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

Software Engineering: CS20006

Assignment – 3: Inheritance Hierarchy, Design, Analysis & Testing *Marks: 100*

Assign Date: *15th February, 2021* Submit Date: *23:55, 4th March, 2021*

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

1 Specification

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

1.1 Requirement Statement

The entities involved in the booking system design include:

- Station: Every Station is identified by its name. Booking is done between any two Stations.
- Railways: It is the Indian railways. It has a collection of Stations with pairwise distance between Stations known a priori. Naturally, there can be only one Railways, called IndianRailways, in the system.
- Date: Any valid date in dd/MMM/yy format.
- BookingClass: There are several BookingClasses for travel (as in Indian Railways fare classes explained). Each BookingClass has the following attributes:
  - *Name*: Name of the BookingClass
  - *Fare Load Factor*: The factor by which the fare for travel by this BookingClass would be loaded over the base fare. This may change from time to time.
  - *Seat / Berth*: Whether the BookingClass provides sleeping berths or just seats. This will not change in future.
  - *AC / Non-AC*: Whether BookingClass is air-conditioned or otherwise. This will not change in future.
  - *# of Tiers*: How many tiers exist in the coach for this BookingClass. This will not change in future.
  - *Luxury / Ordinary*: Whether this BookingClass is considered luxurious by the Government. This may change from time to time.

  New booking classes may be added in future.
- Booking: A Booking is requested with the following information:
  - *fromStation*: Station from which the travel starts for the Booking. This is given by the name of the Station
  - *toStation*: Station at which the travel ends for the Booking. This is given by the name of the Station
  - *date*: Date of travel for the Booking
  - *bookingClass*: BookingClass for the Booking
  - *passenger*: Details of the passenger including name, aadhaar number, date of birth, gender, mobile number, and category of the passenger. *This is for future extension and optional for now.*

  On request of a Booking, the same is processed and fare is computed based on the business logic given in Section 1.3. The Booking is then confirmed with *PNR* and other details on the output. *PNR* is serially allocated starting with 1.
- Passenger: A Passenger may have the following details:
  - *name*: Name of the passenger

- *aadhaar #*: Aadhaar Number to be used as a unique ID
- *dateOfBirth*: Date of birth to be used for verification of age
- *gender*: Gender of the passenger: *male* or *female*
- *mobile #*: Mobile number (optional)
- *category*: One of *General*, *Ladies*, *Senior Citizen*, *Divyaang*, *Tatkal*, *Premium Tatkal*

1.2 Assumptions

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

- IndianRailways has a given set of Stations with distances known a priori. The list of Stations and distances between them are given as Master Data in Section 1.4. No new station can be added to the IndianRailways and distance between pair of stations do not change.
- A Booking, as requested, is always available - between any pair of Stations, on any Date, and for any BookingClass
- No passenger information is considered for the Booking

1.3 Business Logic

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

- *Base Fare Rate*: The base fare for every KM of travel = Rs. 0.5. This may change from time to time.
- *Base Fare*: The base fare between two stations is computed by multiplying the distance between the stations with the base fare for every KM of travel. The base fare applies to the *Sleeper* booking class.
- *Loaded Fare*: For booking classes other the *Sleeper*, the fare is loaded by a factor with respect to the *Sleeper* booking class fare as shown in the *Booking Class Matrix* (Section 1.4.2). The load factor may change from time to time.
- *AC Surcharge*: Further, for air-conditioned classes, AC surcharge of Rs. 50 will be charged on the *loaded fare*. This may change from time to time.
- *Luxury Tax*: Finally, there is a 25% luxury tax to be imposed for all luxury class bookings on the fare computed with surcharge. This may change from time to time. The luxury classification as well as taxation rate may change from time to time.
- Final fare is rounded to the nearest integer.
- *Date* has no effect on the fare.
- *Passenger* has no effect on the fare as it is being ignored for now.

1.3.1 Example

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

By *AC3Tier*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km =
- Loaded fare for *AC3Tier* = Rs. 723.50 * 1.75 = Rs. 1266.125
- After adding the AC surcharge, we get Rs. 1266.125 + Rs. 50 = Rs. 1316.125 = Rs. 1316/= (rounded)

By *ACFirstClass*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km =
- Loaded fare for *ACFirstClass* = Rs. 723.50 * 3.0 = Rs. 2170.50
- After adding the AC surcharge, we get Rs. 2170.50 + Rs. 50 = Rs. 2220.50
- Finally, we levy the luxury tax to get Rs. 2220.50 * 1.25 = Rs. 2775.625 = Rs. 2776/= (rounded)

1.4 Master Data

1.4.1 Stations

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

Station Distance Matrix

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

Distance between a pair of stations is symmetric

1.4.2 Booking Classes

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

Booking Class Matrix

Booking Class	*Name*	*Fare Load Factor*	*Seat / Berth*	*AC*	*# of Tiers*	*Luxury / Ordinary*	*Remarks*
ACFirstClass (1A)	**AC First Class**	3.00	Berth	Yes	2	Luxury	AC 2 berth coupe
AC2Tier (2A)	**AC 2 Tier**	2.00	Berth	Yes	2	Ordinary	AC 2 berth inside, 2 berth on side
FirstClass (FC)	**First Class**	2.00	Berth	No	2	Luxury	Non-AC 2 berth coupe
AC3Tier (3A)	**AC 3 Tier**	1.75	Berth	Yes	3	Ordinary	AC 3 berth inside, 2 berth on side
ACChairCar (CC)	**AC Chair Car**	1.25	Seat	Yes	0	Ordinary	AC chairs
Sleeper (SL)	**Sleeper**	1.00	Berth	No	3	Ordinary	Non-AC 3 berth inside, 2 berth on side
SecondSitting (2S)	**Second Sitting**	0.50	Seat	No	0	Ordinary	Bench seating

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

2 Analysis of Specification

We first analyze the specifications to identify the classes and hierarchy for the design. We also try to extract possible constraints on the design.

- Station and Date are simple data classes.
- Class Railways should be a singleton and should contain the master data of stations and distances. The singleton should be constant as no station can be added and distances cannot be changed.

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

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

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

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

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

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

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

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

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

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

3 High Level Design

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

3.1 Design Principles

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

- *Flexible & Extensible Design*
  - The design should be flexible. That is, it should be easy to change the changeable parameters (like base rate, load factor etc.) easily from the Application space. This should should not need re-building of the library of classes.
  - The design should be extensible. That is, it should be easy to add new behaviour (classes) wherever indicated in the specification (like Booking Classes, Booking, Passenger, etc.). This should not require a re-coding of the existing applications.
- *Minimal Design*
  - Only the stated models and behaviour should be coded. No extra class or method should be coded.
  - *Less code, less error* principle to be followed.
- *Reliable Design*
  - Reliability should be a priority. Everything should work as designed and coded.
  - Data members, methods and objects should be made constant wherever possible.
  - Parameters should be appropriately defaulted wherever possible
- *Testable Design*

- Every class should support the output streaming operator for checking intermittent output if needed.
- Every class should be tested with an appropriate test application for its unit functionality (Section 6.1).
- Test Applications (Section 6.2) and regression test suites should be designed for testing the application on (at least) the common scenarios of use.

3.2 Classes

- Class Station HAS-A name.
- Class Railways is a singleton called IndianRailways. It has a collection of the Stations and their mutual distances. IndianRailways is a constant object.
- Class Date is discussed in the lecture modules.
- Class BookingClasses HAS-A loadFactor. Remaining attributes may be encoded on the methods in the hierarchy classes.
- Class Booking HAS-A fromStation, toStation date, and bookingClass from the booking request where every station name, date and booking class are assumed to have been given correctly. Further it HAS-A fare computed and PNR allocated. Optionally, it may HAS-A bookingStatus (which would be true for this assignment always) and bookingMessage (which may be "BOOKING SUCCEEDED" for this assignment always).

  Booking should support Passenger as a null-able parameter for future extension.
- You may add any class, any data member to a class, or any hierarchy as you need for implementation. Justify your design choice for them.

3.3 Interfaces

- Constructors / Destructors: Proper constructor and destructor for every class
- Copy Functions: Provide user-defined Copy Constructor and / or Copy Assignment Operator for a class if used in the design (should not be needed). Otherwise, block them.
- Provide output streaming operator for every class to help output process as well as debugging
- Class Station to have GetName() for accessing its name and GetDistance(.) to get distance to another station.
- Class Railways to have GetDistance(., .) to get distance between a pair of stations. It should also have proper interface for making it a singleton IndianRailways
- Class BookingClasses to have GetLoadFactor(), GetName(), IsSitting(), IsAC(), GetNumberOfTiers(), and IsLuxury() to get access to various BookingClasses properties. Depending on the polymorphic hierarchy, these methods may be non-polymorphic and / or polymorphic (and in some case `pure`) in BookingClasses and its various derived classes. Consider making them `const` methods.
- Class Booking to have ComputeFare() to implement the fare computation logic. Should it be `virtual` (polymorphic) for future extensions?
- You may add any interface to a class (or `private` / `protected` methods) as you need for implementation. Justify your design choice for them.

3.4 Constants

The following should be static constants in appropriate classes:

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

3.5 Statics

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

3.6 Errors & Exceptions

- All Booking requests are taken to be correct. That is, the Staions as mentioned - do exist, the Date is valid (in future), and no invalid BookingClass is requested
- There is no error in input, processing, or output.
- No error or exception handling to be incorporated in the design for this assignment. However, structure the code flow well so that they can be incorporated later with minimal changes (adhering to the need of flexibility).

4 Low-Level Design

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

4.1 Design Principles

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

- *Encapsulation*
  - Maximize encapsulation for every class
  - Use `private` access specifier for all data members that are not needed by derived classes, if any. Use `protected` otherwise.
  - Use `public` access specifier for interface methods and static constants and *friend* functions only.
- *STL Containers*
  - Use STL containers (like `vector`, `map`, `hashmap`, `list`, etc.) and their iterators. Do not use arrays
  - Use iterators for STL containers. Do not use bare `for` loops.
- *Pointers & References*
  - Minimize the use of pointers. Use pointers only if you need null-able entities
  - If you use pointer for dynamically allocated objects (should be minimized), remember to `delete` at an appropriate position.
  - Use `const` reference wherever possible.

4.2 Design of Classes, Data Members & Methods

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

5 Implementation

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

5.1 Basic Coding Guidelines

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

5.2 Code Organization

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

6 Test Plan

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

6.1 Unit Tests

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

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

6.2 Application Test

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

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

In addition, a sample test application for booking is given in Section C with the expected output in Section C.1. Your codes should pass this test application too.

7 Tasks

The following tasks are to be completed for the assignment:

1. **Design**: Complete the HLD and the LLD. Document the salient points from your design in `Design.txt`. Follow the quality guidelines and design principles outlined above.
2. **Implementation**: Implement the LLD in C++ following the basic coding guidelines (Section A).
3. **Test Planning**: Write a unit test and application test plan in `Testplan.txt` covering all scenarios. Note that no wrong input or erroneous data situation is to be handled. For example, a Date specified will always be valid. So plan tests based only on correct input data.
4. **Testing**: Implement unit test and application test codes and perform testing. For application testing, test with the application given in Section C as well as the application developed by you from the testplan.
5. **Bundle and Submissions**: Name and bundle your files as given in Section 8 and submit to Moodle.

8 Submission of Files

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

1. `Documents.zip`
   (a) `Design.txt`: The design document stating the design details (especially LLD) with principles and guidelines followed

(b) `Testplan.txt`: The testplan document stating scenarios for unit tests (with golden output if needed) and the scenarios of the test application.

2. `Source.zip`:

   (a) Source (`.cpp`) and header (`.h`) files for implementation.

   (b) Source (`.cpp`) and header (`.h`) files for test application.

   (c) README file that describes the contents of every file in the `Source.zip`. Also, mention the compiler (with version, and compiler options, if any) that you have used.

3. `Outputs.zip`

   (a) Output from the given test application (Section C)

   (b) Output from the your test application developed from the test plan

   - The output file can be generated by redirecting the output to a text file or by copy-paste from the console in a text file.
   - There is no need to include the `a.out` file.

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

9 Marks

The marks are distributed as follows:

Design		**[20]**
Breakup		
Non-static & static data members	*[4]*	
Non-static[1] & static member functions signatures	*[4]*	
friend function signatures	*[2]*	
Design of BookingClasses Hierarchy	*[10]*	
Implementation		**[25]**
Breakup		
Non-static & static member functions	*[15]*	
Static data members	*[5]*	
friend function	*[5]*	
Test Planning		**[20]**
Breakup		
Unit Test Scenarios & Goldens (Completeness of scenarios)	*[15]*	
Application Test Scenarios	*[5]*	
Testing		**[15]**
Breakup		
Unit Test Application (adherence to test plan)	*[7]*	
Own Test Application (adherence to test plan)		
Output	*[5]*	
On given Test Application (Section C)		
Output	*[3]*	
Quality of Design & Implementation		**[20]**
Breakup		
Adherence to Design Protocols		
Singletons	*[3]*	
const-ness	*[3]*	
Coding Guidelines	*[5]*	
Extensibility & Flexibity	*[4]*	
Code Comments	*[5]*	

1 Non-static include non-polymorphic as well as polymorphic member functions

A Coding Guidelines

It is advised to follow the guidelines below while coding:

- Use CamelCase for naming variables, classes, types and functions
- Every name should be indicative of its semantics
- Start every variable with a lower case letter
- Start every function and class with an upper case letter
- Use a trailing underscore (_) for every non-static data member
- Use a leading 's' for every static data member
- Do not use any global variable or function (except `main()`, and `friends`)
- No constant value should be written within the code - should be put in the application as static
- Prefer to pass parameters by value for build-in type and by const reference for UDT
- Every polymorphic hierarchy must provide a `virtual` destructor in the base class
- Prefer C++ style casting (like `static_cast<int>(x)` over C Style casting (like `(int)`)
- The project should compile without any compiler warning
- Indent code properly
- Comment the code liberally and meaningfully
- Adopt more guidelines as you prefer. Try to document them

B Unit Testing Fraction Class

As an example of unit test, let us consider the `Fraction` class we have developed in Assignment 2. We illustrate the test for its one overloaded constructor (`Fraction(int = 1, int = 1)`) and `operator+` only. For this we enumerate the different possible cases to test in a unit test plan.

B.1 Unit Test Plan for `Fraction`

We elucidate the unit test plan for constructor and add operator.

B.1.1 Test Scenarios for Construction of Objects

We consider the `Fraction(int = 1, int = 1)` constructor. The scenarios (including normalization, sign handling, and default) are:

- Normalization
  1. Improper fraction in reduced form
  2. Improper fraction in irreduced form
  3. Proper fraction in reduced form
  4. Proper fraction in irreduced form
  5. Fraction 0 with arbitrary denominator
- Sign handling
  1. Fraction with negative numerator
  2. Fraction with negative denominator
  3. Fraction with negative numerator & denominator
- Default parameters
  1. Fraction with only numerator
  2. Fraction with no parameter

B.1.2 Test Scenarios for Addition Operator

We consider the overloaded add operator `friend Fraction operator+(const Fraction&, const Fraction&)`. The scenarios are (considering the given constructor):

1. Add two fractions
2. Add a fraction with an integer
3. Add an integer with a fraction

Rest of the `Fraction` class can be tested by preparing a similar plan.

B.2 Unit Test Implementation for `Fraction`

For unit testing, we write a static function in the class that has this test code. In the application, we use the 'golden output' for every test and compare for equality. If the expected output is not obtained, a message on test error is printed.

B.2.1 `Fraction` Class Code

Here is the relevant parts of the class including the static test function signature

```
#ifndef __FRACTION_HXX// Control inclusion of header files
#define __FRACTION_HXX

/************ C++ Headers ************************************/
#include <iostream>// Defines istream & ostream for IO
using namespace std;
```

```
/************ CLASS Declaration ******************************/
class Fraction {

public:

// CONSTRUCTORS
// ------------
Fraction(int = 1, int = 1);// Uses default parameters. Overloads to
// Fraction(int, int);
// Fraction(int);
// Fraction();

// BINARY ARITHMETIC OPERATORS USING FRIEND FUNCTIONS
// --------------------------------------------------
friend Fraction operator+(const Fraction&, const Fraction&);

// Other member functions, static functions, friend functions

// ...

// STATIC UNIT TEST FUNCTION
// ------------------------
static void UnitTestFraction(); // Test application for Fraction

// Data members

// ...
#endif // __FRACTION_HXX
```

B.2.2 `Fraction` Class Unit Test Application Code

```
// To unit test class Fraction
void Fraction::UnitTestFraction() {
    // Check difference cases of fraction construction
    Fraction f1(5, 3);   // Improper fraction in reduced form
    Fraction f2(15, 9);  // Improper fraction in irreduced form
    Fraction f3(3, 5);   // Proper fraction in reduced form
    Fraction f4(9, 15);  // Proper fraction in irreduced form
    Fraction f5(0, 2);   // Fraction 0 with arbitrary denominator
    Fraction f6(-2, 3);  // Fraction with negative numerator
    Fraction f7(2, -3);  // Fraction with negative denominator
    Fraction f8(-2, -3); // Fraction with negative numerator & denominator
    Fraction f9(5);      // Fraction with only numerator
    Fraction f10;        // Fraction with no parameter

    // Check if every object is constructed in the desired way
    if (f1.iNumerator_ != 5 || f1.uiDenominator_ != 3)    // Check members
        cout << "Fraction Consturction Error on Fraction(5, 3)" << endl;

    if (f2.iNumerator_ != 5 || f2.uiDenominator_ != 3) // Check members & reduction
        cout << "Fraction Consturction Error on Fraction(15, 9)" << endl;

    if (f3.iNumerator_ != 3 || f3.uiDenominator_ != 5) // Check members
        cout << "Fraction Consturction Error on Fraction(3, 5)" << endl;

    if (f4.iNumerator_ != 3 || f4.uiDenominator_ != 5) // Check members & reduce
        cout << "Fraction Consturction Error on Fraction(9, 15)" << endl;

    if (f5.iNumerator_ != 0 || f5.uiDenominator_ != 1) // Check members with denominator = 1
        cout << "Fraction Consturction Error on Fraction(0, 2)" << endl;
```

```
    if (f6.iNumerator_ != -2 || f6.uiDenominator_ != 3) // Check members
        cout << "Fraction Consturction Error on Fraction(-2, 3)" << endl;

    if (f7.iNumerator_ != -2 || f7.uiDenominator_ != 3) // Check members & sign flip
        cout << "Fraction Consturction Error on Fraction(2, -3)" << endl;

    if (f8.iNumerator_ != 2 || f8.uiDenominator_ != 3) // Check members & sign flip
        cout << "Fraction Consturction Error on Fraction(-2, -3)" << endl;

    if (f9.iNumerator_ != 5 || f9.uiDenominator_ != 1) // Check default on second parameter
        cout << "Fraction Consturction Error on Fraction(5)" << endl;

    if (f10.iNumerator_ != 1 || f10.uiDenominator_ != 1) // Check default on both parameters
        cout << "Fraction Consturction Error on Fraction" << endl;

    // Check addition of two fractions
    Fraction f11 = f1 + f3; // Add two fractions
    Fraction f12 = f1 + 1;  // Add a fraction with an integer
    Fraction f13 = 1 + f3;  // Add an integer with a fraction

    if (f11.iNumerator_ != 34 || f11.uiDenominator_ != 15) // Check members on add
        cout << "Fraction Addition Error on Fraction(5, 3) + Fraction(3, 5)" << endl;

    if (f12.iNumerator_ != 8 || f12.uiDenominator_ != 3) // Check members on add
        cout << "Fraction Addition Error on Fraction(5, 3) + 1" << endl;

    if (f13.iNumerator_ != 8 || f13.uiDenominator_ != 5) // Check members on add
        cout << "Fraction Addition Error on 1 + Fraction(3, 5)" << endl;

    return;
}
```

C Test Application for Booking

```
// Test application for booking
void BookingApplication() {

    // Bookings by different booking classes

    // <BookingClasses>::Type() returns the constant object of the respective type
    Booking b1(Station("Mumbai"), Station("Delhi"), Date(15, 2, 2021), ACFirstClass::Type());
    Booking b2(Station("Kolkata"), Station("Delhi"), Date(5, 3, 2021), AC2Tier::Type());
    Booking b3(Station("Mumbai"), Station("Kolkata"), Date(17, 3, 2021), FirstClass::Type());
    Booking b4(Station("Mumbai"), Station("Delhi"), Date(23, 3, 2021), AC3Tier::Type());
    Booking b5(Station("Chennai"), Station("Delhi"), Date(25, 4, 2021), ACChairCar::Type());
    Booking b6(Station("Chennai"), Station("Kolkata"), Date(7, 5, 2021), Sleeper::Type());
    Booking b7(Station("Mumbai"), Station("Delhi"), Date(19, 5, 2021), SecondSitting::Type());
    Booking b8(Station("Delhi"), Station("Mumbai"), Date(22, 5, 2021), SecondSitting::Type());

    // Output the bookings done where sBookings is the collection of bookings done
    vector<Booking*>::iterator it;
    for (it = Booking::sBookings.begin(); it < Booking::sBookings.end(); ++it) {
        cout << *(*it);
    }

    return;
}

int main() {

    BookingApplication();

    return 0;
}
```

Your implementation of classes needs to compile with the above application and output details of every booking done. A sample output could look as follows. It is not necessary to match every line of the output. But the same information should be available in your output.

C.1 Test Output

```
BOOKING SUCCEEDED:
PNR Number = 1
From Station = Mumbai
To Station = Delhi
Travel Date = 15/Feb/2021
Travel Class = AC First Class
 : Mode: Sleeping
 : Comfort: AC
 : Bunks: 2
 : Luxury: Yes
Fare = 2776

BOOKING SUCCEEDED:
PNR Number = 2
From Station = Kolkata
To Station = Delhi
Travel Date = 5/Mar/2021
Travel Class = AC 2 Tier
 : Mode: Sleeping
 : Comfort: AC
 : Bunks: 2
 : Luxury: No
```

```
Fare = 1522

BOOKING SUCCEEDED:
PNR Number = 3
From Station = Mumbai
To Station = Kolkata
Travel Date = 17/Mar/2021
Travel Class = First Class
 : Mode: Sleeping
 : Comfort: Non-AC
 : Bunks: 2
 : Luxury: Yes
Fare = 2518

BOOKING SUCCEEDED:
PNR Number = 4
From Station = Mumbai
To Station = Delhi
Travel Date = 23/Mar/2021
Travel Class = AC 3 Tier
 : Mode: Sleeping
 : Comfort: AC
 : Bunks: 3
 : Luxury: No
Fare = 1316

BOOKING SUCCEEDED:
PNR Number = 5
From Station = Chennai
To Station = Delhi
Travel Date = 25/Apr/2021
Travel Class = AC Chair Car
 : Mode: Sitting
 : Comfort: AC
 : Bunks: 0
 : Luxury: No
Fare = 1413

BOOKING SUCCEEDED:
PNR Number = 6
From Station = Chennai
To Station = Kolkata
Travel Date = 7/May/2021
Travel Class = Sleeper
 : Mode: Sleeping
 : Comfort: Non-AC
 : Bunks: 3
 : Luxury: No
Fare = 830

BOOKING SUCCEEDED:
PNR Number = 7
From Station = Mumbai
To Station = Delhi
Travel Date = 19/May/2021
Travel Class = Second Sitting
 : Mode: Sitting
 : Comfort: Non-AC
 : Bunks: 0
 : Luxury: No
Fare = 362
```

```
BOOKING SUCCEEDED:
PNR Number = 8
From Station = Delhi
To Station = Mumbai
Travel Date = 22/May/2021
Travel Class = Second Sitting
 : Mode: Sitting
 : Comfort: Non-AC
 : Bunks: 0
 : Luxury: No
Fare = 362
```

The above test application is given as a sample. In addition, you should write your own unit and application tests.

D Clarifications

D.1 HAS-A relationship

A HAS-A relationship simply means having a component of an object (part of object) – it is *Composition* or *Aggregation*. It translates to having data members.
For example,

```
Car HAS_A Registration_Number
Car HAS_A Engine
Car HAS_A Model_Name
Car HAS_A Wheel // 4 wheeler
Car HAS_A Owner

Wheel HAS_A SerialNumber
Wheel HAS_A Make
Wheel HAS_A Radius

Engine HAS_A Number
Engine HAS_A Capacity // in cc
```

is modeled as

```
class Engine {
protected: // Engines could be specialized
    const string engineNumber_; // every engine has a unique number that cannot change
    const int capacity_;        // every engine has a unique capacity that cannot change
};

class Wheel {
protected: // Wheels could be specialized
    const string SerialNumber_; // every wheel has a unique serial number that cannot change
    const string& make_;        // Every wheel has a make that can be from a set of known makers
                                // Hence, reference to static maker names should be used
                                // Make cannot change
    const double radius_;       // Radius cannot change
};

class Car {
protected: // Cars could be specialized
    const string registrationNumber_;   // Registration number cannot change
    const Engine engine_;               // Engine cannot change
    const string& modelName_;           // Every car has a model that can be from a set of
                                        // known models. Hence, reference to static model
                                        // names should be used. Model cannot change
    Wheel wheels_[4];                   // Wheels may change; but must all be the same
                                        // make / radius
    string& ownerName_;                 // Owner can change for re-sale
}
```

We should not use private inheritance for HAS-A.

Note: Besides showing HAS-A, the above example also illustrates different situations for `const`-ness and reference

D.2 Private Inheritance

Note the following from the presentation:

- Private inheritance *means nothing during software design, only during software implementation*
- Private inheritance means *is-implemented-in-terms of.* It is usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions

- It is good for enforcing a policy over a hierarchy. For example the following example of an *uncopyable* base class, which can be used to disable the copy constructor and assignment operator in derived classes:

```
class Uncopyable {
    protected:
        Uncopyable() {}
        ~Uncopyable() {}

    private:
        Uncopyable(const Uncopyable&);
        Uncopyable& operator=(const Uncopyable&);
};

class Derived : private Uncopyable { ... };
```

We should not use private inheritance unless we really need it. It is usually rare for most projects.

D.3 Virtual Destructor

The guidelines have the following:

Every polymorphic hierarchy must provide a virtual destructor in the base class

This has attracted the following observation:

An abstract base classes cannot have a constructor or a destructor. So does this mean that this guideline is redundant?

So let us try to understood an Abstract Base Class better. *It is a base class which has at least one pure virtual function*. **That's it**. *And hence we* **cannot instantiate** *it; but we can* **always construct** *objects of it*.

In any hierarchy, we must first construct the base class object before we can construct the derived class object (reverse for the destructor). *So all classes* **concrete** *or* **abstract**, *must have constructor and destructor (which either you provide or get free from the compiler)*. A class is Abstract means that we cannot have an instantiation for it. But it will always get constructed and destructed as a base part of the derived class.

Finally, if the destructor is not virtual in a polymorphic hierarchy, then the destruction will not get dynamically dispatched to the derived class destructor and slicing will result. Let us look at the example below:

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

class Base {
    const int id_;
    static int sObjId;

protected:
    Base() : id_(sObjId++) { cout << "Base Constructor for object id = " << id_ << "\n"; }

public:
    virtual ~Base();        // Line 1
    //virtual ~Base() = 0;  // Line 2

    virtual string GetName() const = 0; // Pure virtual function

    int GetId() const { return id_; }
};

Base::~Base() { cout << "Base Destructor for object id = " << id_ << "\n"; }
```

```
class DerivedOne : public Base {
    const string name_;
public:
    DerivedOne(const string& name) : Base(), name_(name) { cout << "DerivedOne Constructor\n"; }
    ~DerivedOne() { cout << "DerivedOne Destructor\n"; }

    string GetName() const {
        return "DerivedOne: " + name_ + "\n";
    }
};

class DerivedTwo : public Base {
    const string name_;
public:
    DerivedTwo(const string& name) : Base(), name_(name) { cout << "DerivedTwo Constructor\n"; }
    ~DerivedTwo() { cout << "DerivedTwo Destructor\n"; }

    string GetName() const {
        return "DerivedTwo: " + name_ + "\n";
    }
};

int Base::sObjId = 0;

int main() {
    //Base b; // Line 3: Does not compile - Base is abstract

    const Base *p = new DerivedOne("obj1");
    cout << p->GetName();

    const Base *q = new DerivedTwo("obj2");
    cout << q->GetName();

    delete q;
    delete p;

    return 0;
}
```

`class Base` is abstract due to `GetName()` and we cannot instantiate (un-commenting `Line 3` is an error). But it has constructor and destructor both. So, from the above code we get the following output as expected.

```
Base Constructor for object id = 0
DerivedOne Constructor
DerivedOne: obj1
Base Constructor for object id = 1
DerivedTwo Constructor
DerivedTwo: obj2
DerivedTwo Destructor
Base Destructor for object id = 1
DerivedOne Destructor
Base Destructor for object id = 0
```

Now, in `Line 1`, remove the word `virtual`, and the output changes to:

```
Base Constructor for object id = 0
DerivedOne Constructor
DerivedOne: obj1
Base Constructor for object id = 1
DerivedTwo Constructor
DerivedTwo: obj2
```

```
Base Destructor for object id = 1
Base Destructor for object id = 0
```

Clearly, the destructors of `DerivedOne` and `DerivedTwo` are not getting called as the destructor of `Base` is non-polymorphic. This is called **slicing**, as we just chop off the the head of the object leaving the torso behind. Interestingly, we can make the destructor itself pure virtual as well in (comment `Line 1` and un-comment `Line 2`), and still provide its implementation, and we will get correct output as above. Often that is the default way around to start a polymorphic hierarchy:

```
class Base {
public:
    virtual ~Base() = 0;
};

// Derived classes ...
```

Hence, **Every polymorphic hierarchy must provide a virtual destructor in the base class**

D.4 Source Header File and Library Conventions

D.4.1 Source File

Every C++ (C) source file should have `.cpp` (`.c`) extension. Many compilers decide the language rules to apply based on the file extension.

Normally, in a C++ project, even *pure* C code can be written appropriately for a C++ compiler (check the header file conventions for that). So `.c` extension may not be used at all. These were needed earlier (over a decade or more earlier) when C++/C mix in a project was needed to be handled separately either by C compilation (`.c`) or by C++ compilation (`.cpp`). In fact, some compilers also had a support for `.cxx` where the compiler automatically detected whether to compile by C or by C++.

So, we shall use `.cpp` only.

D.4.2 Header File

Every header file (C++ or C) should be `.h` because we should write C in C++ compilable manner. However, since a header file is included in some source/s, its extension really does not matter to the compiler. It is more for the readability of the program.

Historically, `.hpp` extension was used by some companies to mean a C++ header and distinguish from a C `.h` header. These are legacy now. Some companies, in keeping with `.cxx`, used `.hxx` too.

So, we shall use `.h` only.

Once-only Inclusion of Header Files: Enclose every project header file with CPP guards as follows:

```
// Start of "Station.h"
#ifndef _STATION_H
#define _STATION_H

// Header file codes

#endif //_STATION_H
// End of "Station.h"

// Start of "Railways.h"
#ifndef _RAILWAYS_H
#define _RAILWAYS_H

#include "Station.h"

// Header file codes
```

```
#endif //_RAILWAYS_H
// End of "Railways.h"
```

Now when we include, say in, `Booking.cpp` as:

```
// Start of Booking.cpp
#include "Station.h"
#include "Railways.h"

// Source file codes

// End of Booking.cpp
```

the second inclusion of `Station.h` through `Railways.h` is excluded by the CPP guards. Otehrwise, we shall face re-definition errors.

In the C and C++ programming languages, `#pragma once` is a non-standard but widely supported pre-processor directive designed to cause the current source file to be included only once in a single compilation. It is better to avoid it (and use the scheme given above) for better portability.

D.4.3 Include Library

We have to be, however, more careful when we use the include library names as explained below:

- **C++ Standard Library Headers**: Include a C++ standard library header by name within angle brackets (<>):

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

  All symbols are within the `std` namespace.

  With angle brackets (<>), the header will be searched in the C++ standard library path defined in the system.

- **C Standard Library Headers**: In a C++ source, include a a C standard library header by name within angle brackets after prefixing it with `c` and dropping the `.h`. So `stdio.h` is included as:

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

  All symbols are again within the `std` namespace.

  With angle brackets (<>), the header will be searched in the C++ / C standard library path defined in the system. `cstdio` is actually the C++ version of `stdio.h`.

  In a *pure* C source (that will be compiled as C), include a C standard library header by name (with `.h` extension) within angle brackets (<>). So `stdio.h` is included as:

  ```
  #include <stdio.h>
  ```

  All symbols are within the global namespace.

  With angle brackets (<>), the header will be searched in the C standard library path defined in the system.

  Never include a C standard library header in C++ as the `.h` version

- **Project Headers**: Include a project header by name within a pair of double quotes (`""`):

```
#include "Railways.h"
```

The symbols will be in the global namespace or the namespaces defined in the project headers.

With double quotes (`""`), the header will be searched in the project path defined for the project.

D.5 `static`, `const`, Reference & Singleton

- What should be Singleton?

  There are several singletons on the project including the `IndianRailways`, `BookingClasses` etc.

  It would be best to implement them as Meyer's singleton (example is given in Section D.6.3) as no life-time management will be needed. In the solution discussed in the class, we create the singleton dynamically. So the responsibility to release it remains with programmer before `main()` terminates. With a number of singletons in the project, it is very likely to miss the release and the resources would leak (the singleton may be holding some system resource like database access).

- What should be the access specifier?

  Always use `private` for data members, `protected` for data members needed by the inherited classes, and `public` otherwise. Constructor and Destructor should be `private` for singletons.

- What should be `static`?

  Any object that should be available from the beginning to the end of the program, should be a `static`. Examples are collection of `Stations`, distances between `Stations` etc.

- What should be `const`? A value / object, by default, should be `const` unless it needs to change during the computation.

  - What should be `const` value member?

    A value / object that cannot change after the construction of an object. Like `name_` of a `Station` etc. It is constructed / destructed with the object.
  - What should be `const` reference member?

    A value / object that cannot change after the construction of an object and refers to an object that is defined (constructed / destructed) by some other object already.
  - What should be `const` member function?

    By default every member function should be `const`. Make it non-const if it is needed to change values and / or consider `mutable` to selectively change only a few data members in an otherwise `const` object.

  `const`-ness should be treat more semantically than syntactially. For more details check Section D.6.3

- What should be the access specification?

  - `private` / `protected`: All data members, static or non-static, should be `protected` for a base class and `private` otherwise. Constructor / Destructor etc. should be `protected` for a singleton that is a base class, and `private` otherwise.
  - `public`: All interface methods need to be `public`.

D.6 A Few Simple Design Principles

D.6.1 Class Members

- For items like `name_` (or `loadFactor_`), it is better to use simple `string` (or `double`). Define a new class when it has behaviour, validation etc. Like `Date` (dd/mmm/yy: 01/Mar/21 or dd/mm/yy: 01/03/21).

  In the same line, we should make `name_` a private member of, say, `class Station` like `const string name_;`

- Yes, it is good to block copy functions (by providing them in private) if you do not need them so that in case they are used, you will get a compilation error. Note that in most contexts you should be using call-/return-by-reference for UDTs (and call-/return-by-value for built-in types) and hence you would rarely need copying. Define properly when you need.

D.6.2 Hierarchy Design

It is time to vaccinate for COVID-19. Let us look at the priority rules[1] for vaccination to decide on a model of the population for a vaccination schedule allocation application. In decreasing order of priority, vaccine should be administered too:

1. Healthcare Workers (HCWs)
2. Frontline Workers (FLWs). FLW is a generalization of HCW
3. Senior Citizens (SCs), that is age > 59 years
4. Person with Co-Morbidity (PCMs), where age > 44 years and has one of the following Co-Morbidity
   - Diabetes
   - Heart failure with hospital admission in the past one year
   - Post-cardiac transplant
   - Valvular heart disease
   - End-stage kidney disease on haemodialysis
   - Severe respiratory disease with hospitalisation in the last two years
   - Primary immunodeficiency diseases
   - HIV infection
   - Angina and hypertension/diabetes on treatment
   - ...
5. More conditions of Phase 3 ...

Looking at the above, we can identify the following attributes:

1. **Nature of Work**: *Non-Frontline*, *Frontline*, and *Healthcare*
2. **Age**: *Below 45*, *Between 45 and 59*, and *Above 59*
3. **Co-Morbidity**: Some 20 odd types. For simplicity we take this as Boolean: *Non-Co-Morbid* and *Co-Morbid*
4. **Gender**: *Female* and *Male*. It is natural to have some criteria in future on this.
5. **State**: *States on India*. It is natural to have some criteria in future on this.

It is easy to see that the attributes will be available in the population is a totally mixed manner and in total we have $3 \times 3 \times 2 \times 2 = 36$ possible combination classes (ignoring the explosive possibility of States). Question is – how do we put them in a hierarchy? Naturally, every class will multiply inherited these properties in some path or other. That would not be a hierarchical design, it will be an anarchy.

So we decide to *rank the attributes by their uniqueness*. Model single inheritance on one or two of them and keep the rest as pure attribute values to be taken care of in the member function algorithms. For example, if we consider age as the most priority attribute (as we do in many population-related applications), we shall end up in a chaos as HCW, FLW, and Non-FLW would all exist in all ages. So age is a bad choice as a primary.

Hence, for a proper hierarchy, we should look at the natural flow of the model. Here, we can easily see that HCW in (1) is most specialized, FLW in (2) is the next, and (3) onwards it is the non-FLW. So Nature of Work should be the primary hierarchy parameter. Next, we see that other attributes have no effect on HCW and FLW. So only non-FLW is to be classified further. Here (3) predominates the age rule. Finally the Co-Morbidity takes over. Going by this principle from the analysis, we get 9 concrete classes and 3 levels.

[1] These are sample, incomplete, inaccurate rules not to be used in real COVID vaccination decision making

```
// General population
// Abstract Base Class
class Person { };                       // (1) to (5)

// Frontline Workers
class FLW : public Person { };          // (2)

// Non-Frontline Workers
class nonFLW : public Person { };       // (3) to (5)

// Healthcare Workers
class HCW : public FLW { };             // (1)

// Senior Citizen. Age > 59
class SC : public nonFLW { };           // (3)

// Mid-Age Citizen. Age < 59 & > 44
class MC : public nonFLW { };           // (4) to (5)

// Young Citizen. Age < 45
class YC : public nonFLW { };           // (5)

// Co-Morbidity
class CM : public MC { };               // (4)

// NonCo-Morbidity
class nonCM : public MC { };            // (5)
```

We can make it more compact, if we remove `class CM` and `class nonCM` from the hierarchy and just treat co-morbidity as a boolean attribute. This gives us 7 concrete classes and 2 levels. Naturally, gender and state both are also treated as attributes and we do not consider them for the hierarchy design.

So, the thumb rule would be to rank the attributes for the computation model and use one or two of them to create a two or three level hierarchy (ISA). Rest of the attributes should remain as attributes (HAS-A).

D.6.3 Semantics of `const`-ness

We often get confused between different shades of `const`-ness and their treatment in software design. It can feature in a variety of different contexts including:

- **Natural Constants** [`static const` in library]: Like π, e, c, etc.
- **Universal Constants** [`static const` in library]: Like days of the week etc.
- **Design Constants**
  - ***Value Constants***: Values that are constant for
    - Life time of a particular object in a particular execution of the software [`const` data member]: Like Basic Pay of an Employee
    - Life time of a particular object in every execution of the software [`const` data member]: Like Date of Birth of an Employee
    - Life time of all objects in a particular execution of the software [`static const` in application]: Like Month of Pay Computation
    - Life time of all objects in every execution of the software in a period, but can periodically change [`static const` in application]: Like the rate of Dearness Allowance of Employees (changes every six months to a year)
    - Life time of all objects in every execution of the software in a period, but can occasionally change – a change which is likely to software version change [`static const` in library]: Like Pay Scales / Levels of Employees (may change once a five to ten years)

– ***Type Constant*** [const or non-const singleton in code contexts]: Like **Gender** can be of one of two types – *male* and *female*; **Dexterity** can be of one of two types – *left* and *right*; **Verb tenses in English** can be of one of three types – *present*, *past* and *future*, etc. These are type constants that can be coded by representative singleton type objects. These objects stand for the type and may or may not be `const`. Here is a code example for gender.

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

class Gender { const string name_;
protected:
    Gender(const string& name) : name_(name) {}
public:
    virtual const string Print() const = 0;
};

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

        return theObj;
    }
    const string Print() const { return "I am a man"; }
};

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

        return theObj;
    }
    const string Print() const { return "I am a woman"; }
};

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

class Person {
    const string name_;
    const Gender& gender_;
public:
    Person(const string& name, const Gender& gender) :
        name_(name), gender_(gender) {}

    friend ostream& operator<<(ostream& os, const Person& p) {
        os << p.name_ << " says: " << p.gender_.Print() << endl;
        return os;
    }
};
```

```
int main() {
    Person p1("Ramen Bag", Male::Type());       // Type constants by singleton
    Person p2("Elisa Tang", Female::Type());    // Type constants by singleton

    cout << p1;
    cout << p2;

    return 0;
}
-----
Outputs:

Ramen Bag says: I am a man
Elisa Tang says: I am a woman
```

  – ***Behavior Constant*** [`const` in code contexts]: Like `const` pointer, reference, member function etc.

- **Implementation Constants** [`static const` in library / application]: Like array size
- **... Constant**:

D.6.4 Testing

- Every class must be tested for all its functionality in the Unit Test. Hence, it is advisable to keep every class simple – catering to one or two core ideas of the design only. You'll have more classes – each for every unique concept your design has. This is often better than having few complex classes.
- Application Test should test for all scenarios of intended use of the software. It is good to prepare an exhaustive test plan for the scenarios first and then code each into the application.
- We may or may not have separate header files for testing.
- While all unit tests may be written directly for this assignment, you may consider using test frameworks like *Google Test*, *Boost.Test* etc. It will be good fun.
- *Testing Abstract Class*

  How can we test an abstract class? We cannot instantiate and create objects for the same.

  Usually, these can be tested using *mock* (*dummy* / *fake*) objects of classes created by (*minimally*) specializing the abstract class with minimal functionalities for the polymorphic methods. (How to unit test abstract classes: extend with stubs?)

  Alternately, we may just test through the objects of the derived concrete classes. Usually, the approach of mock-based testing is needed and used for abstract classes that are complex (not ideal). In our case, the base classes should be very simple and quite testable from their specialization.

D.6.5 Null-able

- Null-able parameters can be used for parameters (in an interface) to keep provision for future expansion and / or optional behavior. Easiest way to implement it is to use pointer with `NULL` as default. For example,

```
// Using Pointer
class Passenger {
// ...
};

void DoBooking( // Parameters ...,
    const Passenger* = NULL); // Nullable
```

  Alternately, we could have a specific constant static object of the class (being passed as parameter) interpreted as null and use as default parameter value. For example,

```
// Using Reference
class Passenger {
// ...
public:
    static const Passenger nullPassenger;   // Semantically treated as a null passenger
};

const Passenger Passenger::nullPassenger;

void DoBooking( // Parameters ...,
    const Passenger& = Passenger::nullPassenger); // Nullable
```

D.7 Sample: Representing an Undirected Weighted Graph

Here we show an example of how to represent a (fixed) weighed undirected graph. This will help understand various aspects of the design that have been highlighted above. Consider the following project files carefully. Try to build and check out.

D.7.1 Node.h

```
#ifndef __NODE_H
#define __NODE_H

// ***** Node class definition of a graph
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

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

// Nodes of a weighted undirected graph
class Node {
    const string name_;

public:
    Node(const string& n) : name_(n) {
#if _DEBUG
    cout << "Node " << n << " created" << endl;
#endif // _DEBUG
    }

#if _DEBUG
    ~Node() {
        cout << "Node " << name_ << " destroyed" << endl;
    }
#endif // _DEBUG

    const string& GetName() const throw() { return name_; }

    friend ostream& operator<<(ostream& os, const Node& s) {
        os << s.name_;
        return os;
    }
};
#endif // __NODE_H
```

D.7.2 Node.cpp

```
// ***** Node class implementation of a graph
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

// ***** Project Headers
#include "Node.h"
```

D.7.3 Graph.h

```
#ifndef __GRAPH_H
#define __GRAPH_H

// ***** Graph class definition
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

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

// ***** Project Headers
#include "Node.h"
#include "Exception.h"

// Forward declaration
class Node;

// Weighted undirected graph
class Graph {
    Graph();
    ~Graph();

    // List of nodes as name-node pairs
    static map<const string, const Node*> sNodes;

    // List of weights as <node, node>-weight pairs
    static map<pair<const Node*, const Node*>, int> sWeights;

public:
    // Singletone Graph
    static const Graph& TheGraph() {
        // Local static singletone object
        // Gets instantiated on the first call
        // Gets cleaned up with other static objects after main()
        static const Graph theGraph;

        return theGraph;
    }

    // Gets the weight between two nodes
    // Throws Bad_Node if the name of either node does not exist
    // Throws Bad_Edge if the nodes are not connected
    int GetWeight(const string& srcName, const string& dstName) const
```

```
        throw (Bad_Node, Bad_Edge) // Exception specification - may not compile in some compilers
    ;

    // Gets the node of a given name
    const Node* GetNode(const string& name) const
        throw (Bad_Node) // Exception specification - may not compile in some compilers
    ;
};
#endif // __GRAPH_H
```

D.7.4 Graph.cpp

```
// ***** Graph class implementation
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

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

// ***** Project Headers
#include "Graph.h"
#include "Exception.h"

// ***** static Definitions
map<const string, const Node*> Graph::sNodes;
map<pair<const Node*, const Node*>, int> Graph::sWeights;

// ***** Implementation of Graph
Graph::Graph() {
    // May read the node names and weights from a file
    sNodes["A"] = new Node("A"); // Dynamically created, needs deletion
    sNodes["B"] = new Node("B");
    sNodes["C"] = new Node("C");
    sNodes["D"] = new Node("D");

    sWeights[make_pair(sNodes["A"], sNodes["B"])] = 3;
    sWeights[make_pair(sNodes["A"], sNodes["C"])] = 2;
    sWeights[make_pair(sNodes["B"], sNodes["D"])] = 9;
    sWeights[make_pair(sNodes["C"], sNodes["D"])] = 7;

#if _DEBUG
    cout << "Graph created" << endl << endl;
#endif // _DEBUG
}

Graph::~Graph() {
    // Cleans up the nodes created
    map<const string, const Node*>::iterator it;
    for (it = sNodes.begin(); it != sNodes.end(); ++it)
        delete it->second;

#if _DEBUG
    cout << "Graph destroyed" << endl;
#endif // _DEBUG
}
```

```
const Node* Graph::GetNode(const string& name) const
    throw (Bad_Node) // Exception specification - may not compile in some compilers
{
    // Looks for a node with matching name
    map<const string, const Node*>::iterator it;
    for (it = sNodes.begin(); it != sNodes.end(); ++it)
        if (it->first == name)
            return it->second;

    throw Bad_Node("No node as " + name);
}

int Graph::GetWeight(const string& srcName, const string& dstName) const
    throw (Bad_Node, Bad_Edge) // Exception specification - may not compile in some compilers
{
    int weight = 0;
    try {
        // Get the nodes
        const Node* psNode = GetNode(srcName);
        const Node* pdNode = GetNode(dstName);

        // Get the weight between the nodes if they are connected
        map<pair<const Node*, const Node*>, int>::iterator weight_it;
        weight_it = sWeights.find(make_pair(psNode, pdNode));
        if (weight_it == sWeights.end()) {
            weight_it = sWeights.find(make_pair(pdNode, psNode));
            if (weight_it == sWeights.end())
                throw Bad_Edge("No edge between " + srcName + " and " + dstName);
        }
        weight = weight_it->second;
    }
    catch (Bad_Node&) {
        throw;
    }

    return weight;
}
```

D.7.5 Exception.h

```
#ifndef __EXCEPTION_H
#define __EXCEPTION_H

// ***** Exception class definitions for the project
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

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

// My exception class
class Exception : public exception {
    string message_;
public:
    Exception(const string& msg) : message_(msg) { }
    const char* what() {
```

```
        return message_.c_str();
    }
};

// Thrown for a missing node for a name
class Bad_Node : public Exception {
public:
    Bad_Node(const string& msg) : Exception(msg) { }
};

// Thrown for a missing edge between two nodes
class Bad_Edge : public Exception {
public:
    Bad_Edge(const string& msg) : Exception(msg) { }
};
#endif // __EXCEPTION_H
```

D.7.6 App.cpp

```
// ***** Test Application for Graph
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

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

// ***** Project Headers
#include "Graph.h"
#include "Exception.h"

// Application wrapper of GetWeight to handle exceptions
int GetLinkWeight(const string& n1, const string& n2) throw (Bad_Node, Bad_Edge) {
    int w = 0;
    try {
        w = Graph::TheGraph().GetWeight(n1, n2);
        cout << "Weight between " << n1 << " & " << n2 << " is = " << w << endl;
    }
    catch (Bad_Node& bn) {
        cout << bn.what() << endl;
    }
    catch (Bad_Edge& bl) {
        cout << bl.what() << endl;
    }

    return w;
}

int main() {
    // Test cases checking with golden output
    // These will assert on mis-matched output on Debug Build
    // All asserts will vanish in a Release Build and there will be no output

    // Both nodes exist with weight given in the called order
    assert(3 == GetLinkWeight("A", "B"));
```

```
    // Both nodes exist with weight given in reverse of the called order
    assert(3 == GetLinkWeight("B", "A"));

    // Both nodes exist but are not connected
    assert(0 == GetLinkWeight("B", "C"));

    // First node does not exist
    assert(0 == GetLinkWeight("E", "B"));

    // Second node does not exist
    assert(0 == GetLinkWeight("A", "F"));

    // Both nodes do not exist - first node to be reported
    assert(0 == GetLinkWeight("G", "H"));

    cout << endl;

    return 0;
}
```

D.7.7 Output in Debug build (_DEBUG is on)

Note that debug messages are given to track creation and destruction of objects:

```
Node A created
Node B created
Node C created
Node D created
Graph created

Weight between A & B is = 3
Weight between B & A is = 3
No edge between B and C
No node as E
No node as F
No node as G

Node A destroyed
Node B destroyed
Node C destroyed
Node D destroyed
Graph destroyed
```

D.8 Queries

D.8.1 Suryam Arnav Kalra

Sir , I have some doubts in this assignment.

1. Section 3.2 : Sir , it is written that the class Station HAS-A name , class BookingClasses HAS-A loadFactor but sir since name is just a string and loadFactor is just a double value, do I have to create separate classes for name and loadFactor or do I simply define them as string and double in their respective classes?
2. Section 3.3 : Sir , it is written to block the copy functions so do I write them in private with no implementation?
3. Sir , for the unit tests do I have to create a unit test for each class with all it's member functions / utilities tested ?
4. SIr , for the applications testing file what extra shall I add since the file given seems to cover all the possibilities that we want to test for this application ?

D.8.2 Yashica Patodia

I am having a confusion regarding the HAS-A portion of the assignment.

1. From what I understand, private inheritance of classes is a HAS-A relationship but a HAS-A relationship is not necessarily a private inheritance. Please correct me if I am wrong.

   So here, my doubt is should 'name' be a private attribute of class 'Station' or should I make a class 'name' and then do private inheritance. I am unable to understand how to implement a HAS-A relationship on properties.

D.8.3 Rajas Bhatt

I have the following clarifications related to Assignment 3 of CS20006

1. In section 3.2 it has been written that "Class Station HAS-A Name". In normal circumstances, this denotes aggregation. As covered in the class, aggregation can be done in multiple ways. However, I think that using a c++ string instead of a custom Name class should work here. I don't know what will be better.
2. In section 3.2 it has been written that "Booking should support Passenger as a NULL-able parameter for future extension". What does this mean?
3. In many instances, HAS-A relationships have been mentioned. Do these denote simple aggregation or necessarily Private Inheritance as covered in the class?
4. Can we use .hpp files instead of .h files?
5. In section 3.3 , bullet 4, it has been said that Class station should have getDistance() to get the distance to any other station, but aren't we implementing all distance calculation machinery in the Railways class?
6. While designing unit tests for each class, do we really need to have header files? I don't think this is necessary at all.
7. I have decided to keep BookingClasses as an Abstract base class (having two virtual functions), which has two derived classes (Seat and Berth) and they have my actual classes (like ACChairCar) . But the design is such that attributes, like name, ac/non-ac must be kept in the BookingClasses class itself. Now, since the BookingClasses is abstract, while constructing an object for my ACChairCar class, I cannot assign the attributes name_, ac/non-ac_ etc. What should I do?

D.8.4 Nakul Aggarwal

1. Sir, in the class Railways, why do we have to assign stations and pairwise-distances to static data members? If static data members are used then they will have to be non-const because they are to be changed upon instantiation of the singleton object. But at the same time keeping this constant information specific to the singleton instance of the Railways class in non-const attributes can be problematic. Can't we simply use non-static–const–private data members (to store stations and pairwise-distances) as was illustrated in one of the lectures?
2. I have a doubt in one guideline (page 9): **Every polymorphic hierarchy must provide a virtual destructor in the base class**. The only polymorphic hierarchy that we have in this project is the booking-class hierarchy. In this hierarchy all the base classes are abstract; and abstract base classes cannot have a constructor or a destructor. So does this mean that this guideline is redundant or have I misinterpreted it?

D.8.5 Nisarg Upadhyaya

1. Regarding statics in the singleton class Railways: What is the reason for making sStations and sDistStations statics in the Railways class. The design requires the class to be a singleton. The data members (private or public) if made static can be accessed using the scope resolution operator. I am not sure about this but intuitively it feels wrong to allow such access for a singleton class. Shouldn't the access to any method/attribute be using the name of the singleton class, i.e., Railways::IndianRailways.data_ or Railways::IndianRailways.func()

2. Regarding BookingClasses hierarchy: I couldn't comprehend the line "Rather, we would use single inheritance on the strongest property and use the rest as HAS-A with polymorphic value based on the leaf class" on page 4. I understand that we are trying to create a single/double level hierarchy of abstract base classes with the first level on the basis of Seat/Berth inheriting from the root BookingClasses followed by the classes for different tiers inheriting from the Berth class. Then we can create any singleton concrete booking class inheriting from a leaf class of the aforementioned hierarchy that matches the Seat/Berth and tier specification required by the concrete class. What is the part "use the rest as HAS-A with polymorphic value based on the leaf class" trying to convey?
3. Regarding constants: Why have changeable parameters like load factors, base fare rate, etc. been made static constants? Under 3.1 Design Principles, it has been asked that it should be easy to change these parameters from the application space.
4. Regarding copy functions: Does blocking of copy function mean we explicitly set them as private or provide no implementation and only definition for the same?
5. Regarding testing: Separate source and header files are required only for application testing right? For unit tests, we can have a static unit test function for each class whose implementation can be provided in the implementation file along with the implementation of the class as done in the unit testing of Fraction class? Also, confirming that no output is required for the unit tests developed.

D.8.6 Ashutosh Kumar Singh

1. So when you say that we encode the attributes on the methods, it means that there does not have an explicit member variable for AC/Non-AC, luxury, etc., right? We just return an appropriate boolean value from the virtual functions isAC() or isLuxury() as the case is. Is this correct?
2. Also, sir in the format of the date on page 1, it is given dd/mmm/yy, I feel that it is an error. Could you please tell what the correct format is? I believe if we follow the modules, then it should be dd/mm/yyyy.
3. According to what I understood, "golden output" means that it is the theoretical output that we feel is correct. Is this right or is there something more to it?
4. There are various initializations which are to be done, for example creating the IndianRailways object, then as it is said that " it should be easy to change the changeable parameters (likebase rate, load factor etc.) easily from the Application space", so does this mean that these static constants should be initialized in Application.cpp? What I was thinking of is to initialize all the changeable statics in Application.cpp and all the non-changeable statics in their respective .cpp files itself.
5. I wanted to know where should the function bodies for each unit test function for each class be kept. Should each unit test function be in the respective source file of the class, or should all of them be in a different file like something called UnitTests.cpp?

   Also, sir how are the unit test functions to be executed? I thought of putting all the unit test functions in a single file like UnitTests.cpp as mentioned, and then write a temporary main function in that class and then call each unit test function one by one. Is this how it is done, or is there some other procedure that is followed?

D.8.7 Aryan Singh

1. I was going through the lectures of singleton class for implementing the Railways assignment. In that I noticed that we explicitly call delete Printer::printer();

   So Sir is it a good practice to call the destructor at the end of Main() or we should ignore calling delete .

D.8.8 Yindukuri Jayanth Phani Sai

1. I would like to know whether we are supposed to implement tests manually or we are allowed to use frameworks like *Google Test*, *Boost.Test* etc For the Unit Test Implementations part of the Assignment 3. I felt that learning and working with new frameworks in C++ might be fun.

D.8.9 Deep Majumder

1. I really love languages with "strong" compilers that statically detect a lot of bugs. My first "real" language was Java and I used to be quite impressed by its strictness. That is until I found out about Rust, which eliminates memory errors (and seg-faults) statically. Also Kotlin - I really find it useful to interoperate with JVM based libraries, while having a stricter compiler.

   PPD: I love *strong (type-safe)* compilers too. But they come with an efficiency cost. None of these languages match (or come even near) the efficiency of C++ which often is a critical issue. So C++ is being elevated in its type-safety through various generations of C++11, 14, 17, 20, 23.

   But enough of ranting - I would like to discuss with you what I feel about Singletons (especially since I am using it quite a lot now). As you mentioned in class, Singletons have two uses - one for objects that should exist uniquely and otherwise for creating types with an identity. The first one is of course quite necessary - and while we need hacks like Meyer's singleton to represent it properly, while in Java we simply make everything static, it is still okay.

   PPD: I would disagree. Meyer's singleton is **not** a hack. It is one of the most beautiful implementation of a singleton devoid of any runtime support and / or resource management wrappers (like Smart Pointers). Further, I would rather call *types with an identity* as *type constants with behavior*.

   The other use, as you mentioned, is to represent things like days of the week with types and have a compiler guarantee. However, one problem I can see is that since C++ class hierarchies are essentially open, we can easily add a "eight" day by subclassing it in our application code.

   PPD: You are right. Refering to `Gender` example, we can try to subclass `Male` and / or `Female`. It cannot be done because the constructors are `private`. However, if they are written with less rigour (constructor in `protected`, it takes a parameter etc.), you would be able to sub-class `MoonMale` from `Male` and `MoonFemale` from `Female` into the applications space as follows:

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

class Gender { const string name_;
protected: Gender(const string& name) : name_(name) {}
public: virtual const string Print() const = 0;
};
class Male : public Gender { static const string sName;
protected: // Constructor moved to protected and provided with a defaulted parameter
    Male(const string& name = "") : Gender(name.empty()? Male::sName: name) {}
public:
    static const Gender& Type() { static const Male theObj; return theObj; }
    const string Print() const { return "I am a man"; }
};
class Female : public Gender { static const string sName;
protected: // Constructor moved to protected and provided with a defaulted parameter
    Female(const string& name = "") : Gender(name.empty()? Female::sName: name) { }
public:
    static const Gender& Type() { static const Female theObj; return theObj; }
    const string Print() const { return "I am a woman"; }
};

const string Male::sName = "Male";
const string Female::sName = "Female";

class Person { const string name_;
    const Gender& gender_;
public:
    Person(const string& name, const Gender& gender) : name_(name), gender_(gender) {}

    friend ostream& operator<<(ostream& os, const Person& p) {
        os << p.name_ << " says: " << p.gender_.Print() << endl;
        return os;
    }
```

```cpp
};

// In Application
class MoonMale : public Male {
    MoonMale() : Male(MoonMale::sName) {}
    static const string sName;
public:
    static const Gender& Type() { static const MoonMale theObj; return theObj; }
    const string Print() const { return "I am a moon man"; }
};

// In Application
class MoonFemale : public Female {
    MoonFemale() : Female(MoonFemale::sName) {}
    static const string sName;
public:
    static const Gender& Type() { static const MoonFemale theObj; return theObj; }
    const string Print() const { return "I am a moon woman"; }
};

// In Application
const string MoonMale::sName = "MoonMale";
const string MoonFemale::sName = "MoonFemale";

int main() {
    Person p1("Ramen Bag", Male::Type());
    Person p2("Elisa Tang", Female::Type());
    Person mp1("$__X9 @oo8", MoonMale::Type());
    Person mp2("!__V7 ?zv6", MoonFemale::Type());

    cout << p1;
    cout << p2;
    cout << mp1;
    cout << mp2;

    return 0;
}
```

PPD: This will be easy to fix. We need to use `private` inheritance – C++'s version of `final`[2] in Java (of course, C++11 has direct support of `final` with Java-like semantics). Now you cannot sub-class. The question still remains what if we sub-class from the base directly. You cannot stop that. Possible solution is to use a hack (kind of) - provide a `typedef` wrapper on these type constants in the library space within the base class. So you may be able to define an eight day, but you would not be able to use it interchangeably in the library calls. Bad, but effective. This is the required changes in the `Gender` for it.

[2] A class declared as a `final` class, cannot be subclassed

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

// Forward declarations needed for the typedef's
class Male;
class Female;

class Gender { const string name_;
protected Gender(const string& name) : name_(name) {}
public: virtual const string Print() const = 0;

    // Types are hardcoded here
    typedef Male HumanMale;
    typedef Female HumanFemale;
};

class Male : public Gender { Male() : Gender(Male::sName) {}
    static const string sName;
public:
    static const Gender& Type() { static const Male theObj; return theObj; }
    const string Print() const { return "I am a man"; }
};

class Female : public Gender { Female() : Gender(Female::sName) { }
    static const string sName;
public:
    static const Gender& Type() { static const Female theObj; return theObj; }
    const string Print() const { return "I am a woman"; }
};

const string Male::sName = "Male";
const string Female::sName = "Female";

class Person {
    const string name_;
    const Gender& gender_;
public:
    Person(const string& name, const Gender& gender) : name_(name), gender_(gender) {}

    friend ostream& operator<<(ostream& os, const Person& p) {
        os << p.name_ << " says: " << p.gender_.Print() << endl;
        return os;
    }
};

int main() {
    // You cannot generate these types in library space (Gender) namespace
    Person p1("Ramen Bag", Gender::HumanMale::Type());
    Person p2("Elisa Tang", Gender::HumanFemale::Type());

    cout << p1;
    cout << p2;

    return 0;
}
```

PPD: So we have something better, though hacky. That is the price for performance.

It is a silly thing to do and will never get past code review - but it is still a problem that the compiler potentially could have solved but did not. Inspired by the Java enum solution, I tried looking into C++ scoped enums (which came in as late as C++11). On the face of it, it looks okay:

```
enum class DayOfWeek { SUN, MON, TUE, WED, THU, FRI, SAT };
```

However, I found out that underneath scoped enums, is really an underlying integer data type (like regular enums). So we can use `static_cast` and to a disastrous effect. Like: `DayOfWeek apocalypse = static_cast<DayOfWeek>(7)`. This is thoroughly dis-satisfying and insufficient.

As a comparison, the Haskell solution would be something like:

```
data Class = ACFirstClass
           | FirstClass
           | ACThreeTier
           | ACTwoTier
           | Sleeper
           | ACChairCar
           | SecondSitting

class BookingClass a where
    getName       :: a -> String
    getNumTiers   :: a -> Int
    isSitting     :: a -> Bool
    isAc          :: a -> Bool
    getLoadFactor :: a -> Double
```

The Java solution will be pretty similar, using an enum which implements an interface. Long story short - the C++ solution, as far as I know, is quite clunky and not even bulletproof. So Sir, I would like to know if it is possible to make a better, safer solution to this problem in C++, perhaps using more advanced C++? Or are we stuck with this? Because although C++ doesn't have the same language beauty of Haskell or Rust, it is still the industrial standard and I really need to know it well. So Sir, could you please help me out with this?

PPD: I wish I had an elegant solution. We can ask Dr. Bjarne Stroustrup or Dr. Scott Meyers or Dr. Herb Sutter or Dr. Andrei Alexandrescu or some other strong expert of C++ / Java and similar languages. I would suggest that you write to them for their recommendations. Meanwhile, let me observe that the examples given by you (somewhat misguided by my `DayOfWeek` analogy), are about enumerated (in semantic sense) *values*. And there could be other simpler solutions for it, like in case of `Date`'s. This example is taken from **Item 18: Make interfaces easy to use correctly and hard to use incorrectly** in **Effective C++, 3*rd* Ed.** by *Scott Meyers*.

```
struct Day {
    explicit Day(int d) : val(d) {}
    int val;
};

struct Month {
    explicit Month(int m) : val(m) {}
    int val;
};

struct Year {
    explicit Year(int y) : val(y){}
    int val;
};

class Date {
public:
    Date(const Month& m, const Day& d, const Year& y);
...
};

Date d(30, 3, 1995);                    // error! wrong types
Date d(Day(30), Month(3), Year(1995));  // error! wrong types
Date d(Month(3), Day(30), Year(1995));  // okay, types are correct
```

PPD: Now `Month` can have 12 valid choices only. Like as follows:

```
class Month {
public:
    static Month Jan() { return Month(1); }     // functions returning all valid
    static Month Feb() { return Month(2); }     // Month values; see below for
    ...                                         // why these are functions, not
    static Month Dec() { return Month(12); }    // objects
    ...                                         // other member functions

private:
explicit Month(int m);  // prevent creation of new
                        // Month values
    ...                 // month-specific data
};

Date d(Month::Mar(), Day(30), Year(1995));
```

PPD: So the above simple design style can be used to have type-specific constant values. However, our real need in the context are *Type Constants* with *with Behavior*. So I chose the solution I provided. But I would never claim that it is indeed the *best* solution and will look forward to having a better way in C++. Anyone who can educate me on it – I would simply love it.

D.8.10 Rajat Bachhawat

1. In section 5.2 Code Organization, it is mentioned that we have to create an 'Application.cpp'. Is this the same test application that we have to make for carrying out application testing as mentioned in Task 7.4 Testing? Or is this a separate .cpp file that we have to create with a proper text based interface to accept user input and make bookings?

D.8.11 Parth Jindal

1. I had some doubts about how to go about testing abstract classes.What I had in mind was to make a wrapper class for them and create its fake object and test the polymorphic methods in it. Any directions for the same sir?

   PPD: Added a point under Testing (Section D.6.4) to explain.

2. Also from a design perspective,when and why should singleton classes have static data members associated with it.Is it only making things semantically sound or does it have an underlying computational meaning to it.

   PPD: I think we have already discussed different aspects of Singleton and static in depth in the clarifications and in response to Deep. Please go through them.

3. The Booking class is acting like a container here for all other class objects such as bookingClasses, Date, Person etc. Hence, Would the Application test differ from The UnitTest of this class apart from any private method(which I have not implemented here)?

   PPD: This is not a case of container or **HAS-A** relationship or aggregation (commonly called *Strong Aggregation*. Because a `Booking` does not contain a `Date`, or a `Passenger` etc. as a part. Rather, it is a case of **HAS** relationship or *Weak Aggregation* where we can say that `Passenger` has a kind of membership to the `Booking` only. More on this will be discussed in UML.

D.8.12 Anurat Bhattacharya

1. I was facing a problem during unit testing of Abstract Base Classes.Abstract Base Classes cannot be directly instantiated since it has some pure virtual functions.So I googled about it.What I got is that I need to create a mock object inheriting from the abstract base class with minimal functionalities and use it only for testing. How to unit test abstract classes: extend with stubs?

   So regarding a mock object,Do we need to override the pure virtual functions and make them return some default value and then test the already defined methods of the abstract Base class.Or is there any other method for Unit Testing of Abstract Base Classes without writing mock objects.

   PPD: Added a point under Testing (Section D.6.4) to explain.

E Solution

E.1 Design

The design of the classes are outlined in UML Class Diagram in Figure 1:

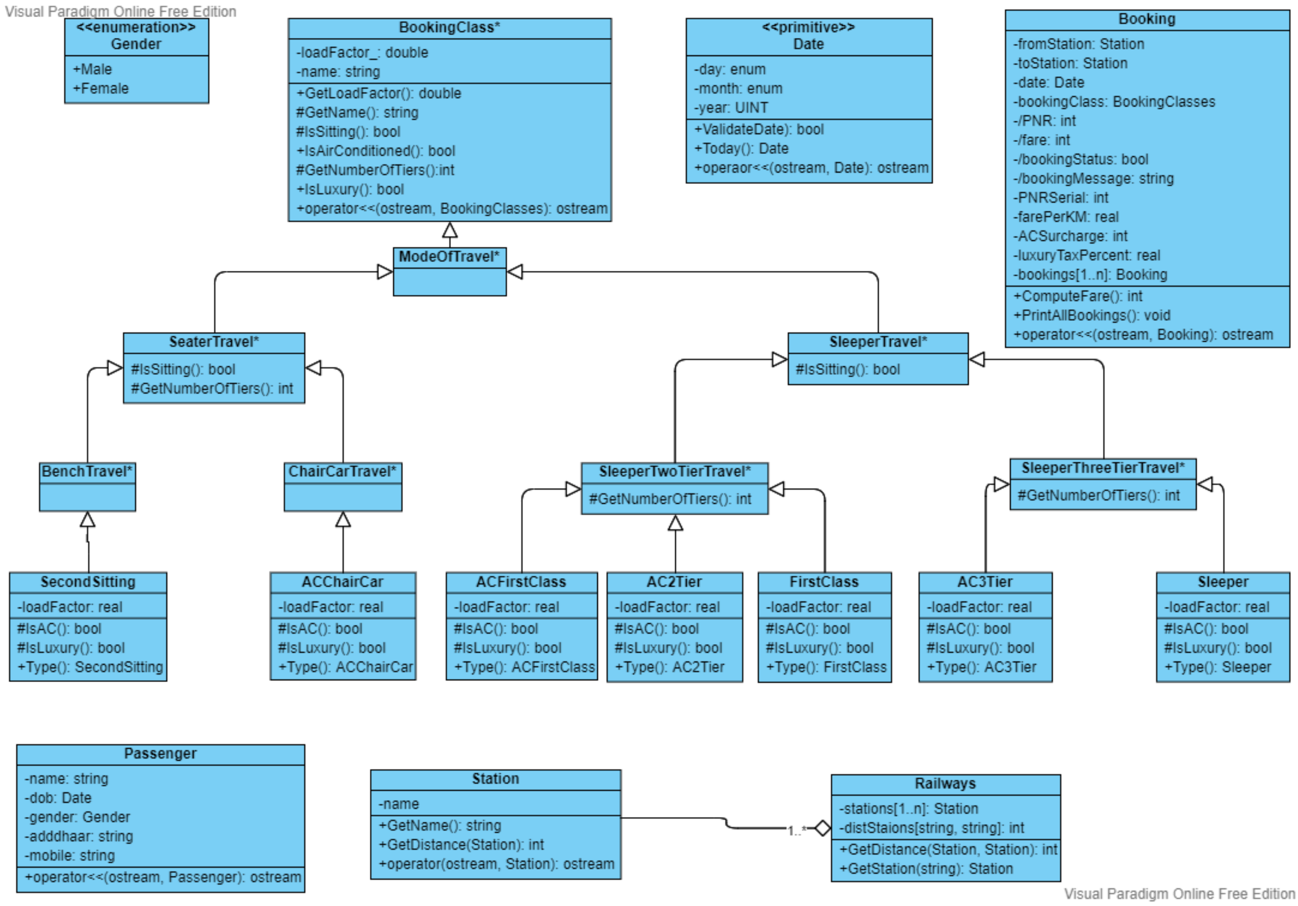

Figure 1: UML Class Diagram

The analysis and design are discussed at length in Sections 2, 3, 4, and D. However, the following major changes have been made to the design and the test application (Section C) during the implementation.

1. All non-static and static data members are put in `private`.

2. The list of stations and distances are made non-static data members in Railways instead of static data members. Semantically, these members (`vector` and `map`) do not carry and *static* data of the application (those are encapsulated in the constructor of Railways). Hence, these are needed only as long as there is `IndianRailways`.

3. The interface of `Booking` takes two temporary Station objects, constructed from their respective names. These must already be available in `IndianRailways`. So it is unnecessary to construct and destruct these temporary objects. And the interface can directly take Station names as input and get the (constant) station objects from `IndianRailways`. So the effective interface changes from:

```
Booking b1(Station("Mumbai"), Station("Delhi"), Date(15, 2, 2021), ACFirstClass::Type());
```

   to

```
Booking b1("Mumbai", "Delhi", Date(15, 2, 2021), ACFirstClass::Type());
```

4. Further the interface of `Booking` also takes a Date objects, which can be constructed from a string It is unnecessary to construct and destruct these temporary objects. So the effective interface changes from:

```
Booking b1("Mumbai", "Delhi", Date(15, 2, 2021), ACFirstClass::Type());
```

to

```
Booking b1("Mumbai", "Delhi", "15/02/2021", ACFirstClass::Type());
```

5. The list of bookings are printed from the application for which the list of bookings need to be exposed. This is, therefore, replaced by a public static method `PrintAllBookings()` in Booking.

Rest of the design details as evident from the UML Class Diagram (Figure 1, HDL) and the class codes with comments in the implementation (LLD).

E.2 Testing

As the current assignment does not expect wrong inputs, we have to only check for correctness of processing logic. That is test if the fare computation business logic is working fine for every case of BookingClass and Station.

E.2.1 Test Plan

Hence we have the following cases in the Test Plan:

1. BookingClass
   (a) ACFirstClass
   (b) AC2Tier
   (c) FirstClass
   (d) AC3Tier
   (e) ACChairCar
   (f) Sleeper
   (g) SecondSitting
2. Station
   (a) Check for every station name
   (b) Check for symmetric ordering to stations
   (c) Check for every pair of stations
3. Application
   (a) All bookings execute correctly
   (b) Booking list is printed correctly
   (c) All objects are constructed correctly as needed. Singletons are constructed only once
   (d) All objects are destructed before the program terminates

E.2.2 Test Suite: Test Cases & Golden

1. BookingClass
   (a) ACFirstClass

   ```
   Station("Mumbai")
   Station("Delhi")
   Date(15, 2, 2021)
   ACFirstClass::Type()

   Output: 2776
   ```

   (b) AC2Tier

```
Station("Kolkata")
Station("Delhi")
Date(5, 3, 2021)
AC2Tier::Type()

Output: 1522
```

(c) FirstClass

```
Station("Mumbai")
Station("Kolkata")
Date(17, 3, 2021)
FirstClass::Type());

Output: 2518
```

(d) AC3Tier

```
Station("Mumbai")
Station("Bangalore")
Date(23, 3, 2021)
AC3Tier::Type()

Output: 908
```

(e) ACChairCar

```
Station("Chennai")
Station("Delhi")
Date(25, 4, 2021)
ACChairCar::Type()

Output: 1413
```

(f) Sleeper

```
Station("Chennai")
Station("Kolkata")
Date(7, 5, 2021)
Sleeper::Type()

Output: 830
```

(g) SecondSitting

```
Station("Mumbai")
Station("Delhi")
Date(19, 5, 2021)
SecondSitting::Type()

Output: 362
```

2. Station

(a) Check for every station name: Checked by cases 1(a) through 1(g)

(b) Check for symmetric ordering of stations

```
Station("Mumbai")
Station("Delhi")
Date(19, 5, 2021)
SecondSitting::Type()
```

```
Output: 362

Station("Delhi")
Station("Mumbai")
Date(22, 5, 2021)
SecondSitting::Type()

Output: 362
```

(c) Check for every pair of stations

3. Application

   (a) All bookings execute correctly

   (b) Booking list is printed correctly

   (c) Construction check
   Put a print message in the Constructor under _DEBUG. This will print debugging message in debug build and skip in release build

   (d) Destruction check
   Put a print message in the Destructor under _DEBUG. This will print debugging message in debug build and skip in release build

E.2.3 Test Report

1. BookingClass

   (a) ACFirstClass: PASS

   (b) AC2Tier: PASS

   (c) FirstClass: PASS

   (d) AC3Tier: PASS

   (e) ACChairCar: PASS

   (f) Sleeper: PASS

   (g) SecondSitting: PASS

2. Station

   (a) Check for every station name: PASS

   (b) Check for symmetric ordering of stations: PASS

   (c) Check for every pair of stations: NOT DONE

3. Application

   (a) All bookings execute correctly: PASS

   (b) Booking list is printed correctly: PASS

   (c) Construction check: PASS

```
Welcome to Booking Application

Booking Class = AC First Class Constructed
Mumbai Constructed
Delhi Constructed
Bangalore Constructed
Kolkata Constructed
Chennai Constructed
IndianRailways Constructed
Date = 15/Feb/2021 Constructed
Booking = 1 Constructed
Booking Class = AC 2 Tier Constructed
```

```
Date = 5/Mar/2021 Constructed
Booking = 2 Constructed
Booking Class = First Class Constructed
Date = 17/Mar/2021 Constructed
Booking = 3 Constructed
Booking Class = AC 3 Tier Constructed
Date = 23/Mar/2021 Constructed
Booking = 4 Constructed
Booking Class = AC Chair Car Constructed
Date = 25/Apr/2021 Constructed
Booking = 5 Constructed
Booking Class = Sleeper Constructed
Date = 7/May/2021 Constructed
Booking = 6 Constructed
Booking Class = Second Sitting Constructed
Date = 19/May/2021 Constructed
Booking = 7 Constructed
Date = 22/May/2021 Constructed
Booking = 8 Constructed

The following bookings have been done today:
```

(d) Destruction check: PASS

```
End of Booking list

Booking = 8 Destructed
Date = 22/May/2021 Destructed
Booking = 7 Destructed
Date = 19/May/2021 Destructed
Booking = 6 Destructed
Date = 7/May/2021 Destructed
Booking = 5 Destructed
Date = 25/Apr/2021 Destructed
Booking = 4 Destructed
Date = 23/Mar/2021 Destructed
Booking = 3 Destructed
Date = 17/Mar/2021 Destructed
Booking = 2 Destructed
Date = 5/Mar/2021 Destructed
Booking = 1 Destructed
Date = 15/Feb/2021 Destructed

Goodbye from Booking Application

Booking Class = Second Sitting Destructed
Booking Class = Sleeper Destructed
Booking Class = AC Chair Car Destructed
Booking Class = AC 3 Tier Destructed
Booking Class = First Class Destructed
Booking Class = AC 2 Tier Destructed
Mumbai Destructed
Delhi Destructed
Bangalore Destructed
Kolkata Destructed
Chennai Destructed
IndianRailways Destructed
Booking Class = AC First Class Destructed
```

Note:

- All automatic (Booking, Date) ,dynamic (Station), and (local) static (Railways, different Book-

ingClasses) objects are constructed in function `BookingApplication()`. The automatic objects also get destructed function `BookingApplication()`. However, the static objects get destructed after `main()` and the dynamic objects go as a part of them.
If the static objects were global, they would also have got constructed before starting `main()`.

- Booking objects are destructed (with the embedded Date data member) in the reverse order of construction as they are automatic.
- Station objects (dynamically allocated) are destructed in the order of construction (which is the order in which they got into the list).
- The Railways object is constructed after Station objects and destructed after Station objects as these Station objects are managed by it.
- BookingClass objects behave truly as singleton - SecondSitting is used twice but is constructed and destructed once only.
- ACFirstClass object gets constructed before `IndianRailways` and gets destructed after it because it is used in the first Booking object (parameters are evaluated before a function - constructor here - is called).

E.3 Implementation

A sample implementation code is given as a single file below with the application test.

```
// ***** Booking application
// ***** Author: P P Das
// ***** Date: 01-Mar-2021
// ***** Version: 1.0
// ***** Known bugs: None

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

// Manage a date
class Date {
    // Year, Month, and Day types
    typedef unsigned int UINT;
    enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };

    // Data members
    UINT date_;
    Month month_;
    UINT year_;

    // Names of months & Days for output messages
    static const string sMonthNames[12];
    static const string sDayNames[7];
public:
    Date(UINT d, UINT m, UINT y) : date_(d), month_((Month)m), year_(y)
    {
#ifdef _DEBUG
        cout << "Date = " << *this << " Constructed" << endl;
#endif
    }

    Date(const string& date)
    {
        // Parse "dd/mm/yyyy" string for date
        size_t i = date.find_first_of("/");
```

```
        date_ = atoi((date.substr(0, i)).c_str());
        size_t j = date.find_first_of("/", i + 1);
        month_ = static_cast<Month>(atoi((date.substr(i + 1, j - i)).c_str()));
        year_ = atoi((date.substr(j + 1, date.length() - j)).c_str());
#ifdef _DEBUG
        cout << "Date = " << *this << " Constructed" << endl;
#endif
    }

    ~Date()
    {
#ifdef _DEBUG
        cout << "Date = " << *this << " Destructed" << endl;
#endif
    }


    // Not implemented
    bool ValidDate() { /* Check validity */ return true; }

    // Not implemented
    const Date Today()
    { /* Compute day from date using time.h */ return Date(1, 3, 2021); }

    friend ostream& operator<<(ostream& os, const Date& d);
};

// Names of months & Days for output messages
const string Date::sMonthNames[12] =
    { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };

const string Date::sDayNames[7] =
    { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };

// Out-stream Date
ostream& operator<<(ostream& os, const Date& d) {
    os << d.date_ << "/" << Date::sMonthNames[d.month_ - 1] << "/" << d.year_;
    return os;
}

// Forward declaration for Railways
class Station;

// Railwas is singleton, constant and contains stations
class Railways {
    Railways();
    ~Railways();

    // Stations
    vector<const Station*> stations_;

    // Distances between stations
    map<pair<string, string>, int> distStations_;
public:
    // The Railways
    static const Railways& IndianRailways() {
        static const Railways theRailways;

        return theRailways;
    }
```

```
    // Get a station by name
    const Station& GetStation(const string& name) const;

    // Distance between two stations
    int GetDistance(const Station& fromStation, const Station& toStation) const;
};

// Stations - simple named class
class Station {
    const string name_;

public:
    Station(const string& n) : name_(n)
    {
#ifdef _DEBUG
        cout << *this << " Constructed" << endl;
#endif
    }
    ~Station()
    {
#ifdef _DEBUG
        cout << *this << " Destructed" << endl;
#endif
    }

    int GetDistance(const Station& station) const;
    const string& GetName() const { return name_; }

    friend ostream& operator<<(ostream& os, const Station& s) {
        os << s.name_;
        return os;
    }
};

int Station::GetDistance(const Station& station) const {
    return Railways::IndianRailways().GetDistance(*this, station);
}

Railways::Railways() {
    // Initialize stations
    stations_.push_back(new Station("Mumbai"));    // Dynamically created - needs delete
    stations_.push_back(new Station("Delhi"));
    stations_.push_back(new Station("Bangalore"));
    stations_.push_back(new Station("Kolkata"));
    stations_.push_back(new Station("Chennai"));

    // Initialize distances
    distStations_[make_pair("Mumbai", "Delhi")] = 1447;
    distStations_[make_pair("Mumbai", "Bangalore")] = 981;
    distStations_[make_pair("Mumbai", "Kolkata")] = 2014;
    distStations_[make_pair("Mumbai", "Chennai")] = 1338;
    distStations_[make_pair("Delhi", "Bangalore")] = 2150;
    distStations_[make_pair("Delhi", "Kolkata")] = 1472;
    distStations_[make_pair("Delhi", "Chennai")] = 2180;
    distStations_[make_pair("Bangalore", "Kolkata")] = 1871;
    distStations_[make_pair("Bangalore", "Chennai")] = 350;
    distStations_[make_pair("Kolkata", "Chennai")] = 1659;

#ifdef _DEBUG
    cout << "IndianRailways Constructed" << endl;
#endif
```

```
}

Railways::~Railways() {
    // Cleans up the stations created
    vector<const Station*>::iterator it;
    for (it = stations_.begin(); it != stations_.end(); ++it)
        delete *it;

#ifdef _DEBUG
    cout << "IndianRailways Destructed" << endl;
#endif
}

// Get a station by name
const Station& Railways::GetStation(const string& name) const {
    vector<const Station*>::const_iterator it;
    for (it = stations_.begin(); it != stations_.end(); ++it)
        if ((*it)->GetName() == name)
            break;

    return *(*it);
}

// Distance between stations
int Railways::GetDistance(const Station& fromStation, const Station& toStation) const {
    map<pair<string, string>, int>::const_iterator dist_it;

    // Check the from-to pair
    dist_it = distStations_.find(make_pair(fromStation.GetName(), toStation.GetName()));
    if (dist_it == distStations_.end()) { // Does not exist
        // Check the to-from pair. Must exist by symmetry
        dist_it = distStations_.find(make_pair(toStation.GetName(), fromStation.GetName()));
    }

    return dist_it->second;
}

// Booking Class Hierarchy rooted at BookingClass follow
// This goes after the class diagram and master data of
// booking classes
class BookingClass { // Abstract Base Class
    const double loadFactor_;
    const string name_;

protected:
    BookingClass(double lf = 1.0, const string& name = "") :
        loadFactor_(lf), name_(name) {
#ifdef _DEBUG
        cout << "Booking Class = " << GetName() << " Constructed" << endl;
#endif
    }
    ~BookingClass() {
#ifdef _DEBUG
        cout << "Booking Class = " << GetName() << " Destructed" << endl;
#endif
    }

    // Not used outside the hierarchy
    const string& GetName() const
    { return name_; }
    virtual bool IsSitting() const = 0;
```

```
    virtual int  GetNumberOfTiers() const = 0;

public:
    // Used in fare computation
    double GetLoadFactor() const { return loadFactor_; }
    virtual bool IsAC() const = 0;
    virtual bool IsLuxury() const = 0;

    friend ostream& operator<<(ostream& os, const BookingClass& c);
};

// Out-stream Booking Classes
ostream& operator<<(ostream& os, const BookingClass& c) {
    os << "Travel Class = " << c.GetName() << endl;
    os << " : Mode: " << ((c.IsSitting()) ? "Sitting" : "Sleeping") << endl;
    os << " : Comfort: " << ((c.IsAC()) ? "AC" : "Non-AC") << endl;
    os << " : Bunks: " << c.GetNumberOfTiers() << endl;
    os << " : Luxury: " << ((c.IsLuxury()) ? "Yes" : "No");
    return os;
}

class ModeOfTravel : public BookingClass { // Abstract Base Class
protected:
    ModeOfTravel(double lf, const string& name) : BookingClass(lf, name) { }
};

class SeaterTravel : public ModeOfTravel { // Abstract Base Class
protected:
    SeaterTravel(double lf, const string& name) : ModeOfTravel(lf, name) { }
    bool IsSitting() const            { return true; }
    int  GetNumberOfTiers() const    { return 0; }
};

class ChairCarTravel : public SeaterTravel { // Abstract Base Class
protected:
    ChairCarTravel(double lf, const string& name) : SeaterTravel(lf, name) { }
};

class BenchTravel : public SeaterTravel { // Abstract Base Class
protected:
    BenchTravel(double lf, const string& name) : SeaterTravel(lf, name) { }
};

class SleeperTravel : public ModeOfTravel { // Abstract Base Class
protected:
    SleeperTravel(double lf, const string& name) : ModeOfTravel(lf, name) { }
    bool IsSitting() const    { return false; }
};

class SleeperTwoTierTravel : public SleeperTravel { // Abstract Base Class
protected:
    SleeperTwoTierTravel(double lf, const string& name) :SleeperTravel(lf, name) { }
    int GetNumberOfTiers() const    { return 2; }
};

class SleeperThreeTierTravel : public SleeperTravel { // Abstract Base Class
protected:
    SleeperThreeTierTravel(double lf, const string& name) : SleeperTravel(lf, name) { }
    int GetNumberOfTiers() const    { return 3; }
};
```

```
class ACFirstClass : public SleeperTwoTierTravel {
    ACFirstClass(
        double lf = sLoadFactorACFirstClass,
        const string& name = "AC First Class") :
        SleeperTwoTierTravel(lf, name) { }
    ~ACFirstClass() { }

    static const double sLoadFactorACFirstClass;

protected:
    bool IsAC() const                { return true; }
    bool IsLuxury() const            { return true; }

public:
    static const ACFirstClass& Type() {
        static const ACFirstClass theACFirstClass;

        return theACFirstClass;
    }
};

class AC2Tier : public SleeperTwoTierTravel {
    AC2Tier(
        double lf = sLoadFactorAC2Tier,
        const string& name = "AC 2 Tier") :
        SleeperTwoTierTravel(lf, name) { }
    ~AC2Tier() { }

    static const double sLoadFactorAC2Tier;

protected:
    bool IsAC() const                { return true; }
    bool IsLuxury() const            { return false; }

public:
    static const AC2Tier& Type() {
        static const AC2Tier theAC2Tier;

        return theAC2Tier;
    }
};

class FirstClass : public SleeperTwoTierTravel {
    FirstClass(
        double lf = sLoadFactorFirstClass,
        const string& name = "First Class") :
        SleeperTwoTierTravel(lf, name) { }
    ~FirstClass() { }

    static const double sLoadFactorFirstClass;

protected:
    bool IsAC() const                { return false; }
    bool IsLuxury() const            { return true; }

public:
    static const FirstClass& Type() {
        static const FirstClass theFirstClass;

        return theFirstClass;
    }
```

```
};

class AC3Tier : public SleeperThreeTierTravel {
    AC3Tier(
        double lf = sLoadFactorAC3Tier,
        const string& name = "AC 3 Tier") :
        SleeperThreeTierTravel(lf, name) { }
    ~AC3Tier() { }

    static const double sLoadFactorAC3Tier;

protected:
    bool IsAC()    const                { return true; }
    bool IsLuxury() const            { return false; }

public:
    static const AC3Tier& Type() {
        static const AC3Tier theAC3Tier;

        return theAC3Tier;
    }
};

class ACChairCar : public ChairCarTravel {
    ACChairCar(
        double lf = sLoadFactorACChairCar,
        const string& name = "AC Chair Car") :
        ChairCarTravel(lf, name) { }
    ~ACChairCar() { }

    static const double sLoadFactorACChairCar;
protected:
    bool IsAC() const                { return true; }
    bool IsLuxury() const            { return false; }

public:
    static const ACChairCar& Type() {
        static const ACChairCar theACChairCar;

        return theACChairCar;
    }
};

class Sleeper : public SleeperThreeTierTravel {
    Sleeper(
        double lf = sLoadFactorSleeper,
        const string& name = "Sleeper") :
        SleeperThreeTierTravel(lf, name) { }
    ~Sleeper() { }

    static const double sLoadFactorSleeper;

protected:
    bool IsAC() const                { return false; }
    bool IsLuxury() const            { return false; }

public:
    static const Sleeper& Type() {
        static const Sleeper theSleeper;

        return theSleeper;
```

```
    }
};

class SecondSitting : public BenchTravel {
    SecondSitting(
        double lf = sLoadFactorSecondSitting,
        const string& name = "Second Sitting") :
        BenchTravel(lf, name) { }
    ~SecondSitting() { }

    static const double sLoadFactorSecondSitting;

protected:
    bool IsAC() const               { return false; }
    bool IsLuxury() const           { return false; }

public:
    static const SecondSitting& Type() {
        static const SecondSitting theSecondSitting;

        return theSecondSitting;
    }
};

// Load factors of various booking classes - from master data
const double ACFirstClass::sLoadFactorACFirstClass = 3.0;
const double AC2Tier::sLoadFactorAC2Tier = 2.0;
const double FirstClass::sLoadFactorFirstClass = 2.0;
const double AC3Tier::sLoadFactorAC3Tier = 1.75;
const double ACChairCar::sLoadFactorACChairCar = 1.25;
const double Sleeper::sLoadFactorSleeper = 1.0;
const double SecondSitting::sLoadFactorSecondSitting = 0.5;

// Male - Female enumerated type
typedef enum { male, female } Gender;

class Passenger { // Abstract Base Class - not used
    const string name_;
    const Date dob_;
    const Gender gender_;
    const string aadhaar_;
    const string mobile_;
public:
    virtual ~Passenger() = 0;

    friend ostream& operator<<(ostream& os, const Passenger& p);
};

class Booking {
    // Booking input data
    const Station&          fromStation_;
    const Station&          toStation_;
    const Date              date_;
    const BookingClass*      bookingClass_;

    // Booking completion data
    int                      PNR_;
    int                      fare_;

    // Booking status data
    bool                   bookingStatus_;        // Sucessful or not
```

```
    string                    bookingMessage_;    // Booking message for output

    // Static booking parameters
    static int sBookingPNRSerial;            // PNR
    static const double sBaseFarePerKM;        // Base rate
    static const int sACSurcharge;            // AC Charge
    static const double sLuxuryTaxPercent;    // Luxury mark up

    // Collection of all boookings for the session
    static vector<Booking*> sBookings;

public:
    Booking(
        const string& fs,
        const string& ts,
        const string& d,
        const BookingClass& cs,
        Passenger* pax = NULL    // Skipped for now
        );
    ~Booking() {
#ifdef _DEBUG
        cout << "Booking = " << PNR_ << " Destructed" << endl;
#endif
    }

    // Fare computation business logic
    int ComputeFare();

    friend ostream& operator<<(ostream& os, const Booking& b);

    static void PrintAllBookings() {
        cout << endl << "The following bookings have been done today:" << endl << endl;
        // Output the bookings done
        vector<Booking*>::iterator it;
        for (it = Booking::sBookings.begin(); it < Booking::sBookings.end(); ++it) {
            cout << *(*it);
        }
        cout << endl << "End of Booking list" << endl << endl;
    }
};

Booking::Booking(
    const string& fs, const string& ts,
    const string& d,
    const BookingClass& cs,
    Passenger* pax) :
    fromStation_(Railways::IndianRailways().GetStation(fs)),
    toStation_(Railways::IndianRailways().GetStation(ts)),
    date_(d), bookingClass_(&cs),
    fare_(0),
    bookingStatus_(true), // Booking status will be always successful
    bookingMessage_("BOOKING SUCCEEDED: ")
{
    fare_ = ComputeFare(); // Compute fare by business logic

    if (bookingStatus_) // If successful, issue the booking
        PNR_ = sBookingPNRSerial++;

    sBookings.push_back(this); // Add booking to collection

#ifdef _DEBUG
```

```
    cout << "Booking = " << PNR_ << " Constructed" << endl;
#endif
}

// Compute fare by business logic
int Booking::ComputeFare() {
    int distance = fromStation_.GetDistance(toStation_);
    double baseFare = distance * sBaseFarePerKM;
    double loadedFare = baseFare * bookingClass_->GetLoadFactor();

    double fare = loadedFare;
    if (bookingClass_->IsAC())
        fare += Booking::sACSurcharge;

    if (bookingClass_->IsLuxury())
        fare = fare * (1 + Booking::sLuxuryTaxPercent / 100);

    int finalFare = static_cast<int>(fare + 0.5);

    return finalFare;
}

// Out-stream Booking
ostream& operator<<(ostream& os, const Booking& b) {
    os << b.bookingMessage_ << endl;
    os << "PNR Number = ";
    if (b.bookingStatus_)
        cout << b.PNR_;
    else
        cout << "None";
    cout << endl;
    os << "From Station = " << b.fromStation_ << endl;
    os << "To Station = " << b.toStation_ << endl;
    os << "Travel Date = " << b.date_ << endl;
    os << *(b.bookingClass_) << endl;
    if (b.bookingStatus_)
        os << "Fare = " << b.fare_ << endl;
    cout << endl;
    return os;
}

// Static booking parameters
int Booking::sBookingPNRSerial = 1;
const double Booking::sBaseFarePerKM = 0.5;
const int Booking::sACSurcharge = 50;
const double Booking::sLuxuryTaxPercent = 25;

// Collection of all boookings for the session
vector<Booking*> Booking::sBookings;

// Test application for booking
void BookingApplication() {

    // Bookings by different booking classes

    // <BookingClass>::Type() returns the constant object of the respective type
    Booking b1("Mumbai", "Delhi", "15/02/2021", ACFirstClass::Type());
    Booking b2("Kolkata", "Delhi", "05/03/2021", AC2Tier::Type());
    Booking b3("Mumbai", "Kolkata", "17/03/2021", FirstClass::Type());
    Booking b4("Mumbai", "Bangalore", "23/03/2021", AC3Tier::Type());
    Booking b5("Chennai", "Delhi", "25/04/2021", ACChairCar::Type());
```

```
    Booking b6("Chennai", "Kolkata", "07/05/2021", Sleeper::Type());
    Booking b7("Mumbai", "Delhi", "19/05/2021", SecondSitting::Type());
    Booking b8("Delhi", "Mumbai", "22/05/2021", SecondSitting::Type());

    Booking::PrintAllBookings();

    return;
}

int main() {

    cout << "Welcome to Booking Application" << endl << endl;
    BookingApplication();
    cout << endl << "Goodbye from Booking Application" << endl << endl;


    return 0;
}
```