Design a Movie Ticket Booking System

We'll cover the following

An online movie ticket booking system facilitates the purchasing of movie tickets to its customers. E-ticketing systems allow customers to browse through movies currently playing and book seats, anywhere and anytime.

Requirements and Goals of the System

Our ticket booking service should meet the following requirements:

1. It should be able to list the cities where affiliate cinemas are located.
2. Each cinema can have multiple halls and each hall can run one movie show at a time.
3. Each Movie will have multiple shows.
4. Customers should be able to search movies by their title, language, genre, release date, and city name.
5. Once the customer selects a movie, the service should display the cinemas running that movie and its available shows.
6. The customer should be able to select a show at a particular cinema and book their tickets.
7. The service should show the customer the seating arrangement of the cinema hall. The customer should be able to select multiple seats according to their preference.
8. The customer should be able to distinguish between available seats and booked ones.
9. The system should send notifications whenever there is a new movie, as well as when a booking is made or canceled.
10. Customers of our system should be able to pay with credit cards or cash.
11. The system should ensure that no two customers can reserve the same seat.
12. Customers should be able to add a discount coupon to their payment.

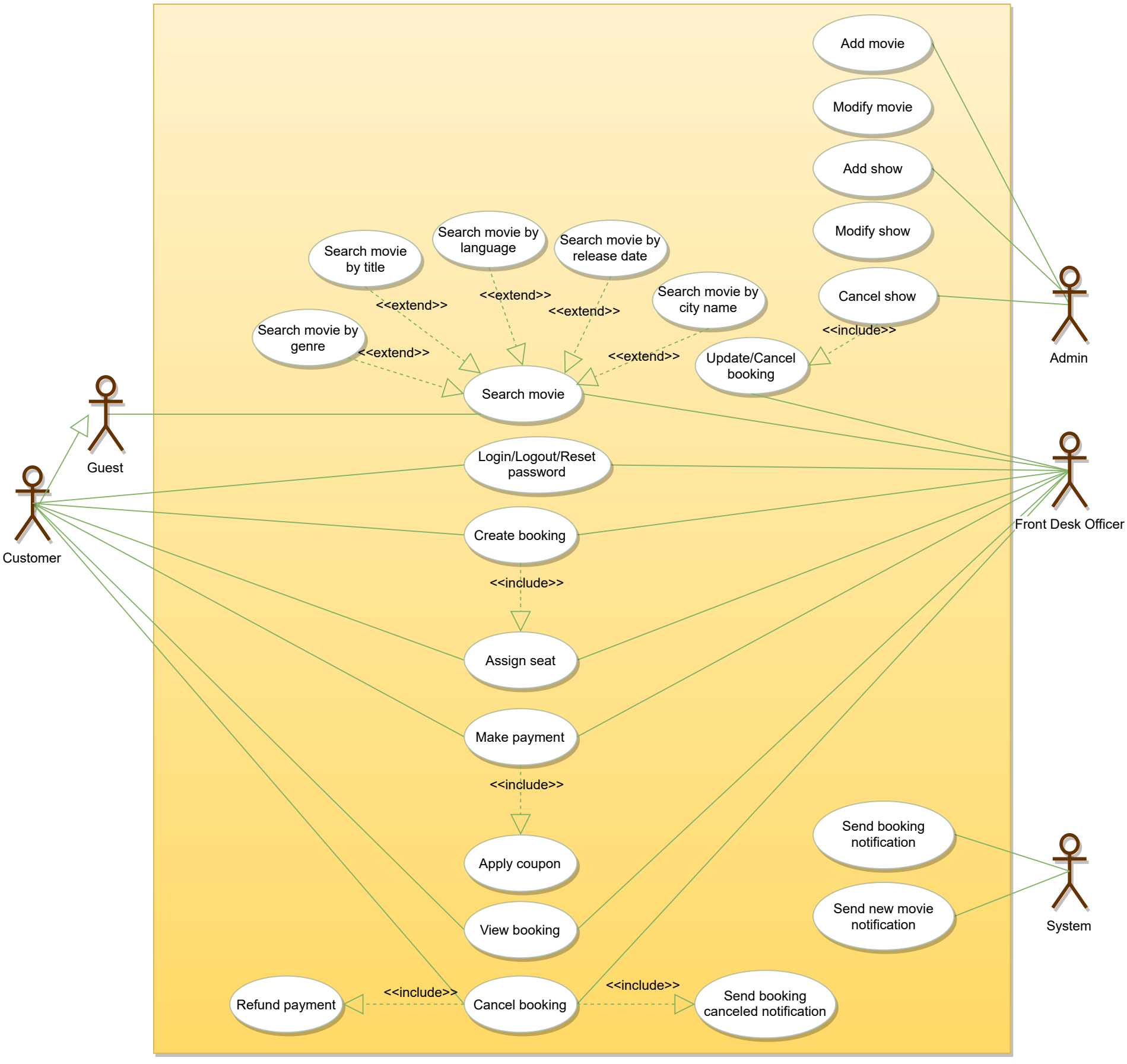
Use case diagram

We have five main Actors in our system:

- **Admin:** Responsible for adding new movies and their shows, canceling any movie or show, blocking/unblocking customers, etc.
- **FrontDeskOfficer:** Can book/cancel tickets.
- **Customer:** Can view movie schedules, book, and cancel tickets.
- **Guest:** All guests can search movies but to book seats they have to become a registered member.
- **System:** Mainly responsible for sending notifications for new movies, bookings, cancellations, etc.

Here are the top use cases of the Movie Ticket Booking System:

- **Search movies:** To search movies by title, genre, language, release date, and city name.
- **Create/Modify/View booking:** To book a movie show ticket, cancel it or view details about the show.
- **Make payment for booking:** To pay for the booking.
- **Add a coupon to the payment:** To add a discount coupon to the payment.
- **Assign Seat:** Customers will be shown a seat map to let them select seats for their booking.
- **Refund payment:** Upon cancellation, customers will be refunded the payment amount as long as the cancellation occurs within the allowed time frame.



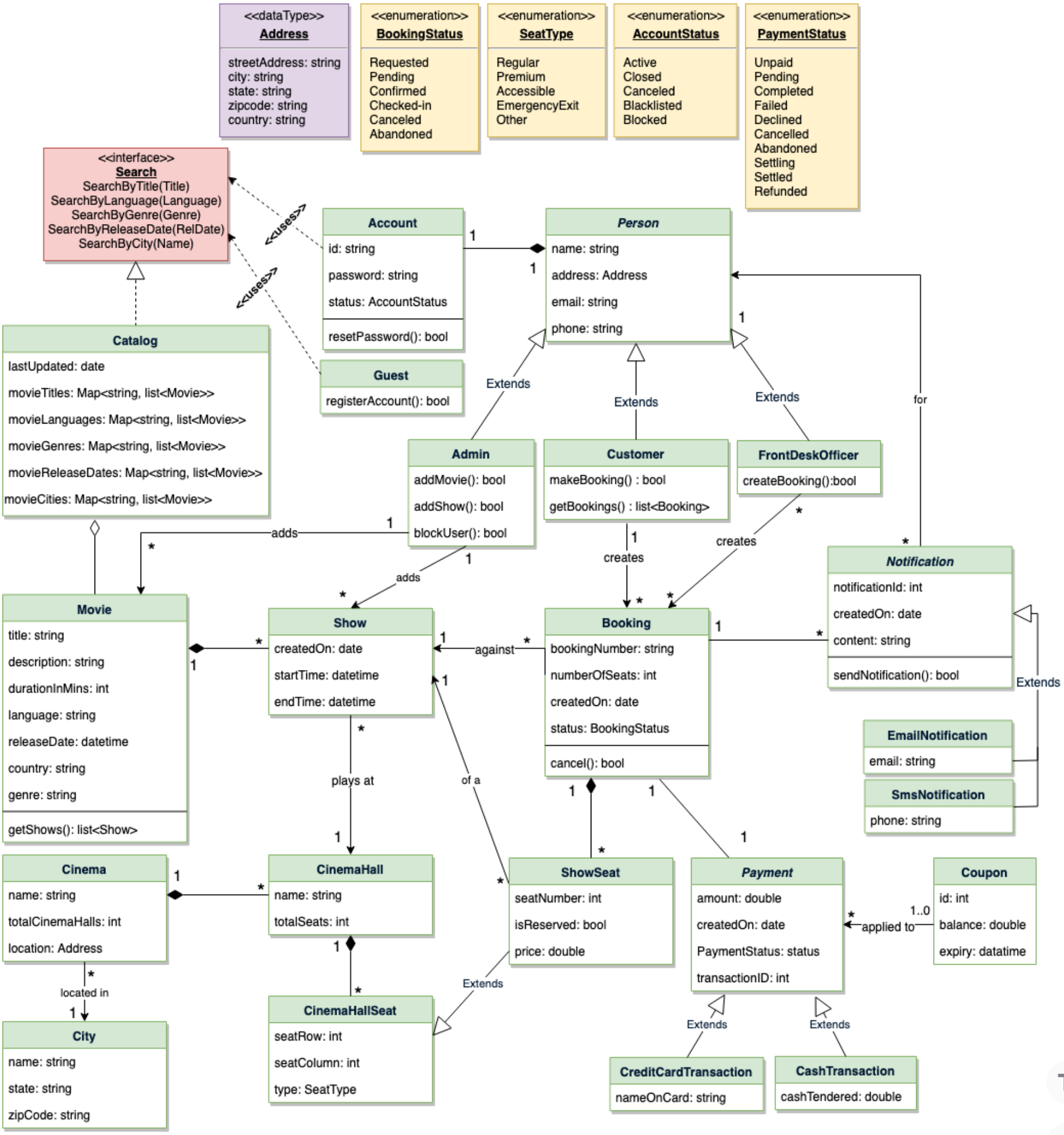
Use case diagram

Class diagram

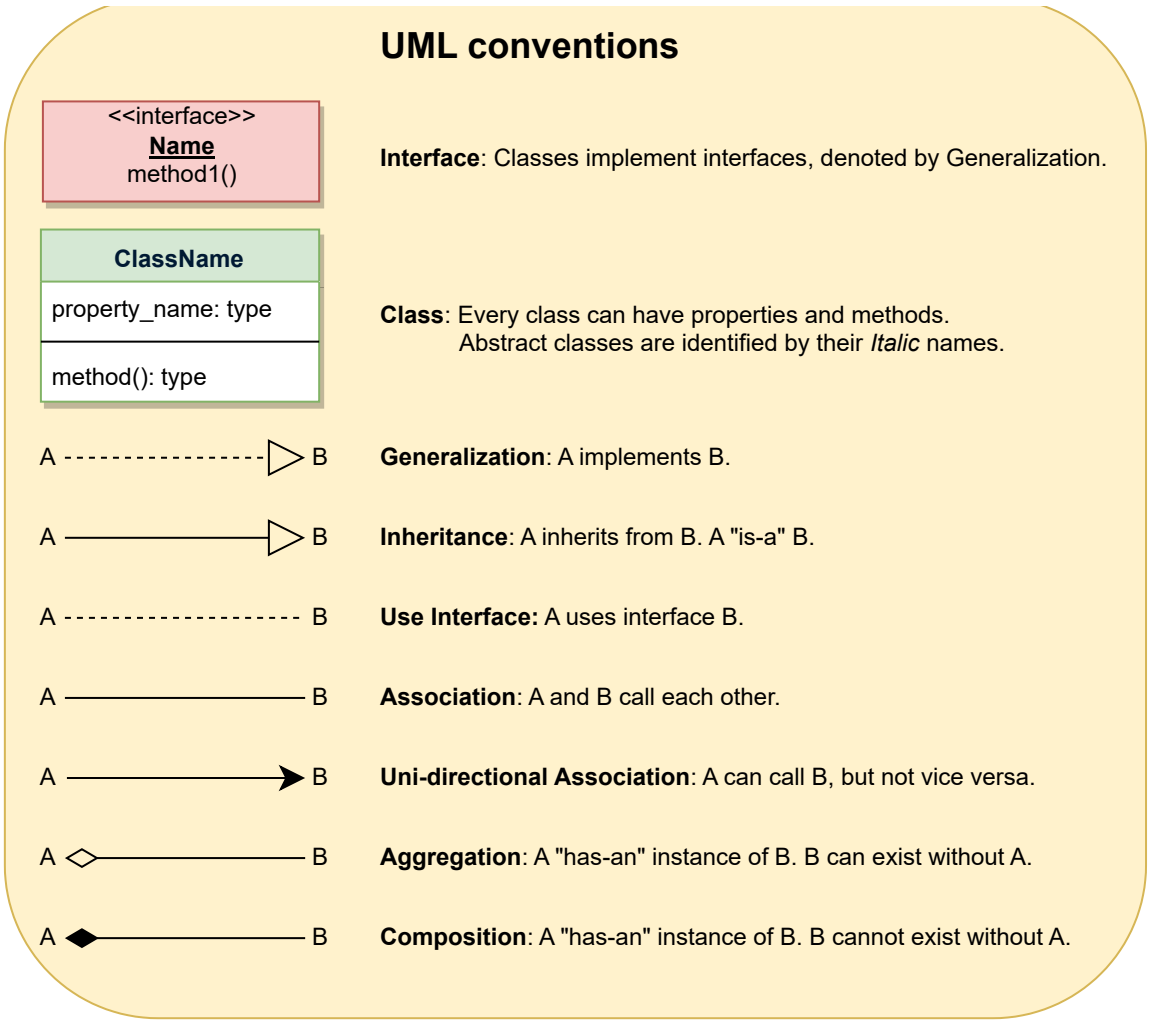
Here are the main classes of the Movie Ticket Booking System:

- **Account:** Admin will be able to add/remove movies and shows, as well as block/unblock accounts. Customers can search for movies and make bookings for shows. FrontDeskOffice can book tickets for movie shows.

- **Guest:** Guests can search and view movies descriptions. To make a booking for a show they have to become a registered member.
- **Cinema:** The main part of the organization for which this software has been designed. It has attributes like ‘name’ to distinguish it from other cinemas.
- **CinemaHall:** Each cinema will have multiple halls containing multiple seats.
- **City:** Each city can have multiple cinemas.
- **Movie:** The main entity of the system. Movies have attributes like title, description, language, genre, release date, city name, etc.
- **Show:** Each movie can have many shows; each show will be played in a cinema hall.
- **CinemaHallSeat:** Each cinema hall will have many seats.
- **ShowSeat:** Each ShowSeat will correspond to a movie Show and a CinemaHallSeat. Customers will make a booking against a ShowSeat.
- **Booking:** A booking is against a movie show and has attributes like a unique booking number, number of seats, and status.
- **Payment:** Responsible for collecting payments from customers.
- **Notification:** Will take care of sending notifications to customers.

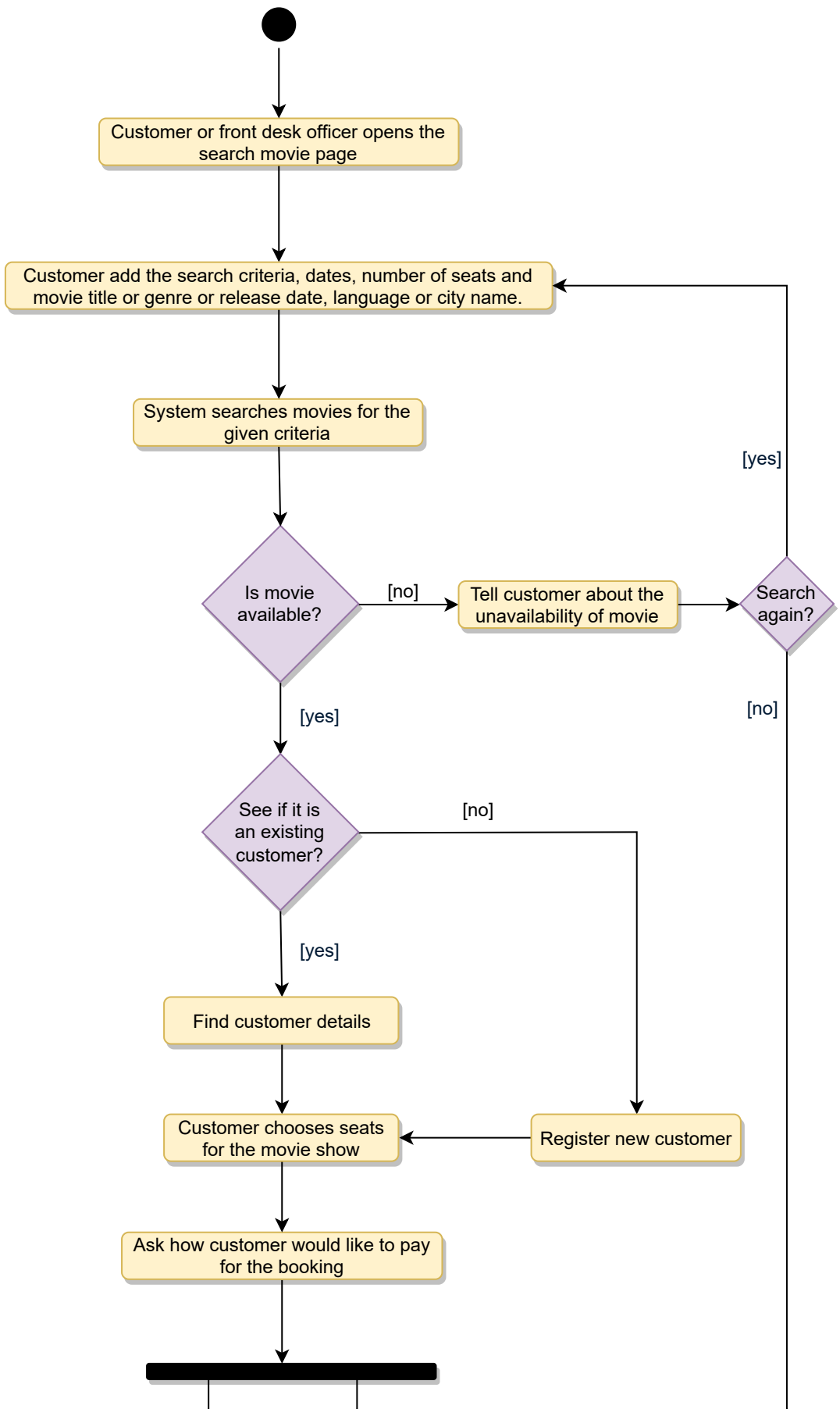


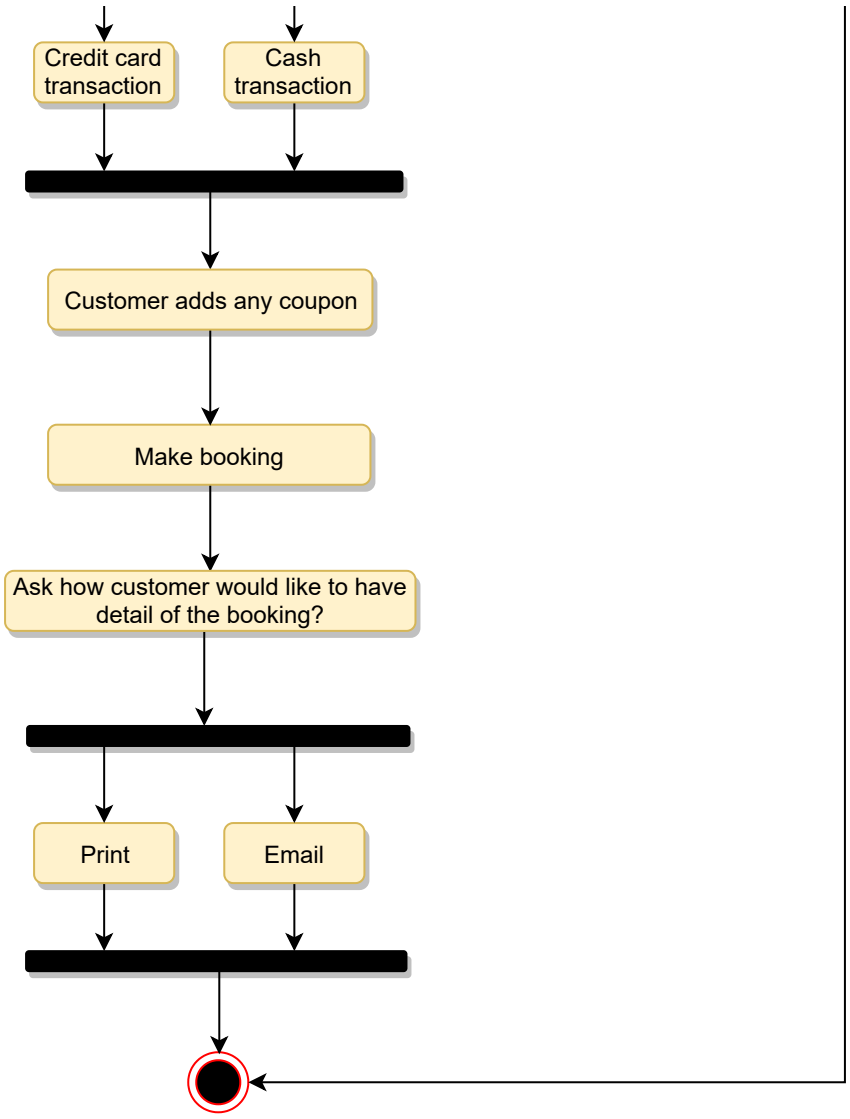
Class diagram



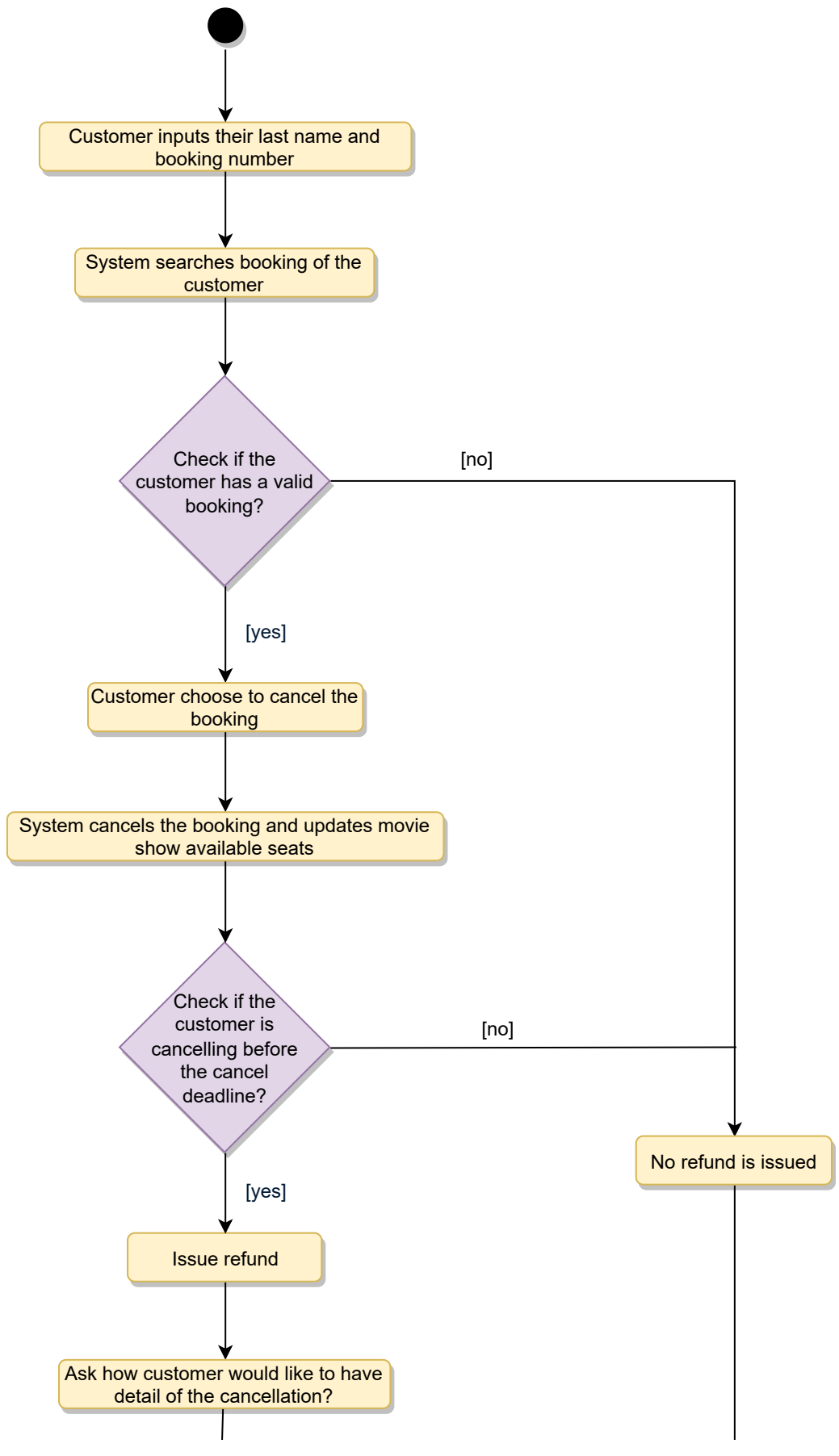
Activity Diagram

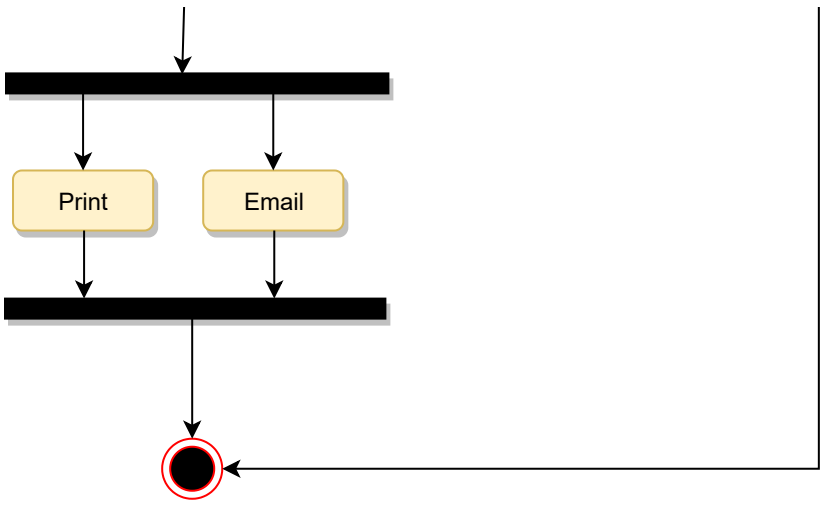
- **Make a booking:** Any customer can perform this activity. Here are the steps to book a ticket for a show:





- **Cancel a booking:** Customer can cancel their bookings. Here are the steps to cancel a booking:





Code

Here are the high-level definitions for the classes described above.

Enums, data types, and constants: Here are the required enums, data types, and constants:

Java Python

```
class BookingStatus(Enum):
  REQUESTED, PENDING, CONFIRMED, CHECKED_IN, CANCELED, ABANDONED = 1, 2, 3, 4, 5, 6


class SeatType(Enum):
  REGULAR, PREMIUM, ACCESSIBLE, SHIPPED, EMERGENCY_EXIT, OTHER = 1, 2, 3, 4, 5, 6


class AccountStatus(Enum):
  ACTIVE, BLOCKED, BANNED, COMPROMISED, ARCHIVED, UNKNOWN = 1, 2, 3, 4, 5, 6


class PaymentStatus(Enum):
  UNPAID, PENDING, COMPLETED, FILLED, DECLINED, CANCELLED, ABANDONED, SETTLING, SETTLED, REFUNDED = 1, 2, 3, 4, 5, (


class Address:
  def __init__(self, street, city, state, zip_code, country):
    self.__street_address = street
    self.__city = city
    self.__state = state
    self.__zip_code = zip_code
    self.__country = country

```

Account, Customer, Admin, FrontDeskOfficer, and Guest: These classes represent the different people that interact with our system:

Java Python

```
# For simplicity, we are not defining getter and setter functions. The reader can
# assume that all class attributes are private and accessed through their respective
# public getter methods and modified only through their public methods function.


class Account:
  def __init__(self, id, password, status=AccountStatus.Active):
    self.__id = id
    self.__password = password
    self.__status = status

  def reset_password(self):
    None


# from abc import ABC, abstractmethod
class Person(ABC):
  def __init__(self, name, address, email, phone, account):
    self.__name = name
```

```
      self.__name = name
      self.__address = address
      self.__email = email
      self.__phone = phone
      self.__account = account


class Customer(Person):
  def make_booking(self, booking):
    None

  def get_bookings(self):
```

Show and Movie: A movie will have many shows:

Java | Python

```
class Show:
  def __init__(self, id, played_at, movie, start_time, end_time):
    self.__show_id = id
    self.__created_on = datetime.date.today()
    self.__start_time = start_time
    self.__end_time = end_time
    self.__played_at = played_at
    self.__movie = movie


class Movie:
  def __init__(self, title, description, duration_in_mins, language, release_date, country, genre, added_by):
    self.__title = title
    self.__description = description
    self.__duration_in_mins = duration_in_mins
    self.__language = language
    self.__release_date = release_date
    self.__country = country
    self.__genre = genre
    self.__movie_added_by = added_by

    self.__shows = []

  def get_shows(self):
    None
```

Booking, ShowSeat, and Payment: Customers will reserve seats with a booking and make a payment:

Java | Python

```
class Booking:
  def __init__(self, booking_number, number_of_seats, status, show, show_seats, payment):
    self.__booking_number = booking_number
    self.__number_of_seats = number_of_seats
    self.__created_on = datetime.date.today()
    self.__status = status
    self.__show = show
    self.__seats = show_seats
    self.__payment = payment

  def make_payment(self, payment):
    None

  def cancel(self):
    None

  def assign_seats(self, seats):
    None


class ShowSeat(CinemaHallSeat):
  def __init__(self, id, is_reserved, price):
    self.__show_seat_id = id
    self.__is_reserved = is_reserved
```

```
    self.__price = price


class Payment:
  def __init__(self, amount, transaction_id, payment_status):
    self.__amount = amount
```

City, Cinema, and CinemaHall: Each city can have many cinemas and each cinema can have many cinema halls:

Java | Python

```
class City:
  def __init__(self, name, state, zip_code):
    self.__name = name
    self.__state = state
    self.__zip_code = zip_code


class Cinema:
  def __init__(self, name, total_cinema_halls, address, halls):
    self.__name = name
    self.__total_cinema_halls = total_cinema_halls
    self.__location = address
    self.__halls = halls


class CinemaHall:
  def __init__(self, name, total_seats, seats, shows):
    self.__name = name
    self.__total_seats = total_seats

    self.__seats = seats
    self.__shows = shows
```

Search interface and Catalog: Catalog will implement Search to facilitate searching of products.

Java | Python

```
from abc import ABC, abstractmethod

class Search(ABC):
  def search_by_title(self, title):
    None

  def search_by_language(self, language):
    None

  def search_by_genre(self, genre):
    None

  def search_by_release_date(self, rel_date):
    None

  def search_by_city(self, city_name):
    None


class Catalog(Search):
  def __init__(self):
    self.__movie_titles = {}
    self.__movie_languages = {}
    self.__movie_genres = {}
    self.__movie_release_dates = {}
    self.__movie_cities = {}

    def search_by_title(self, title):
      return self.__movie_titles.get(title)
```

Concurrency

How to handle concurrency; such that no two users are able to book the same seat? We can use transactions in SQL databases to avoid any clashes. For example, if we are using SQL server we can utilize Transaction Isolation Levels to lock the rows before we update them. Note: within a transaction, if we read rows we get a write-lock on them so that they can't be updated by anyone else. Here is the sample code:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

    -- Suppose we intend to reserve three seats (IDs: 54, 55, 56) for ShowID=99
    Select * From ShowSeat where ShowID=99 && ShowSeatID in (54, 55, 56) && isReserved=0

    -- if the number of rows returned by the above statement is NOT three, we can return failure to the user.
    update ShowSeat table...
    update Booking table ...

COMMIT TRANSACTION;
```

'Serializable' is the highest isolation level and guarantees safety from Dirty, Nonrepeatable, and Phantoms reads.

Once the above database transaction is successful, we can safely assume that the reservation has been marked successfully and no two customers will be able to reserve the same seat.

Here is the sample Java code:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.ResultSet;

public class Customer extends Person {

  public boolean makeBooking(Booking booking) {
    List<ShowSeat> seats = booking.getSeats();
    Integer seatIds[] = new Integer[seats.size()];
    int index = 0;
    for(ShowSeat seat : seats) {
      seatIds[index++] = seat.getShowSeatId();
    }

    Connection dbConnection = null;
    try {
      dbConnection = getDBConnection();
      dbConnection.setAutoCommit(false);
      // 'Serializable' is the highest isolation level and guarantees safety from
      // Dirty, Nonrepeatable, and Phantoms reads
      dbConnection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

      Statement st = dbConnection.createStatement();
      String selectSQL = "Select * From ShowSeat where ShowID=? && ShowSeatID in (?) && isReserved=0";
      PreparedStatement preparedStatement = dbConnection.prepareStatement(selectSQL);
```

Read JDBC Transaction Isolation Levels for details.

Back

Design Stack Overflow

Next

Design an ATM