Unit Test & Exception Document of Various Classes

-Contains both assert unit test examples and try-catch exception handling

Content:

Passenger:

We can have exceptions due to various reasons, like mobile number should be of 10 digits, aadhaar should be of 12 digits, date of birth should have a right format.
I created various test cases to generate exceptions on invalid inputs

Date of birth:

- 01/01/2030 ->bad passenger thrown
- 35/01/2010 ->bad date thrown
  This month can't have more than 30 days
- 29/02/2001 ->bad date thrown
  Feb can't have 29 days in a non-leap year
- 01/01/2010 -> correct value

Aadhar:

- "123456" -> bad passenger thrown thrown
  Aadhaar can't have less than 12 characters
- "1234567891234abcde" -> bad passenger
  Aadhaar can't have more than 12 characters, and it has to be composed of integers from 1 to 9

Mobile number:

- "1234567890" ->no error
- "123456" -> bad passenger thrown
  A passenger can't have less than 10 digits
- "123456789123456789" -> bad passenger thrown
  A passenger can't have more than 10 digits
- "" -> no error as phone number is optional
  Mobile number of a passenger can be empty

Examples:

- try{

```
const Passenger p1 = Passenger::createPassenger("Rohit", "", "Raj",
"123456789012", "09/08/1970", Gender::Male::Type(), "94310", 4, 3, "Blind") ;
        assert(p1.GetName() == "Rohit");
        assert(p1.GetAadharNo() == "123456789012" );
        assert(p1.GetDateOfBirth() == "09/08/2001");
        assert(p1.GetGender() == "Male");
        assert(p1.GetMobileNumber() == "94310");
assert(p1.GetCategory() == "Divyaang");}
Catch(const Bad_Passenger &e)
{throw e.what();}
```

Caught Bad Passenger -> Mobile number is invalid

-> It should be of 10 digits

- try{

const Passenger p1 = Passenger::createPassenger("Ram", "", "Singh", "123456789987", "34/04/2002", Gender::Male::Type(), "9431049093", 4, 3, “Blind”) ;

assert(p1.GetName() == "Rohit");
assert(p1.GetAadharNo() == "123456789987");
assert(p1.GetDateOfBirth() == "34/04/2002”);
assert(p1.GetGender() == "Male");
assert(p1.GetMobileNumber() == "9431049093");
assert(p1.GetCategory() == "Divyaang");}
Catch(const Bad_Passenger &e)
{throw e.what();}

Caught Bad Date

-> Month can’t have more than 31 days

- try{

const Passenger p1 = Passenger::createPassenger("", "Singh", "", "123452234432", "30/04/2002", Gender::Male::Type(), "6261171632", 1, 2, "Tatkal") ;

assert(p1.GetName() == "Ram");
assert(p1.GetAadharNo() == "123452234432");
assert(p1.GetDateOfBirth() == "30/04/2002");
assert(p1.GetGender() == "Male");
assert(p1.GetMobileNumber() == "6261171632");
assert(p1.GetCategory() == "Tatkal");
Catch(const Bad_Passenger &e)
{throw e.what();}

Caught Bad Passenger -> It has invalid name

-> Either first or last name must be present

- const Passenger p1 = Passenger::createPassenger("Shivam", "Singh", "", "123452234412", "30/04/2002", Gender::Male::Type(), "6261171632", 1, 2, "Tatkal") ;
  assert(p1.GetName() == "Shivam");
  assert(p1.GetAadharNo() == "123452234412");
  assert(p1.GetDateOfBirth() == "30/04/2002");
  assert(p1.GetGender() == "Male");
  assert(p1.GetMobileNumber() == "6261171632");
  assert(p1.GetCategory() == "Tatkal");

Correct Assert

- const Passenger p1 = Passenger::createPassenger("Shivam", "Singh", "Bedi", "112233445566", "20/04/2002", Gender::Male::Type(), "1234567890", 1, 2, "Tatkal") ;
  assert(p1.GetName() == "Shivam");
  assert(p1.GetAadharNo() == "112233445566");
  assert(p1.GetDateOfBirth() == "20/04/2002");
  assert(p1.GetGender() == "Male");
  assert(p1.GetMobileNumber() == "1234567890");
  assert(p1.GetCategory() == "Tatkal");

Correct Assert

DATE:

Here we check the different invalid dates and date formats. The following are tested:

Greater than operator (overloaded)
Overloaded difference operator
Overloaded assignment and equate operator
Also name and no. of day month is also being tested etc

- Format: String that is sent to create date must be of correct format otherwise error is detected
  ->”abcd” -> error
  ->”01/01/20 -> error…

- Validity: Various invalid dates are tested
  -> 1/1/2030 ->error
  -> 1/1/1830 ->error
  ->35/1/2010 ->error
  ->29/2/2001 ->error
  ->1/1/2010 -> correct value

- CompYear: Checks difference of 2 dates
  CompYear(15/01/2020, 20/01/2020) => 5 days
  CompYear(15/01/2020, 15/01/2020) => 0 days
  CompYear(15/01/2021, 15/01/2022) => 365 days -> 1year

Unit testing:

- try{

```
const Date &date = Date::CreateDate("31/02/2001");
        assert(date.GiveDate() == 31);
        assert(date.GiveMonth() == "Feb");
        assert(date.GiveMonthNo() == 2);
        assert(date.GiveYear() == 2001);
}
Catch(const Bad_Date &e)
{throw e.what();}
```

Caught Bad Date-> Invalid Date
-> 31 days in february

- const Date &date = Date::CreateDate("11/02/2001");
  assert(date.GiveDate() == 11);
  assert(date.GiveMonth() == "Feb");
  assert(date.GiveMonthNo() == 2);
  assert(date.GiveYear() == 2001);

Correct assert

- const Date &date = Date::CreateDate("31/12/2018");
  assert(date.GiveDate() == 31);
  assert(date.GiveMonth() == "Dec");
  assert(date.GiveMonthNo() == 4);
  assert(date.GiveYear() == 2018);

Correct assert

STATION:

Here I have checked if the name of a created station is correctly matched.
Distances are checked in railways.

```
Station s1 = Station("Mumbai");
Station s2 = Station("Delhi");
Station s3 = Station("Kolkata");
```

- Name, comparison

assert(s1.GetName() == "Mumbai"); -> Correct Output
assert(s2.GetName() == "Delhi"); ->Correct Output

Correct asserts

assert(s1 == Station("Mumbai")); ->Correct Output
assert(s2 == Station("Delhi")); ->Correct Output

Correct Asserts

RAILWAYS:

Here I have checked if the distances are correct between pairs of created stations, in a dummy railway created.
From the last example,
Station s1 = Station("Mumbai");
Station s2 = Station("Delhi");
Station s3 = Station("Kolkata");

- Distance:
  assert(GetDistance(s1, s2) == 1447); ->no error
  assert(GetDistance(s3, s2) == 2014); ->no error
  To check if distances are correctly assigned

Correct Assert

GENDER :

Here I have created male and female objects and I've checked if they generate correct outputs for GetTitle, GetName, IsMale

- Title:
  Male returns "Mr."
  Female returns "Mrs."
- Name:
  Male returns "Male"
  Female returns "Female"
- IsMale returns true for male and false for female

```
-->const Gender& Male_ = Gender::Male::Type() ;
        assert(Male_.GetTitle() == "Mr.") ;
        assert(Male_.GetName() == "Male") ;
        assert(Male_.IsMale(Male_) == true);
```

Correct Assert

```
        const Gender& Female_ = Gender::Female::Type() ;
        assert(Female_.GetTitle() == "Ms.") ;
        assert(Female_.GetName() == "Female") ;
        assert(Female_.IsMale(Female_) == false);
```

Correct Assert

This code checks all the above cases.

BOOKING CLASSES:

The booking class has a simple hierarchy with intermediate abstract classes and every child class is a concrete leaf class overriding the functions of the parent base class
Here I have created instance for all the child classes in the parametric polymorphism and checked all outputs of functions
Get name checks the name of the class so it should return “AC First Class” for ACFirstClass class. Get load factor returns the load factor for that particular class. Similarly for IsAc, IsSitting and so on…

Booking Classes is a parametric polymorphic hierarchy with the following child classes:
ACFirstClass, AC2Tier, AC3Tier, ACChairCar, FirstClass, Sleeper, SecondSitting, ExecutiveChairCar

- ## AC First Class:

I created the following asserts to check the various methods in the class

```
    const BookingClasses& ACFirstClass_ =
BookingClasses::ACFirstClass::Type() ;
    assert(ACFirstClass_.GetName() == "AC First Class") ;
    assert(ACFirstClass_.GetLoadFactor() == 6.50) ;
    assert(ACFirstClass_.IsAC() == true) ;
    assert(ACFirstClass_.IsLuxury() == true) ;
```

```
assert(ACFirstClass_.IsSitting() == false) ;
assert(ACFirstClass_.GetNumberOfTiers() == 2) ;
assert(ACFirstClass_.GetMinTatkalCharge() == 400) ;
assert(ACFirstClass_.GetMaxTatkalCharge() == 500) ;
assert(ACFirstClass_.GetReservationCharge() == 60.00) ;
assert(ACFirstClass_.GetMinTatkalDistance() == 500) ;
assert(ACFirstClass_.GetTatkalLoadFactor() == 0.3)  ;
```

Correct Assert

They have correct outputs because the methods are as follows:

IsLuxury() should return true.
IsAc() should return true.
GetLoadFactor() should return 6.50.
GetName() should return “Ac 2Tier”.
IsSitting() should return false.
GetMinTatkalCharge() should returns 400.
GetMaxTatkalCharge() should returns 500.
GetReservationCharge() should returns 60.00.
GetMinTatkalDistance() should returns 500.
GetTatkalLoadFactor() should returns 0.30.
GetNumberOfTiers() should return 2.

- Ac2Tier class:

```
const BookingClasses& AC2Tier_ = BookingClasses::AC2Tier::Type() ;
    assert(AC2Tier_.GetName() == "AC 2 Tier") ;
    assert(AC2Tier_.GetLoadFactor() == 4.00) ;
    assert(AC2Tier_.IsAC() == true) ;
    assert(AC2Tier_.IsLuxury() == false) ;
    assert(AC2Tier_.IsSitting() == false) ;
    assert(AC2Tier_.GetNumberOfTiers() == 2) ;
    assert(AC2Tier_.GetMinTatkalCharge() == 400) ;
    assert(AC2Tier_.GetMaxTatkalCharge() == 500) ;
    assert(AC2Tier_.GetReservationCharge() == 50.00) ;
    assert(AC2Tier_.GetMinTatkalDistance() == 500) ;
    assert(AC2Tier_.GetTatkalLoadFactor() == 0.3)  ;
```

Correct Assert

Expected:

IsLuxury() should return true.
IsAc() should return false.
GetLoadFactor() should return 4.00.
GetName() should return “Ac 2Tier”.
IsSitting() should return false.
GetMinTatkalCharge() should returns 400.
GetMaxTatkalCharge() should returns 500.
GetReservationCharge() should returns 50.00.
GetMinTatkalDistance() should returns 500.
GetTatkalLoadFactor() should returns 0.30.
GetNumberOfTiers() should return 2.

- AC3Tier:

```
const BookingClasses& AC3Tier_ = BookingClasses::AC3Tier::Type() ;
        assert(AC3Tier_.GetName() == "AC 3 Tier") ;
        assert(AC3Tier_.GetLoadFactor() == 2.50) ;
        assert(AC3Tier_.IsAC() == true) ;
        assert(AC3Tier_.IsLuxury() == false) ;
        assert(AC3Tier_.IsSitting() == false) ;
        assert(AC3Tier_.GetNumberOfTiers() == 3) ;
        assert(AC3Tier_.GetMinTatkalCharge() == 300) ;
        assert(AC3Tier_.GetMaxTatkalCharge() == 400) ;
        assert(AC3Tier_.GetReservationCharge() == 40.00) ;
        assert(AC3Tier_.GetMinTatkalDistance() == 500) ;
        assert(AC3Tier_.GetTatkalLoadFactor() == 0.3)  ;
```

Correct Assert

Expected:

1. sLuxury() should return false.
2. IsAc() should return true.
3. GetLoadFactor() should return 1.00.
4. GetName() should return “ Ac3Tier ”.
5. IsSitting() should return false.
6. GetMinTatkalCharge() should returns 300.
7. GetMaxTatkalCharge() should returns 400.
8. GetReservationCharge() should returns 40.00.
9. GetMinTatkalDistance() should returns 500.
10. GetTatkalLoadFactor() should returns 0.30.
11. GetNumberOfTiers() should return 3.

And similarly for the rest of the booking classes.

BOOKING CATEGORIES:

The booking categories class is a simple hierarchy to provide concession factors according to the category of the passenger. It contains the following child classes
General, SeniorCitizen, Ladies, Tatkal, PremiumTatkal, Divyaang

And other divyaang-classes are provided through divyaang abstract class which contains the following classes
TB, Orthopedic, Cancer, Blind

Unit tests of this are done by creating every type of booking category and then is checked/ asserted for correctness using getter functions.

Although we have assumed that this stays errorless but simple get can be tested like: TB::Type().GetName() -> Output: TB

BOOKING:

This is a very important test. Here, we will try to do bookings with booking categories, booking classes, passengers.

- try{
Date d1 = Date::createDate("7/04/2021") , d2 = Date::createDate("20/09/2021") ;
Passenger p = Passenger::createPassenger("Hari", "", "Kumar", "123456654321", "08/04/2001" , Gender::Male::Type(), "1234567890", 4, 12, "Blind");

```
		Booking b1 = Booking::DoBooking(Station("Delhi"), Station("Delhi"),
"10/04/2021" , "24/07/2021", BookingClasses::AC2Tier::Type(),
BookingCategories::General::Type(), p);
		assert(b1.GetFare() == 2994) ;
		// write the test number
	}
	catch (Bad_Booking& ex){
		cout << ex.what() << '\n';
	}
```

->Bad booking Caught because from station and to station are the same

- ```
  try{
  		Date d1 = Date::createDate("7/04/2021") , d2 =
  Date::createDate("20/09/2021") ;
  		Passenger p = Passenger::createPassenger("Hari", "", "Kumar",
  "123456654321", "08/04/2001" , Gender::Male::Type(), "1234567890", 4, 12,
  "Blind");
  		Booking b1 = Booking::DoBooking(Station("Mumbai"), Station("Delhi"),
  "10/04/2021" , "04/04/2021", BookingClasses::AC2Tier::Type(),
  BookingCategories::General::Type(), p);
  		assert(b1.GetFare() == 2994) ;
  		// write the test number
  	}
  	catch (Bad_Booking& ex){
  		cout << ex.what() << '\n';
  	}
  ```

->Bad Booking Caught because date of travel is more than date of booking

- ```
  try{
  		Date d1 = Date::createDate("07/04/2021") , d2 =
  Date::createDate("20/09/2021") ;
  		Passenger p = Passenger::createPassenger("Hari", "", "Kumar",
  "123456654321", "08/04/2001" , Gender::Male::Type(), "1234567890", 3, 12,
  "Blind");
  		Booking b1 = Booking::DoBooking(Station("Mumbai"), Station("Delhi"),
  "10/04/2021" , "04/04/2021", BookingClasses::AC2Tier::Type(),
  BookingCategories::General::Type(), p);
  		assert(b1.GetFare() == 2994) ;
  		// write the test number
  	}
  	catch (Bad_Booking& ex){
  ```

```
        cout << ex.what() << '\n';
    }
```

->Bad Booking Caught because divyaang category doesn’t match Blind

- try{
```
        Date d1 = Date::createDate("07/04/2021") , d2 =
Date::createDate("20/09/2021") ;
        Passenger p = Passenger::createPassenger("Hari", "", "Kumar",
"123456654321", "08/04/2001" , Gender::Male::Type(), "1234567890", 4, 12,
"Blind");
        Booking b1 = Booking::DoBooking(Station("Mumbai"), Station("Delhi"),
"10/04/2021" , "04/04/2021", BookingClasses::AC2Tier::Type(),
BookingCategories::General::Type(), p);
        assert(b1.GetFare() == 2994) ;
        // write the test number
    }
    catch (Bad_Booking& ex){
        cout << ex.what() << '\n';
    }
```

->Bad Booking Caught because divyaang category doesn’t match Blind