We come across a situation in the `Booking` class hierarchy where we have to invoke the constructor of an appropriate sub-class depending on a sub-class from the `BookingCategory` hierarchy. Here we use the virtual construcion idiom to perform this task efficiently. Details about this are evident in the implementation.

Now, we delve deep into each class and look at the design paradigms for each of them.

2 Class `Station`

2.1 private data members / member functions / operator functions

- `const string name_` : Attribute to store the name of each `Station`. It has been made const as the name of a Station object will not change in future after it has been created.

- `Station(const string&)` : Constructor which takes a string which is a const reference as argument.

- `Station& operator=(const Station&)` : Blocked copy assignment operator

- `Station(const Station&)` : Blocked copy constructor

2.2 public member functions

- `~Station()` : Destructor.

- `static Station& CreateStation(const string&)` : This static function returns a `Station` object after checking if the string passed to it has any errors. If not, then it invokes the constructor of the `Station` class.

- `string GetName() const` : Returns the name of a `Station` object. The method is made const because it does not need to change the attributes of any object.

- `int GetDistance(const Station&) const` : This method takes a `Station` object as a const reference as it does not need to change it. It calls the `GetDistance` method of the `Railways` class to get the distance between two stations.

- `friend ostream& operator<<(ostream&, const Station&)` : Output streaming operator.

3 Class `Railways`

`Railways` is a singleton class.

3.1 private data members / member functions / operator functions

- `static const vector<Station> sStations` : A vector of `Station` objects that stores the master data, i.e., the list of all Stations.
  It is made static as it needs to be accessed from the beginning till the end of the program.
  It is made const because the list of Stations is already given and never changes in the future.

- `static const map<pair<string, string>, int> sDistStations` : This stores the pairwise distances between all Stations.
  It is implemented as a `map<pair<string, string>, int>`.
  The key is a pair of strings which is actually the pair of Stations and the value is an integer which is the distance between these 2 Stations.
  It is also made static as it is required at multiple times during the execution and const because it never changes in the future.

  Both `sStations` and `sDistStations` are initialized in `Railways.cpp` at the beginning itself because these are constants which will stay constant throughout all runs of the application. Hence, it is best to initialize them in the library itself.

- `Railways()` : Empty Constructor.

- `~Railways()` : Destructor.

- `Railways(const Railways&)` : Blocked copy constructor.

- `Railways& operator=(const Railways&)` : Blocked copy assignment operator.

3.2 public member functions

- `static const Railways& IndianRailways()` : This function implements the Meyer's Singleton and returns the singleton object.
  It has to be made static because this function itself returns the single instance. If it is not static, then for the first call to this function we will not have any instance, and we will not be able to call it. Hence it has to be made static.

- `Station GetStation(const string& name) const` : This function returns the reference to a `Station` object by retrieving it from the vector `sStations`. It is made const as it does not need to change the state of any object.

- `int GetDistance(const Station&, const Station&) const` : This is a const function as it only returns the distance between two `Station`s and does not need to change the state of any object. The arguments as usual are passed a const references.
  This function retrieves the distance between Stations from the map `sDistStations` by using the pair of Station names as the key.

- `friend ostream& operator<<(ostream&, const Railways&)` : Output streaming operator.

4 Class `Date`

4.1 private data members / member functions / operator functions

- `static const vector<string> sMonthNames` : A vector of strings to store the names of the 12 months.
  It is made static as we need it at multiple times and places during the execution.
  It is const as it will never change as the names of the 12 months cannot change.

- `static const vector<string> sDayNames` : A vector of strings to store the names of the 7 days of the week.
  It is made static as we need it at multiple instances during the execution.
  It is const as it will never change as the names of the days of the week cannot change.

  Both `sMonthNames` and `sDayNames` never change in the future and remain constant throughout all runs of the program, hence they are initialized in the library itself at the beginning of `Date.cpp`.

- `const int date_, const Month month_, const int year_` : These have been made const as they will not change after the object has been created.

- `Date(int, int, int)` : Constructor which takes three integers as arguments (passed by value).

- `Date& operator=(const Date&)` : Blocked copy assignment operator.

4.2 public member functions

- `static Date& CreateDate(const string&)` : A static function that returns a `Date` object after checking if the string passed to it has any errors. If not, then it invokes the constructor of the `Date` class.

- `Date(const Date&)` : Copy Constructor with usual semantics. It has been defined as it is needed in the initializer list of the class `Booking`.

- `~Date()` : Destructor.

- `Day day() const` : This function returns an integer from 0-6 depending on the day of the week with 0 denoting Sunday.
  It uses the `ctime` library to perform this task. It is made const as it does not change the state of the `Date` object.

- `static Date Today()` : `static` function to return the `Date` of the present day.

- `friend int operator-(const Date&, const Date&)` : Friend function that overloads the - operator to return the difference between two dates.

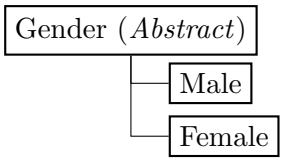
- `bool operator<=(const Date&)` : Overloaded operator to compare two dates.

- `bool operator==(const Date&)` : Equality operator to check equality of two dates.

- `friend ostream& operator<<(ostream&, const Date&)` : Output Streaming Operator.

5 Class and Hierarchy of `Gender`

The class `Gender` and its child classes `Male` and `Female` have been modelled using parametric polymorphism and inclusion polymorphism.



5.1 class `Gender`

5.1.1 private data members / member functions / operator functions

- `const string name_` : const data member that stores the name of the gender.

5.1.2 protected member functions

- `Gender(const string& name)` : Constructor.
- `virtual ~Gender()` : Virtual Destructor for a polymorphic hierarchy.

5.1.3 public member functions

- `const string& GetName() const` : Function to get the name of the `Gender`. It is made const as it does not change the state of any object.
- `virtual const string GetTitle() const = 0` : Pure virtual function.
- `static bool IsMale(const Gender&)` : `static` function to check if an object is of `Male` or `Female` type.

`Male` and `Female` are singleton classes modelled using static sub-typing (templates).

5.2 classes `Male` and `Female` denoted by the template `GenderTypes<T>`

5.2.1 private data members / member functions / operator functions

- `static const string sName, static const string sSalutation` : static constants. These are not needed in the application space and are hence initialized in `Gender.cpp` itself.
- `GenderTypes(const string& name = GenderTypes<T>::sName)` : Constructor.
- `~GenderTypes()` : Destructor.

5.2.2 public member functions

- `static const GenderTypes<T>& Type()` : Function to return the singleton object.
- `const string GetTitle() const` : Overriden function to get the salutation for a gender.

6 Class `Passenger`

6.1 private data members / member functions / operator functions

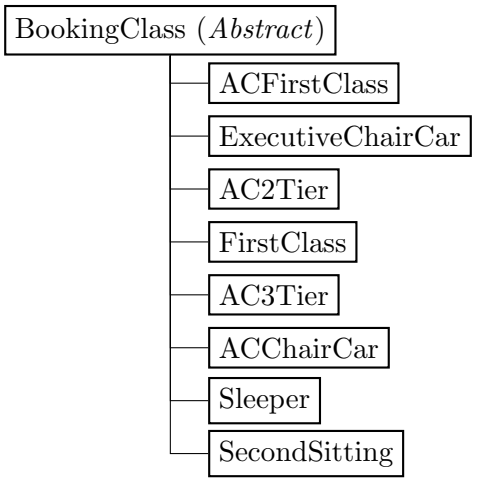
- `const Name name_, const Date dateOfBirth_, const Gender& gender_, const string& aadhaar_, const string& mobile_, const Divyaang& disability_, const string& disabilityID_)` : Data members to store information about a `Passenger`. All are made const because none of them can change once entered.
- `Passenger(const Name, const Date, const Gender&, const string&, const string&, const Divyaang&, const string&)` : Constructor.

6.2 public member functions

- `static Passenger& CreatePassenger(const Name, const Date, const Gender&, const string&, const string&, const Divyaang&, const string&)` : A static function that returns a `Passenger` object after checking if the attributes passed to it have any errors. If not, then it invokes the constructor of the `Passenger` class.
- `const Date GetDateOfBirth() const` : Returns the `dateOfBirth_`. Made const as it will not change the state of the object.
- `const Gender& GetGender() const` : Returns the `gender_`. Made const as it will not change the state of the object.
- `const Divyaang& GetDisability() const` : Returns the `disability_`. Made const as it will not change the state of the object.

7 Class and Hierarchy of `BookingClass`

The class `BookingClass` and its 8 child classes are modelled using a mix of static sub-typing and inclusion polymorphism.



7.1 class BookingClass

7.1.1 private data members / member functions / operator functions

- `const string name_` : const data member that stores the name of the booking class.

7.1.2 protected member functions

- `BookingClass(const string& name)` : Constructor.
- `virtual ~BookingClass()` : Virtual Destructor for a polymorphic hierarchy.

7.1.3 public member functions

- `const string& GetName() const` : Function to get the name of the `BookingClass`. It is made const as it does not change the state of any object.
- `friend ostream& operator<<(ostream&, const BookingClass&)` : Output Streaming Operator.
- There are a number of pure virtual functions which can be listed down as follows :
  - `virtual bool IsAC() const = 0`
  - `virtual bool IsLuxury() const = 0`
  - `virtual bool IsSitting() const = 0`
  - `virtual double GetLoadFactor() const = 0`
  - `virtual int GetNumberOfTiers() const = 0`
  - `virtual double GetReservationCharge() const = 0`
  - `virtual double GetTatkalFactor() const = 0`
  - `virtual double GetTatkalMinCharge() const = 0`
  - `virtual double GetTatkalMaxCharge() const = 0`
  - `virtual int GetMinTatkalDistance() const = 0`

Now, we have 8 booking classes as shown in the figure - `ACFirstClass, ExecutiveChairCar, AC2Tier, FirstClass, AC3Tier, ACChairCar, Sleeper` and `SecondSitting`. These are singleton classes modelled using static sub-typing (templates).

7.2 The 8 Booking Classes denoted by the template BookingClassType<T>

7.2.1 private data members / member functions / operator functions

- `static const string sName` : static constant to set the name of the booking class.
- `static const bool sIsSitting, static const bool sIsAC, static const int sNumberOfTiers` : These static constants will never be changed, hence they are intialized in the library itself.

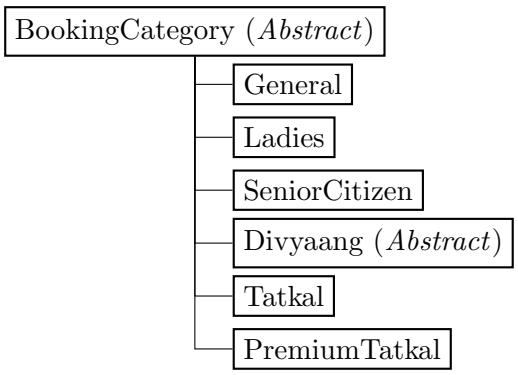
- `static const double sLoadFactor, static const bool sIsLuxury, static const double sReservationCharge, static const double sTatkalLoadFactor, static const double sMinTatkalCharge, static const double sMaxTatkalCharge, static const int sMinTatkalDistance` : These are changeable static constants, hence they are intialized in the application space.
- `BookingClassType(const string& name = BookingClassType<T>::sName)` : Constructor.
- `~BookingClassType()` : Destructor.

7.2.2 public member functions

- `static const BookingClassType<T>& Type()` : Function to return the singleton object.
- All the pure virtual functions are again redefined to make these classes concrete classes.

8 Class and Hierarchy of `BookingCategory`

The class `BookingCategory` and its child classes are modelled using static sub-typing by inclusion and parametric polymorphism.



8.1 class `BookingCategory`

8.1.1 private data members / member functions / operator functions

- `const string name_` : const data member that stores the name of the booking category.

8.1.2 protected member functions

- `BookingCategory(const string& name)` : Constructor.
- `virtual ~BookingCategory()` : Virtual Destructor for a polymorphic hierarchy.

8.1.3 public member functions

- `const string& GetName() const` : Function to get the name of the `BookingCategory`.
- `virtual bool IsEligible(Passenger&) const = 0` : Pure virtual function, it serves to check the eligibility of a person in the leaf classes (booking categories).

- `friend ostream& operator<<(ostream&, const BookingCategory&)` : Output Streaming Operator.

Now, `BookingCategory` has 5 concerete singleton child classes - `General, Ladies, SeniorCitizen, Tatkal, PremiumTatkal` modelled as templates using the name `BookingCategoryType<T>`. Also, `BookingCategory` has an abstract child class `Divyaang` which has its own hierarchy as described in the next section.

8.2 The 5 concrete Booking Categories denoted by the template `BookingCategoryType<T>`

8.2.1 private data members / member functions / operator functions

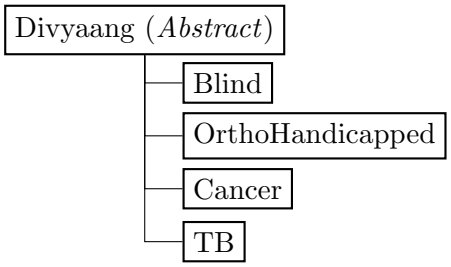
- `static const string sName` : static constant to set the name of the booking class.
- `BookingCategoryType(const string& name = BookingCategoryType<T>::sName)` : Constructor.
- `~BookingCategoryType()` : Destructor.

8.2.2 public member functions

- `static const BookingCategoryType<T>& Type()` : Function to return the singleton object.
- `bool IsEligible(Passenger&) const` : Implementing the pure virtual function defined in `BookingCategory`.
- `Booking* CreateBooking(...)` : A function made in accordance with the virtual construction idiom to invoke the correct constructor of the `Booking` hierarchy in accordance with the class in the `BookingCategory` hierarchy.

9 Class and Hierarchy of `Divyaang`

`Divyaang` is just an abstract class derived from the class `BookingCategory`.
It has 4 concerete singleton child classes - `Blind, OrthoHandicapped, Cancer, TB` , each standing for a disability type.



9.1 class `Divyaang`

9.1.1 public member functions

- `friend ostream& operator<<(ostream&, const Divyaang&)` : Output Streaming Operator.

9.2 The 4 concrete Divyaang Categories (Disability Types) denoted by the template `DivyaangType<T>`

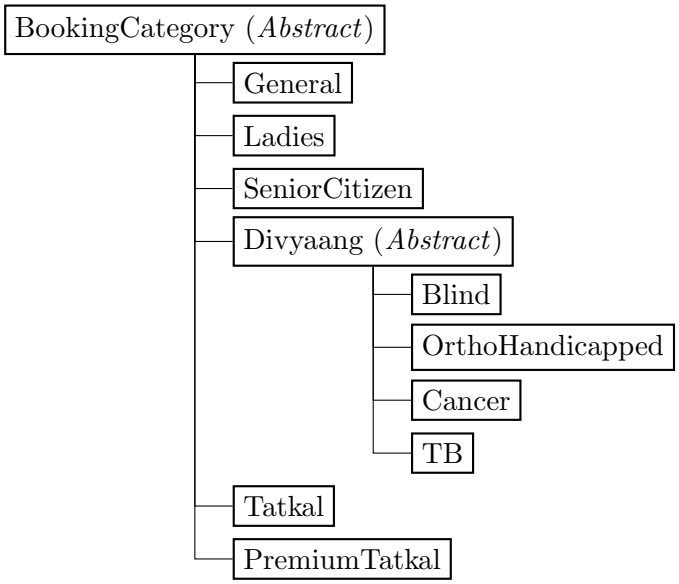
9.2.1 private data members / member functions / operator functions

- `static const string sName` : static constant to set the name of the booking class.
- `DivyaangType(const string& name = DivyaangType<T>::sName)` : Constructor.
- `~DivyaangType()` : Destructor.

9.2.2 public member functions

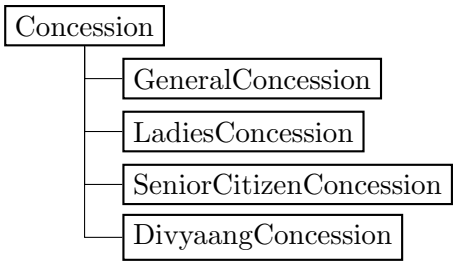
- `static const DivyaangType<T>& Type()` : Function to return the singleton object.
- `bool IsEligible(Passenger&) const` : Implementing the pure virtual function defined in `BookingCategory`.
- `Booking* CreateBooking(...)` : A function made in accordance with the virtual construction idiom to invoke the correct constructor of the `Booking` hierarchy in accordance with the class in the `BookingCategory` hierarchy.

Thus, combining the `Divyaang` hierarchy with the `BookingCategory` hieracrchy, the final hierarchy of booking categories looks like this :



10 Class and Hierarchy of `Concession`

The `Concession` hierarchy has been designed to store the data of the various concession factors for various booking categories.
This child classes of this hierarchy have very less properties in common, hence this is modelled using ad-hoc polymorphism.
The class `Concession` is the base class and it has 4 singleton child classes - `GeneralConcession, LadiesConcession, SeniorCitizenConcession, DivyaangConcession`



10.1 class `Concession`

10.1.1 private data members

- `const string name_` : const string variable to store the name of the `Concession` class.

10.1.2 protected data members

- `Concession(const string&)` : Constructor
- `~Concession()` : Destructor

10.2 class `GeneralConcession`

10.2.1 private data members / member functions

- `static const double sFactor` : static constant to store the concession factor for the General category. This is currently 0.
- `GeneralConcession(const string&)` : Constructor
- `~GeneralConcession()` : Destructor

10.2.2 public member functions

- `static const GeneralConcession& Type()` : Function to return the singleton object.
- `double GetFactor()` : Function to return the concession factor.
- `friend ostream& operator<<(ostream&, const GeneralConcession&)`: Output Streaming Operator.

10.3 class `LadiesConcession`

10.3.1 private data members / member functions

- `static const double sFactor` : static constant to store the concession factor for the Ladies category. This is currently 0.
- `LadiesConcession(const string&)` : Constructor
- `~LadiesConcession()` : Destructor

10.3.2 public member functions

- `static const LadiesConcession& Type()` : Function to return the singleton object.
- `double GetFactor()` : Function to return the concession factor.
- `friend ostream& operator<<(ostream&, const LadiesConcession&)`: Output Streaming Operator.

10.4 class `SeniorCitizenConcession`

10.4.1 private data members / member functions

- `static const double sFactorMale` : static constant to store the concession factor for a male senior citizen.
- `static const double sFactorFemale` : static constant to store the concession factor for a female senior citizen.
- `SeniorCitizenConcession(const string&)` : Constructor
- `~SeniorCitizenConcession()` : Destructor

10.4.2 public member functions

- `static const SeniorCitizenConcession& Type()` : Function to return the singleton object.
- `double GetFactor(Passenger&)` : Function to return the concession factor based on the `Gender` of the `Passenger`.
- `friend ostream& operator<<(ostream&, const SeniorCitizenConcession&)`: Output Streaming Operator.

10.5 class `DivyaangConcession`

10.5.1 private data members / member functions

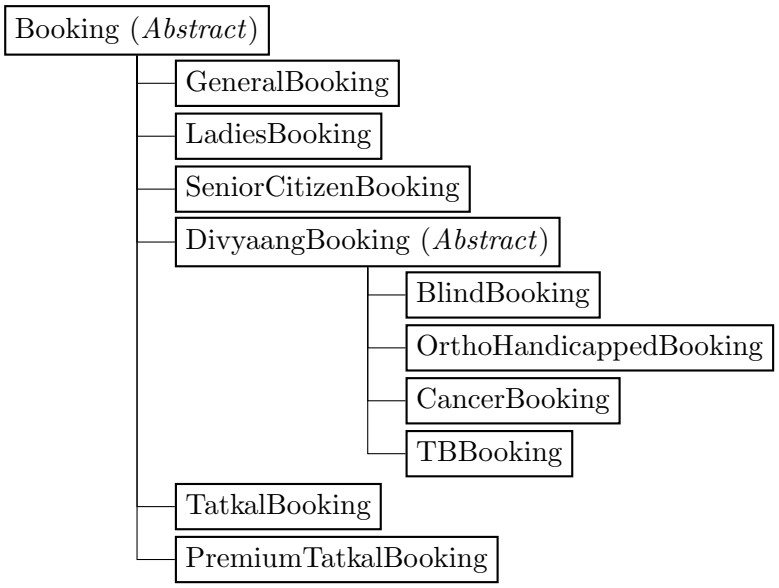
- `static const map<pair<BookingClass*, Divyaang*>, double> sFactor` : The concession factor here also depends on the `BookingClass` and `Divyaang` category.
- `DivyaangConcession(const string&)` : Constructor
- `~DivyaangConcession()` : Destructor

10.5.2 public member functions

- `static const DivyaangConcession& Type()` : Function to return the singleton object.
- `double GetFactor(BookingClass&, Divyaang&)` : Function to return the concession factor.
- `friend ostream& operator<<(ostream&, const DivyaangConcession&)`: Output Streaming Operator.

11 Class and Hierarchy of `Booking`

The class `Booking` and its child classes are modelled using static sub-typing by inclusion and parametric polymorphism and they exactly mirror the `BookingCategory` hierarchy.



11.1 class `Booking`

11.1.1 private data members

- `static int sBookingPNRSerial` : A static variable to keep track of the next `PNR_` to be allocated. It is made static as we need a single copy of this for all the Booking objects. Hence it cannot be object-specific. It is not made const as its value increases by one every time a new Booking is made.

11.1.2 protected data members / member functions

- `const Station fromStation_` : Made const as once the booking is made, the station from which the booking is done will never change.
- `const Station& toStation_` : Made const as once the booking is made, the station upto which the booking is done will never change.
- `const Date dateOfBooking_` : This is also made const as date of travel will not change once the booking has been done.
- `const BookingClass& bookingClass_` : Made const as the booking class will not change after the booking. It has been made a reference because all booking classes are singleton classes, hence there is only one instance (object) for each of them. Thus, whenever we need to use a booking class object, we will have to use a reference to that single instance.
- `const BookingCategory& bookingCategory_` : Similar to `bookingClass_`, this is also made const.
- `Passenger passenger_` : `Passenger` information, as a reference to the object.

- `int fare_` : The fare computed for the booking using the `ComputeFare()` method.
- `const int PNR_` : Made const as the `PNR_` is kind of a unique ID for each booking, and hence will always remain constant for the booking. It is not made const as its value increases by one every time a new booking is made.
- `Booking(const Booking&)` : Blocked copy constructor.
- `Booking& operator=(const Booking&)` : Blocked copy assignment operator.
- `Booking(...)` : Constructor.
- `~Booking()` : Virtual Destructor for a polymorphic hierarchy.

11.1.3 public data members / member functions

- `static const double sBaseFarePerKM` : static constant that is initialized from the application space.
- `static vector<Booking*> sBookings` : This is a vector of pointers to objects of the `Booking` class. This is basically a list of all the Bookings made till now. It is made static as we need a single copy of the list irrespective of the objects. It is however not const as it gets updated whenever a new Booking is made.
- `friend ostream& operator<<(ostream&, const Booking&)` : Output streaming operator.
- `Booking* ReserveBooking(...)` : Function which is called from the application space to create a booking. This, in accordance with the virtual construction idiom calls the appropriate `CreateBooking(...)` function of the `BookingCategory` hierarchy.
- `virtual int ComputeFare() const = 0` : Pure virtual function which implements the fare computation logic.

Similar to the `BookingCategory` hierarchy, `Booking` has 5 concrete child classes - `GeneralBooking, LadiesBooking, SeniorCitizenBooking, TatkalBooking, PremiumTatkalBooking` modelled as templates using the name `BookingCategoryType<T>`.
Also, `Booking` has an abstract child class `DivyaangBooking`.
`DivyaangBooking` further has 4 concrete child classes - `BlindBooking, OrthoHandicappedBooking, CancerBooking, TBBooking`.
All the concrete leaf classes have the following member functions : are singleton classes and implement the `ComputeFare()` function.

- Constructor.
- Destructor.
- `Type()` - To get the singleton object.
- `int ComputeFare() const` - Implements the fare computation logic.

12 Class and Hierarchy of `Exceptions`