Software Engineering CS20006 -- Theory Assignment O4
Nakul Aggarwal 19CS10044
31 March 2021

UML DIAGRAMS

Use Case Diagram

Preface

- *Use Case Diagrams* are behaviour diagrams used to describe a set of actions (*use cases*) that systems perform in collaboration with one or more external users of the system (actors).
- Here the *use case diagram* for the *Booking Application* is designed.

Actors

- The only actor in the system is any *user* who is using the *application*.

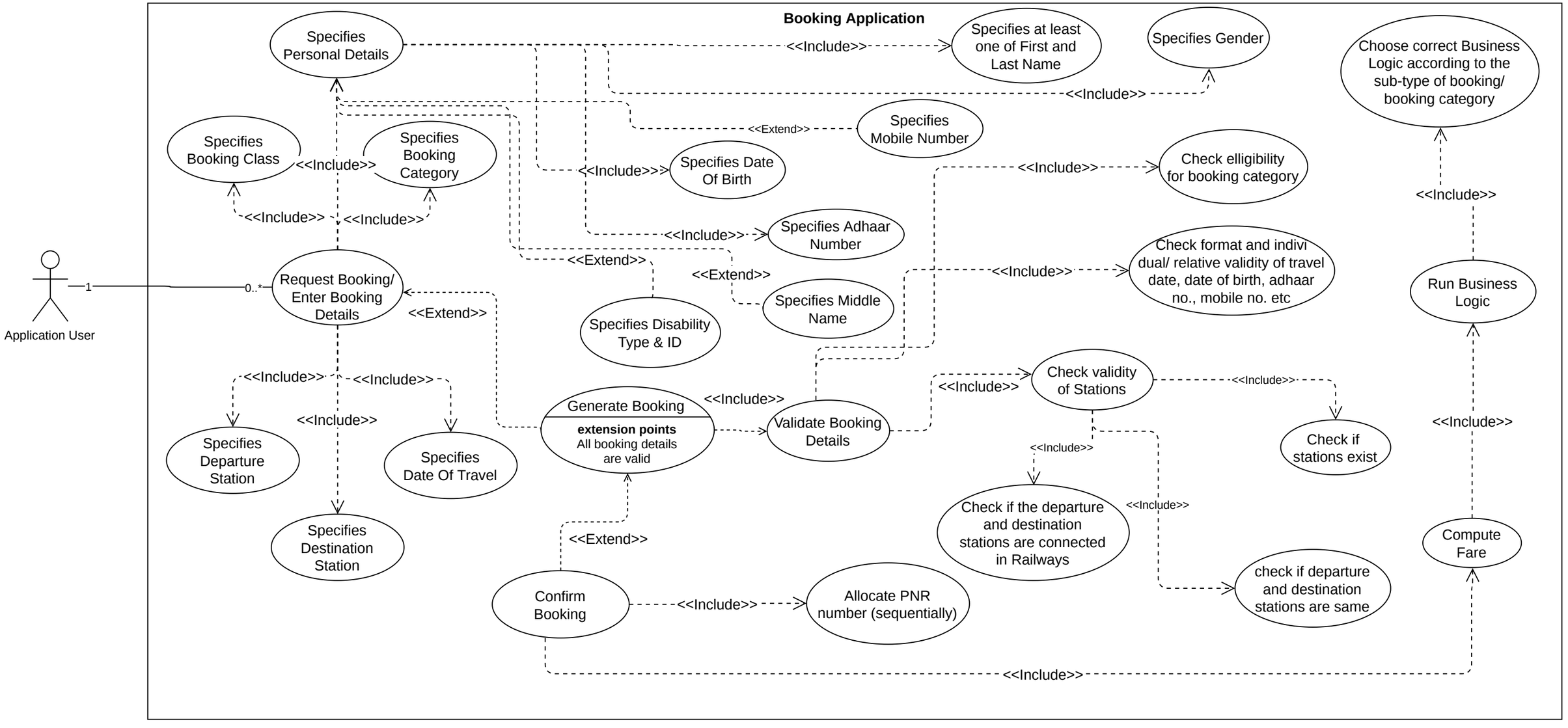
Use Case

- There is only one use case that is available to the *user* and that is *requesting a booking* or *enter booking details* to do booking. This is what the *booking application* stands for and no other functionalities and privileges are given to the user.
- *Booking request* being the only use case available to the actor doesnt make it the only use case in the system. There are several layers of *sub* and *sub-sub* and *sub-sub-sub* use cases behind the *primary* use case that check the validity of the provided details and finally confirm the task.

Relationships Among Use Cases

- There are *two kinds of relationships* that were observed among the various use cases in this system -- <<*include*>> and <<*extend*>>
- <<*include*>> relationship is used when one use case *includes* the behaviour of another use case in its sequence of events and actions and performance. This inclusion models a *compulsory* inclusion, that is, if use case A includes use case B, the use case B will be performed every time the use case A is performed.
- *include* relationship is the most frequently used relationship in this design because the involvement of most of the use cases into other use cases is *compulsory*. Take *Specifies Personal Details* for instance. It is *included* by *Enter Booking Details* because the latter use case *must* perform the former use case in all scenarios.

- <<*extend*>> relationship among use cases on the other hand is used to show optional system behaviour. This relationship is used to model two types of use cases, the one that are performed by the choice of the actor/user and the others that are performed if and only if some conditions are satisfied.
- The use cases *Specifies Mobile Number, Specifies Middle Name*, *Specifies Disability Type & ID* are all optional in the sense that these details need not be provided in order to be able to define a *passenger*. So it is the choice of the *actor* to provide or not provide these details and hence an *extend* relationship with the umbrella use case *Specifies Personal Details* is used, as opposed to *include* relationship.
- Notice the use case *Confirm Booking*. Whether or not this use case is performed is not controlled by the *actor* but is controlled by the the validity of the details provided by the actor. This use case is if *include*-ed in the use case *Generate Booking*, it will be performed every time the booking is told to be generated, irrespective of whether the details are valid or not. But this is not correct.
  Therefore here an *optional system behaviour* has to be modelled. *Confirm Booking* shares an *extend* relationship with *Generate Booking* with the *extension point -- "all booking details are valid"*. Like this the former use case will be performed only when the *extension condition* is applicable.
- Formally speaking, *extension points* are certain conditions only under which an optional system behaviour is extended.


Booking Application
Specifies Personal Details
Specifies at least one of First and Last Name
Specifies Gender
Choose correct Business Logic according to the sub-type of booking/ booking category
<<Include>>
<<Include>>
Specifies Mobile Number
<<Extend>>
Specifies Booking Class
Specifies Booking Category
<<Include>>
<<Include>>
<<Include>>
Specifies Date Of Birth
<<Include>>
Check elligibility for booking category
<<Include>>
<<Include>>
Specifies Adhaar Number
<<Include>>
Application User
1
0..*
Request Booking/ Enter Booking Details
<<Extend>>
<<Extend>>
Specifies Middle Name
Specifies Disability Type & ID
<<Include>>
Check format and indivi dual/ relative validity of travel date, date of birth, adhaar no., mobile no. etc
Run Business Logic
<<Include>>
<<Include>>
<<Include>>
<<Include>>
<<Include>>
Generate Booking
extension points
All booking details are valid
<<Include>>
Validate Booking Details
<<Include>>
Check validity of Stations
<<Include>>
Check if stations exist
<<Include>>
Specifies Departure Station
Specifies Date Of Travel
<<Include>>
<<Include>>
Specifies Destination Station
<<Extend>>
Check if the departure and destination stations are connected in Railways
<<Include>>
check if departure and destination stations are same
Compute Fare
Confirm Booking
<<Include>>
Allocate PNR number (sequentially)
<<Include>>

Like in a GUI application, the user has to click a button after providing all details to start the booking procedure. The user after entering all the details might abruptly shut down the console or not give a final "trigger" to "Book Ticket". Therefore this use case is also "extend"ed rather than "include"d. In a text-based interface, the user might have to give a final trigger, say press *ENTER* key to start the process

This is the only use case that is directly available to the user. This is what the booking application is about. In summary; this use case "includes" input of personal details of the user and the details specific to the booking/travel ticket. This first layer of sub-use-cases further includes another layer of sub-sub-use-cases that validate the provided information. Finally, if all the validations are successful, booking is confirmed.

When the user has to construct a Booking object, a Passenger object has to be passed as an arguement. Therefore, this use case must be "included" in order to take as input the details of the passenger

<<extend>> relationship among use cases is to highlight optional system behaviour. The attributes such as "middle name", "mobile number", "disability type" and "disability ID" are optional attributes and hence need not necessary "include" the master use case -- *Specifies Personal Details*

Subject/Heading

Diagram Frame

There is only one actor and that is the person who is using the application. Once the user is able to book successfully, she becomes a passenger

Association

Actor

Multiplicity

Use Case

Include Relationship

Extension Points

Extend Relationship

Booking Application

Specifies Personal Details

Specifies at least one of First and Last Name

Specifies Gender

Specifies Booking Class

Specifies Booking Category

Specifies Mobile Number

Specifies Date Of Birth

Specifies Adhaar Number

Specifies Middle Name

Specifies Disability Type & ID

Request Booking/ Enter Booking Details

Application User

1

0..*

Specifies Departure Station

Specifies Date Of Travel

Specifies Destination Station

Generate Booking

extension points
All booking details are valid

Validate Booking Details

Confirm Booking

Allocate PNR number (sequentially)

Check validity of Stations

Check if stations exist

Check if the departure and destination stations are connected in Railways

check if departure and destination stations are same

Check elligibility for booking category

Check format and indivi dual/ relative validity of travel date, date of birth, adhaar no., mobile no. etc

Choose correct Business Logic according to the sub-type of booking/ booking category

Run Business Logic

Compute Fare

<<Include>>

<<Extend>>

The fare computation logic for any booking depends on the booking category it has. Besides, even for the same booking category, the reservation charge/loading factor might depend on the booking class. Therefore once a booking object is created, an appropriate logic and an appropriate set of values will have to be considered in computing its fare.

This use case highlights at a high level the test of the validity, format and existence of all the values that are provided as input. Here "etc" might include booking class, booking category, date of reservation, disability type, gender. Relative validity means for example, date of travel can be valid if seen absolutely but relative to date of reservation, it should be after it and within 1 year from it

"Confirm Booking" includes two sub-use cases. First the PNR number is allocated sequentially (in a strictly increasing fashion from 1). Secondly, fare is computed for the booking. These two sub-use cases complete the booking procedure with success.

"Generate Booking" use case necessarily includes the validation of all details and hence "include" relationship is used with the "Validate Booking Details" sub-use case. But the other sub-use-case of "Confirm Booking" is *conditional* because it can take place if and only if all the details are validated. Hence here "extend" relationship is used. There is only one *extension point -- "All booking details are valid"*

Class Diagram

Preface

- *Class Diagram* is a *UML structure diagram* which shows structure of the designed system at the level of classes and interfaces, shows their features, constraints and relationships -- like associations, generalizations and dependencies.
- Here the *class diagram* for the *Booking Application* is designed.

Classes

- Various kinds of classes are designed in this class diagram -- *stand-alone classes* (ones that are neither an instance of a template nor a part of hierarchy), *abstract classes, template classes*, *singleton classes* and obviously some *derived/concrete classes*.
- In summary, the following classes are shown in the *class diagram*
    - *Date*
    - *Railways*
    - *Station*
    - *Passenger*
    - *Divyaang (complete inclusion parametric-polymorphic hierarchy)*
    - *Gender (complete inclusion parametric-polymorphic hierarchy)*
    - *BookingClass (complete inclusion parametric-polymorphic hierarchy)*
    - *BookingCategory (complete inclusion parametric-polymorphic hierarchy)*
    - *Booking (complete inclusion parametric-polymorphic hierarchy)*
    - *Concessions (complete ad-hoc-polymorphic hierarchy)*
- The features of every class are shown on a very low level of abstraction, with specification of all the *static/non-static data members* and *static/non-static/ abstract member functions*.

Relationships and Dependencies

- Various kinds of relationships and dependencies are used in the diagram -- *association, composition, generalization, binding dependency* and *constraints*.
- *Association* is the most frequently used relationship in this diagram, perhaps because of the flexibility it provides in representing almost any situation. Here several bunches of similar *association* relationships are used (along with obviously some other unrelated *associations*), in order to model the internal features of the booking application at a lower level. Some important ones from them include --
    - *Booking-BookingCategory* tethering -- By the virtue of the *virtual construction idiom*, these two hierarchies are tightly intertwined and this is

shown by a bunch of association relationships that run from each of the static sub types of *Booking* to the corresponding sub type of *BookingCategory*. This bunch relationships shows that for every *BookingCategory* a *specialized Booking class* is defined.

- *Concession dependencies* -- A bunch of association relationships runs from each of those *Booking sub-types* for which concessions are available, to the classes whose attributes determine the concession factor applicabe to any instance of the sub type. For example, the concession for *SeniorCitizenBooking* depends on the gender of the passenger and hence an association relationship runs from *SeniorCitizenBooking* to *Passenger* with the relationship description -- *"concession factor depends on gender of"*.

Other groups of association relationships highlight *fare computation*, *eligibility of BookingCategory sub-types* and *data encapsulated by classes in Concessions hierarchy*, which are explicitly annotated in the second diagram.

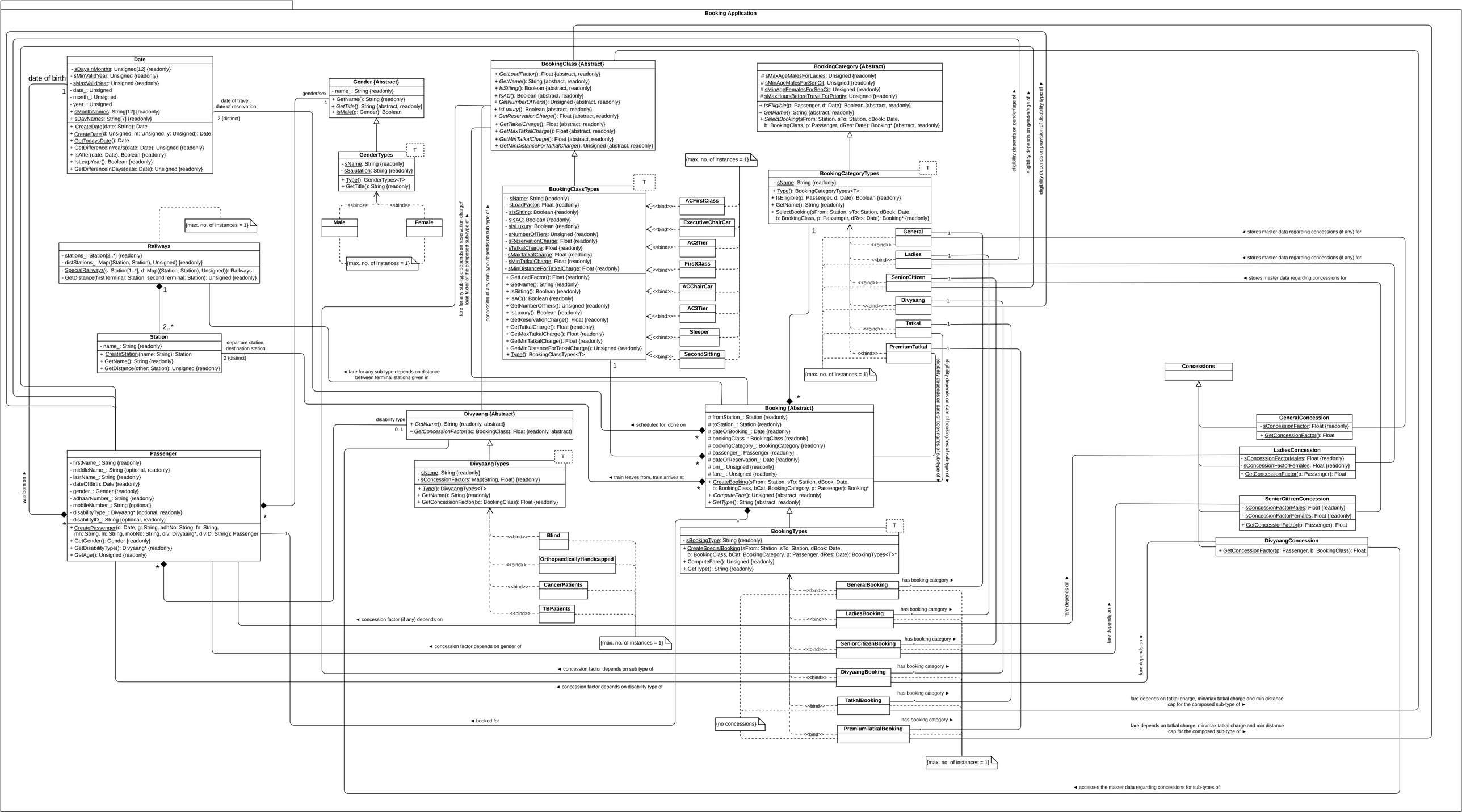
- *Composition* or *strong aggregations* are used to depict *HAS-A* relationships. These relationships are easy to see when the attributes of the classes are shown at a very low level.
- The classes that are *composed* of instances of other classes include *Railways, Passenger* and *Booking*. The relationship is a *strong* aggregation because these classes are *meaningless* without the instances of other classes it is composed of. For example, a *Railways* is meaningless without *Station*, *Booking* is meaningless without *Passenger* and *Passenger* is meaningless without *Gender* or *Date* of birth. It can sometimes also be percieved as a relationship of *necessary co-existence*. If there is a *Station*, there must be a *Railways* network it is a part of; and if there is a *Railways,* there must be a *Station* it comprises.

- *Generalization* relationship is shared between generalized and specialized classes. For example, *Concessions* is a generalization of *GeneralConcession* and *BookingCategory* is a generalization of *BookingCategoryTypes* template.

- *Binding dependencies* here are used to represent depict the classes that are an instance of a template. Classes modelled on a template share a *binding* relationship with their template.
- For example, *Male* is a *bound* to *GenderTypes* because it modelled on this template. Same is the case with all static sub-types of *Booking, BookingClass, BookingCategory* and *Divyaang*.

- *Constraints* are used to showcase some characteristic behaviour of some classes. Here constraints are mainly used to represent *singleton classes*. A constraint of *“max number of instances = 1”* is put on class instances of each of the template classes, and also *Railways*, to depict that these are *singleton classes* and can be instantiated at most once.
- Constraints can also be used to provide *non-semantic characteristics* like *GeneralBooking*, *TatkalBooking* and *PremiumTatkalBooking* having *no concessions* is a constraint given in the specification document of the application.

Booking Application



Names of static data members and methods are underlined

- symbol is an indicator of *private access specifier in C++*

Names of abstract methods are *italicised*

+ symbol is an indicator of *public access specifier in C++*

Abstract class notation

Constraint

Abstract Class

Domain covered by the *class diagram*

Template Class

symbol is an indicator of *protected access specifier in C++*

Generalization relationship

Template class notation

Binding dependency

Class

Atributes Compartment

Operations Compartment

Associate end name / role name

Composition / Strong aggregation relationship (highlights HAS-A relationship)

Comma separated role names

Constraint on multiple compositions of same type.

Reading order

Relationship description

readonly properties/operations highlight *const* data members/member functions in C++

Association relationship (binary)

Multiplicity

Booking Application

Date
- sDaysInMonths: Unsigned[12] {readonly}
- sMinValidYear: Unsigned {readonly}
- sMaxValidYear: Unsigned {readonly}
- date_: Unsigned
- month_: Unsigned
- year_: Unsigned
- sMonthNames: String[12] {readonly}
- sDayNames: String[7] {readonly}

+ CreateDate(date: String): Date
+ CreateDate(d: Unsigned, m: Unsigned, y: Unsigned): Date
+ GetTodaysDate(): Date
+ GetDifferenceInYears(date: Date): Unsigned {readonly}
+ IsAfter(date: Date): Boolean {readonly}
+ IsLeapYear(): Boolean {readonly}
+ GetDifferenceInDays(date: Date): Unsigned {readonly}

date of birth

1

date of travel, date of reservation

2 {distinct}

Gender {Abstract}
- name_: String {readonly}

+ GetName(): String {readonly}
+ *GetTitle*(): String {abstract, readonly}
+ IsMale(g: Gender): Boolean

gender/sex

1

GenderTypes
- sName: String {readonly}
- sSalutation: String {readonly}

+ Type(): GenderTypes<T>
+ GetTitle(): String {readonly}

T

<<bind>>

Male

Female

{max. no. of instances = 1}

Railways
- stations_: Station[2..*] {readonly}
- distStations_: Map({Station, Station), Unsigned} {readonly}

- SpecialRailways(s: Station[1..*], d: Map({Station, Station), Unsigned)): Railways
- GetDistance(firstTerminal: Station, secondTerminal: Station): Unsigned {readonly}

1

2..*

Station
- name_: String {readonly}

+ CreateStation(name: String): Station
+ GetName(): String {readonly}
+ GetDistance(other: Station): Unsigned {readonly}

departure station, destination station

2 {distinct}

◄ fare for any sub-type depends on distance between terminal stations given in

Passenger
- firstName_: String {readonly},
- middleName_: String {optional, readonly}
- lastName_: String {readonly}
- dateOfBirth: Date {readonly}
- gender_: Gender {readonly}
- adhaarNumber_: String {readonly}
- mobileNumber_: String {optional}
- disabilityType_: Divyaang* {optional, readonly}
- disabilityID_: String {optional, readonly}

+ CreatePassenger(dl: Date, g: String, adhNo: String, fn: String, mn: String, ln: String, mobNo: String, div: Divyaang*, divID: String): Passenger
+ GetGender(): Gender {readonly}
+ GetDisabilityType(): Divyaang* {readonly}
+ GetAge(): Unsigned {readonly}

was born on ▲

*

1

BookingClass {Abstract}

+ *GetLoadFactor*(): Float {abstract, readonly}
+ *GetName*(): String {abstract, readonly}
+ *IsSitting*(): Boolean {abstract, readonly}
+ *IsAC*(): Boolean {abstract, readonly}
+ *GetNumberOfTiers*(): Unsigned {abstract, readonly}
+ *IsLuxury*(): Boolean {abstract, readonly}
+ *GetReservationCharge*(): Float {abstract, readonly}
+ *GetTatkalCharge*(): Float {abstract, readonly}
+ *GetMaxTatkalCharge*(): Float {abstract, readonly}
+ *GetMinTatkalCharge*(): Float {abstract, readonly}
+ *GetMinDistanceForTatkalCharge*(): Unsigned {abstract, readonly}

BookingClassTypes
- sName: String {readonly}
- sLoadFactor: Float {readonly}
- sIsSitting: Boolean {readonly}
- sIsAC: Boolean {readonly}
- sIsLuxury: Boolean {readonly}
- sNumberOfTiers: Unsigned {readonly}
- sReservationCharge: Float {readonly}
- sTatkalCharge: Float {readonly}
- sMaxTatkalCharge: Float {readonly}
- sMinTatkalCharge: Float {readonly}
- sMinDistanceForTatkalCharge: Float {readonly}

+ GetLoadFactor(): Float {readonly}
+ GetName(): String {readonly}
+ IsSitting(): Boolean {readonly}
+ IsAC(): Boolean {readonly}
+ GetNumberOfTiers(): Unsigned {readonly}
+ IsLuxury(): Boolean {readonly}
+ GetReservationCharge(): Float {readonly}
+ GetTatkalCharge(): Float {readonly}
+ GetMaxTatkalCharge(): Float {readonly}
+ GetMinTatkalCharge(): Float {readonly}
+ GetMinDistanceForTatkalCharge(): Unsigned {readonly}
+ Type(): BookingClassTypes<T>

T

<<bind>>

ACFirstClass

ExecutiveChairCar

AC2Tier

FirstClass

ACChairCar

AC3Tier

Sleeper

SecondSitting

{max. no. of instances = 1}

fare for any sub-type depends on reservation charge/ total fation of the composed sub-type of ▲

concession of any sub-type depends on sub-type of ▲

BookingCategory {Abstract}
sMaxAgeMalesForLadies: Unsigned {readonly}
sMinAgeMalesForSenCit: Unsigned {readonly}
sMinAgeFemalesForSenCit: Unsigned {readonly}
sMaxHoursBeforeTravelForPriority: Unsigned {readonly}

+ *IsEligible*(p: Passenger, d: Date): Boolean {abstract, readonly}
+ *GetName*(): String {abstract, readonly}
+ *SelectBooking*(sFrom: Station, sTo: Station, dBook: Date, b: BookingClass, p: Passenger, dRes: Date): Booking* {abstract, readonly}

eligibility depends on ▲ of booking type

eligibility depends on specific ▲ of concession

eligibility depends on provision of disability type of ▲

BookingCategoryTypes
- sName: String {readonly}

+ Type(): BookingCategoryTypes<T>
+ IsEligible(p: Passenger, d: Date): Boolean {readonly}
+ GetName(): String {readonly}
+ SelectBooking(sFrom: Station, sTo: Station, dBook: Date, b: BookingClass, p: Passenger, dRes: Date): Booking* {readonly}

T

1

<<bind>>

General

Ladies

SeniorCitizen

Divyaang

Tatkal

PremiumTatkal

{max. no. of instances = 1}

◄ stores master data regarding concessions (if any) for

◄ stores master data regarding concessions (if any) for

◄ stores master data regarding concessions for

Concessions

GeneralConcession
- sConcessionFactor: Float {readonly}

+ GetConcessionFactor(): Float

LadiesConcession
- sConcessionFactorMales: Float {readonly}
- sConcessionFactorFemales: Float {readonly}

+ GetConcessionFactor(p: Passenger): Float

SeniorCitizenConcession
- sConcessionFactorMales: Float {readonly}
- sConcessionFactorFemales: Float {readonly}

+ GetConcessionFactor(p: Passenger): Float

DivyaangConcession

+ GetConcessionFactor(p: Passenger, b: BookingClass): Float

Booking {Abstract}
fromStation_: Station {readonly}
toStation_: Station {readonly}
dateOfBooking_: Date {readonly}
bookingClass_: BookingClass {readonly}
bookingCategory_: BookingCategory {readonly}
passenger_: Passenger {readonly}
dateOfReservation_: Date {readonly}
pnr_: Unsigned {readonly}
fare_: Unsigned {readonly}

+ CreateBooking(sFrom: Station, sTo: Station, dBook: Date, b: BookingClass, bCat: BookingCategory, p: Passenger): Booking*
+ *ComputeFare*(): Unsigned {abstract, readonly}
+ *GetType*(): String {abstract, readonly}

*

◄ scheduled for, done on

◄ train leaves from, train arrives at

eligibility depends on date of booking/no's of sub-type of ▲

BookingTypes
- sBookingType: String {readonly}

+ CreateSpecialBooking (sFrom: Station, sTo: Station, dBook: Date, b: BookingClass, bCat: BookingCategory, p: Passenger, dRes: Date): BookingTypes<T>*
+ ComputeFare(): Unsigned {readonly}
+ GetType(): String {readonly}

T

<<bind>>

GeneralBooking

LadiesBooking

SeniorCitizenBooking

DivyaangBooking

TatkalBooking

PremiumTatkalBooking

has booking category ►

{no concessions}

{max. no. of instances = 1}

fare depends on ▲

fare depends on ▲

fare depends on ▲

fare depends on tatkal charge, min/max tatkal charge and min distance cap for the composed sub-type of ►

fare depends on tatkal charge, min/max tatkal charge and min distance cap for the composed sub-type of ►

◄ accesses the master data regarding concessions for sub-types of

Divyaang {Abstract}
+ *GetName*(): String {readonly, abstract}
+ *GetConcessionFactor*(bc: BookingClass): Float {readonly, abstract}

disability type

0..1

DivyaangTypes
- sName: String {readonly}
- sConcessionFactors: Map(String, Float) {readonly}

+ Type(): DivyaangTypes<T>
+ GetName(): String {readonly}
+ GetConcessionFactor(bc: BookingClass): Float {readonly}

T

Blind

OrthopaedicallyHandicapped

CancerPatients

TBPatients

◄ concession factor (if any) depends on

◄ concession factor depends on gender of

◄ concession factor depends on sub type of

◄ concession factor depends on disability type of

◄ booked for

{max. no. of instances = 1}

How is the eligibility of a passenger for a *BookingCategory sub-type* decided?

What is the significance of the data encapsulated by the classes in *Concessions* hierarchy?

How are the *static sub-types of Booking* tethered with the *static sub-types of BookingCategory*? For every concrete sub-type of *BookingCategory*, a specialized *Booking* sub-type is designed.

What are some common factors among all the *Booking* sub-types that are considered while computing fare? (these include distance between terminal stations, loading/reservation charges of *BookingClass* sub-type etc)

Abstract Class

Template Class

How are the concession factors applicable to an instance of any *Booking sub-type* depend on various factors/attributes?

How is fare of any instance of any *Booking* sub-type computed from the values of other types, or the values stored in the *static* data of other classes? This bunch of relationships models the dependence of fare of a booking on different factors, specific to each sub-type.

Sequence Diagram

Preface

- *Sequence Diagram* is the most common kind of *interaction diagram* which focuses on the message interchange between a numbe rof lifelines.
- It is a *UML behaviour diagram* that depicts the inter-object behaviour of a system, ordered by time.
- The major activities that support the client side of the *booking application* are -- *Booking Request* and *Make Passenger*. Note that *Booking Request* is the *primary activity* that the application is used for. But requesting a booking, or in context of *C++*, constructing a booking object requires *Passenger* as an argument. Therefore, *Passenger* also has to be instantiated on the client-side which becomes the secondary activity.
  Both the activities should have there own separate *sequence diagrams*.

Lifelines

- *Lifeline* is an element which represents an individual participant in the interaction.
- Here interaction is happening among *classes* (as designed in the *class diagram*) and also between the *classes* and the *application side*/*client side*. Reading through the specifications and the last two UML diagrams, the following major participants can be identified for the *Booking Request* sequence diagram.
  - *Application*
  - *Booking*
  - *Railways*
  - *Date*
  - *BookingCategory Sub Types*
  - *Booking Sub Types*

  Similarly, the following major participants can be identified for the *Booking Request* sequence diagram.
  - *Application*
  - *Passenger*
  - *Date*

Messages

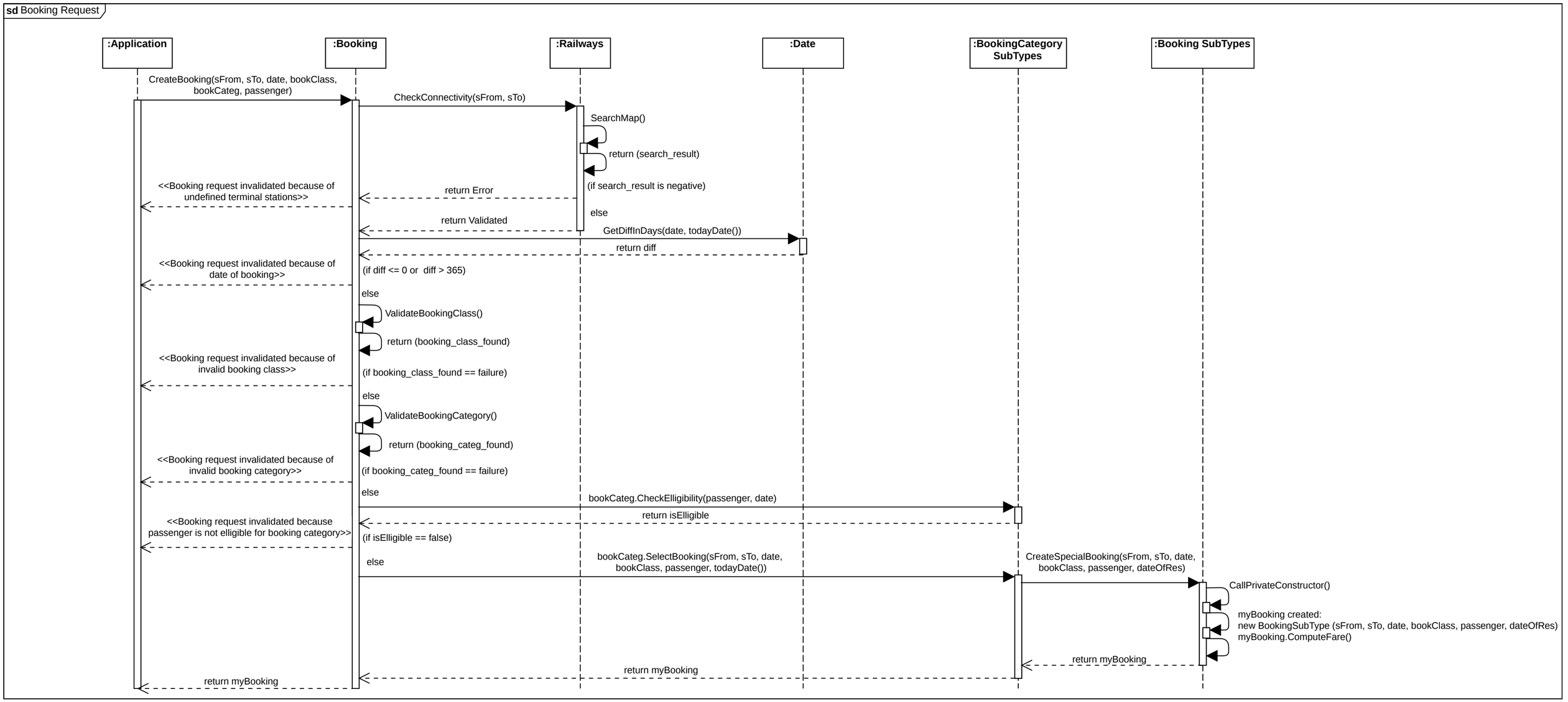
- The kinds of messages that are used in both the sequence diagrams are -- *synchronous message* and *reply message*. Both are *messages by action type*.
- The messages in *Booking Request* sequence diagram can be divided into two groups.
  - Messages meant to validate the details that are ultimately used to *create booking.*

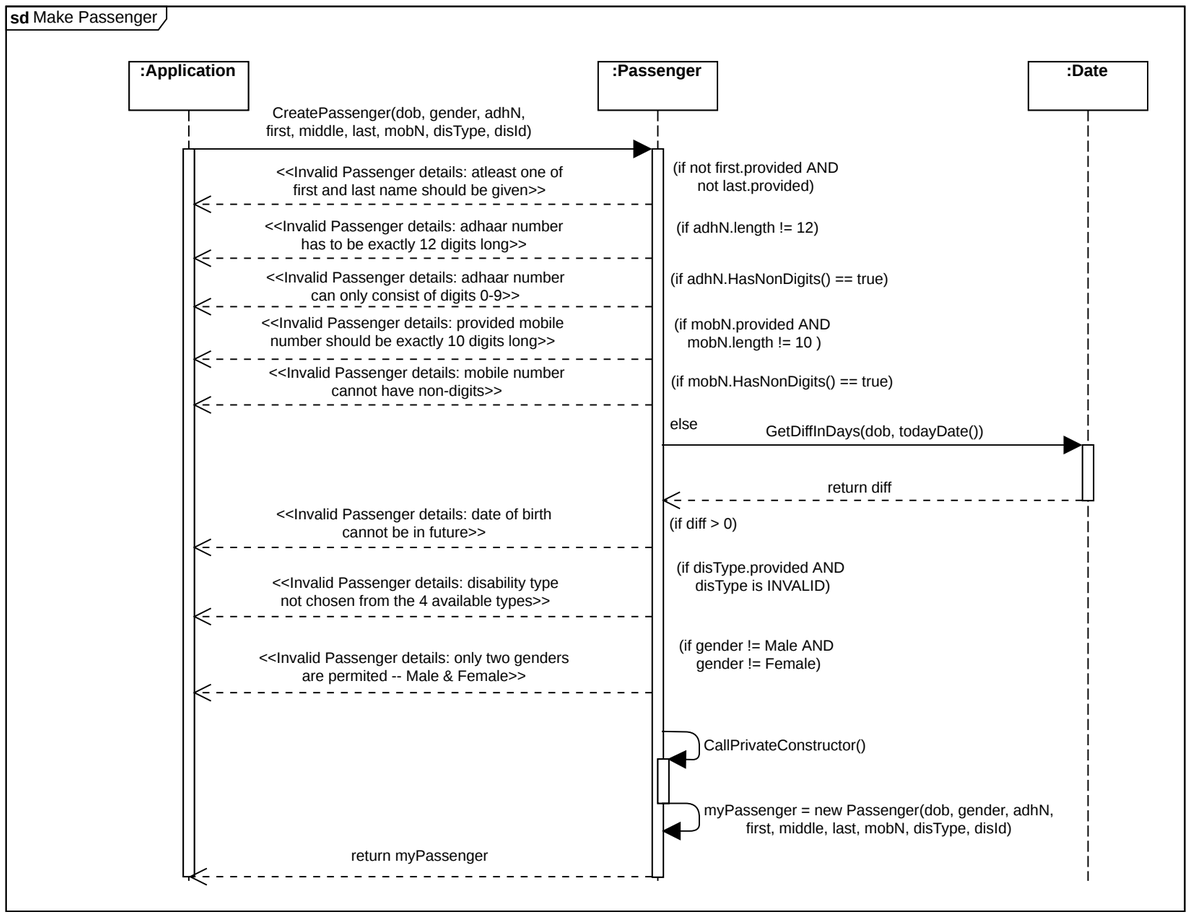
    - Messages that (in terms of C++) represent the virtual construction idiom in which depending on the *static sub type* of *BookingCategory*, the accurate *sub type* of *Booking* is instantiated using these details (only if they are validated)
- Similarly, the messages in *Make Passenger* sequence diagram can also be divided into two groups.
    - Messages meant to validate the details that are ultimately used to *create a passenger.*
    - Messages meant to actually *create* a passenger with these details when they are validated.

Interaction Fragments

- There are various kinds of *interaction fragments* but not all of them are relevant here. The ones that are shown in these sequence diagrams include -- *occurrence*, *execution/activation* and *overlapping execution*.
- *Occurrence* is interaction fragment which represents a moment in time (event) at the beginning or end of a message or at the beginning or end of an execution. Naturally there are two types of occurrences applicable to the lifetime of an execution -- *start occurrence* and *finish occurrence*. Besides, there are *message occurrences* also used here that represent events of sending and receiving signals/messages.
- *Execution (Activation)* is an interaction fragment which represents a period in the participant's lifetime when it is executing a unit of behaviour or action within the lifetime, or sending a signal to another participant or waiting for a reply message from another participant.

  All the lifelines here have an *execution* interaction fragment. Some lifelines that exhibit activity/behaviour(s) for a longer span (like *Application*) have longer *execution periods* and others like *Date* have very short *execution periods*.
- *Overlapping execution specifications* on the same lifeline are represented by overlapping rectangles. These are used to highlight some *key behaviours* happening internally in a *lifeline*. For example *SearchMap* message is sent to an overlapped execution fragment in *Railways* lifeline just to highlight the vital routine or algorithm that is(are) used during the period of *execution*.


sd Booking Request
:Application
:Booking
:Railways
:Date
:BookingCategory SubTypes
:Booking SubTypes
CreateBooking(sFrom, sTo, date, bookClass, bookCateg, passenger)
CheckConnectivity(sFrom, sTo)
SearchMap()
return (search_result)
(if search_result is negative)
return Error
<<Booking request invalidated because of undefined terminal stations>>
else
return Validated
GetDiffInDays(date, todayDate())
return diff
<<Booking request invalidated because of date of booking>>
(if diff <= 0 or diff > 365)
else
ValidateBookingClass()
return (booking_class_found)
<<Booking request invalidated because of invalid booking class>>
(if booking_class_found == failure)
else
ValidateBookingCategory()
return (booking_categ_found)
<<Booking request invalidated because of invalid booking category>>
(if booking_categ_found == failure)
else
bookCateg.CheckElligibility(passenger, date)
return isElligible
<<Booking request invalidated because passenger is not elligible for booking category>>
(if isElligible == false)
else
bookCateg.SelectBooking(sFrom, sTo, date, bookClass, passenger, todayDate())
CreateSpecialBooking(sFrom, sTo, date, bookClass, passenger, dateOfRes)
CallPrivateConstructor()
myBooking created:
new BookingSubType (sFrom, sTo, date, bookClass, passenger, dateOfRes)
myBooking.ComputeFare()
return myBooking
return myBooking
return myBooking


sd Make Passenger
:Application
:Passenger
:Date
CreatePassenger(dob, gender, adhN, first, middle, last, mobN, disType, disId)
<<Invalid Passenger details: atleast one of first and last name should be given>>
(if not first.provided AND not last.provided)
<<Invalid Passenger details: adhaar number has to be exactly 12 digits long>>
(if adhN.length != 12)
<<Invalid Passenger details: adhaar number can only consist of digits 0-9>>
(if adhN.HasNonDigits() == true)
<<Invalid Passenger details: provided mobile number should be exactly 10 digits long>>
(if mobN.provided AND mobN.length != 10)
<<Invalid Passenger details: mobile number cannot have non-digits>>
(if mobN.HasNonDigits() == true)
else
GetDiffInDays(dob, todayDate())
return diff
<<Invalid Passenger details: date of birth cannot be in future>>
(if diff > 0)
<<Invalid Passenger details: disability type not chosen from the 4 available types>>
(if disType.provided AND disType is INVALID)
<<Invalid Passenger details: only two genders are permited -- Male & Female>>
(if gender != Male AND gender != Female)
CallPrivateConstructor()
myPassenger = new Passenger(dob, gender, adhN, first, middle, last, mobN, disType, disId)
return myPassenger

Sequence Diagram for Booking Application -- *Booking Request*

Lifelines

Heading/Subject

Lifeline -- Class Name

Lifeline head

Lifeline tail: this vertical line represents the lifetime of the participant

sd Booking Request

:Application | **:Booking** | **:Railways** | **:Date** | **:BookingCategory SubTypes** | **:Booking SubTypes**

Start Occurence

Synchronous Message

Execution Specification

Return Message

Overlapping Execution

Check for inconsistencies/invalidities in the arguements-- One by one check the validity of each of the arguements. If any erroneous scenario is encountered, do not proceed further and return an appropriate message to the *Application* side.

Finish Occurence

CreateBooking(sFrom, sTo, date, bookClass, bookCateg, passenger)

CheckConnectivity(sFrom, sTo)

SearchMap()

return (search_result)

(if search_result is negative)

return Error

<<Booking request invalidated because of undefined terminal stations>>

else

return Validated

GetDiffInDays(date, todayDate())

return diff

<<Booking request invalidated because of date of booking>>

(if diff <= 0 or diff > 365)

else

ValidateBookingClass()

return (booking_class_found)

<<Booking request invalidated because of invalid booking class>>

(if booking_class_found == failure)

else

ValidateBookingCategory()

return (booking_categ_found)

<<Booking request invalidated because of invalid booking category>>

(if booking_categ_found == failure)

else

bookCateg.CheckElligibility(passenger, date)

return isElligible

<<Booking request invalidated because passenger is not elligible for booking category>>

(if isElligible == false)

else

bookCateg.SelectBooking(sFrom, sTo, date, bookClass, passenger, todayDate())

CreateSpecialBooking(sFrom, sTo, date, bookClass, passenger, dateOfRes)

CallPrivateConstructor()

myBooking created:
new BookingSubType (sFrom, sTo, date, bookClass, passenger, dateOfRes)
myBooking.ComputeFare()

return myBooking

return myBooking

return myBooking

Execution/Activation : represents a period in a paricipant's lifetime

Return message based on condition(s)

Sequence Diagram for Booking Application -- *Make Passenger*

This sequence of message interchanges is secondary to the *Booking request* sequence diagram. As seen there, a *Passenger* object (in terms of C++) is needed to create a booking. But creating a passenger itself is a sequence of interactions. These two are the major interactions between the application and other lifelines at the back-end.

Lifeline -- Class Name

Start Occurence

Synchronous Message

Follow the same strategy as in the Booking Request sequence diagram. Check for inconsistencies/invalidities in the arguements-- One by one check the validity of each of the arguements. If any erroneous scenario is encountered, do not proceed further and return an appropriate message to the *Application* side.

Return Message

Finish Occurence

sd Make Passenger

:Application | **:Passenger** | **:Date**

CreatePassenger(dob, gender, adhN, first, middle, last, mobN, disType, disId)

<<Invalid Passenger details: atleast one of first and last name should be given>>

(if not first.provided AND not last.provided)

<<Invalid Passenger details: adhaar number has to be exactly 12 digits long>>

(if adhN.length != 12)

<<Invalid Passenger details: adhaar number can only consist of digits 0-9>>

(if adhN.HasNonDigits() == true)

<<Invalid Passenger details: provided mobile number should be exactly 10 digits long>>

(if mobN.provided AND mobN.length != 10)

<<Invalid Passenger details: mobile number cannot have non-digits>>

(if mobN.HasNonDigits() == true)

else

GetDiffInDays(dob, todayDate())

return diff

<<Invalid Passenger details: date of birth cannot be in future>>

(if diff > 0)

<<Invalid Passenger details: disability type not chosen from the 4 available types>>

(if disType.provided AND disType is INVALID)

<<Invalid Passenger details: only two genders are permited -- Male & Female>>

(if gender != Male AND gender != Female)

CallPrivateConstructor()

myPassenger = new Passenger(dob, gender, adhN, first, middle, last, mobN, disType, disId)

return myPassenger

Return message based on condition(s)

Lifeline head

(1.) Call to polymorphic method of *BookingCategory* hierarchy gets dynamically dispatched to the overridden method of the appropriate static sub-type.

(2.) The overridden method in the static sub type of *BookingCategory* calls the *public static method* of the corresponding *Booking sub-type.*

(3.) The *static member function* in the *Booking sub-type* calls the private constructor to instantiate it.

(4.) Finally return the constructed object.

Lifelines

Lets try to understand the *virtual construction idiom* here in *C++* context. *Sequence Diagrams* make this design more understandable

Execution/Activation : represents a period in a paricipant's lifetime

Lifeline tail: this vertical line represents the lifetime of the participant

Overlapping Execution

Communication Diagram

Preface

- *Communication Diagram* shows interactions between objects and/or parts (represented as lifelines) using *sequenced messages* in a free-form arrangement.
- It is a *UML behaviour diagram* that depicts the inter-object behaviour of a system, ordered by space.

Frame

- *Communication diagram* is shown within a rectangular frame with the name in a compartment in the upper left corner. The frame can be of various types like *interaction* and *sd (short frame)*
- In this design, an *interaction frame* called *Booking Application* is used.

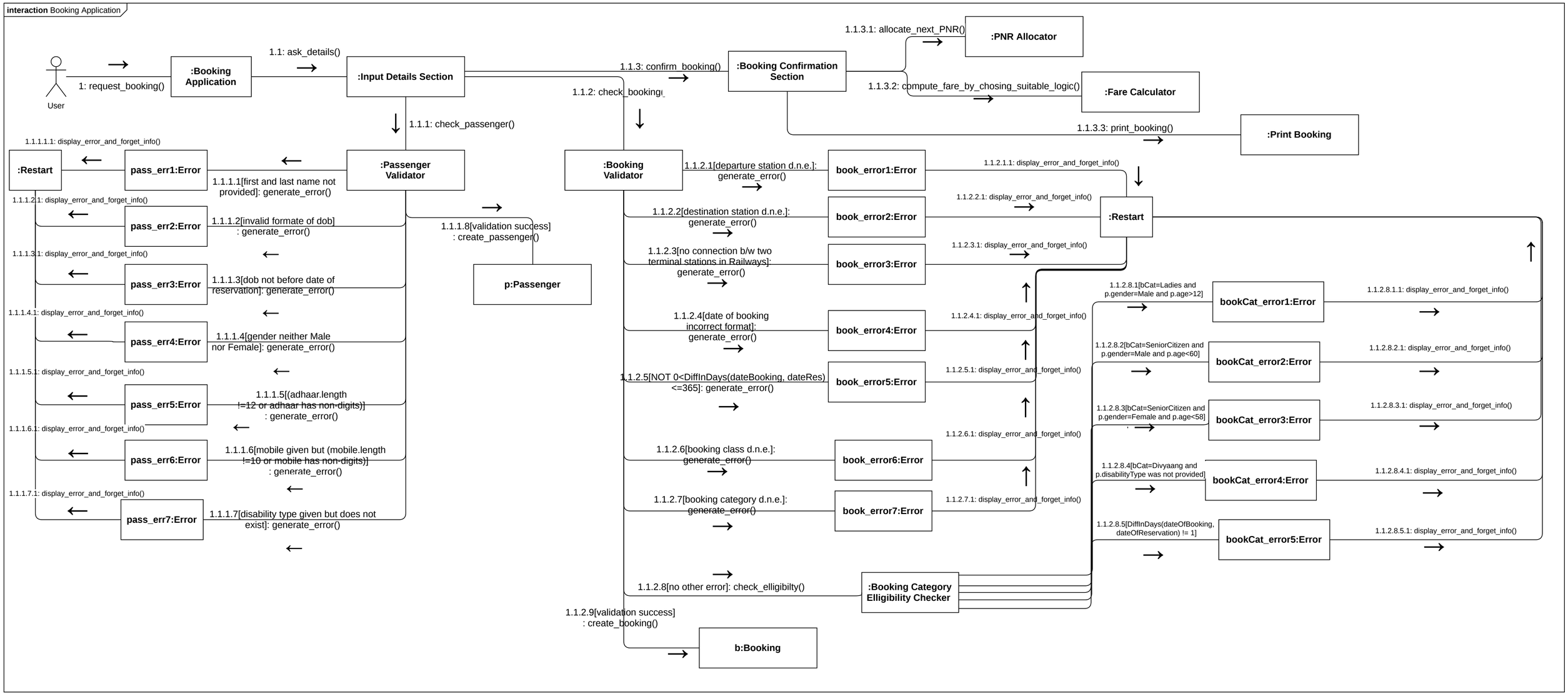
Lifeline

- *Lifeline* is a specialization of named element which represents an individual participant in the interaction.
- Lifelines have a *class name* and may or may not have an *object name*. Names (class names) of the lifelines used in the diagram are --
    - *User*
    - *Booking Application*
    - *Input Details Section*
    - *Passenger Validator*
    - *Booking Validator*
    - *Booking Category Eligibility Checker*
    - *Error*
    - *Restart*
    - *Booking*
    - *Passenger*
    - *Booking Confirmation Section*
    - *PNR Allocator*
    - *Fare Calculator*
    - *Print Booking*
- Some participants like *Passenger* and *Booking* are shown with their *object names*, simply to reflect that in context of a programming language, say *C++*, the objects of these classes are *constructed* as a result of all the preceeding interactions that are mostly meant to validate the input details.
- Lifelines with *Error* class name are allotted an object name so that they can be re-used several times, without having to introduce a new class of lifelines all together. In context of *C++*, this can be perceived as different kinds of

exceptions, each one of which is thrown for a specific and a very special kind of inconsistency in the input details.

Messages

- Messages are shown as a line with a *sequence expression* and arrow above the line.
- Like in *sequence diagram* here also there is a class of messages that is meant to check the invalidities in the input details. But, obviously, the syntax of the messages is different.
- Here for the messages that are meant to check the *erroneous scenarios*, a *guard* condition is provided along with the sequential order. Different possible error conditions are checked at different depths of nesting.


interaction Booking Application
User
1: request_booking()
:Booking Application
1.1: ask_details()
:Input Details Section
1.1.1: check_passenger()
:Passenger Validator
1.1.1.1[first and last name not provided]: generate_error()
pass_err1:Error
1.1.1.1.1: display_error_and_forget_info()
:Restart
1.1.1.2[invalid formate of dob] : generate_error()
pass_err2:Error
1.1.1.2.1: display_error_and_forget_info()
1.1.1.3[dob not before date of reservation]: generate_error()
pass_err3:Error
1.1.1.3.1: display_error_and_forget_info()
1.1.1.4[gender neither Male nor Female]: generate_error()
pass_err4:Error
1.1.1.4.1: display_error_and_forget_info()
1.1.1.5[(adhaar.length !=12 or adhaar has non-digits)] : generate_error()
pass_err5:Error
1.1.1.5.1: display_error_and_forget_info()
1.1.1.6[mobile given but (mobile.length !=10 or mobile has non-digits)] : generate_error()
pass_err6:Error
1.1.1.6.1: display_error_and_forget_info()
1.1.1.7[disability type given but does not exist]: generate_error()
pass_err7:Error
1.1.1.7.1: display_error_and_forget_info()
1.1.1.8[validation success] : create_passenger()
p:Passenger
1.1.2: check_booking()
:Booking Validator
1.1.2.1[departure station d.n.e.]: generate_error()
book_error1:Error
1.1.2.1.1: display_error_and_forget_info()
1.1.2.2[destination station d.n.e.]: generate_error()
book_error2:Error
1.1.2.2.1: display_error_and_forget_info()
1.1.2.3[no connection b/w two terminal stations in Railways]: generate_error()
book_error3:Error
1.1.2.3.1: display_error_and_forget_info()
1.1.2.4[date of booking incorrect format]: generate_error()
book_error4:Error
1.1.2.4.1: display_error_and_forget_info()
1.1.2.5[NOT 0<DiffInDays(dateBooking, dateRes) <=365]: generate_error()
book_error5:Error
1.1.2.5.1: display_error_and_forget_info()
1.1.2.6[booking class d.n.e.]: generate_error()
book_error6:Error
1.1.2.6.1: display_error_and_forget_info()
1.1.2.7[booking category d.n.e.]: generate_error()
book_error7:Error
1.1.2.7.1: display_error_and_forget_info()
1.1.2.8[no other error]: check_elligibilty()
:Booking Category Elligibility Checker
1.1.2.9[validation success] : create_booking()
b:Booking
1.1.3: confirm_booking()
:Booking Confirmation Section
1.1.3.1: allocate_next_PNR()
:PNR Allocator
1.1.3.2: compute_fare_by_chosing_suitable_logic()
:Fare Calculator
1.1.3.3: print_booking()
:Print Booking
:Restart
1.1.2.8.1[bCat=Ladies and p.gender=Male and p.age>12]
bookCat_error1:Error
1.1.2.8.1.1: display_error_and_forget_info()
1.1.2.8.2[bCat=SeniorCitizen and p.gender=Male and p.age<60]
bookCat_error2:Error
1.1.2.8.2.1: display_error_and_forget_info()
1.1.2.8.3[bCat=SeniorCitizen and p.gender=Female and p.age<58]
bookCat_error3:Error
1.1.2.8.3.1: display_error_and_forget_info()
1.1.2.8.4[bCat=Divyaang and p.disabilityType was not provided]
bookCat_error4:Error
1.1.2.8.4.1: display_error_and_forget_info()
1.1.2.8.5[DiffInDays(dateOfBooking, dateOfReservation) != 1]
bookCat_error5:Error
1.1.2.8.5.1: display_error_and_forget_info()

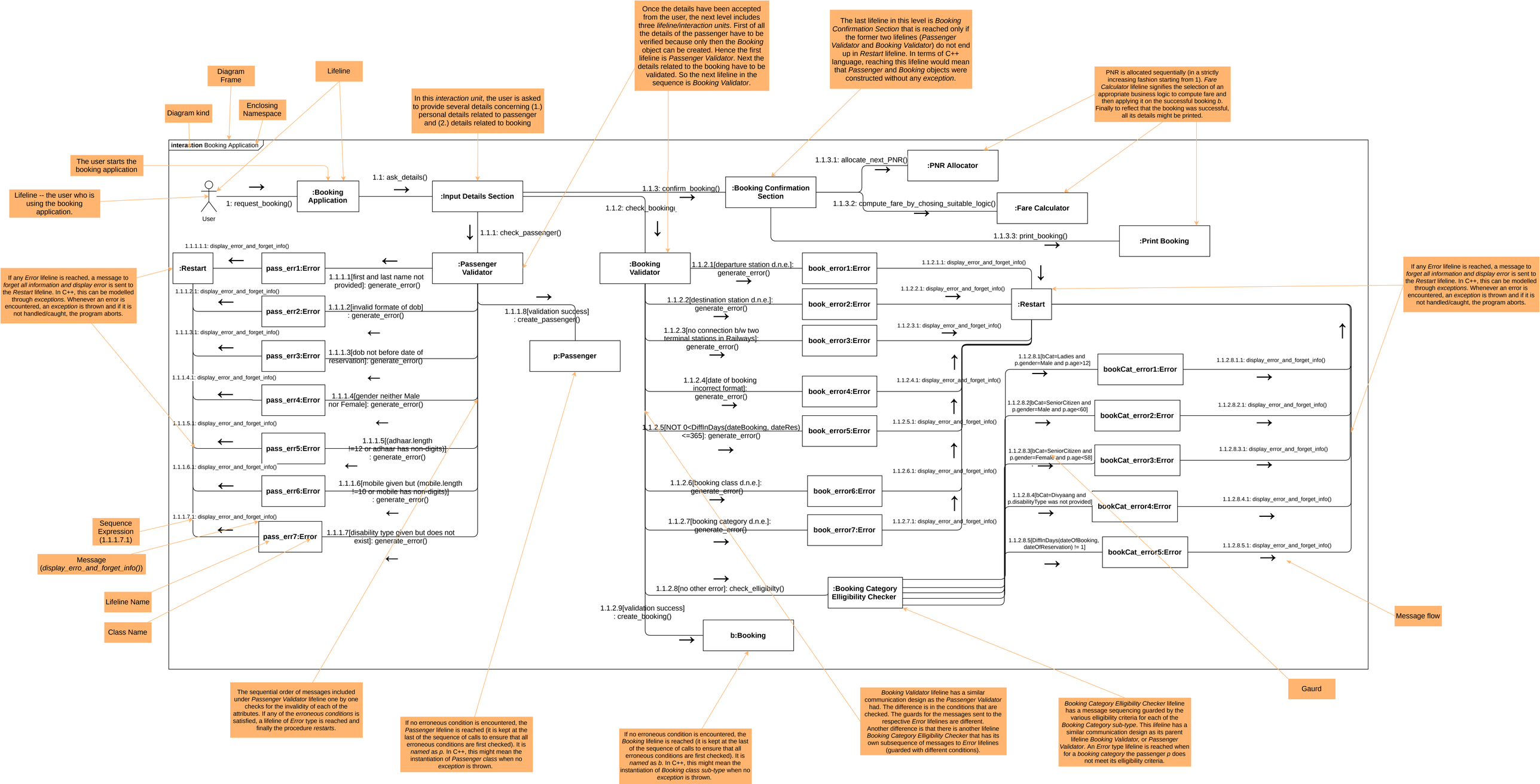


Diagram Frame
Lifeline
Diagram kind
Enclosing Namespace
Once the details have been accepted from the user, the next level includes three *lifeline/interaction units*. First of all the details of the passenger have to be verified because only then the *Booking* object can be created. Hence the first lifeline is *Passenger Validator*. Next the details related to the booking have to be validated. So the next lifeline in the sequence is *Booking Validator*.
The last lifeline in this level is *Booking Confirmation Section* that is reached only if the former two lifelines (*Passenger Validator* and *Booking Validator*) do not end up in *Restart* lifeline. In terms of C++ language, reaching this lifeline would mean that *Passenger* and *Booking* objects were constructed without any *exception*.
PNR is allocated sequentially (in a strictly increasing fashion starting from 1). *Fare Calculator* lifeline signifies the selection of an appropriate business logic to compute fare and then applying it on the successful booking *b*. Finally to reflect that the booking was successful, all its details might be printed.
In this *interaction unit*, the user is asked to provide several details concerning (1.) personal details related to passenger and (2.) details related to booking
interaction Booking Application
The user starts the booking application
Lifeline -- the user who is using the booking application.
User
1: request_booking()
:Booking Application
1.1: ask_details()
:Input Details Section
1.1.1: check_passenger()
1.1.2: check_booking()
1.1.3: confirm_booking()
:Booking Confirmation Section
1.1.3.1: allocate_next_PNR()
:PNR Allocator
1.1.3.2: compute_fare_by_chosing_suitable_logic()
:Fare Calculator
1.1.3.3: print_booking()
:Print Booking
:Passenger Validator
:Booking Validator
1.1.1.1.1: display_error_and_forget_info()
:Restart
pass_err1:Error
1.1.1.1[first and last name not provided]: generate_error()
1.1.1.2.1: display_error_and_forget_info()
pass_err2:Error
1.1.1.2[invalid formate of dob] : generate_error()
1.1.1.3.1: display_error_and_forget_info()
pass_err3:Error
1.1.1.3[dob not before date of reservation]: generate_error()
1.1.1.4.1: display_error_and_forget_info()
pass_err4:Error
1.1.1.4[gender neither Male nor Female]: generate_error()
1.1.1.5.1: display_error_and_forget_info()
pass_err5:Error
1.1.1.5[(adhaar.length !=12 or adhaar has non-digits)] : generate_error()
1.1.1.6.1: display_error_and_forget_info()
pass_err6:Error
1.1.1.6[mobile given but (mobile.length !=10 or mobile has non-digits)] : generate_error()
1.1.1.7.1: display_error_and_forget_info()
pass_err7:Error
1.1.1.7[disability type given but does not exist]: generate_error()
1.1.1.8[validation success] : create_passenger()
p:Passenger
1.1.2.1[departure station d.n.e.]: generate_error()
book_error1:Error
1.1.2.1.1: display_error_and_forget_info()
1.1.2.2[destination station d.n.e.]: generate_error()
book_error2:Error
1.1.2.2.1: display_error_and_forget_info()
1.1.2.3[no connection b/w two terminal stations in Railways]: generate_error()
book_error3:Error
1.1.2.3.1: display_error_and_forget_info()
1.1.2.4[date of booking incorrect format]: generate_error()
book_error4:Error
1.1.2.4.1: display_error_and_forget_info()
1.1.2.5[NOT 0<DiffInDays(dateBooking, dateRes) <=365]: generate_error()
book_error5:Error
1.1.2.5.1: display_error_and_forget_info()
1.1.2.6[booking class d.n.e.]: generate_error()
book_error6:Error
1.1.2.6.1: display_error_and_forget_info()
1.1.2.7[booking category d.n.e.]: generate_error()
book_error7:Error
1.1.2.7.1: display_error_and_forget_info()
1.1.2.8[no other error]: check_elligibilty()
:Booking Category Elligibility Checker
1.1.2.8.1[bCat=Ladies and p.gender=Male and p.age>12]
bookCat_error1:Error
1.1.2.8.1.1: display_error_and_forget_info()
1.1.2.8.2[bCat=SeniorCitizen and p.gender=Male and p.age<60]
bookCat_error2:Error
1.1.2.8.2.1: display_error_and_forget_info()
1.1.2.8.3[bCat=SeniorCitizen and p.gender=Female and p.age<58]
bookCat_error3:Error
1.1.2.8.3.1: display_error_and_forget_info()
1.1.2.8.4[bCat=Divyaang and p.disabilityType was not provided]
bookCat_error4:Error
1.1.2.8.4.1: display_error_and_forget_info()
1.1.2.8.5[DiffInDays(dateOfBooking, dateOfReservation) != 1]
bookCat_error5:Error
1.1.2.8.5.1: display_error_and_forget_info()
1.1.2.9[validation success] : create_booking()
b:Booking
If any *Error* lifeline is reached, a message to *forget all information and display error* is sent to the *Restart* lifeline. In C++, this can be modelled through *exceptions*. Whenever an error is encountered, an *exception* is thrown and if it is not handled/caught, the program aborts.
If any *Error* lifeline is reached, a message to *forget all information and display error* is sent to the *Restart* lifeline. In C++, this can be modelled through *exceptions*. Whenever an error is encountered, an *exception* is thrown and if it is not handled/caught, the program aborts.
Sequence Expression (1.1.1.7.1)
Message (*display_erro_and_forget_info()*)
Lifeline Name
Class Name
Message flow
Gaurd
The sequential order of messages included under *Passenger Validator* lifeline one by one checks for the invalidity of each of the attributes. If any of the *erroneous conditions* is satisfied, a lifeline of *Error* type is reached and finally the procedure *restarts*.
If no erroneous condition is encountered, the *Passenger* lifeline is reached (it is kept at the last of the sequence of calls to ensure that all erroneous conditions are first checked). It is *named as p*. In C++, this might mean the instantiation of *Passenger class* when no *exception* is thrown.
If no erroneous condition is encountered, the *Booking* lifeline is reached (it is kept at the last of the sequence of calls to ensure that all erroneous conditions are first checked). It is *named as b*. In C++, this might mean the instantiation of *Booking class sub-type* when no *exception* is thrown.
Booking Validator lifeline has a similar communication design as the *Passenger Validator* had. The difference is in the conditions that are checked. The guards for the messages sent to the respective *Error* lifelines are different.
Another difference is that there is another lifeline *Booking Category Elligibility Checker* that has its own subsequence of messages to *Error* lifelines (guarded with different conditions).
Booking Category Elligibility Checker lifeline has a message sequencing guarded by the various elligibility criteria for each of the *Booking Category sub-type*. This *lifeline* has a similar communication design as its parent lifeline *Booking Validator*, or *Passenger Validator*. An *Error* type lifeline is reached when for a *booking category* the passenger *p* does not meet its elligibility criteria.

Activity Diagram

Preface

- *Activity Diagram* is a *UML behavior diagram* which shows flow of control or object flow with emphasis on the sequence and conditions of the flow.
- Though the *booking application* we are designing objectively stands for one activity, that is *Requesting A Booking* but subjectively it is composed of quite a few smaller sub-activites.

Activity

- *Activity* is a paramterized behaviour represented as a coordinated flow of *actions*.
- The main *activity* in the diagram is actually an *activity partition* that is the *activity group* for actions having some common characteristics. The main activity partition includes two units -- *User* and *Booking Application*.
  In the *User* side of the partition, the actions are performed by the person who is using the application, like starting the program and filling in the details. While on the *Booking Application* side of the partition, everything is governed by the code we have written in the *implementation*.
- Though the *User* side of the partition is simple as far as the complexity and the number of the actions is concerned, on the other side, the actions can be very large in number because of various situations that are needed to be handled by the application. Therefore it is best to club all the compatible actions together into a single *activity* and define it separately. So the constituent *activities* of the *Booking Application* side of the *main activity partition* include -- *Validate Details* and *Confirm Booking*. *Validate Details* is further composed of another activity called *Check Eligibility of Passenger for Booking Category* and *Confirm Booking* constitutes another activity called *Compute Fare*.
- So there are *4 activities* and *1 activity partition (having 2 partitions)* in the activity diagram. *A*ctivity can have an activity parameter that is actually the *output* of the activity. *Validate Details* activity has *Validation Result* as the parameter (boolean type) and *Check Eligibility of Passenger for Booking Category* has *Eligibility Result* as the parameter (boolean type).

Activity Edge

- *Activity edge* is an abstract class for the directed connections along which tokens or data objects flow between activity nodes. It includes -- *control edges* and *object flow edges*.
- In the diagram, *connectors* are used heavily to reduce the clutter due to activity edges.
- *Object flow edges* are used to show flow of objects. For example *Error* object is created whenever an error is encountered and then that object is displayed. This

is an object flow of *error*. Similarly, in *Confirm Booking* activity, *Passenger* and *Booking* objects are created to show the flow of objects once the details are validated.

- *Guards* are used very often, especially for the edges connected to the *decision nodes* to represent *conditional flow of actions*.

Controls

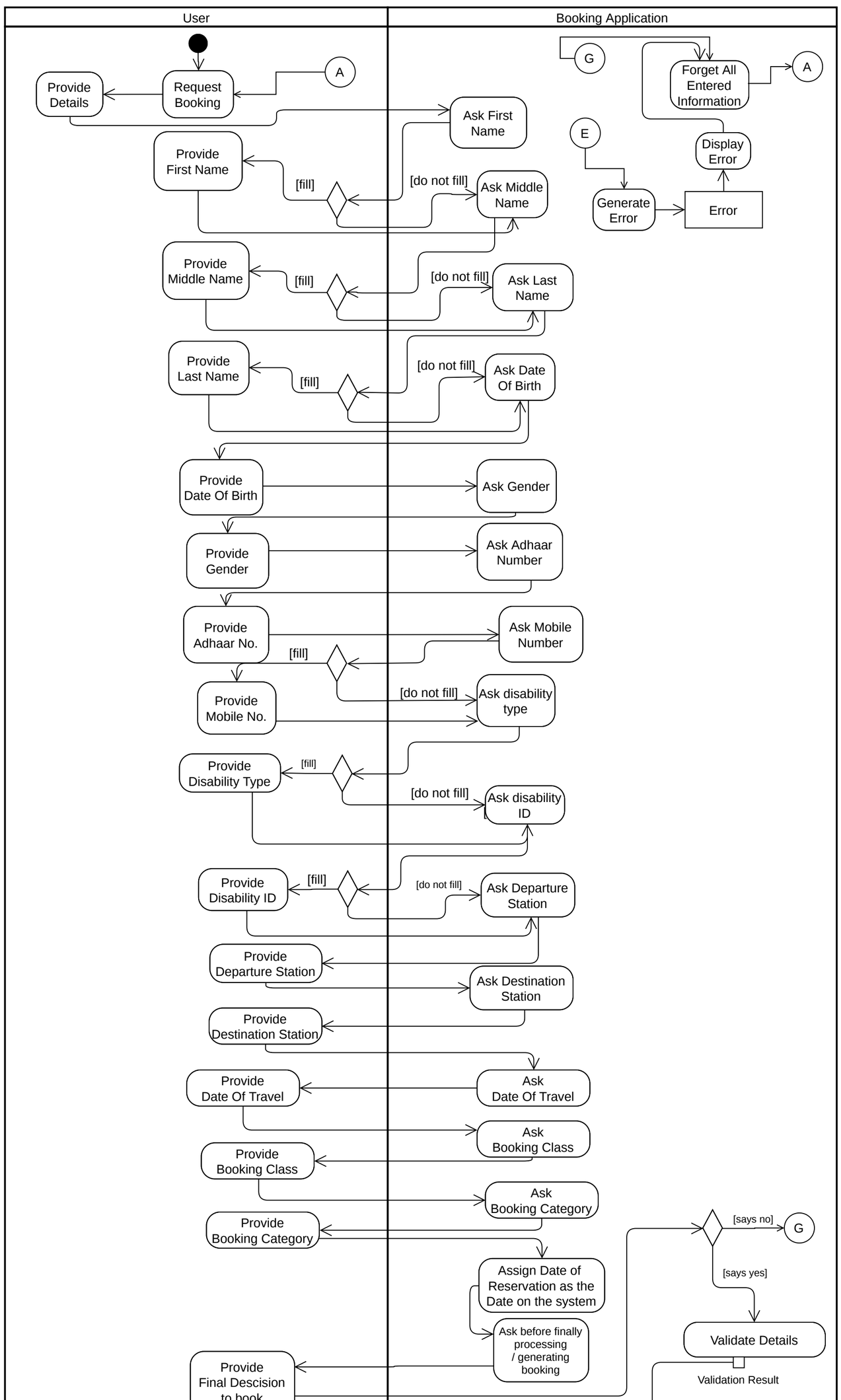
- *Control node* is an activity node used to coordinate the flows between other nodes. Several types of *control nodes* are used in the diagram like -- *initial node, activity final node* and *decision node*.
- Initial node is used in every activity/activity partition. Each one has a unique *initial node*.
- Activity final node is also used in every activity/activity partition. The final states in the activities *Check Eligibility of Passenger for Booking Category* and *Validate Details* are associated with the an output of *boolean* type.
- *Decision nodes* are very frequently used. In fact almost all of the *Validate Details* and *Check Eligibility of Passenger for Booking Category* activities constitute nesting of decision nodes to check for the various erroneous scenarios. Besides, decision nodes are also used in the activity partition to take input for the *optional* input details like *middle name, mobile number* etc.

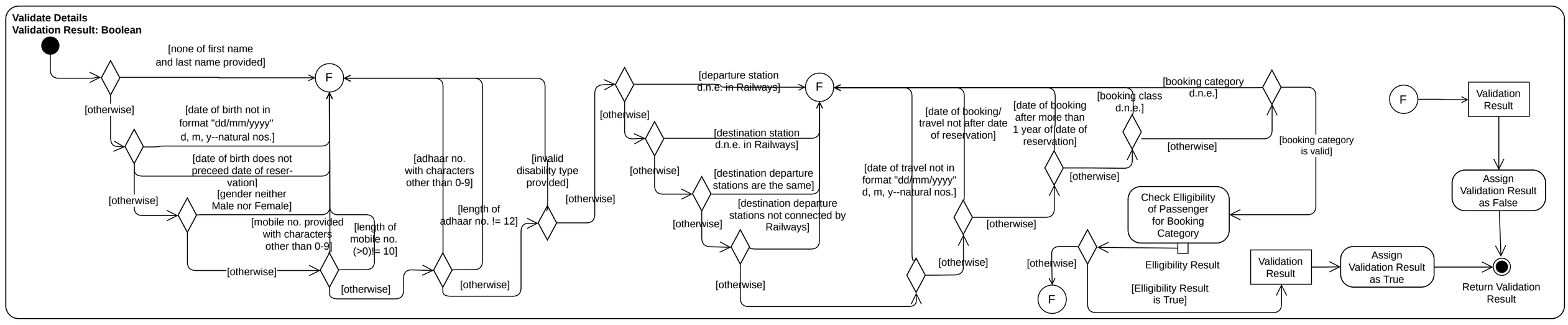
Objects

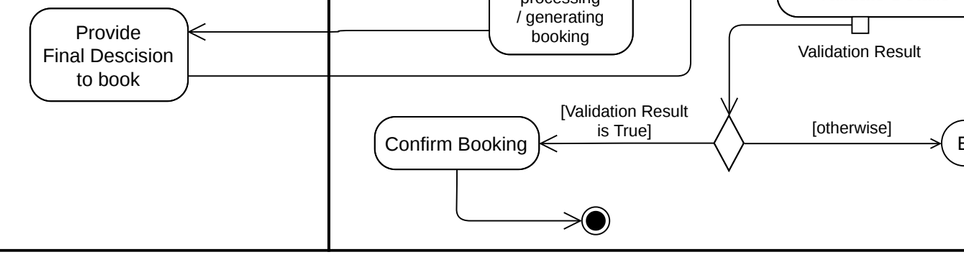
- There are certain *object nodes* used in this diagram to represent creation or instantiation of a class. These object nodes include *Passenger*, *Booking* and *Error*.
- Other object node that is used is *pin* that is used to show the outputs of the activities like *Validate Details* and *Check Eligibility of Passenger for Booking Category*.

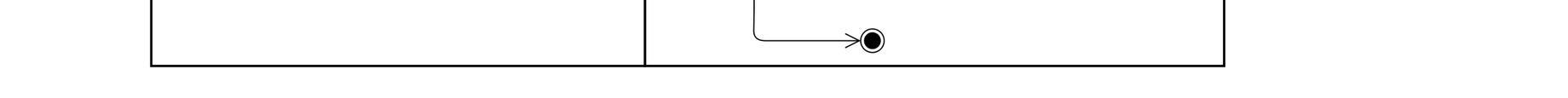
Actions

- *Action* is the named element which represents a single atomic step within activity that is not further decomposed within the activity.
- There are a lot of *actions* in this diagram. Some of them are *Ask Gender*, *Provide First Name, Create Booking* etc.
- Note that elements like *Validate Details* that might look like actions are not actions because they in no way represent a single atomic step. These are, as discussed, *activites*.


User
Booking Application
Request Booking
Provide Details
A
Ask First Name
Provide First Name
[fill]
[do not fill]
Ask Middle Name
Provide Middle Name
[fill]
[do not fill]
Ask Last Name
Provide Last Name
[fill]
[do not fill]
Ask Date Of Birth
Provide Date Of Birth
Ask Gender
Provide Gender
Ask Adhaar Number
Provide Adhaar No.
[fill]
Ask Mobile Number
Provide Mobile No.
[do not fill]
Ask disability type
Provide Disability Type
[fill]
[do not fill]
Ask disability ID
Provide Disability ID
[fill]
[do not fill]
Ask Departure Station
Provide Departure Station
Ask Destination Station
Provide Destination Station
Provide Date Of Travel
Ask Date Of Travel
Provide Booking Class
Ask Booking Class
Provide Booking Category
Ask Booking Category
Assign Date of Reservation as the Date on the system
Ask before finally processing / generating booking
Provide Final Descision to book
[says no]
G
[says yes]
Validate Details
Validation Result
[Validation Result is True]
[otherwise]
E
Confirm Booking
G
Forget All Entered Information
A
E
Display Error
Generate Error
Error


Validate Details
Validation Result: Boolean
[none of first name and last name provided]
[otherwise]
[date of birth not in format "dd/mm/yyyy" d, m, y--natural nos.]
[date of birth does not preceed date of reser-vation]
[otherwise]
[gender neither Male nor Female]
[otherwise]
[mobile no. provided with characters other than 0-9]
[length of mobile no. (>0)!= 10]
[otherwise]
[otherwise]
[adhaar no. with characters other than 0-9]
[length of adhaar no. != 12]
[otherwise]
[invalid disability type provided]
[otherwise]
[departure station d.n.e. in Railways]
[otherwise]
[destination station d.n.e. in Railways]
[otherwise]
[destination departure stations are the same]
[otherwise]
[destination departure stations not connected by Railways]
[otherwise]
[date of travel not in format "dd/mm/yyyy" d, m, y--natural nos.]
[otherwise]
[date of booking/ travel not after date of reservation]
[otherwise]
[date of booking after more than 1 year of date of reservation]
[otherwise]
[booking class d.n.e.]
[otherwise]
[booking category d.n.e.]
[booking category is valid]
Check Elligibility of Passenger for Booking Category
Elligibility Result
[Elligibility Result is True]
[otherwise]
F
Validation Result
Assign Validation Result as False
Validation Result
Assign Validation Result as True
Return Validation Result
F


Compute Fare
Choose Business/Fare Computation Logic based on the type/class of booking
Run the logic and store the computed fare in the booking's attributes


Confirm Booking
Create Passenger
Passenger
Allocate next available PNR for the booking
Create Booking
Booking
Compute Fare
Print All Details with Success Message

Check Elligibility Of Passenger for Booking Category
Elligibility Result: Boolean
[Booking Category is General]
[otherwise]
[Booking Category is Ladies]
[otherwise]
[Gender of Passenger : Female]
[otherwise]
[otherwise]
[Age at the time of booking is not more than 12]
[otherwise]
[otherwise]
[Booking Category is SeniorCitizen]
[otherwise]
[Gender of Passenger : Female]
[otherwise]
[Age at the time of booking is not less than 58]
[otherwise]
[Age at the time of booking is not less than 60]
[otherwise]
[otherwise]
[Booking Category is Divyaang]
[otherwise]
[Passenger did not fill disability type]
[otherwise]
[otherwise]
[otherwise]
[date of travel after 1 day of date of reservation]
R
T
R
T
Elligibility Result
Assign Elligibility Result as False
Elligibility Result
Assign Elligibility Result as True
Return Elligibility Result

Activity Partitions

Connector: When the user fills in all details but does not proceed to generate the booking

User

Booking Application

Connector: If all the details could not be validated

Restart from *Request Booking* action

Initial Node

Activity Edge

Object Flow Edge

Action

Object Node

Gaurd

Object Flow of *Error* between *Generate Error* and *Display Error*

Decision Node

Provide details related to *Passenger*

Provide details related to *Booking*

These details are *optional* and hence *decision* flows are used to decide if these values will be filled or not. Note that though *first name* and *last name* are both *optional*, for a valid *Passenger* atleast one of them should be provided

Pin

Validation Result is output from *Validate Details* activity

Connectors

Activity Final Node

Validate Details and *Confirm Booking* are *activities* whose *coordinated action-flow diagram* is defined separately

Activity

Activity Name

Activity Parameter

Parameter Name

Parameter Type

Initial Node

One by one check the validity of all the details

Decision Nodes: Checking Validity is nothing but a collection of decision statements. For every property we need to *decide* if it is compatible or not

Connector: An invalid attribute/detail was encountered

Object Node -- Return Value

Activity Final Node -- return *validation result*

Elligibility Result is output from *Check Elligibility of Passenger for Booking Category* activity

Check Elligibility of Passenger for Booking Category is an *activity* whose *coordinated action-flow diagram* is described separately

Object Nodes

Activity Final Node

Implement a similar *coordination of decision nodes* as in *Validate Details* activity by sub-suming the elligibility criteria for every *booking category*

Connector R: Acts like a *reject configuration*. It is reached when the *passenger* is not *elligible* for the *booking category* that is selected

Connector T: Acts like an *accept configuration*

Object Nodes: Return Values

Activity Final Node: return Eligibility Result

Activity; NOT an *action*

State Machine Diagram

Preface

- *State Machine Diagram* is a behaviour diagram which shows discrete behaviour of a part of designed system through finite state transitions.
- The state machine that is designed here is a *behavioral state machine* that specifies discrete behaviour through finite state transitions.

Behavioral States

- It is a subclass of vertex (vertex is named element which is an abstraction of a node in a state machine graph) that models a situation during which some invariant condition holds. The kinds of states used in the diagram include -- *simple states* and *submachine states*.
- *Simple state* does not have any substates. In the diagram these include *enterPassengerInformation, allocatePNR, computeFare* etc. For each of these states a name is provided and also the internal *do* activities are mentioned.
- The diagram includes two *submachine states* -- *ValidatePassenger* and *ValidateBooking*. The substates and internal transitions in these submachine states are defined separately outside of the main state machine *BookingApplication*. *Submachine state* is a decomposition mechanism that allows factoring of common behaviors and their reuse. Here they are used simply to *modularize* the various processes that are happening in order to *create and confirm a booking*.

Pseudo States

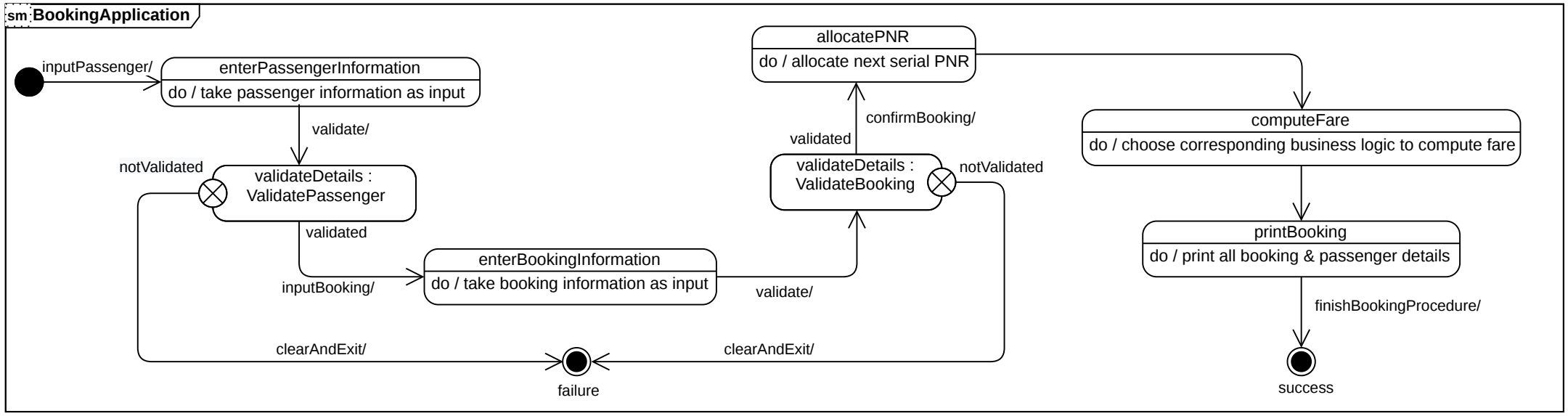
- It is again a subclass of vertex that is used to connect multiple transitions into more complex transition paths. Though there are many kinds of pseudostates, most of them are not relevant here. The ones used are -- *initial pseudostate, terminal pseudostate, exit point pseudostate, choice pseudostate* and *final state*.
- The state machine *BookingApplication* has one *initial pseudostate* and each of the two submachine states have there own *initial pseudostates*.
- Terminal pseudostates are used in the definitions of the submachine states. These imply termination of execution of the state. Terminal pseudostate in the submachine states are reached when an inconsistency in some input detail is found. For example if some booking detail is invalid (like passenger is ineligible for the booking category), the terminal pseudostate is directly reached in order to exit (from the *exit point pseudostate*). Similarly if some passenger detail is invalid (like gender is niether male nore female), the terminal pseudostate is directly reached in order to exit the state.
- The *declaration* of the submachine states in the state machine *BookingApplication* comprises of an *exit point pseudostate*. When the

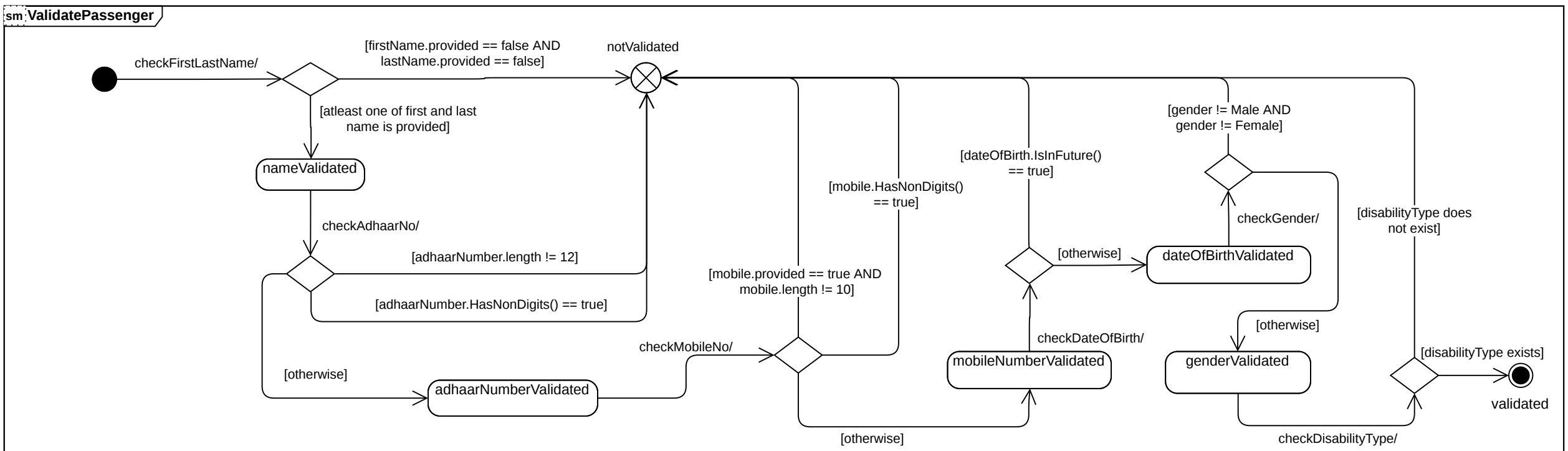
submachine states exit their states by reaching the terminal pseudostate, not the final state, then in the state machine it is shown as an outgoing edge coming out of the *exit point* pseudostate. Otherwise, it is simply shown as coming out of the rectangular box's boundary. This helps in differentiating the two possibilities that might have caused the submachine state to exit. Coming out of the exit point of the submachine states means that the details could not be validated.

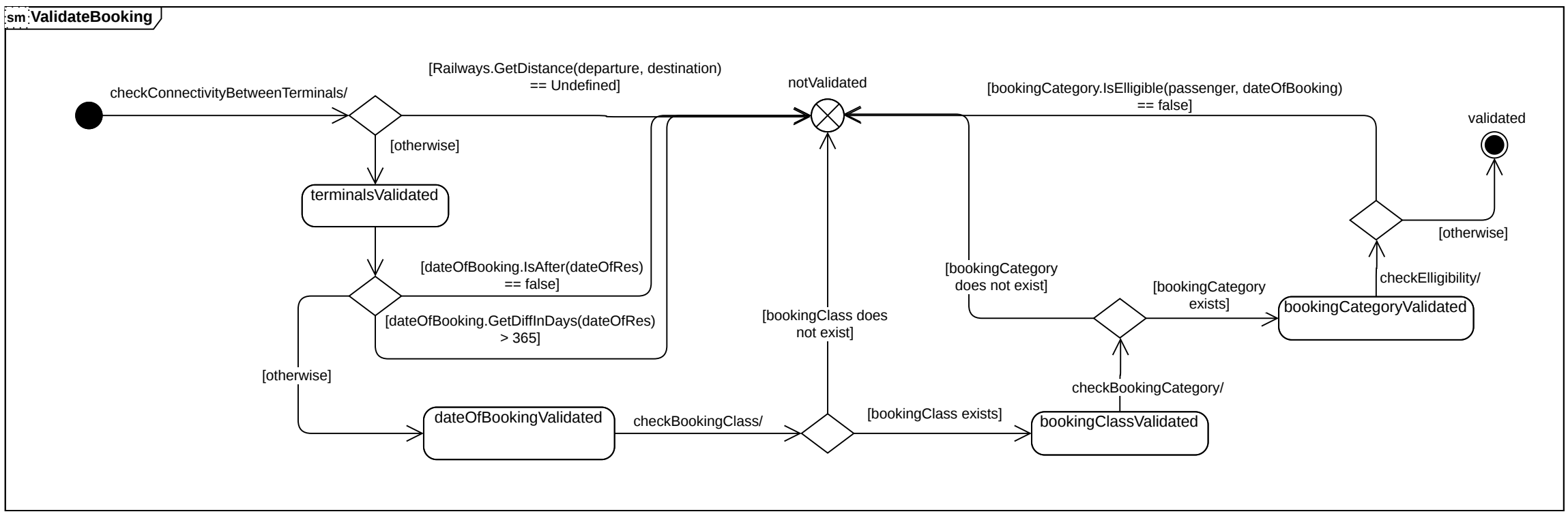
- *Choice pseudostates* are heavily used in the submachine states to check for various erroneous conditions. One by one each input parameter/detail is checked and choice pseudostates are used in deciding the transition based on the result of the *check* performed. This choice is governed by conditions given in *guards*.
- *Two final states* are present in the state machine; one to represent successful booking and the other to represent failed booking. The submachine states also have their own final states. Coming out of the submachine states after reaching their final state means that all the details were validated.

Behavioral Transitions

- A transition is a directed relationship between a source vertex and a target vertex. Notations for behavioral transitions might include *triggers*, *guards* and *behavior expressions*.
- Here the triggers are self sufficient to describe and identify the transitions uniquely and also to understand their purpose unambiguously. Therefore expressions are not used. Eg -- *checkBookingClass/* , *confirmBooking/* etc.
- For the outgoing transitions of the decision nodes, there are no triggers because the next state is solely based on the condition that is satisfied. Therefore, the *outgoing* transitions of the decision nodes have only *guards*.
- Some transitions might not have any notation. These transitions spontaneously take place from one terminal state to the other without any input trigger or guard condition. For example -- *allocatePNR* to *computeFare* to *printBooking*.


sm BookingApplication
inputPassenger/
enterPassengerInformation
do / take passenger information as input
validate/
notValidated
validateDetails :
ValidatePassenger
validated
inputBooking/
enterBookingInformation
do / take booking information as input
validate/
validateDetails :
ValidateBooking
notValidated
validated
confirmBooking/
allocatePNR
do / allocate next serial PNR
computeFare
do / choose corresponding business logic to compute fare
printBooking
do / print all booking & passenger details
finishBookingProcedure/
success
clearAndExit/
clearAndExit/
failure


sm ValidatePassenger
checkFirstLastName/
[firstName.provided == false AND
lastName.provided == false]
notValidated
[atleast one of first and last
name is provided]
nameValidated
checkAdhaarNo/
[adhaarNumber.length != 12]
[adhaarNumber.HasNonDigits() == true]
[otherwise]
adhaarNumberValidated
[mobile.provided == true AND
mobile.length != 10]
checkMobileNo/
[otherwise]
[mobile.HasNonDigits()
== true]
[dateOfBirth.IsInFuture()
== true]
[otherwise]
dateOfBirthValidated
checkDateOfBirth/
mobileNumberValidated
[gender != Male AND
gender != Female]
checkGender/
[otherwise]
genderValidated
[disabilityType does
not exist]
[disabilityType exists]
validated
checkDisabilityType/


sm ValidateBooking
checkConnectivityBetweenTerminals/
[Railways.GetDistance(departure, destination)
== Undefined]
notValidated
[otherwise]
terminalsValidated
[dateOfBooking.IsAfter(dateOfRes)
== false]
[dateOfBooking.GetDiffInDays(dateOfRes)
> 365]
[otherwise]
dateOfBookingValidated
checkBookingClass/
[bookingClass does
not exist]
[bookingClass exists]
bookingClassValidated
checkBookingCategory/
[bookingCategory
does not exist]
[bookingCategory.IsElligible(passenger, dateOfBooking)
== false]
[bookingCategory
exists]
bookingCategoryValidated
checkElligibility/
[otherwise]
validated

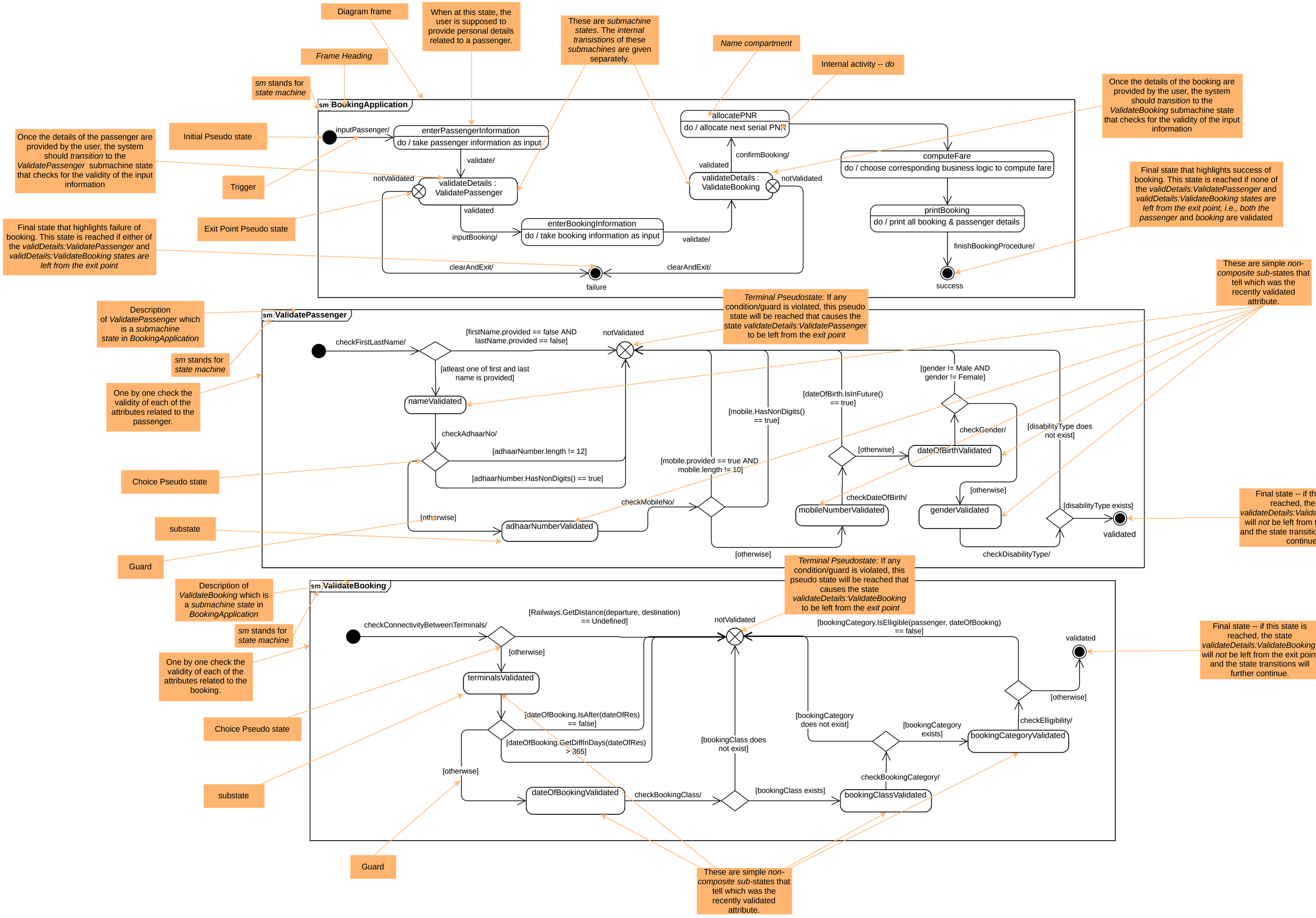


Diagram frame
When at this state, the user is supposed to provide personal details related to a passenger.
These are submachine states. The internal transistions of these submachines are given separately.
Name compartment
Internal activity -- do
Frame Heading
sm stands for state machine
sm :BookingApplication
Once the details of the booking are provided by the user, the system should transition to the ValidateBooking submachine state that checks for the validity of the input information
Once the details of the passenger are provided by the user, the system should transition to the ValidatePassenger submachine state that checks for the validity of the input information
Initial Pseudo state
Trigger
Exit Point Pseudo state
Final state that highlights failure of booking. This state is reached if either of the validDetails:ValidatePassenger and validDetails:ValidateBooking states are left from the exit point
Final state that highlights success of booking. This state is reached if none of the validDetails:ValidatePassenger and validDetails:ValidateBooking states are left from the exit point, i.e., both the passenger and booking are validated
inputPassenger/
enterPassengerInformation
do / take passenger information as input
validate/
notValidated
validateDetails : ValidatePassenger
validated
inputBooking/
enterBookingInformation
do / take booking information as input
validate/
validateDetails : ValidateBooking
notValidated
validated
confirmBooking/
allocatePNR
do / allocate next serial PNR
computeFare
do / choose corresponding business logic to compute fare
printBooking
do / print all booking & passenger details
finishBookingProcedure/
clearAndExit/
clearAndExit/
failure
success
These are simple non-composite sub-states that tell which was the recently validated attribute.
Description of ValidatePassenger which is a submachine state in BookingApplication
sm stands for state machine
One by one check the validity of each of the attributes related to the passenger.
Choice Pseudo state
substate
Guard
Terminal Pseudostate: If any condition/guard is violated, this pseudo state will be reached that causes the state validateDetails:ValidatePassenger to be left from the exit point
sm :ValidatePassenger
checkFirstLastName/
[firstName.provided == false AND lastName.provided == false]
notValidated
[atleast one of first and last name is provided]
nameValidated
checkAdhaarNo/
[adhaarNumber.length != 12]
[adhaarNumber.HasNonDigits() == true]
[otherwise]
adhaarNumberValidated
checkMobileNo/
[mobile.HasNonDigits() == true]
[mobile.provided == true AND mobile.length != 10]
[otherwise]
mobileNumberValidated
checkDateOfBirth/
[dateOfBirth.IsInFuture() == true]
[otherwise]
dateOfBirthValidated
checkGender/
[gender != Male AND gender != Female]
[otherwise]
genderValidated
checkDisabilityType/
[disabilityType does not exist]
[disabilityType exists]
validated
Final state -- if this state is reached, the state validateDetails:ValidatePassenger will not be left from the exit point and the state transitions will further continue.
Description of ValidateBooking which is a submachine state in BookingApplication
sm stands for state machine
One by one check the validity of each of the attributes related to the booking.
Choice Pseudo state
substate
Guard
Terminal Pseudostate: If any condition/guard is violated, this pseudo state will be reached that causes the state validateDetails:ValidateBooking to be left from the exit point
sm :ValidateBooking
checkConnectivityBetweenTerminals/
[Railways.GetDistance(departure, destination) == Undefined]
notValidated
[otherwise]
terminalsValidated
[dateOfBooking.IsAfter(dateOfRes) == false]
[dateOfBooking.GetDiffInDays(dateOfRes) > 365]
[otherwise]
dateOfBookingValidated
checkBookingClass/
[bookingClass does not exist]
[bookingClass exists]
bookingClassValidated
checkBookingCategory/
[bookingCategory does not exist]
[bookingCategory exists]
bookingCategoryValidated
checkEligibility/
[bookingCategory.IsElligible(passenger, dateOfBooking) == false]
[otherwise]
validated
Final state -- if this state is reached, the state validateDetails:ValidateBooking will not be left from the exit point and the state transitions will further continue.
These are simple non-composite sub-states that tell which was the recently validated attribute.