Object Oriented Analysis and Design

Object Oriented Analysis and Design is a structured method for analyzing and designing a system by applying object-oriented concepts. This design process consists of an investigation into the objects constituting the system. It starts by first identifying the objects of the system and then figuring out the interactions between various objects.

The process of object oriented analysis and design can be described as:

1. **Identifying the objects in a system.**
2. **Defining relationships between objects.**
3. **Establishing the interface of each object.**
4. **Making a design, which can be converted to executables using object-oriented languages.**

We need a standard method/tool to document all this information; for this purpose we use UML. UML can be considered as the successor of object-oriented (OO) analysis and design. UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are a representation of object-oriented concepts only. Thus, before learning UML, it is essential to understand object-oriented concepts.

Let's find out how we can model using UML.

Use Case Diagram: Used to describe a set of user scenarios, this diagram illustrates the functionality provided by the system.

Class Diagram: Used to describe structure and behavior in the use cases, this diagram provides a conceptual model of the system in terms of entities and their relationships.

Activity Diagram: Used to model the functional flow-of-control between two or more class objects.

Sequence Diagram: Used to describe interactions among classes in terms of an exchange of messages over time.

These are the different types of relationships between classes:

Association: If two classes in a model need to communicate with each other, there must be a link between them. This link can be represented by an association. Associations can be represented in a class diagram by a line between these classes with an arrow indicating the navigation direction.

- By default, associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship. In the diagram below, the association between Pilot and FlightInstance is bi-directional, as both classes know each other.
- By contrast, in a unidirectional association, two classes are related - but only one class knows that the relationship exists. In the below example, only Flight class knows about Aircraft; hence it is a unidirectional association

Multiplicity: Multiplicity indicates how many instances of a class participate in the relationship. It is a constraint that specifies the range of permitted cardinalities between two classes. For example, in the diagram below, one FlightInstance will have two Pilots, while a Pilot can have many FlightInstances. A ranged multiplicity can be expressed as “0…*” which means “zero to many" or as “2…4” which means “two to four”.

Aggregation: Aggregation is a special type of association used to model a “whole to its parts” relationship. In a basic aggregation relationship, the lifecycle of a PART class is independent of the WHOLE class’s lifecycle. In other words, aggregation implies a relationship where the child can exist independently of the parent. In the above diagram, Aircraft can exist without Airline.

Composition: The composition aggregation relationship is just another form of the aggregation relationship, but the child class’s instance lifecycle is dependent on the parent class’s instance lifecycle. In other words, Composition implies a relationship where the child cannot exist independent of the parent. In the above example, WeeklySchedule is composed in Flight which means when Flight lifecycle ends, WeeklySchedule automatically gets destroyed.

Generalization: Generalization is the mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. In the above diagram, Crew, Pilot, and Admin, all are Person.

Dependency: A dependency relationship is a relationship in which one class, the client, uses or depends on another class, the supplier. In the above diagram, FlightReservation depends on Payment.

Design a Library Management System

A Library Management System is a software built to handle the primary housekeeping functions of a library. Libraries rely on library management systems to manage asset collections as well as relationships with their members. Library management systems help libraries keep track of the books and their checkouts, as well as members' subscriptions and profiles. Library management systems also involve maintaining the database for entering new books and recording books that have been borrowed with their respective due dates.

System Requirements

1. Any library member should be able to search books by their title, author, subject category as well by the publication date.
2. Each book will have a unique identification number and other details including a rack number which will help to physically locate the book.
3. There could be more than one copy of a book, and library members should be able to check-out and reserve any copy. We will call each copy of a book a book item.
4. The system should be able to retrieve information like who took a particular book or what are the books checked-out by a specific library member.
5. There should be a maximum limit (5) on how many books a member can check-out.
6. There should be a maximum limit (10) on how many days a member can keep a book.
7. The system should be able to collect fines for books returned after the due date.
8. Members should be able to reserve books that are not currently available.
9. The system should be able to send notifications whenever the reserved books become available, as well as when the book is not returned within the due date.
10. Each book and member card will have a unique barcode. The system will be able to read barcodes from books and members' library cards.

We have three main actors in our system:

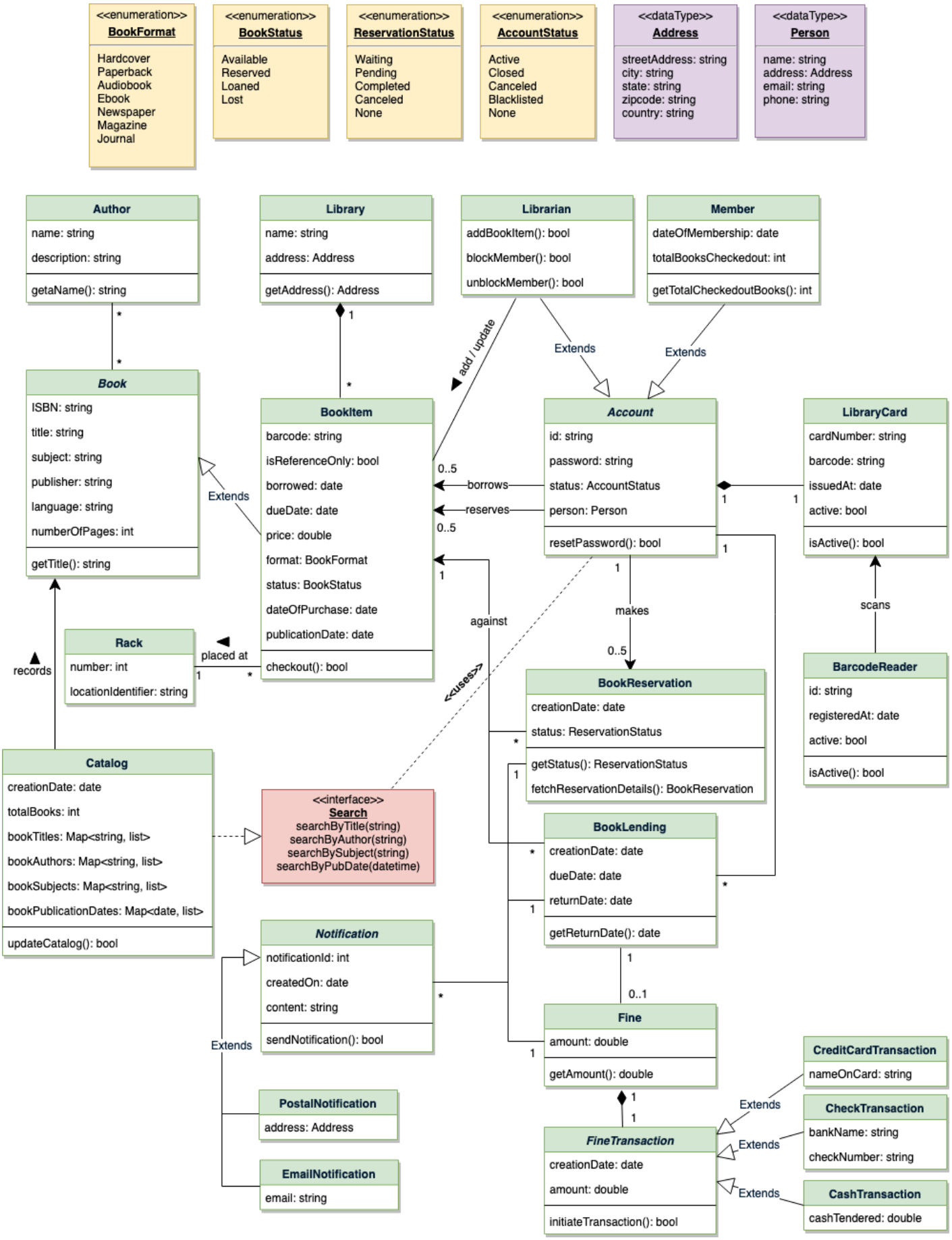
- **Librarian:** Mainly responsible for adding and modifying books, book items, and users. The Librarian can also issue, reserve, and return book items.
- **Member:** All members can search the catalog, as well as check-out, reserve, renew, and return a book.
- **System:** Mainly responsible for sending notifications for overdue books, canceled reservations, etc.

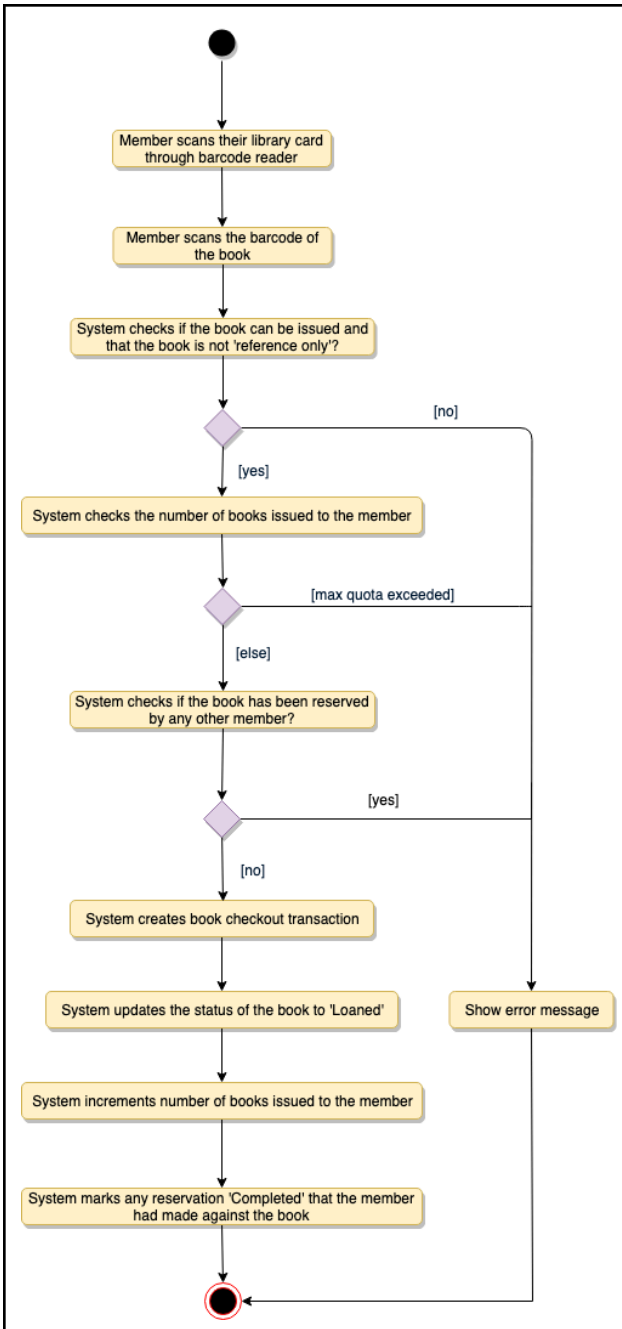
Here are the top use cases of the Library Management System:

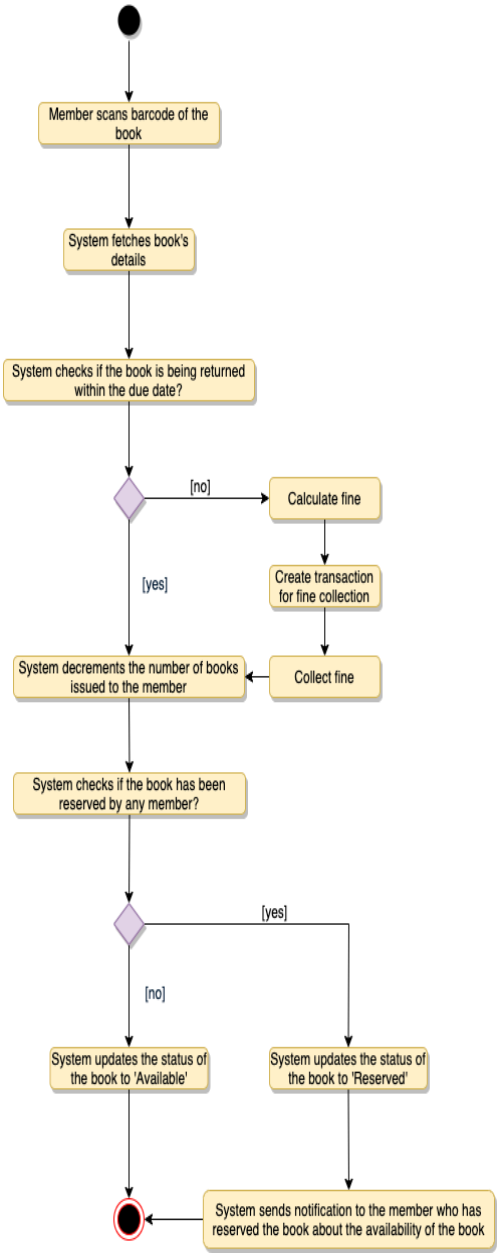
- **Add/Remove/Edit book:** To add, remove or modify a book or book item.
- **Search catalog:** To search books by title, author, subject or publication date.
- **Register new account/cancel membership:** To add a new member or cancel the membership of an existing member.
- **Check-out book:** To borrow a book from the library.
- **Reserve book:** To reserve a book which is not currently available.
- **Renew a book:** To reborrow an already checked-out book.
- **Return a book:** To return a book to the library which was issued to a member.

Here are the main classes of our Library Management System:

- **Library:** The central part of the organization for which this software has been designed. It has attributes like ‘Name’ to distinguish it from any other libraries and ‘Address’ to describe its location.
- **Book:** The basic building block of the system. Every book will have ISBN, Title, Subject, Publishers, etc.
- Any book can have multiple copies, each copy will be considered a book item in our system. Each book item will have a unique barcode.
- **Account:** We will have two types of accounts in the system, one will be a general member, and the other will be a librarian.
- **LibraryCard:** Each library user will be issued a library card, which will be used to identify users while issuing or returning books.
- **BookReservation: R**esponsible for managing reservations against book items.
- **BookLending: Manage the checking-out of book items.**
- **Catalog:** Catalogs contain a list of books sorted on certain criteria. Our system will support searching through four catalogs: Title, Author, Subject, and Publish-date.
- **Fine:** This class will be responsible for calculating and collecting fines from library members.
- **Author:** This class will encapsulate a book author.
- **Rack:** Books will be placed on racks. Each rack will be identified by a rack number and will have a location identifier to describe the physical location of the rack in the library.
- **Notification:** This class will take care of sending notifications to library members.


<<enumeration>>
BookFormat
Hardcover
Paperback
Audiobook
Ebook
Newspaper
Magazine
Journal
<<enumeration>>
BookStatus
Available
Reserved
Loaned
Lost
<<enumeration>>
ReservationStatus
Waiting
Pending
Completed
Canceled
None
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<dataType>>
Person
name: string
address: Address
email: string
phone: string
Author
name: string
description: string
getaName(): string
Library
name: string
address: Address
getAddress(): Address
Librarian
addBookItem(): bool
blockMember(): bool
unblockMember(): bool
Member
dateOfMembership: date
totalBooksCheckedout: int
getTotalCheckedoutBooks(): int
Book
ISBN: string
title: string
subject: string
publisher: string
language: string
numberOfPages: int
getTitle(): string
BookItem
barcode: string
isReferenceOnly: bool
borrowed: date
dueDate: date
price: double
format: BookFormat
status: BookStatus
dateOfPurchase: date
publicationDate: date
checkout(): bool
Account
id: string
password: string
status: AccountStatus
person: Person
resetPassword(): bool
LibraryCard
cardNumber: string
barcode: string
issuedAt: date
active: bool
isActive(): bool
Rack
number: int
locationIdentifier: string
BarcodeReader
id: string
registeredAt: date
active: bool
isActive(): bool
BookReservation
creationDate: date
status: ReservationStatus
getStatus(): ReservationStatus
fetchReservationDetails(): BookReservation
Catalog
creationDate: date
totalBooks: int
bookTitles: Map<string, list>
bookAuthors: Map<string, list>
bookSubjects: Map<string, list>
bookPublicationDates: Map<date, list>
updateCatalog(): bool
<<interface>>
Search
searchByTitle(string)
searchByAuthor(string)
searchBySubject(string)
searchByPubDate(datetime)
BookLending
creationDate: date
dueDate: date
returnDate: date
getReturnDate(): date
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool
Fine
amount: double
getAmount(): double
PostalNotification
address: Address
EmailNotification
email: string
FineTransaction
creationDate: date
amount: double
initiateTransaction(): bool
CreditCardTransaction
nameOnCard: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double
add / update
Extends
Extends
Extends
borrows
reserves
0..5
0..5
against
<<uses>>
makes
0..5
scans
placed at
records
Extends
Extends
Extends
Extends
1
*


Member scans their library card through barcode reader
Member scans the barcode of the book
System checks if the book can be issued and that the book is not 'reference only'?
[no]
[yes]
System checks the number of books issued to the member
[max quota exceeded]
[else]
System checks if the book has been reserved by any other member?
[yes]
[no]
System creates book checkout transaction
System updates the status of the book to 'Loaned'
Show error message
System increments number of books issued to the member
System marks any reservation 'Completed' that the member had made against the book


Member scans barcode of the book
System fetches book's details
System checks if the book is being returned within the due date?
[no]
Calculate fine
[yes]
Create transaction for fine collection
Collect fine
System decrements the number of books issued to the member
System checks if the book has been reserved by any member?
[yes]
[no]
System updates the status of the book to 'Available'
System updates the status of the book to 'Reserved'
System sends notification to the member who has reserved the book about the availability of the book

Design a Parking Lot

A parking lot or car park is a dedicated cleared area that is intended for parking vehicles. In most countries where cars are a major mode of transportation, parking lots are a feature of every city and suburban area. Shopping malls, sports stadiums, megachurches, and similar venues often feature parking lots over large areas.

System Requirements:

We will focus on the following set of requirements while designing the parking lot:

1. The parking lot should have multiple floors where customers can park their cars.
2. The parking lot should have multiple entry and exit points.
3. Customers can collect a parking ticket from the entry points and can pay the parking fee at the exit points on their way out.
4. Customers can pay the tickets at the automated exit panel or to the parking attendant.
5. Customers can pay via both cash and credit cards.
6. Customers should also be able to pay the parking fee at the customer's info portal on each floor. If the customer has paid at the info portal, they don't have to pay at the exit.
7. The system should not allow more vehicles than the maximum capacity of the parking lot. If the parking is full, the system should be able to show a message at the entrance panel and on the parking display board on the ground floor.
8. Each parking floor will have many parking spots. The system should support multiple types of parking spots such as Compact, Large, Handicapped, Motorcycle, etc.
9. The Parking lot should have some parking spots specified for electric cars. These spots should have an electric panel through which customers can pay and charge their vehicles.
10. The system should support parking for different types of vehicles like car, truck, van, motorcycle, etc.
11. Each parking floor should have a display board showing any free parking spot for each spot type.
12. The system should support a per-hour parking fee model. For example, customers have to pay $4 for the first hour, $3.5 for the second and third hours, and $2.5 for all the remaining hours.

Here are the main Actors in our system:

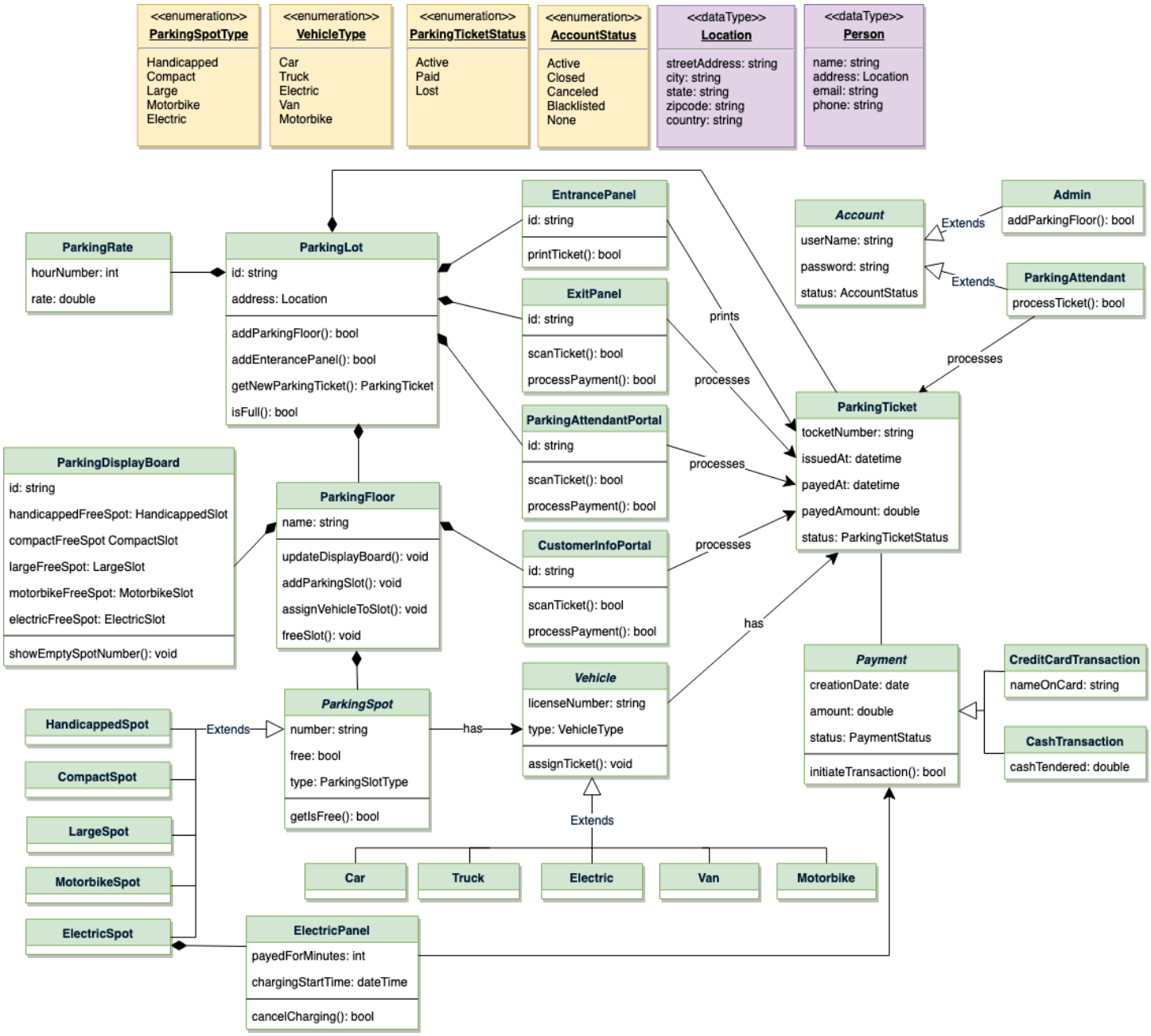
- **Admin:** Mainly responsible for adding and modifying parking floors, parking spots, entrance, and exit panels, adding/removing parking attendants, etc.
- **Customer:** All customers can get a parking ticket and pay for it.
- Parking Attendant: Parking attendants can do all the activities on the customer's behalf, and can take cash for ticket payment.
- **System:** To display messages on different info panels, as well as assigning and removing a vehicle from a parking spot.

Here are the top use cases for Parking Lot:

- **Add/Remove/Edit parking floor:** To add, remove or modify a parking floor from the system. Each floor can have its own display board to show free parking spots.
- **Add/Remove/Edit parking spot:** To add, remove or modify a parking spot on a parking floor.
- **Add/Remove a parking attendant:** To add or remove a parking attendant from the system.
- **Take ticket:** To provide customers with a new parking ticket when entering the parking lot.
- **Scan ticket:** To scan a ticket to find out the total charge.
- **Credit card payment:** To pay the ticket fee with a credit card.
- **Cash payment:** To pay the parking ticket through cash.
- **Add/Modify parking rate:** To allow admin to add or modify the hourly parking rate.

Here are the main classes of our Parking Lot System:

- **ParkingLot:** The central part of the organization for which this software has been designed. It has attributes like 'Name' to distinguish it from any other parking lots and 'Address' to define its location.
- **ParkingFloor:** The parking lot will have many parking floors.
- **ParkingSpot:** Each parking floor will have many parking spots. Our system will support different parking spots 1) Handicapped, 2) Compact, 3) Large, 4) Motorcycle, and 5) Electric.
- **Account:** We will have two types of accounts in the system: one for an Admin, and the other for a parking attendant.
- **Parking ticket:** This class will encapsulate a parking ticket. Customers will take a ticket when they enter the parking lot.
- **Vehicle:** Vehicles will be parked in the parking spots. Our system will support different types of vehicles 1) Car, 2) Truck, 3) Electric, 4) Van and 5) Motorcycle.
- **EntrancePanel and ExitPanel:** EntrancePanel will print tickets, and ExitPanel will facilitate payment of the ticket fee.
- **Payment:** This class will be responsible for making payments. The system will support credit card and cash transactions.
- **ParkingRate:** This class will keep track of the hourly parking rates. It will specify a dollar amount for each hour. For example, for a two hour parking ticket, this class will define the cost for the first and the second hour.
- **ParkingDisplayBoard:** Each parking floor will have a display board to show available parking spots for each spot type. This class will be responsible for displaying the latest availability of free parking spots to the customers.
- **ParkingAttendantPortal:** This class will encapsulate all the operations that an attendant can perform, like scanning tickets and processing payments.
- **CustomerInfoPortal:** This class will encapsulate the info portal that customers use to pay for the parking ticket. Once paid, the info portal will update the ticket to keep track of the payment.
- **ElectricPanel:** Customers will use the electric panels to pay and charge their electric vehicles.


<<enumeration>>
ParkingSpotType
Handicapped
Compact
Large
Motorbike
Electric
<<enumeration>>
VehicleType
Car
Truck
Electric
Van
Motorbike
<<enumeration>>
ParkingTicketStatus
Active
Paid
Lost
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
<<dataType>>
Location
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<dataType>>
Person
name: string
address: Location
email: string
phone: string
ParkingRate
hourNumber: int
rate: double
ParkingLot
id: string
address: Location
addParkingFloor(): bool
addEnteranceePanel(): bool
getNewParkingTicket(): ParkingTicket
isFull(): bool
EntrancePanel
id: string
printTicket(): bool
ExitPanel
id: string
scanTicket(): bool
processPayment(): bool
ParkingAttendantPortal
id: string
scanTicket(): bool
processPayment(): bool
CustomerInfoPortal
id: string
scanTicket(): bool
processPayment(): bool
Account
userName: string
password: string
status: AccountStatus
Admin
addParkingFloor(): bool
ParkingAttendant
processTicket(): bool
Extends
Extends
prints
processes
processes
processes
processes
ParkingTicket
tocketNumber: string
issuedAt: datetime
payedAt: datetime
payedAmount: double
status: ParkingTicketStatus
ParkingDisplayBoard
id: string
handicappedFreeSpot: HandicappedSlot
compactFreeSpot CompactSlot
largeFreeSpot: LargeSlot
motorbikeFreeSpot: MotorbikeSlot
electricFreeSpot: ElectricSlot
showEmptySpotNumber(): void
ParkingFloor
name: string
updateDisplayBoard(): void
addParkingSlot(): void
assignVehicleToSlot(): void
freeSlot(): void
has
Payment
creationDate: date
amount: double
status: PaymentStatus
initiateTransaction(): bool
CreditCardTransaction
nameOnCard: string
CashTransaction
cashTendered: double
HandicappedSpot
CompactSpot
LargeSpot
MotorbikeSpot
ElectricSpot
Extends
ParkingSpot
number: string
free: bool
type: ParkingSlotType
getIsFree(): bool
has
Vehicle
licenseNumber: string
type: VehicleType
assignTicket(): void
Extends
Car
Truck
Electric
Van
Motorbike
ElectricPanel
payedForMinutes: int
chargingStartTime: dateTime
cancelCharging(): bool

Design Amazon - Online Shopping System

System Requirements

We will be designing a system with the following requirements:

1. Users should be able to add new products to sell.
2. Users should be able to search for products by their name or category.
3. Users can search and view all the products, but they will have to become a registered member to buy a product.
4. Users should be able to add/remove/modify product items in their shopping cart.
5. Users can check out and buy items in the shopping cart.
6. Users can rate and add a review for a product.
7. The user should be able to specify a shipping address where their order will be delivered.
8. Users can cancel an order if it has not shipped.
9. Users should get notifications whenever there is a change in the order or shipping status.
10. Users should be able to pay through credit cards or electronic bank transfer.
11. Users should be able to track their shipment to see the current state of their order.

We have four main Actors in our system:

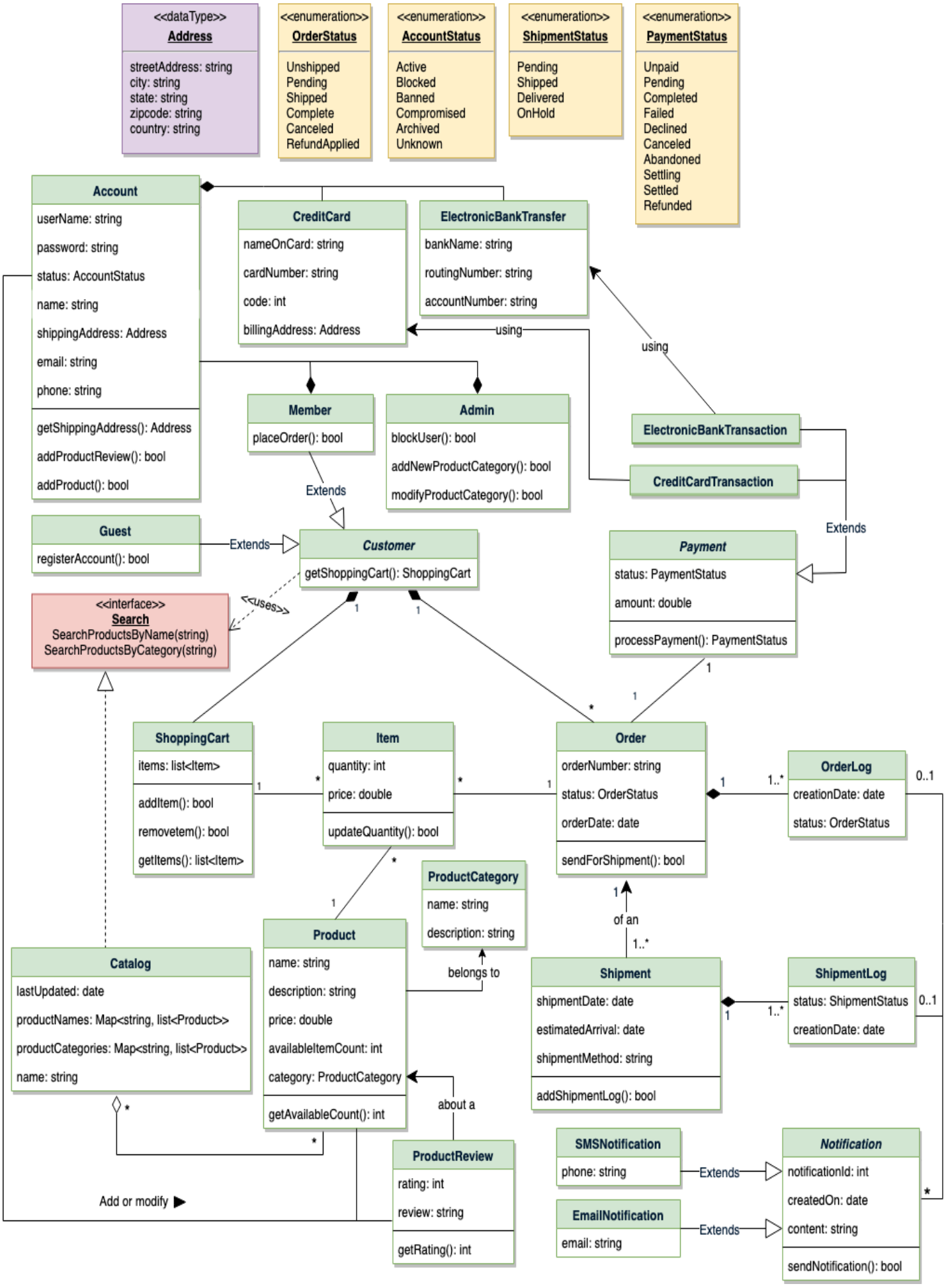
- **Admin**: Mainly responsible for account management and adding or modifying new product categories.
- **Guest**: All guests can search the catalog, add/remove items to the shopping cart, as well as become registered members.
- **Member**: Members can perform all the activities that guests can, in addition to which, they can place orders and add new products to sell.
- **System**: Mainly responsible for sending notifications for orders and shipping updates.

Here are the top use cases of the Online Shopping System:

1. Add/update products; whenever a product is added or modified, we will update the catalog.
2. Search for products by their name or category.
3. Add/remove product items in the shopping cart.
4. Check-out to buy product items in the shopping cart.
5. Make a payment to place an order.
6. Add a new product category.
7. Send notifications to members with shipment updates.

Here are the descriptions of the different classes of our Online Shopping System:

- **Account**: There are two types of registered accounts in the system: one will be an Admin, who is responsible for adding new product categories and blocking/unblocking members; the other, a Member, who can buy/sell products.
- **Guest**: Guests can search for and view products, and add them in the shopping cart. To place an order they have to become a registered member.
- **Catalog**: Users of our system can search for products by their name or category. This class will keep an index of all products for faster search.
- **ProductCategory**: This will encapsulate the different categories of products, such as books, electronics, etc.
- **Product**: This class will encapsulate the entity that the users of our system will be buying and selling. Each Product will belong to a ProductCategory.
- **ProductReview**: Any registered member can add a review about a product.
- **ShoppingCart**: Users will add product items that they intend to buy to the shopping cart.
- **Item:** This class will encapsulate a product item that the users will be buying or placing in the shopping cart. For example, a pen could be a product and if there are 10 pens in the inventory, each of these 10 pens will be considered a product item.
- **Order**: This will encapsulate a buying order to buy everything in the shopping cart.
- **OrderLog**: Will keep a track of the status of orders, such as unshipped, pending, complete, canceled, etc.
- **ShipmentLog**: Will keep a track of the status of shipments, such as pending, shipped, delivered, etc.
- **Notification**: This class will take care of sending notifications to customers.
- **Payment**: This class will encapsulate the payment for an order. Members can pay through credit card or electronic bank transfer.


<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<enumeration>>
OrderStatus
Unshipped
Pending
Shipped
Complete
Canceled
RefundApplied
<<enumeration>>
AccountStatus
Active
Blocked
Banned
Compromised
Archived
Unknown
<<enumeration>>
ShipmentStatus
Pending
Shipped
Delivered
OnHold
<<enumeration>>
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Canceled
Abandoned
Settling
Settled
Refunded
Account
userName: string
password: string
status: AccountStatus
name: string
shippingAddress: Address
email: string
phone: string
getShippingAddress(): Address
addProductReview(): bool
addProduct(): bool
CreditCard
nameOnCard: string
cardNumber: string
code: int
billingAddress: Address
ElectronicBankTransfer
bankName: string
routingNumber: string
accountNumber: string
using
using
Member
placeOrder(): bool
Admin
blockUser(): bool
addNewProductCategory(): bool
modifyProductCategory(): bool
ElectronicBankTransaction
CreditCardTransaction
Extends
Guest
registerAccount(): bool
Extends
Customer
getShoppingCart(): ShoppingCart
Extends
Payment
status: PaymentStatus
amount: double
processPayment(): PaymentStatus
<<interface>>
Search
SearchProductsByName(string)
SearchProductsByCategory(string)
<<uses>>
1
1
1
*
1
ShoppingCart
items: list<Item>
addItem(): bool
removetem(): bool
getItems(): list<Item>
1
*
Item
quantity: int
price: double
updateQuantity(): bool
*
1
Order
orderNumber: string
status: OrderStatus
orderDate: date
sendForShipment(): bool
1
1..*
OrderLog
creationDate: date
status: OrderStatus
0..1
*
1
ProductCategory
name: string
description: string
belongs to
1
of an
1..*
Product
name: string
description: string
price: double
availableItemCount: int
category: ProductCategory
getAvailableCount(): int
Catalog
lastUpdated: date
productNames: Map<string, list<Product>>
productCategories: Map<string, list<Product>>
name: string
*
*
about a
Shipment
shipmentDate: date
estimatedArrival: date
shipmentMethod: string
addShipmentLog(): bool
1
1..*
ShipmentLog
status: ShipmentStatus
creationDate: date
0..1
ProductReview
rating: int
review: string
getRating(): int
Add or modify
SMSNotification
phone: string
Extends
EmailNotification
email: string
Extends
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool
*

Design Stack Overflow

Stack Overflow is one of the largest online communities for developers to learn and share their knowledge. The website provides a platform for its users to ask and answer questions, and through membership and active participation, to vote questions and answers up or down. Users can edit questions and answers in a fashion similar to a wiki.
Users of Stack Overflow can earn reputation points and badges. For example, a person is awarded ten reputation points for receiving an “up” vote on an answer and five points for the “up” vote of a question. They can also receive badges for their valued contributions. A higher reputation lets users unlock new privileges like the ability to vote, comment on, and even edit other people’s posts.

System Requirements

We will be designing a system with the following requirements:

1. Any non-member (guest) can search and view questions. However, to add or upvote a question, they have to become a member.
2. Members should be able to post new questions.
3. Members should be able to add an answer to an open question.
4. Members can add comments to any question or answer.
5. A member can upvote a question, answer or comment.
6. Members can flag a question, answer or comment, for serious problems or moderator attention.
7. Any member can add a bounty to their question to draw attention.
8. Members will earn badges for being helpful.
9. Members can vote to close a question; Moderators can close or reopen any question.
10. Members can add tags to their questions. A tag is a word or phrase that describes the topic of the question.
11. Members can vote to delete extremely off-topic or very low-quality questions.
12. Moderators can close a question or undelete an already deleted question.
13. The system should also be able to identify most frequently used tags in the questions.

We have five main actors in our system:

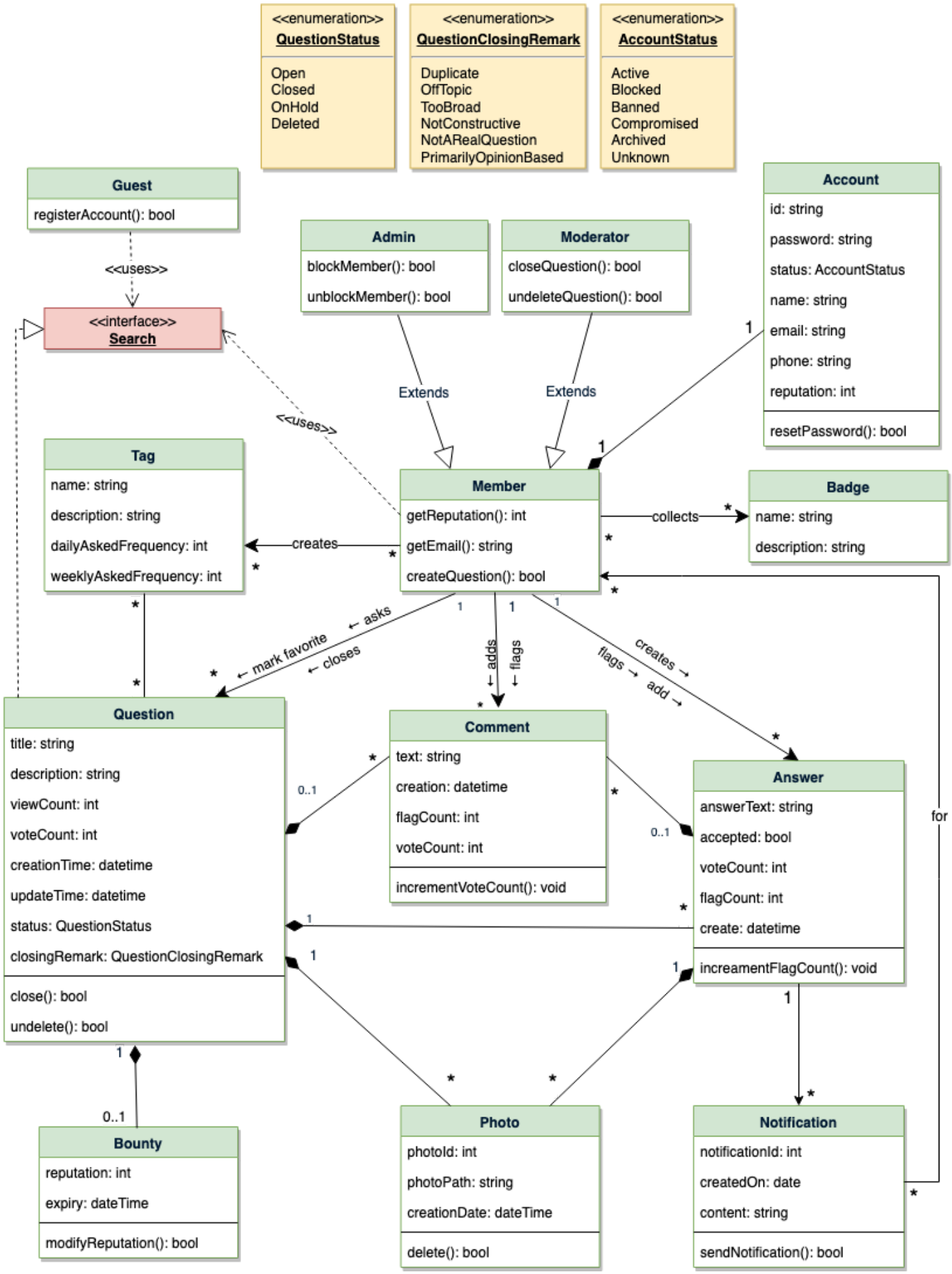
- **Admin**: Mainly responsible for blocking or unblocking members.
- **Guest**: All guests can search and view questions.
- **Member**: Members can perform all activities that guests can, in addition to which they can add/remove questions, answers, and comments. Members can delete and undelete their questions, answers or comments.
- **Moderator**: In addition to all the activities that members can perform, moderators can close/delete/undelete any question.
- **System**: Mainly responsible for sending notifications and assigning badges to members.

Here are the top use cases for Stack Overflow:

1. Search questions.
2. Create a new question with bounty and tags.
3. Add/modify answers to questions.
4. Add comments to questions or answers.
5. Moderators can close, delete, and un-delete any question.

Here are the main classes of Stack Overflow System:

- **Question**: This class is the central part of our system. It has attributes like Title and Description to define the question. In addition to this, we will track the number of times a question has been viewed or voted on. We should also track the status of a question, as well as closing remarks if the question is closed.
- **Answer**: The most important attributes of any answer will be the text and the view count. In addition to that, we will also track the number of times an answer is voted on or flagged. We should also track if the question owner has accepted an answer.
- **Comment**: Similar to answer, comments will have text, and view, vote, and flag counts. Members can add comments to questions and answers.
- **Tag**: Tags will be identified by their names and will have a field for a description to define them. We will also track daily and weekly frequencies at which tags are associated with questions.
- **Badge**: Similar to tags, badges will have a name and description.
- **Photo**: Questions or answers can have photos.
- **Bounty**: Each member, while asking a question, can place a bounty to draw attention. Bounties will have a total reputation and an expiry date.
- **Account**: We will have four types of accounts in the system, guest, member, admin, and moderator. Guests can search and view questions. Members can ask questions and earn reputation by answering questions and from bounties.
- **Notification**: This class will be responsible for sending notifications to members and assigning badges to members based on their reputations.


<<enumeration>>
QuestionStatus
Open
Closed
OnHold
Deleted
<<enumeration>>
QuestionClosingRemark
Duplicate
OffTopic
TooBroad
NotConstructive
NotARealQuestion
PrimarilyOpinionBased
<<enumeration>>
AccountStatus
Active
Blocked
Banned
Compromised
Archived
Unknown
Guest
registerAccount(): bool
<<uses>>
<<interface>>
Search
Admin
blockMember(): bool
unblockMember(): bool
Moderator
closeQuestion(): bool
undeleteQuestion(): bool
Account
id: string
password: string
status: AccountStatus
name: string
email: string
phone: string
reputation: int
resetPassword(): bool
Extends
Extends
<<uses>>
Tag
name: string
description: string
dailyAskedFrequency: int
weeklyAskedFrequency: int
Member
getReputation(): int
getEmail(): string
createQuestion(): bool
Badge
name: string
description: string
collects
creates
asks
mark favorite
closes
adds
flags
flags
add
creates
Question
title: string
description: string
viewCount: int
voteCount: int
creationTime: datetime
updateTime: datetime
status: QuestionStatus
closingRemark: QuestionClosingRemark
close(): bool
undelete(): bool
Comment
text: string
creation: datetime
flagCount: int
voteCount: int
incrementVoteCount(): void
Answer
answerText: string
accepted: bool
voteCount: int
flagCount: int
create: datetime
increamentFlagCount(): void
for
Bounty
reputation: int
expiry: dateTime
modifyReputation(): bool
Photo
photoId: int
photoPath: string
creationDate: dateTime
delete(): bool
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool

Design a Movie Ticket Booking System

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.

System Requirements

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.

We have five main Actors in our system:

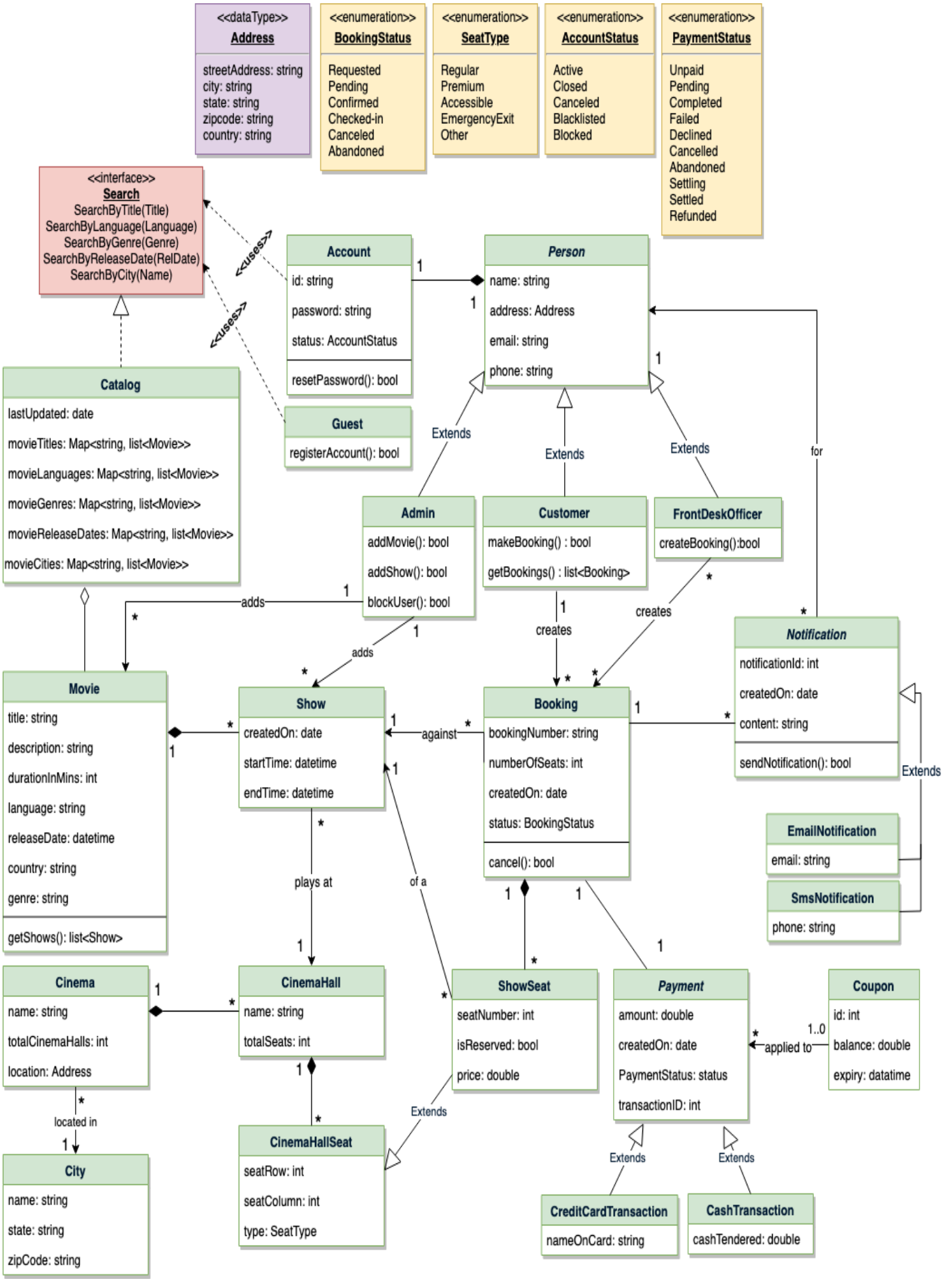
- Admin: Responsible for adding new movies and their shows, canceling any movie or show, blocking/unblocking customers, etc.
- FrontDeskOfficer: Can book/cancel tickets.
- Customer: Can view movie schedules, book, and cancel tickets.
- Guest: All guests can search movies but to book seats they have to 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.

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 movie 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.


<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<enumeration>>
BookingStatus
Requested
Pending
Confirmed
Checked-in
Canceled
Abandoned
<<enumeration>>
SeatType
Regular
Premium
Accessible
EmergencyExit
Other
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
Blocked
<<enumeration>>
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Cancelled
Abandoned
Settling
Settled
Refunded
<<interface>>
Search
SearchByTitle(Title)
SearchByLanguage(Language)
SearchByGenre(Genre)
SearchByReleaseDate(RelDate)
SearchByCity(Name)
<<uses>>
<<uses>>
Account
id: string
password: string
status: AccountStatus
resetPassword(): bool
Person
name: string
address: Address
email: string
phone: string
Guest
registerAccount(): bool
Catalog
lastUpdated: date
movieTitles: Map<string, list<Movie>>
movieLanguages: Map<string, list<Movie>>
movieGenres: Map<string, list<Movie>>
movieReleaseDates: Map<string, list<Movie>>
movieCities: Map<string, list<Movie>>
Extends
Extends
Extends
for
Admin
addMovie(): bool
addShow(): bool
blockUser(): bool
Customer
makeBooking() : bool
getBookings() : list<Booking>
FrontDeskOfficer
createBooking():bool
adds
adds
creates
creates
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool
Movie
title: string
description: string
durationInMins: int
language: string
releaseDate: datetime
country: string
genre: string
getShows(): list<Show>
Show
createdOn: date
startTime: datetime
endTime: datetime
against
Booking
bookingNumber: string
numberOfSeats: int
createdOn: date
status: BookingStatus
cancel(): bool
Extends
EmailNotification
email: string
SmsNotification
phone: string
plays at
of a
Cinema
name: string
totalCinemaHalls: int
location: Address
CinemaHall
name: string
totalSeats: int
ShowSeat
seatNumber: int
isReserved: bool
price: double
Payment
amount: double
createdOn: date
PaymentStatus: status
transactionID: int
Coupon
id: int
balance: double
expiry: datatime
applied to
1..0
located in
Extends
Extends
Extends
City
name: string
state: string
zipCode: string
CinemaHallSeat
seatRow: int
seatColumn: int
type: SeatType
CreditCardTransaction
nameOnCard: string
CashTransaction
cashTendered: double

Design an ATM

An automated teller machine (ATM) is an electronic telecommunications instrument that provides the clients of a financial institution with access to financial transactions in a public space without the need for a cashier or bank teller. ATMs are necessary as not all the bank branches are open every day of the week, and some customers may not be in a position to visit a bank each time they want to withdraw or deposit money.

System Requirements

The main components of the ATM that will affect interactions between the ATM and its users are:

1. Card reader: to read the users' ATM cards.
2. Keypad: to enter information into the ATM e.g. PIN. cards.
3. Screen: to display messages to the users.
4. Cash dispenser: for dispensing cash.
5. Deposit slot: For users to deposit cash or checks.
6. Printer: for printing receipts.
7. Communication/Network Infrastructure: it is assumed that the ATM has a communication infrastructure to communicate with the bank upon any transaction or activity.

The user can have two types of accounts: 1) Checking, and 2) Savings, and should be able to perform the following five transactions on the ATM:

1. Balance inquiry: To see the amount of funds in each account.
2. Deposit cash: To deposit cash.
3. Deposit check: To deposit checks.
4. Withdraw cash: To withdraw money from their checking account.
5. Transfer funds: To transfer funds to another account.

Here are the actors of the ATM system and their use cases:

Operator: The operator will be responsible for the following operations:

1. Turning the ATM ON/OFF using the designated Key-Switch.
2. Refilling the ATM with cash.
3. Refilling the ATM's printer with receipts.
4. Refilling the ATM's printer with INK.
5. Take out deposited cash and checks.

Customer: The ATM customer can perform the following operations:

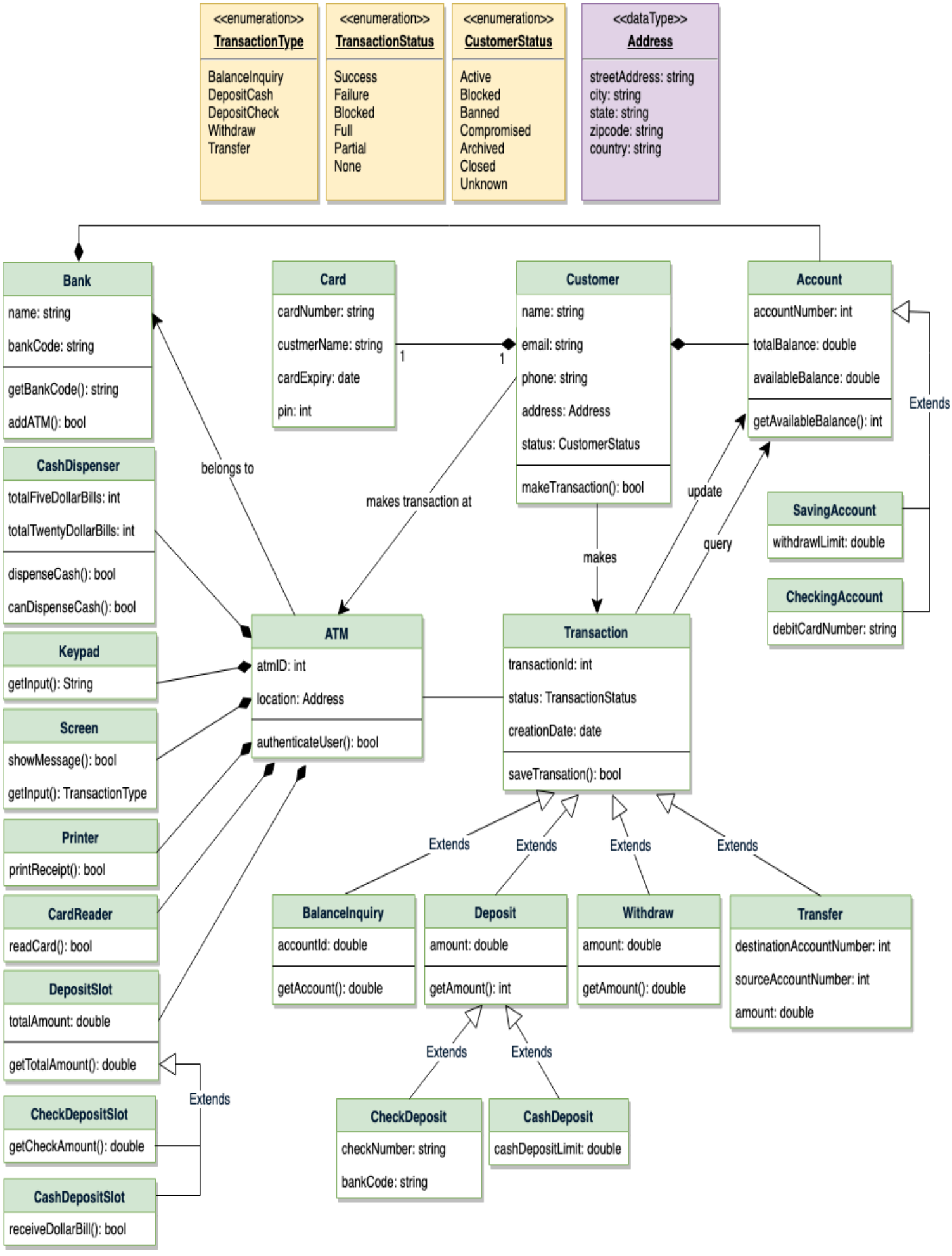
1. Balance inquiry: the user can view his/her account balance.
2. Cash withdrawal: the user can withdraw a certain amount of cash.
3. Deposit funds: the user can deposit cash or checks.
4. Transfer funds: the user can transfer funds to other accounts.

Bank Manager: The Bank Manager can perform the following operations:

1. Generate a report to check total deposits.
2. Generate a report to check total withdrawals.
3. Print total deposits/withdrawal reports.
4. Check the remaining cash in the ATM.

Here are the main classes of the ATM System:

- **ATM**: The main part of the system for which this software has been designed. It has attributes like 'atmID' to distinguish it from other available ATMs, and 'location' which defines the physical address of the ATM.
- **CardReader**: To encapsulate the ATM's card reader used for user authentication.
- **CashDispenser**: To encapsulate the ATM component which will dispense cash.
- **Keypad**: The user will use the ATM's keypad to enter their PIN or amounts.
- **Screen**: Users will be shown all messages on the screen and they will select different transactions by touching the screen.
- **Printer**: To print receipts.
- **DepositSlot**: Users can deposit checks or cash through the deposit slot.
- **Bank**: To encapsulate the bank which owns the ATM. The bank will hold all the account information and the ATM will communicate with the bank to perform customer transactions.
- **Account**: We'll have two types of accounts in the system: 1)Checking and 2)Saving.
- **Customer**: This class will encapsulate the ATM's customer. It will have the customer's basic information like name, email, etc.
- **Card**: Encapsulating the ATM card that the customer will use to authenticate themselves. Each customer can have one card.
- **Transaction**: Encapsulating all transactions that the customer can perform on the ATM, like BalanceInquiry, Deposit, Withdraw, etc.


<<enumeration>>
TransactionType
BalanceInquiry
DepositCash
DepositCheck
Withdraw
Transfer
<<enumeration>>
TransactionStatus
Success
Failure
Blocked
Full
Partial
None
<<enumeration>>
CustomerStatus
Active
Blocked
Banned
Compromised
Archived
Closed
Unknown
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
Bank
name: string
bankCode: string
getBankCode(): string
addATM(): bool
Card
cardNumber: string
custmerName: string
cardExpiry: date
pin: int
Customer
name: string
email: string
phone: string
address: Address
status: CustomerStatus
makeTransaction(): bool
Account
accountNumber: int
totalBalance: double
availableBalance: double
getAvailableBalance(): int
CashDispenser
totalFiveDollarBills: int
totalTwentyDollarBills: int
dispenseCash(): bool
canDispenseCash(): bool
Keypad
getInput(): String
Screen
showMessage(): bool
getInput(): TransactionType
Printer
printReceipt(): bool
CardReader
readCard(): bool
DepositSlot
totalAmount: double
getTotalAmount(): double
CheckDepositSlot
getCheckAmount(): double
CashDepositSlot
receiveDollarBill(): bool
ATM
atmID: int
location: Address
authenticateUser(): bool
Transaction
transactionId: int
status: TransactionStatus
creationDate: date
saveTransation(): bool
SavingAccount
withdrawlLimit: double
CheckingAccount
debitCardNumber: string
BalanceInquiry
accountId: double
getAccount(): double
Deposit
amount: double
getAmount(): int
Withdraw
amount: double
getAmount(): double
Transfer
destinationAccountNumber: int
sourceAccountNumber: int
amount: double
CheckDeposit
checkNumber: string
bankCode: string
CashDeposit
cashDepositLimit: double
belongs to
makes transaction at
1
1
makes
update
query
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Extends

Design an Airline Management System

An Airline Management System is a managerial software which targets to control all operations of an airline. Airlines provide transport services for their passengers. They carry or hire aircraft for this purpose. All operations of an airline company are controlled by their airline management system.
This system involves the scheduling of flights, air ticket reservations, flight cancellations, customer support, and staff management. Daily flight updates can also be retrieved by using the system.

We will focus on the following set of requirements while designing the Airline Management System:

1. Customers should be able to search for flights for a given date and source/destination airport.
2. Customers should be able to reserve a ticket for any scheduled flight. Customers can also build a multi-flight itinerary.
3. Users of the system can check flight schedules, their departure time, available seats, arrival time, and other flight details.
4. Customers can make reservations for multiple passengers under one itinerary.
5. Only the admin of the system can add new aircrafts, flights, and flight schedules. Admin can cancel any pre-scheduled flight (all stakeholders will be notified).
6. Customers can cancel their reservation and itinerary.
7. The system should be able to handle the assignment of pilots and crew members to flights.
8. The system should be able to handle payments for reservations.
9. The system should be able to send notifications to customers whenever a reservation is made/modified or there is an update for their flights.

We have five main Actors in our system:

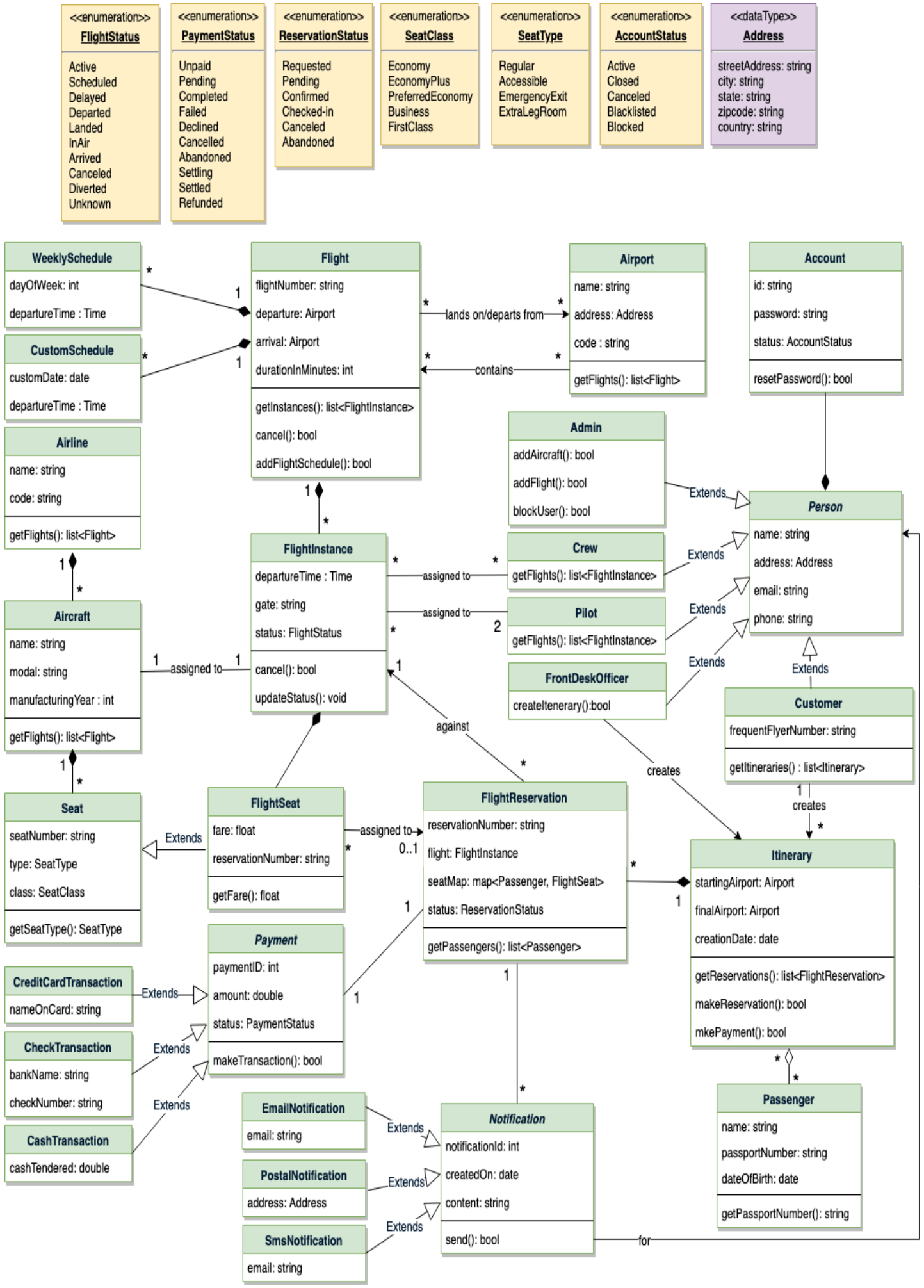
- **Admin**: Responsible for adding new flights and their schedules, canceling any flight, maintaining staff-related work, etc.
- **Front desk officer**: Will be able to reserve/cancel tickets.
- **Customer**: Can view flight schedule, reserve and cancel tickets.
- **Pilot/Crew:** Can view their assigned flights and their schedules.
- **System**: Mainly responsible for sending notifications regarding itinerary changes, flight status updates, etc.

Here are the top use cases of the Airline Management System:

- **Search Flights:** To search the flight schedule to find flights for a suitable date and time.
- **Create/Modify/View reservation**: To reserve a ticket, cancel it, or view details about the flight or ticket.
- **Assign seats to passengers**: To assign seats to passengers for a flight instance with their reservation.
- **Make payment for a reservation:** To pay for the reservation.
- **Update flight schedule**: To make changes in the flight schedule, and to add or remove any flight.
- **Assign pilots and crew**: To assign pilots and crews to flights.

Here are the main classes of our Airline Management System:

- **Airline**: The main part of the organization for which this software has been designed. It has attributes like ‘name’ and an airline code to distinguish the airline from other airlines.
- **Airport**: Each airline operates out of different airports. Each airport has a name, address, and a unique code.
- **Aircraft**: Airlines own or hire aircraft to carry out their flights. Each aircraft has attributes like name, model, manufacturing year, etc.
- **Flight**: The main entity of the system. Each flight will have a flight number, departure and arrival airport, assigned aircraft, etc.
- **FlightInstance**: Each flight can have multiple occurrences; each occurrence will be considered a flight instance in our system. For example, if a British Airways flight from London to Tokyo (flight number: BA212) occurs twice a week, each of these occurrences will be considered a separate flight instance in our system.
- **WeeklySchedule** and CustomSchedule: Flights can have multiple schedules and each schedule will create a flight instance.
- **FlightReservation**: A reservation is made against a flight instance and has attributes like a unique reservation number, list of passengers and their assigned seats, reservation status, etc.
- **Itinerary**: An itinerary can have multiple flights.
- **FlightSeat**: This class will represent all seats of an aircraft assigned to a specific flight instance. All reservations of this flight instance will assign seats to passengers through this class.
- **Payment**: Will be responsible for collecting payments from customers.
- **Notification**: This class will be responsible for sending notifications for flight reservations, flight status update, etc.


<<enumeration>>
FlightStatus
Active
Scheduled
Delayed
Departed
Landed
InAir
Arrived
Canceled
Diverted
Unknown
<<enumeration>>
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Cancelled
Abandoned
Settling
Settled
Refunded
<<enumeration>>
ReservationStatus
Requested
Pending
Confirmed
Checked-in
Canceled
Abandoned
<<enumeration>>
SeatClass
Economy
EconomyPlus
PreferredEconomy
Business
FirstClass
<<enumeration>>
SeatType
Regular
Accessible
EmergencyExit
ExtraLegRoom
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
Blocked
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
WeeklySchedule
dayOfWeek: int
departureTime : Time
CustomSchedule
customDate: date
departureTime : Time
Flight
flightNumber: string
departure: Airport
arrival: Airport
durationInMinutes: int
getInstances(): list<FlightInstance>
cancel(): bool
addFlightSchedule(): bool
lands on/departs from
contains
Airport
name: string
address: Address
code : string
getFlights(): list<Flight>
Account
id: string
password: string
status: AccountStatus
resetPassword(): bool
Airline
name: string
code: string
getFlights(): list<Flight>
Admin
addAircraft(): bool
addFlight(): bool
blockUser(): bool
Extends
Person
name: string
address: Address
email: string
phone: string
FlightInstance
departureTime : Time
gate: string
status: FlightStatus
cancel(): bool
updateStatus(): void
assigned to
Crew
getFlights(): list<FlightInstance>
Pilot
getFlights(): list<FlightInstance>
FrontDeskOfficer
createItenerary():bool
Aircraft
name: string
modal: string
manufacturingYear : int
getFlights(): list<Flight>
against
Customer
frequentFlyerNumber: string
getItineraries() : list<Itinerary>
creates
Seat
seatNumber: string
type: SeatType
class: SeatClass
getSeatType(): SeatType
FlightSeat
fare: float
reservationNumber: string
getFare(): float
FlightReservation
reservationNumber: string
flight: FlightInstance
seatMap: map<Passenger, FlightSeat>
status: ReservationStatus
getPassengers(): list<Passenger>
Itinerary
startingAirport: Airport
finalAirport: Airport
creationDate: date
getReservations(): list<FlightReservation>
makeReservation(): bool
mkePayment(): bool
Payment
paymentID: int
amount: double
status: PaymentStatus
makeTransaction(): bool
CreditCardTransaction
nameOnCard: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double
EmailNotification
email: string
PostalNotification
address: Address
SmsNotification
email: string
Notification
notificationId: int
createdOn: date
content: string
send(): bool
for
Passenger
name: string
passportNumber: string
dateOfBirth: date
getPassportNumber(): string

Design a Hotel Management System

A Hotel Management System is a software built to handle all online hotel activities easily and safely. This System will give the hotel management power and flexibility to manage the entire system from a single online portal. The system allows the manager to keep track of all the available rooms in the system as well as to book rooms and generate bills.

System Requirements

We'll focus on the following set of requirements while designing the Hotel Management System:

1. The system should support the booking of different room types like standard, deluxe, family suite, etc.
2. Guests should be able to search the room inventory and book any available room.
3. The system should be able to retrieve information, such as who booked a particular room, or what rooms were booked by a specific customer.
4. The system should allow customers to cancel their booking - and provide them with a full refund if the cancellation occurs before 24 hours of the check-in date.
5. The system should be able to send notifications whenever the booking is nearing the check-in or check-out date.
6. The system should maintain a room housekeeping log to keep track of all housekeeping tasks.
7. Any customer should be able to add room services and food items.
8. Customers can ask for different amenities.
9. The customers should be able to pay their bills through credit card, check or cash.

Here are the main Actors in our system:

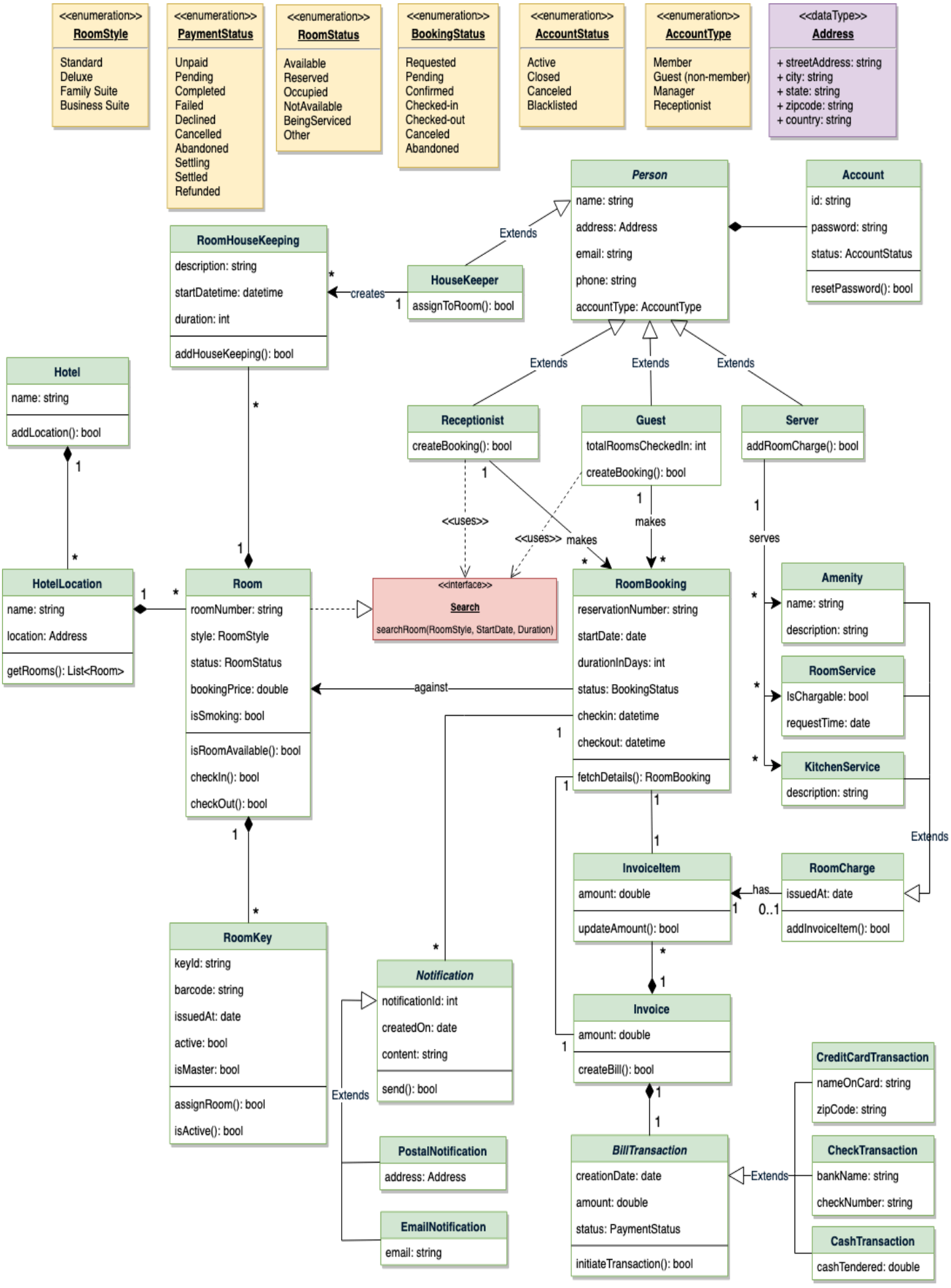
- Guest: All guests can search the available rooms, as well as make a booking.
- Receptionist: Mainly responsible for adding and modifying rooms, creating room bookings, check-in, and check-out customers.
- System: Mainly responsible for sending notifications for room booking, cancellation, etc.
- Manager: Mainly responsible for adding new workers.
- Housekeeper: To add/modify housekeeping records of rooms.
- Server: To add/modify room service records of rooms.

Here are the top use cases of the Hotel Management System:

- **Add/Remove/Edit room:** To add, remove, or modify a room in the system.
- **Search room**: To search for rooms by type and availability.
- **Register or cancel an account**: To add a new member or cancel the membership of an existing member.
- **Book room:** To book a room.
- **Check-in:** To let the guest check-in for their booking.
- **Check-out:** To track the end of the booking and the return of the room keys.
- **Add room charge:** To add a room service charge to the customer's bill.
- **Update housekeeping log:** To add or update the housekeeping entry of a room.

Here are the main classes of our Hotel Management System:

- **Hotel** and **HotelLocation**: Our system will support multiple locations of a hotel.
- **Room**: The basic building block of the system. Every room will be uniquely identified by the room number. Each Room will have attributes like Room Style, Booking Price, etc.
- **Account**: We will have different types of accounts in the system: one will be a guest to search and book rooms, another will be a receptionist. Housekeeping will keep track of the housekeeping records of a room, and a Server will handle room service.
- **RoomBooking**: This class will be responsible for managing bookings for a room.
- **Notification**: Will take care of sending notifications to guests.
- **RoomHouseKeeping**: To keep track of all housekeeping records for rooms.
- **RoomCharge**: Encapsulates the details about different types of room services that guests have requested.
- **Invoice**: Contains different invoice-items for every charge against the room.
- **RoomKey**: Each room can be assigned an electronic key card. Keys will have a barcode and will be uniquely identified by a key-ID.


<<enumeration>>
RoomStyle
Standard
Deluxe
Family Suite
Business Suite
<<enumeration>>
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Cancelled
Abandoned
Settling
Settled
Refunded
<<enumeration>>
RoomStatus
Available
Reserved
Occupied
NotAvailable
BeingServiced
Other
<<enumeration>>
BookingStatus
Requested
Pending
Confirmed
Checked-in
Checked-out
Canceled
Abandoned
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
<<enumeration>>
AccountType
Member
Guest (non-member)
Manager
Receptionist
<<dataType>>
Address
+ streetAddress: string
+ city: string
+ state: string
+ zipcode: string
+ country: string
Person
name: string
address: Address
email: string
phone: string
accountType: AccountType
Account
id: string
password: string
status: AccountStatus
resetPassword(): bool
Extends
RoomHouseKeeping
description: string
startDatetime: datetime
duration: int
addHouseKeeping(): bool
creates
HouseKeeper
assignToRoom(): bool
Hotel
name: string
addLocation(): bool
Extends
Extends
Extends
Receptionist
createBooking(): bool
Guest
totalRoomsCheckedIn: int
createBooking(): bool
Server
addRoomCharge(): bool
<<uses>>
<<uses>>
makes
makes
serves
HotelLocation
name: string
location: Address
getRooms(): List<Room>
Room
roomNumber: string
style: RoomStyle
status: RoomStatus
bookingPrice: double
isSmoking: bool
isRoomAvailable(): bool
checkIn(): bool
checkOut(): bool
<<interface>>
Search
searchRoom(RoomStyle, StartDate, Duration)
RoomBooking
reservationNumber: string
startDate: date
durationInDays: int
status: BookingStatus
checkin: datetime
checkout: datetime
fetchDetails(): RoomBooking
against
Amenity
name: string
description: string
RoomService
IsChargable: bool
requestTime: date
KitchenService
description: string
Extends
InvoiceItem
amount: double
updateAmount(): bool
has
0..1
RoomCharge
issuedAt: date
addInvoiceItem(): bool
RoomKey
keyId: string
barcode: string
issuedAt: date
active: bool
isMaster: bool
assignRoom(): bool
isActive(): bool
Notification
notificationId: int
createdOn: date
content: string
send(): bool
Extends
Invoice
amount: double
createBill(): bool
PostalNotification
address: Address
EmailNotification
email: string
BillTransaction
creationDate: date
amount: double
status: PaymentStatus
initiateTransaction(): bool
Extends
CreditCardTransaction
nameOnCard: string
zipCode: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double

Design a Restaurant Management System

A Restaurant Management System is a software built to handle all restaurant activities in an easy and safe manner. This System will give the Restaurant management power and flexibility to manage the entire system from a single portal. The system allows the manager to keep track of available tables in the system as well as the reservation of tables and bill generation.

System Requirements

We will focus on the following set of requirements while designing the Restaurant Management System:

1. The restaurant will have different branches.
2. Each restaurant branch will have a menu.
3. The menu will have different menu sections, containing different menu items.
4. The waiter should be able to create an order for a table and add meals for each seat.
5. Each meal can have multiple meal items. Each meal item corresponds to a menu item.
6. The system should be able to retrieve information about tables currently available to seat walk-in customers.
7. The system should support the reservation of tables.
8. The receptionist should be able to search for available tables by date/time and reserve a table.
9. The system should allow customers to cancel their reservation.
10. The system should be able to send notifications whenever the reservation time is approaching.
11. The customers should be able to pay their bills through credit card, check or cash.
12. Each restaurant branch can have multiple seating arrangements of tables.

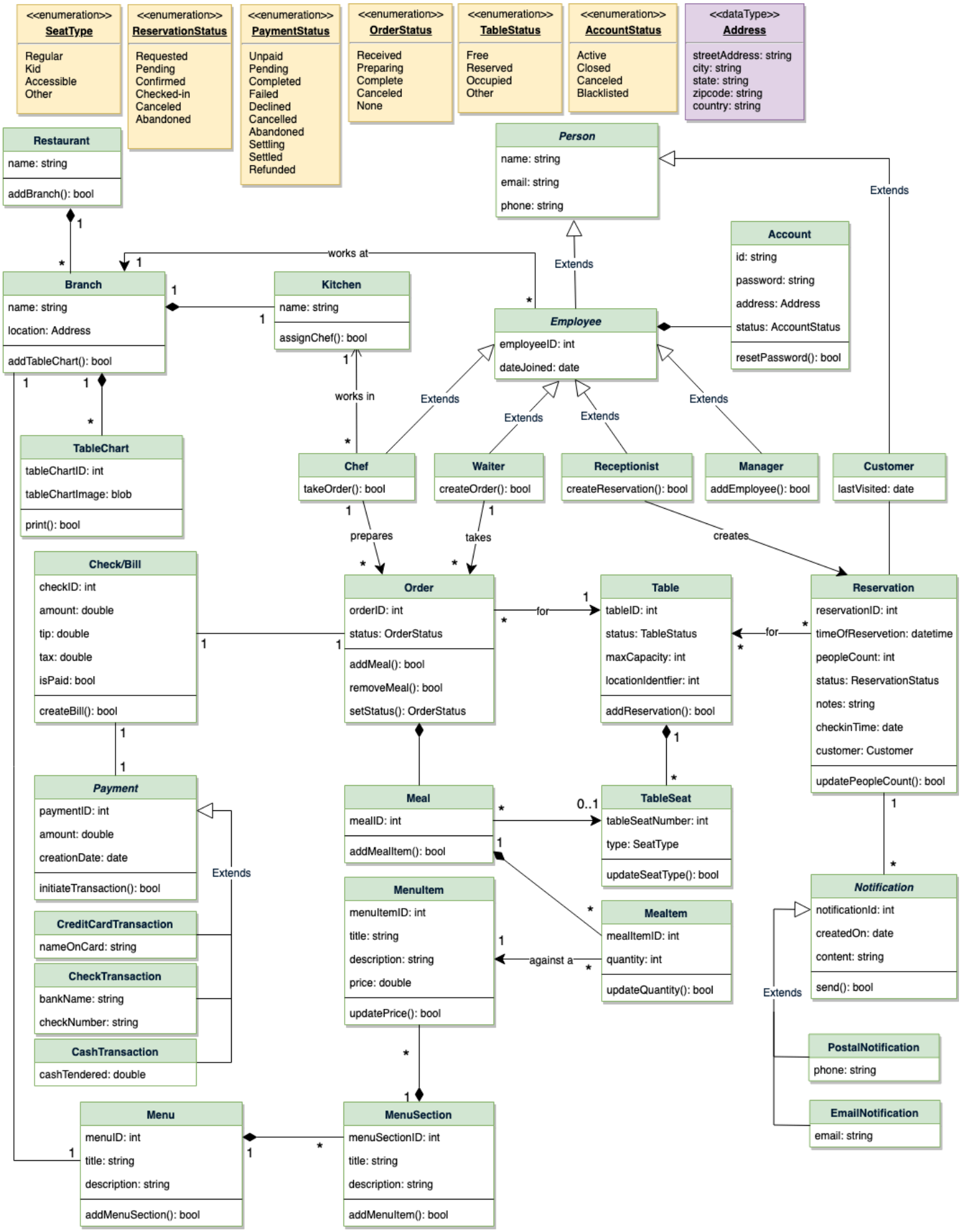
Here are the main Actors in our system:
Receptionist: Mainly responsible for adding and modifying tables and their layout, and creating and canceling table reservations.
Waiter: To take/modify orders.
Manager: Mainly responsible for adding new workers and modifying the menu.
Chef: To view and work on an order.
Cashier: To generate checks and process payments. System: Mainly responsible for sending notifications about table reservations, cancellations, etc.

Here are the top use cases of the Restaurant Management System:

- **Add/Modify tables:** To add, remove, or modify a table in the system.
- **Search tables:** To search for available tables for reservation.
- **Place order:** Add a new order in the system for a table.
- **Update order:** Modify an already placed order, which can include adding/modifying meals or meal items.
- **Create a reservation:** To create a table reservation for a certain date/time for an available table.
- **Cancel reservation:** To cancel an existing reservation.
- **Check-in:** To let the guest check in for their reservation.
- **Make payment:** Pay the check for the food.

Here is the description of the different classes of our Restaurant Management System:

- **Restaurant**: This class represents a restaurant. Each restaurant has registered employees. The employees are part of the restaurant because if the restaurant becomes inactive, all its employees will automatically be deactivated.
- **Branch**: Any restaurants can have multiple branches. Each branch will have its own set of employees and menus.
- **Menu**: All branches will have their own menu.
- **MenuSection** and MenuItem: A menu has zero or more menu sections. Each menu section consists of zero or more menu items.
- **Table** and **TableSeat**: The basic building block of the system. Every table will have a unique identifier, maximum seating capacity, etc. Each table will have multiple seats.
- **Order**: This class encapsulates the order placed by a customer.
- **Meal**: Each order will consist of separate meals for each table seat.
- **Meal Item**: Each Meal will consist of one or more meal items corresponding to a menu item.
- **Account**: We'll have different types of accounts in the system, one will be a receptionist to search and reserve tables and the other, the waiter will place orders in the system.
- **Notification**: Will take care of sending notifications to customers.
- **Bill**: Contains different bill-items for every meal item.


<<enumeration>>
SeatType
Regular
Kid
Accessible
Other
<<enumeration>>
ReservationStatus
Requested
Pending
Confirmed
Checked-in
Canceled
Abandoned
<<enumeration>>
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Cancelled
Abandoned
Settling
Settled
Refunded
<<enumeration>>
OrderStatus
Received
Preparing
Complete
Canceled
None
<<enumeration>>
TableStatus
Free
Reserved
Occupied
Other
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
Restaurant
name: string
addBranch(): bool
Person
name: string
email: string
phone: string
Extends
Account
id: string
password: string
address: Address
status: AccountStatus
resetPassword(): bool
works at
Branch
name: string
location: Address
addTableChart(): bool
Kitchen
name: string
assignChef(): bool
Employee
employeeID: int
dateJoined: date
works in
TableChart
tableChartID: int
tableChartImage: blob
print(): bool
Chef
takeOrder(): bool
Waiter
createOrder(): bool
Receptionist
createReservation(): bool
Manager
addEmployee(): bool
Customer
lastVisited: date
prepares
takes
creates
Check/Bill
checkID: int
amount: double
tip: double
tax: double
isPaid: bool
createBill(): bool
Order
orderID: int
status: OrderStatus
addMeal(): bool
removeMeal(): bool
setStatus(): OrderStatus
for
Table
tableID: int
status: TableStatus
maxCapacity: int
locationIdentfier: int
addReservation(): bool
Reservation
reservationID: int
timeOfReservetion: datetime
peopleCount: int
status: ReservationStatus
notes: string
checkinTime: date
customer: Customer
updatePeopleCount(): bool
Payment
paymentID: int
amount: double
creationDate: date
initiateTransaction(): bool
CreditCardTransaction
nameOnCard: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double
Meal
mealID: int
addMealItem(): bool
TableSeat
tableSeatNumber: int
type: SeatType
updateSeatType(): bool
MenuItem
menuItemID: int
title: string
description: string
price: double
updatePrice(): bool
MealItem
mealItemID: int
quantity: int
updateQuantity(): bool
against a
Notification
notificationId: int
createdOn: date
content: string
send(): bool
PostalNotification
phone: string
EmailNotification
email: string
Menu
menuID: int
title: string
description: string
addMenuSection(): bool
MenuSection
menuSectionID: int
title: string
description: string
addMenuItem(): bool

Design Chess

System Requirements

We'll focus on the following set of requirements while designing the game of chess:

1. The system should support two online players to play a game of chess.
2. All rules of international chess will be followed.
3. Each player will be randomly assigned a side, black or white.
4. Both players will play their moves one after the other. The white side plays the first move.
5. Players can't cancel or roll back their moves.
6. The system should maintain a log of all moves by both players.
7. Each side will start with 8 pawns, 2 rooks, 2 bishops, 2 knights, 1 queen, and 1 king.
8. The game can finish either in a checkmate from one side, forfeit or stalemate (a draw), or resignation.

We have two actors in our system:

- Player: A registered account in the system, who will play the game. The player will play chess moves.
- Admin: To ban/modify players.

Here are the top use cases for chess:

- Player moves a piece: To make a valid move of any chess piece.
- Resign or forfeit a game: A player resigns from/forfeits the game.
- Register new account/Cancel membership: To add a new member or cancel an existing member.
- Update game log: To add a move to the game log.

Here are the main classes for chess:

Player: Player class represents one of the participants playing the game. It keeps track of which side (black or white) the player is playing.

Account: We'll have two types of accounts in the system: one will be a player, and the other will be an admin.

Game: This class controls the flow of a game. It keeps track of all the game moves, which player has the current turn, and the final result of the game.

Box: A box represents one block of the 8x8 grid and an optional piece.

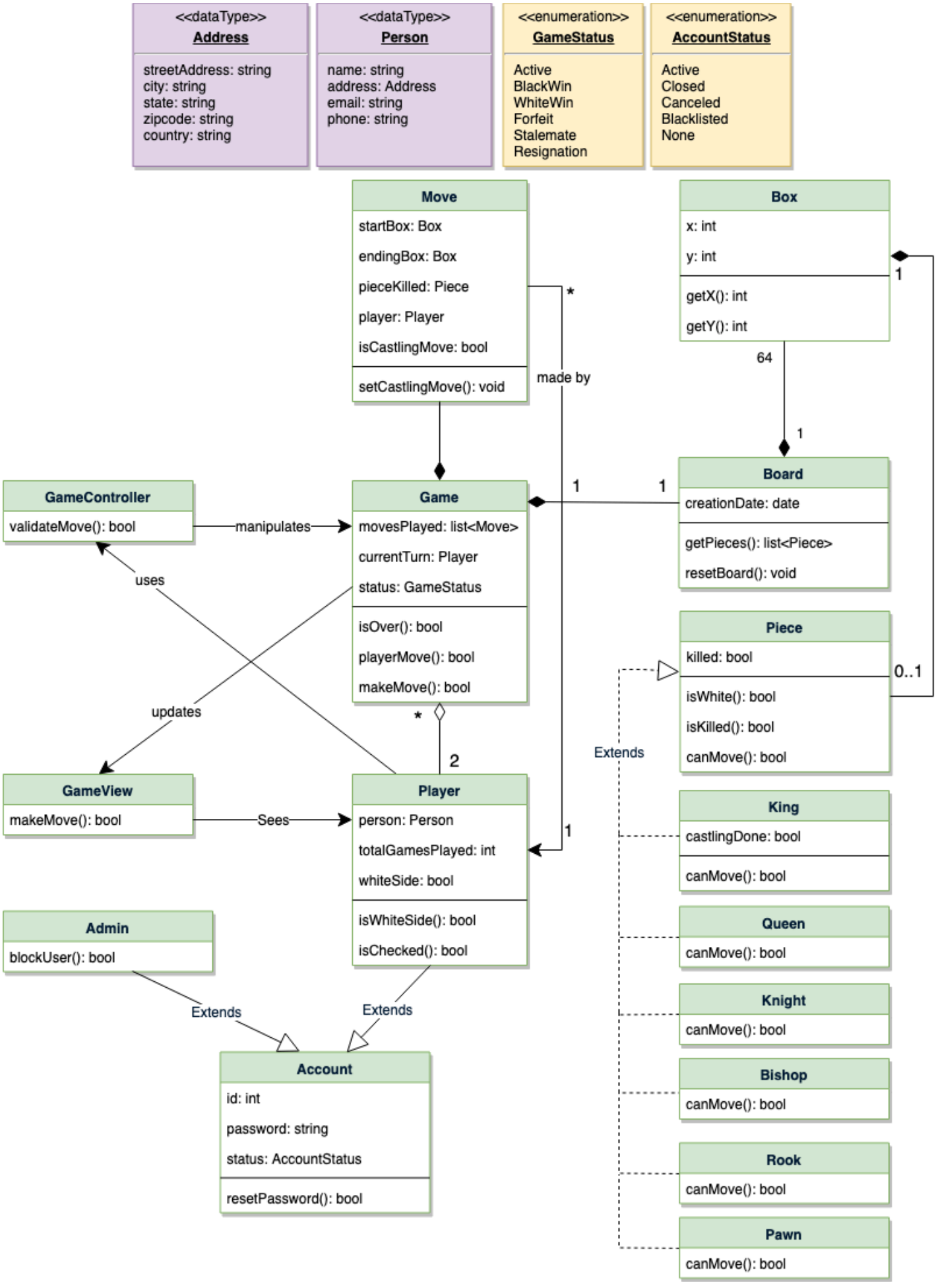
Board: Board is an 8x8 set of boxes containing all active chess pieces.

Piece: The basic building block of the system, every piece will be placed on a box. This class contains the color the piece represents and the status of the piece (that is, if the piece is currently in play or not). This would be an abstract class and all game pieces will extend it.

Move: Represents a game move, containing the starting and ending box. The Move class will also keep track of the player who made the move, if it is a castling move, or if the move resulted in the capture of a piece.

GameController: Player class uses GameController to make moves.

GameView: Game class updates the GameView to show changes to the players.


<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<dataType>>
Person
name: string
address: Address
email: string
phone: string
<<enumeration>>
GameStatus
Active
BlackWin
WhiteWin
Forfeit
Stalemate
Resignation
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
Move
startBox: Box
endingBox: Box
pieceKilled: Piece
player: Player
isCastlingMove: bool
setCastlingMove(): void
Box
x: int
y: int
getX(): int
getY(): int
made by
64
1
GameController
validateMove(): bool
manipulates
Game
movesPlayed: list<Move>
currentTurn: Player
status: GameStatus
isOver(): bool
playerMove(): bool
makeMove(): bool
Board
creationDate: date
getPieces(): list<Piece>
resetBoard(): void
uses
updates
Piece
killed: bool
isWhite(): bool
isKilled(): bool
canMove(): bool
0..1
Extends
GameView
makeMove(): bool
Sees
Player
person: Person
totalGamesPlayed: int
whiteSide: bool
isWhiteSide(): bool
isChecked(): bool
King
castlingDone: bool
canMove(): bool
Queen
canMove(): bool
Knight
canMove(): bool
Bishop
canMove(): bool
Rook
canMove(): bool
Pawn
canMove(): bool
Admin
blockUser(): bool
Extends
Extends
Account
id: int
password: string
status: AccountStatus
resetPassword(): bool

Design an Online Stock Brokerage System

An Online Stock Brokerage System facilitates its users the trade (i.e. buying and selling) of stocks online. It allows clients to keep track of and execute their transactions, and shows performance charts of the different stocks in their portfolios. It also provides security for their transactions and alerts them to predefined levels of changes in stocks, without the use of any middlemen.
The online stock brokerage system automates traditional stock trading using computers and the internet, making the transaction faster and cheaper. This system also gives speedier access to stock reports, current market trends, and real-time stock prices

System Requirements

We will focus on the following set of requirements while designing the online stock brokerage system:

1. Any user of our system should be able to buy and sell stocks.
2. Any user can have multiple watchlists containing multiple stock quotes.
3. Users should be able to place stock trade orders of the following types: 1) market, 2) limit, 3) stop loss and, 4) stop limit.
4. Users can have multiple 'lots' of a stock. This means that if a user has bought a stock multiple times, the system should be able to differentiate between different lots of the same stock.
5. The system should be able to generate reports for quarterly updates and yearly tax statements.
6. Users should be able to deposit and withdraw money either via check, wire, or electronic bank transfer.
7. The system should be able to send notifications whenever trade orders are executed.

We have three main Actors in our system:

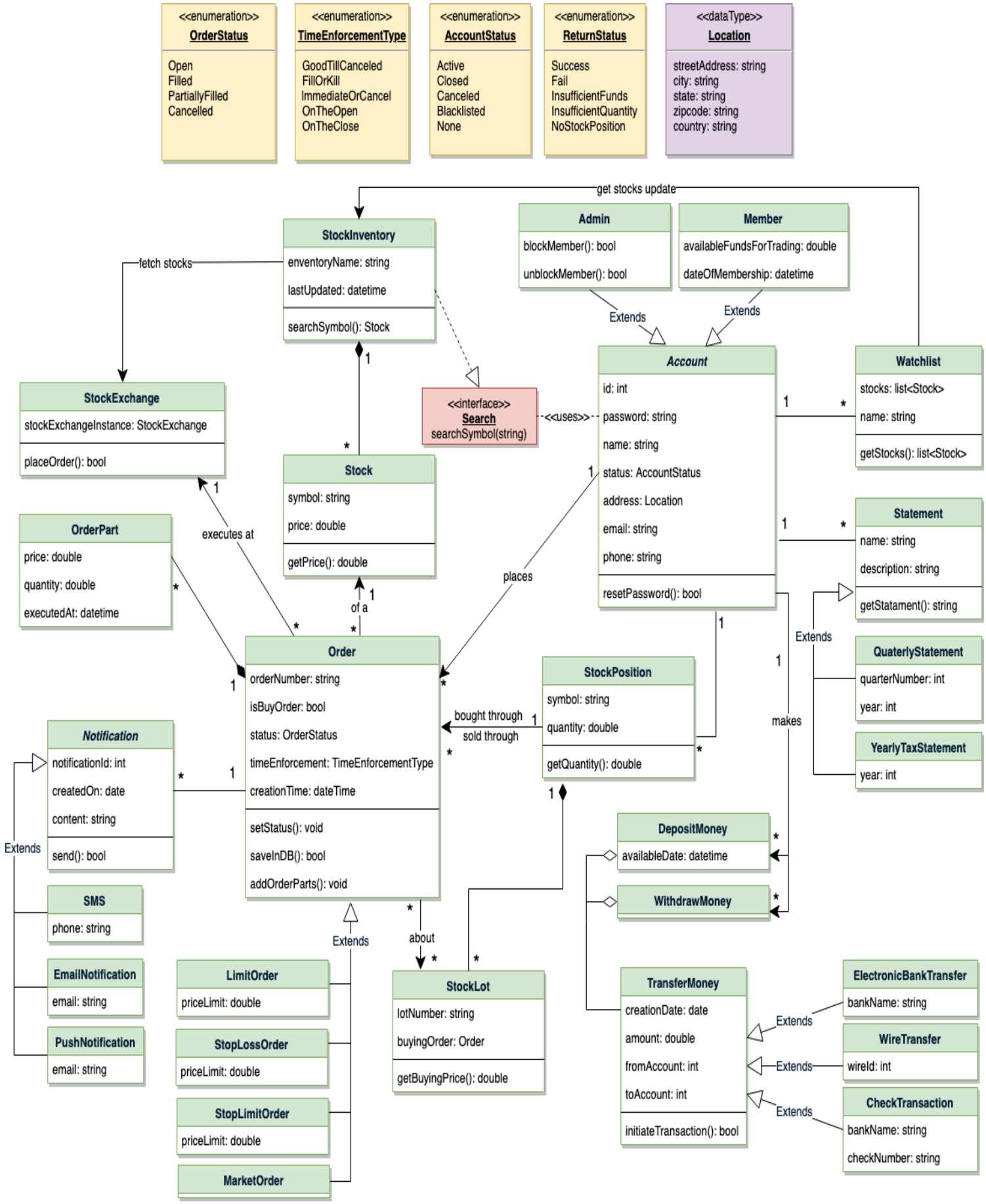
- **Admin**: Mainly responsible for administrative functions like blocking or unblocking members.
- **Member**: All members can search the stock inventory, as well as buy and sell stocks. Members can have multiple watchlists containing multiple stock quotes.
- **System**: Mainly responsible for sending notifications for stock orders and periodically fetching stock quotes from the stock exchange.

Here are the top use cases of the Stock Brokerage System:

- Register new account/Cancel membership: To add a new member or cancel the membership of an existing member.
- Add/Remove/Edit watchlist: To add, remove or modify a watchlist.
- Search stock inventory: To search for stocks by their symbols.
- Place order: To place a buy or sell order on the stock exchange.
- Cancel order: Cancel an already placed order.
- Deposit/Withdraw money: Members can deposit or withdraw money via check, wire or electronic bank transfer.

Here are the main classes of our Online Stock Brokerage System:

- **Account**: Consists of the member's name, address, e-mail, phone, total funds, funds that are available for trading, etc. We'll have two types of accounts in the system: one will be a general member, and the other will be an Admin. The Account class will also contain all the stocks the member is holding.
- **StockExchange**: The stockbroker system will fetch all stocks and their current prices from the stock exchange. StockExchange will be a singleton class encapsulating all interactions with the stock exchange. This class will also be used to place stock trading orders on the stock exchange.
- **Stock**: The basic building block of the system. Every stock will have a symbol, current trading price, etc.
- **StockInventory**: This class will fetch and maintain the latest stock prices from the StockExchange. All system components will read the most recent stock prices from this class.
- **Watchlist**: A watchlist will contain a list of stocks that the member wants to follow.
- **Order**: Members can place stock trading orders whenever they would like to sell or buy stock positions. **The system would support multiple types of orders:**
    - **Market Order**: Market order will enable users to buy or sell stocks immediately at the current market price.
    - **Limit Order**: Limit orders will allow a user to set a price at which they want to buy or sell a stock.
    - **Stop Loss Order**: An order to buy or sell once the stock reaches a certain price.
    - **Stop Limit Order**: The stop-limit order will be executed at a specified price or better after a given stop price has been reached. Once the stop price is reached, the stop-limit order becomes a limit order to buy or sell at the limit price or better.
- **OrderPart**: An order could be fulfilled in multiple parts. For example, a market order to buy 100 stocks could have one part containing 70 stocks at $10 and another part with 30 stocks at $10.05.
- **StockLot**: Any member can buy multiple lots of the same stock at different times. This class will represent these individual lots. For example, the user could have purchased 100 shares of AAPL yesterday and 50 more stocks of AAPL today. While selling, users will be able to select which lot they want to sell first.
- **StockPosition**: This class will contain all the stocks that the user holds.
- **Statement**: All members will have reports for quarterly updates and yearly tax statements.
- **DepositMoney** & **WithdrawMoney**: Members will be able to move money through check, wire or electronic bank transfers.
- **Notification**: Will take care of sending notifications to members.


<<enumeration>>
OrderStatus
Open
Filled
PartiallyFilled
Cancelled
<<enumeration>>
TimeEnforcementType
GoodTillCanceled
FillOrKill
ImmediateOrCancel
OnTheOpen
OnTheClose
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
<<enumeration>>
ReturnStatus
Success
Fail
InsufficientFunds
InsufficientQuantity
NoStockPosition
<<dataType>>
Location
streetAddress: string
city: string
state: string
zipcode: string
country: string
get stocks update
StockInventory
enventoryName: string
lastUpdated: datetime
searchSymbol(): Stock
fetch stocks
Admin
blockMember(): bool
unblockMember(): bool
Member
availableFundsForTrading: double
dateOfMembership: datetime
Extends
Extends
Account
id: int
password: string
name: string
status: AccountStatus
address: Location
email: string
phone: string
resetPassword(): bool
Watchlist
stocks: list<Stock>
name: string
getStocks(): list<Stock>
StockExchange
stockExchangeInstance: StockExchange
placeOrder(): bool
<<interface>>
Search
searchSymbol(string)
<<uses>>
Stock
symbol: string
price: double
getPrice(): double
executes at
places
of a
OrderPart
price: double
quantity: double
executedAt: datetime
Statement
name: string
description: string
getStatament(): string
Extends
QuaterlyStatement
quarterNumber: int
year: int
YearlyTaxStatement
year: int
Order
orderNumber: string
isBuyOrder: bool
status: OrderStatus
timeEnforcement: TimeEnforcementType
creationTime: dateTime
setStatus(): void
saveInDB(): bool
addOrderParts(): void
bought through
sold through
StockPosition
symbol: string
quantity: double
getQuantity(): double
makes
Notification
notificationId: int
createdOn: date
content: string
send(): bool
Extends
SMS
phone: string
EmailNotification
email: string
PushNotification
email: string
DepositMoney
availableDate: datetime
WithdrawMoney
Extends
about
LimitOrder
priceLimit: double
StopLossOrder
priceLimit: double
StopLimitOrder
priceLimit: double
MarketOrder
StockLot
lotNumber: string
buyingOrder: Order
getBuyingPrice(): double
TransferMoney
creationDate: date
amount: double
fromAccount: int
toAccount: int
initiateTransaction(): bool
Extends
Extends
Extends
ElectronicBankTransfer
bankName: string
WireTransfer
wireId: int
CheckTransaction
bankName: string
checkNumber: string

Design a Car Rental System

A Car Rental System is a software built to handle the renting of automobiles for a short period of time, generally ranging from a few hours to a few weeks. A car rental system often has numerous local branches (to allow its user to return a vehicle to a different location), and is primarily located near airports or busy city areas.

System Requirements

We will focus on the following set of requirements while designing our Car Rental System:

1. The system will support the renting of different automobiles like cars, trucks, SUVs, vans, and motorcycles.
2. Each vehicle should be added with a unique barcode and other details, including a parking stall number which helps to locate the vehicle.
3. The system should be able to retrieve information like which member took a particular vehicle or what vehicles have been rented out by a specific member.
4. The system should collect a late-fee for vehicles returned after the due date.
5. Members should be able to search the vehicle inventory and reserve any available vehicle.
6. The system should be able to send notifications whenever the reservation is approaching the pick-up date, as well as when the vehicle is nearing the due date or has not been returned within the due date.
7. The system will be able to read barcodes from vehicles.
8. Members should be able to cancel their reservations.
9. The system should maintain a vehicle log to track all events related to the vehicles.
10. Members can add rental insurance to their reservation.
11. Members can rent additional equipment, like navigation, child seat, ski rack, etc.
12. Members can add additional services to their reservation, such as roadside assistance, additional driver, wifi, etc.

We have four main Actors in our system:

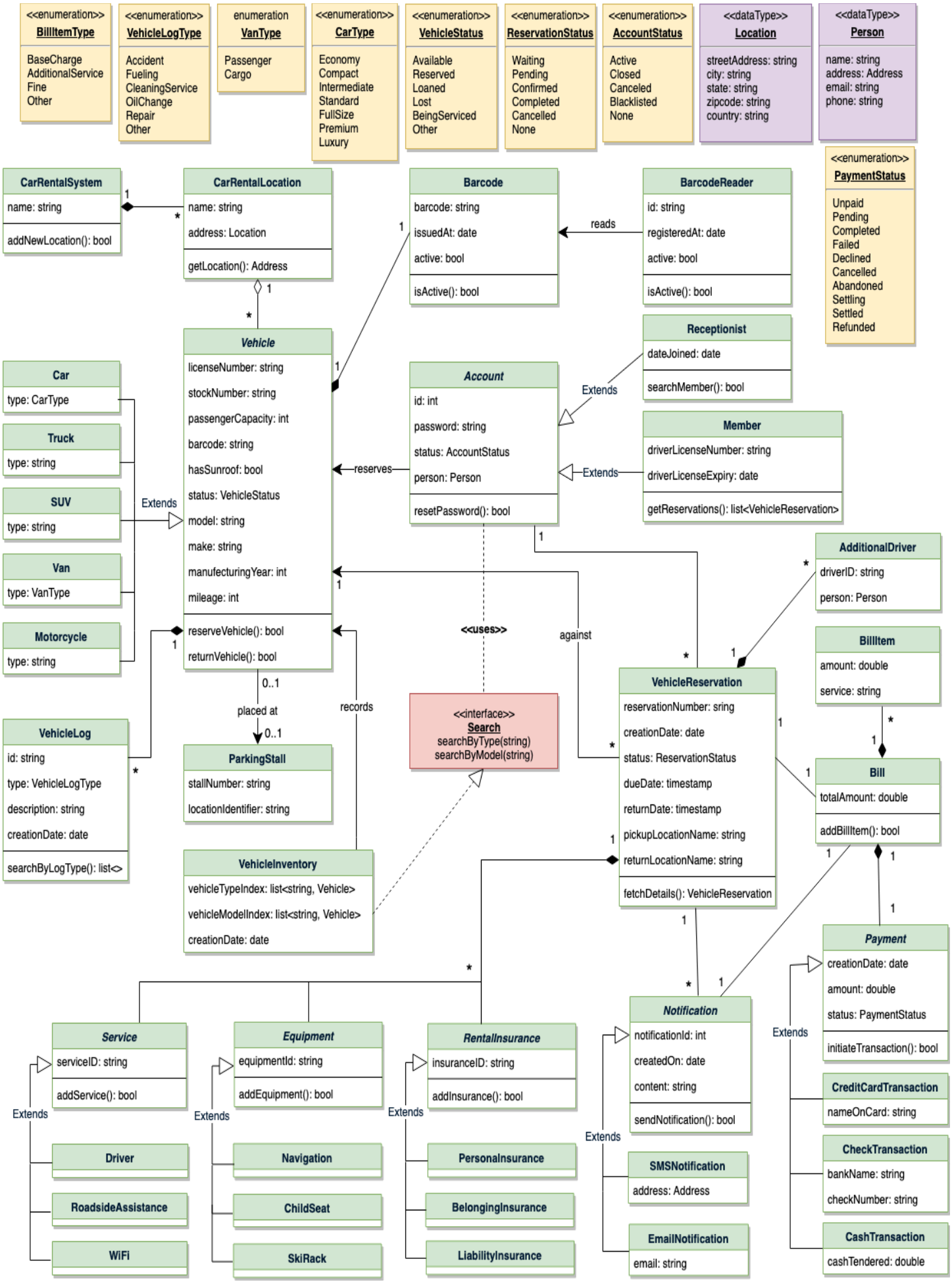
- **Receptionist**: Mainly responsible for adding and modifying vehicles and workers. Receptionists can also reserve vehicles.
- **Member**: All members can search the catalog, as well as reserve, pick-up, and return a vehicle.
- **System**: Mainly responsible for sending notifications about overdue vehicles, canceled reservations, etc.
- **Worker**: Mainly responsible for taking care of a returned vehicle and updating the vehicle log.

Here are the top use cases of the Car Rental System:

- Add/Remove/Edit vehicle: To add, remove or modify a vehicle.
- Search catalog: To search for vehicles by type and availability.
- Register new account/Cancel membership: To add a new member or cancel an existing membership.
- Reserve vehicle: To reserve a vehicle.
- Check-out vehicle: To rent a vehicle.
- Return a vehicle: To return a vehicle which was checked-out to a member.
- Add equipment: To add equipment to a reservation like navigation, child seat, etc.
- Update car log: To add or update a car log entry, such as refueling, cleaning, damage, etc.

Here are the main classes of our Car Rental System:

- **CarRentalSystem**: The main part of the organization for which this software has been designed.
- **CarRentalLocation**: The car rental system will have multiple locations, each location will have attributes like 'Name' to distinguish it from any other locations and 'Address' which defines the address of the rental location.
- **Vehicle**: The basic building block of the system. Every vehicle will have a barcode, license plate number, passenger capacity, model, make, mileage, etc. Vehicles can be of multiple types, like car, truck, SUV, etc.
- **Account**: Mainly, we will have two types of accounts in the system, one will be a general member and the other will be a receptionist. Another account can be of the worker taking care of the returned vehicle.
- **VehicleReservation**: This class will be responsible for managing reservations for a vehicle.
- **Notification**: Will take care of sending notifications to members.
- **VehicleLog**: To keep track of all the events related to a vehicle.
- **RentalInsurance**: Stores details about the various rental insurances that members can add to their reservation.
- **Equipment**: Stores details about the various types of equipment that members can add to their reservation.
- **Service**: Stores details about the various types of service that members can add to their reservation, such as additional drivers, roadside assistance, etc.
- **Bill**: Contains different bill-items for every charge for the reservation.


<<enumeration>>
BillItemType
BaseCharge
AdditionalService
Fine
Other
<<enumeration>>
VehicleLogType
Accident
Fueling
CleaningService
OilChange
Repair
Other
enumeration
VanType
Passenger
Cargo
<<enumeration>>
CarType
Economy
Compact
Intermediate
Standard
FullSize
Premium
Luxury
<<enumeration>>
VehicleStatus
Available
Reserved
Loaned
Lost
BeingServiced
Other
<<enumeration>>
ReservationStatus
Waiting
Pending
Confirmed
Completed
Cancelled
None
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
<<dataType>>
Location
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<dataType>>
Person
name: string
address: Address
email: string
phone: string
<<enumeration>>
PaymentStatus
Unpaid
Pending
Completed
Failed
Declined
Cancelled
Abandoned
Settling
Settled
Refunded
CarRentalSystem
name: string
addNewLocation(): bool
CarRentalLocation
name: string
address: Location
getLocation(): Address
Barcode
barcode: string
issuedAt: date
active: bool
isActive(): bool
reads
BarcodeReader
id: string
registeredAt: date
active: bool
isActive(): bool
Vehicle
licenseNumber: string
stockNumber: string
passengerCapacity: int
barcode: string
hasSunroof: bool
status: VehicleStatus
model: string
make: string
manufacturingYear: int
mileage: int
reserveVehicle(): bool
returnVehicle(): bool
Account
id: int
password: string
status: AccountStatus
person: Person
resetPassword(): bool
reserves
Receptionist
dateJoined: date
searchMember(): bool
Extends
Member
driverLicenseNumber: string
driverLicenseExpiry: date
getReservations(): list<VehicleReservation>
Extends
Car
type: CarType
Truck
type: string
SUV
type: string
Van
type: VanType
Motorcycle
type: string
Extends
<<uses>>
against
AdditionalDriver
driverID: string
person: Person
BillItem
amount: double
service: string
VehicleReservation
reservationNumber: sring
creationDate: date
status: ReservationStatus
dueDate: timestamp
returnDate: timestamp
pickupLocationName: string
returnLocationName: string
fetchDetails(): VehicleReservation
Bill
totalAmount: double
addBillItem(): bool
placed at
0..1
0..1
records
<<interface>>
Search
searchByType(string)
searchByModel(string)
ParkingStall
stallNumber: string
locationIdentifier: string
VehicleLog
id: string
type: VehicleLogType
description: string
creationDate: date
searchByLogType(): list<>
VehicleInventory
vehicleTypeIndex: list<string, Vehicle>
vehicleModelIndex: list<string, Vehicle>
creationDate: date
Service
serviceID: string
addService(): bool
Equipment
equipmentId: string
addEquipment(): bool
RentalInsurance
insuranceID: string
addInsurance(): bool
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool
Payment
creationDate: date
amount: double
status: PaymentStatus
initiateTransaction(): bool
Extends
Extends
Extends
Extends
Extends
Driver
RoadsideAssistance
WiFi
Navigation
ChildSeat
SkiRack
PersonalInsurance
BelongingInsurance
LiabilityInsurance
SMSNotification
address: Address
EmailNotification
email: string
CreditCardTransaction
nameOnCard: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double

Design LinkedIn

A LinkedIn member’s profile page, which emphasizes their skills, employment history, and education, has professional network news feeds with customizable modules.
LinkedIn is very similar to Facebook in terms of its layout and design. These features are more specialized because they cater to professionals, but in general, if you know how to use Facebook or any other similar social network, LinkedIn is somewhat comparable.LinkedIn is a social network for professionals. The main goal of the site is to enable its members to connect with people they know and trust professionally, as well as to find new opportunities to grow their careers.
A LinkedIn member’s profile page, which emphasizes their skills, employment history, and education, has professional network news feeds with customizable modules.
LinkedIn is very similar to Facebook in terms of its layout and design. These features are more specialized because they cater to professionals, but in general, if you know how to use Facebook or any other similar social network, LinkedIn is somewhat comparable.

System Requirements

We will focus on the following set of requirements while designing LinkedIn:

1. Each member should be able to add information about their basic profile, experiences, education, skills, and accomplishments.
2. Any user of our system should be able to search for other members or companies by their name.
3. Members should be able to send or accept connection requests from other members.
4. Any member will be able to request a recommendation from other members.
5. The system should be able to show basic stats about a profile, like the number of profile views, the total number of connections, and the total number of search appearances of the profile.
6. Members should be able to create new posts to share with their connections.
7. Members should be able to add comments to posts, as well as like or share a post or comment.
8. Any member should be able to send messages to other members.
9. The system should send a notification to a member whenever there is a new message, connection invitation or a comment on their post.
10. Members will be able to create a page for a Company and add job postings.
11. Members should be able to create groups and join any group they like.
12. Members should be able to follow other members or companies.

We have three main Actors in our system:

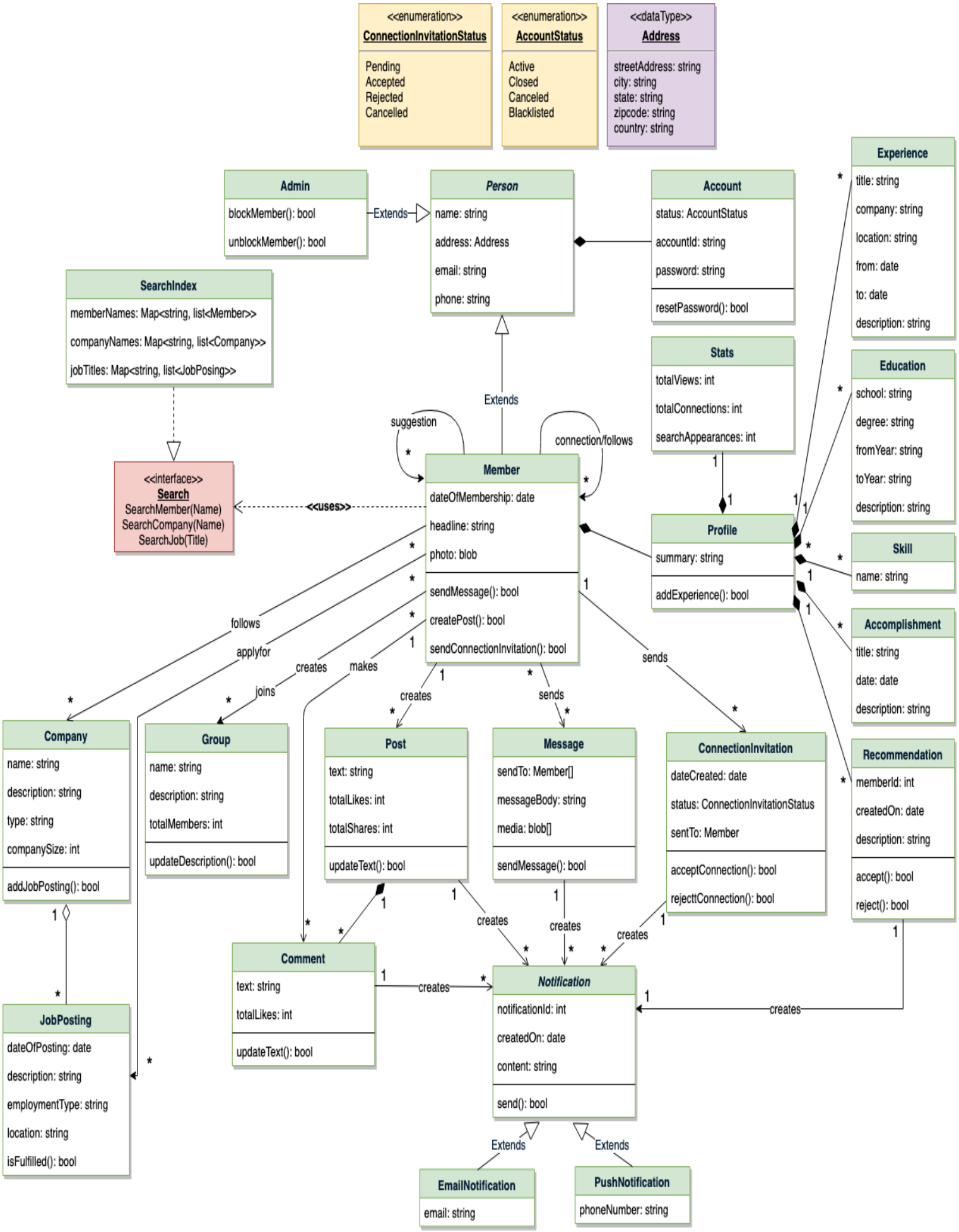
- Member: All members can search for other members, companies or jobs, as well as send requests for connection, create posts, etc.
- Admin: Mainly responsible for admin functions such as blocking and unblocking a member, etc.
- System: Mainly responsible for sending notifications for new messages, connections invites, etc.

Here are the top use cases of our system:

- Add/update profile: Any member should be able to create their profile to reflect their experiences, education, skills, and accomplishments.
- Search: Members can search other members, companies or jobs. Members can send a connection request to other members.
- Follow or Unfollow member or company: Any member can follow or unfollow any other member or a company.
- Send message: Any member can send a message to any of their connections.
- Create post: Any member can create a post to share with their connections, as well as like other posts or add comments to any post.
- Send notifications: The system will be able to send notifications for new messages, connection invites, etc.

Here are the main classes of the LinkedIn system:

- **Member**: This will be the main component of our system. Each member will have a profile which includes their Experiences, Education, Skills, Accomplishments, and Recommendations. Members will be connected to other members and they can follow companies and members. Members will also have suggestions to make connections with other members.
- **Search**: Our system will support searching for other members and companies by their names, and jobs by their titles.
- **Message**: Members can send messages to other members with text and media.
- **Post**: Members can create posts containing text and media.
- **Comment**: Members can add comments to posts as well as like them.
- **Group**: Members can create and join groups.
- **Company**: Company will store all the information about a company's page.
- **JobPosting**: Companies can create a job posting. This class will handle all information about a job.
- **Notification**: Will take care of sending notifications to members.


<<enumeration>>
ConnectionInvitationStatus
Pending
Accepted
Rejected
Cancelled
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
Admin
blockMember(): bool
unblockMember(): bool
Extends
Person
name: string
address: Address
email: string
phone: string
Account
status: AccountStatus
accountId: string
password: string
resetPassword(): bool
Experience
title: string
company: string
location: string
from: date
to: date
description: string
SearchIndex
memberNames: Map<string, list<Member>>
companyNames: Map<string, list<Company>>
jobTitles: Map<string, list<JobPosing>>
Extends
Stats
totalViews: int
totalConnections: int
searchAppearances: int
Education
school: string
degree: string
fromYear: string
toYear: string
description: string
suggestion
connection/follows
<<interface>>
Search
SearchMember(Name)
SearchCompany(Name)
SearchJob(Title)
<<uses>>
Member
dateOfMembership: date
headline: string
photo: blob
sendMessage(): bool
createPost(): bool
sendConnectionInvitation(): bool
Profile
summary: string
addExperience(): bool
Skill
name: string
follows
applyfor
creates
makes
joins
creates
sends
sends
Accomplishment
title: string
date: date
description: string
Company
name: string
description: string
type: string
companySize: int
addJobPosting(): bool
Group
name: string
description: string
totalMembers: int
updateDescription(): bool
Post
text: string
totalLikes: int
totalShares: int
updateText(): bool
Message
sendTo: Member[]
messageBody: string
media: blob[]
sendMessage(): bool
ConnectionInvitation
dateCreated: date
status: ConnectionInvitationStatus
sentTo: Member
acceptConnection(): bool
rejecttConnection(): bool
Recommendation
memberId: int
createdOn: date
description: string
accept(): bool
reject(): bool
creates
creates
creates
creates
Comment
text: string
totalLikes: int
updateText(): bool
creates
Notification
notificationId: int
createdOn: date
content: string
send(): bool
creates
JobPosting
dateOfPosting: date
description: string
employmentType: string
location: string
isFulfilled(): bool
Extends
Extends
EmailNotification
email: string
PushNotification
phoneNumber: string

Design Cricinfo

Cricinfo is a sports news website exclusively for the game of cricket. The site features live coverage of cricket matches containing ball-by-ball commentary and a database for all the historic matches. The site also provides news and articles about cricket.

System Requirements

We will focus on the following set of requirements while designing Cricinfo:

1. The system should keep track of all cricket-playing teams and their matches.
2. The system should show live ball-by-ball commentary of cricket matches.
3. All international cricket rules should be followed.
4. Any team playing a tournament will announce a squad (a set of players) for the tournament.
5. For each match, both teams will announce their playing-eleven from the tournament squad.
6. The system should be able to record stats about players, matches, and tournaments.
7. The system should be able to answer global stats queries like, “Who is the highest wicket taker of all time?”, “Who has scored maximum numbers of 100s in test matches?”, etc.
8. The system should keep track of all ODI, Test and T20 matches.

We have two main Actors in our system:

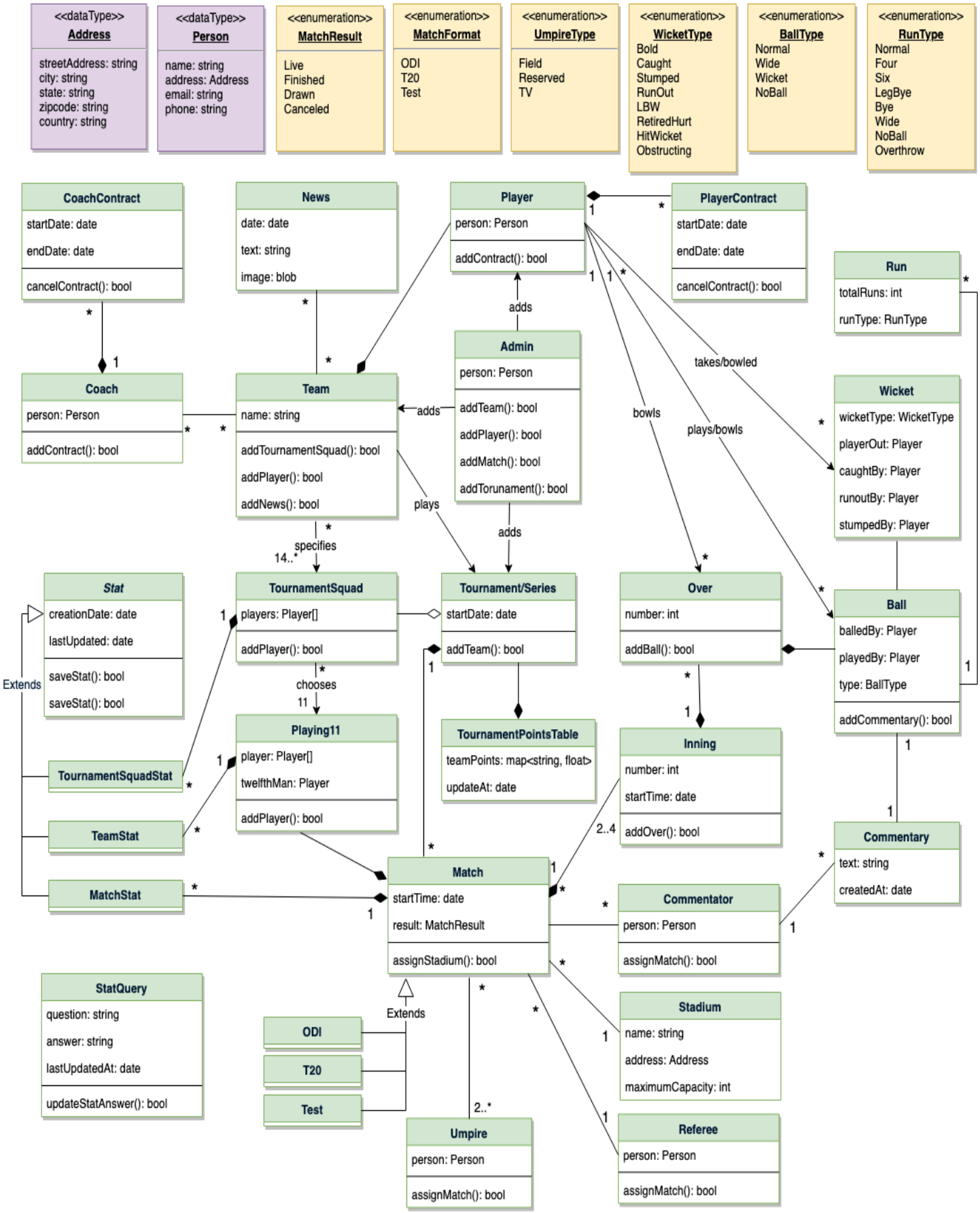
- **Admin**: An Admin will be able to add/modify players, teams, tournaments, and matches, and will also record ball-by-ball details of each match.
- **Commentator**: Commentators will be responsible for adding ball-by-ball commentary for matches.

Here are the top use cases of our system:

- Add/modify teams and players: An Admin will add players to teams and keeps up-to-date information about them in the system.
- Add tournaments and matches: Admins will add tournaments and matches in the system.
- Add ball: Admins will record ball-by-ball details of a match.
- Add stadium, umpire, and referee: The system will keep track of stadiums as well as of the umpires and referees managing the matches.
- Add/update stats: Admins will add stats about matches and tournaments. The system will generate certain stats.
- Add commentary: Add ball-by-ball commentary of matches.

Here are the main classes of the Cricinfo system:

- **Player**: Keeps a record of a cricket player, their basic profile and contracts.
- **Team**: This class manages cricket teams.
- **Tournament**: Manages cricket tournaments and keeps track of the points table for all playing teams.
- **TournamentSquad**: Each team playing a tournament will announce a set of players who will be playing the tournament. TournamentSquad will encapsulate that.
- **Playing11**: Each team playing a match will select 11 players from their announced tournaments squad.
- **Match**: Encapsulates all information of a cricket match. Our system will support three match types: 1) ODI, 2) T20, and 3) Test
- **Innings**: Records all innings of a match.
- **Over**: Records details about an Over.
- **Ball**: Records every detail of a ball, such as the number of runs scored, if it was a wicket-taking ball, etc.
- **Run**: Records the number and type of runs scored on a ball. The different run types are: Wide, LegBy, Four, Six, etc.
- **Commentator** and **Commentary**: The commentator adds ball-by-ball commentary.
- **Umpire** and **Referee**: These classes will store details about umpires and referees, respectively.
- **Stat**: Our system will keep track of the stats for every player, match and tournament.
- **StatQuery**: This class will encapsulate general stat queries and their answers, like “Who has scored the maximum number of 100s in ODIs?” or, “Which bowler has taken the most wickets in test matches?”, etc.


<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<dataType>>
Person
name: string
address: Address
email: string
phone: string
<<enumeration>>
MatchResult
Live
Finished
Drawn
Canceled
<<enumeration>>
MatchFormat
ODI
T20
Test
<<enumeration>>
UmpireType
Field
Reserved
TV
<<enumeration>>
WicketType
Bold
Caught
Stumped
RunOut
LBW
RetiredHurt
HitWicket
Obstructing
<<enumeration>>
BallType
Normal
Wide
Wicket
NoBall
<<enumeration>>
RunType
Normal
Four
Six
LegBye
Bye
Wide
NoBall
Overthrow
CoachContract
startDate: date
endDate: date
cancelContract(): bool
News
date: date
text: string
image: blob
Player
person: Person
addContract(): bool
PlayerContract
startDate: date
endDate: date
cancelContract(): bool
Run
totalRuns: int
runType: RunType
adds
Admin
person: Person
addTeam(): bool
addPlayer(): bool
addMatch(): bool
addTorunament(): bool
takes/bowled
Coach
person: Person
addContract(): bool
Team
name: string
addTournamentSquad(): bool
addPlayer(): bool
addNews(): bool
adds
bowls
plays/bowls
Wicket
wicketType: WicketType
playerOut: Player
caughtBy: Player
runoutBy: Player
stumpedBy: Player
plays
specifies
14..*
adds
Stat
creationDate: date
lastUpdated: date
saveStat(): bool
saveStat(): bool
TournamentSquad
players: Player[]
addPlayer(): bool
Tournament/Series
startDate: date
addTeam(): bool
Over
number: int
addBall(): bool
Ball
balledBy: Player
playedBy: Player
type: BallType
addCommentary(): bool
Extends
chooses
11
Playing11
player: Player[]
twelfthMan: Player
addPlayer(): bool
TournamentPointsTable
teamPoints: map<string, float>
updateAt: date
Inning
number: int
startTime: date
addOver(): bool
2..4
TournamentSquadStat
TeamStat
MatchStat
Match
startTime: date
result: MatchResult
assignStadium(): bool
Commentary
text: string
createdAt: date
Commentator
person: Person
assignMatch(): bool
StatQuery
question: string
answer: string
lastUpdatedAt: date
updateStatAnswer(): bool
Extends
ODI
T20
Test
Stadium
name: string
address: Address
maximumCapacity: int
2..*
Umpire
person: Person
assignMatch(): bool
Referee
person: Person
assignMatch(): bool

Design Facebook - a social network

System Requirements

We will focus on the following set of requirements while designing Facebook:

1. Each member should be able to add information about their basic profile, work experience, education, etc.
2. Any user of our system should be able to search other members, groups or pages by their name.
3. Members should be able to send and accept/reject friend requests from other members.
4. Members should be able to follow other members without becoming their friend.
5. Members should be able to create groups and pages, as well as join already created groups, and follow pages.
6. Members should be able to create new posts to share with their friends.
7. Members should be able to add comments to posts, as well as like or share a post or comment.
8. Members should be able to create privacy lists containing their friends. Members can link any post with a privacy list to make the post visible only to the members of that list.
9. Any member should be able to send messages to other members.
10. Any member should be able to add a recommendation for any page.
11. The system should send a notification to a member whenever there is a new message or friend request or comment on their post.
12. Members should be able to search through posts for a word.

Extended Requirement: Write a function to find a connection suggestion for a member.

We have three main Actors in our system:

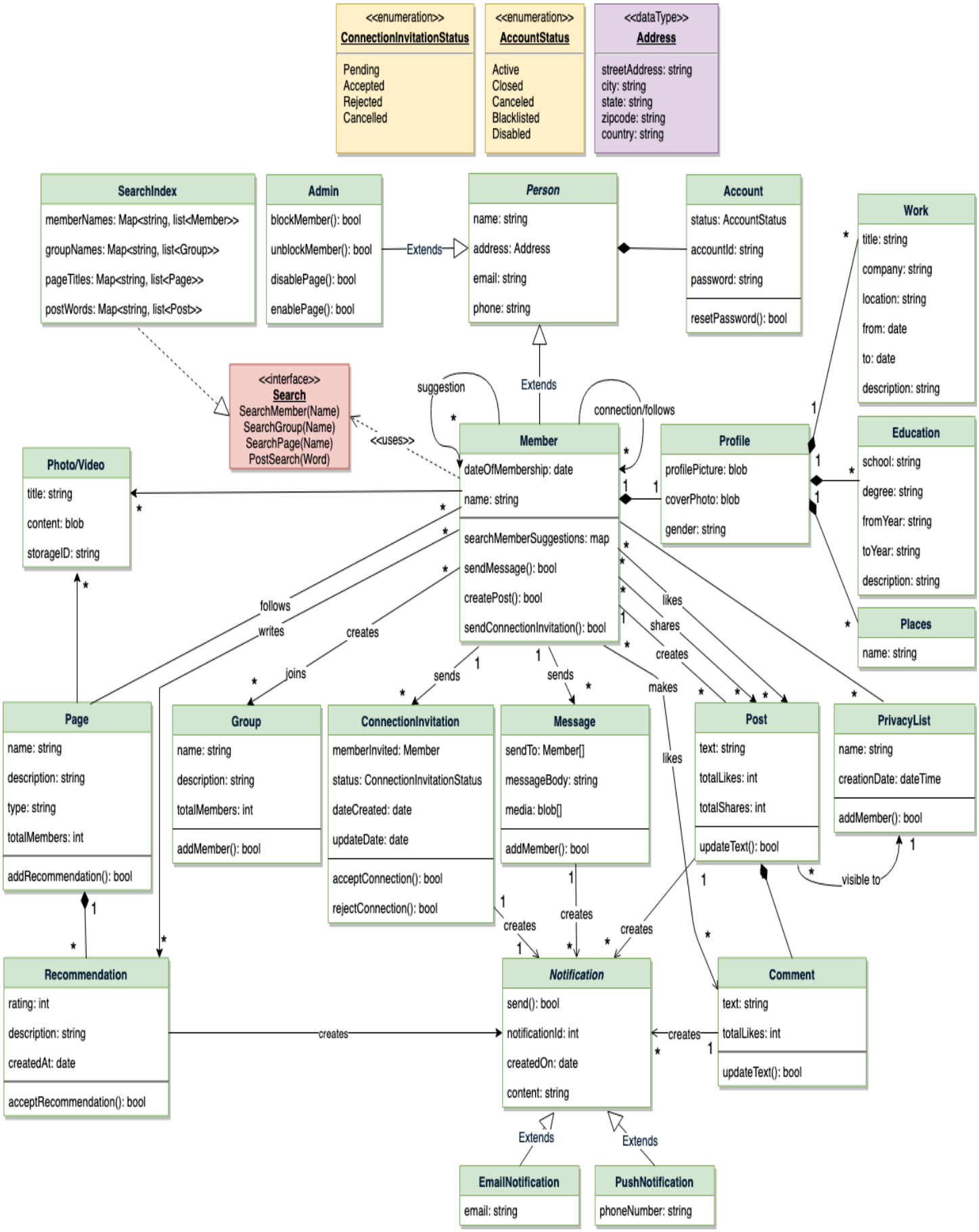
- **Member**: All members can search for other members, groups, pages, or posts, as well as send friend requests, create posts, etc.
- **Admin**: Mainly responsible for admin functions like blocking and unblocking a member, etc.
- **System**: Mainly responsible for sending notifications for new messages, friend requests, etc.

Here are the top use cases of our system:

- Add/update profile: Any member should be able to create their profile to reflect their work experiences, education, etc.
- Search: Members can search for other members, groups or pages. Members can send a friend request to other members.
- Follow or Unfollow a member or a page: Any member can follow or unfollow any other member or page.
- Send message: Any member can send a message to any of their friends.
- Create post: Any member can create a post to share with their friends, as well as like or add comments to any post visible to them.
- Send notification: The system will be able to send notifications for new messages, friend requests, etc.

Here are the main classes of the Facebook system:

- **Member**: This will be the main component of our system. Each member will have a profile which includes their Work Experiences, Education, etc. Members will be connected to other members and they can follow other members and pages. Members will also have suggestions to send friend requests to other members.
- **Search**: Our system will support searching for other members, groups and pages by their names, and through posts for any word.
- **Message**: Members can send messages to other members with text, photos, and videos.
- **Post**: Members can create posts containing text and media, as well as like and share a post.
- **Comment**: Members can add comments to posts as well as like any comment.
- **Group**: Members can create and join groups.
- **PrivacyList**: Members can create privacy lists containing their friends. Members can link any post with a privacy list, to make the post visible only to the members of that list.
- **Page**: Members can create pages that other members can follow, and share messages there.
- **Notification**: This class will take care of sending notifications to members. The system will be able to send a push notification or an email.


<<enumeration>>
ConnectionInvitationStatus
Pending
Accepted
Rejected
Cancelled
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
Disabled
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
SearchIndex
memberNames: Map<string, list<Member>>
groupNames: Map<string, list<Group>>
pageTitles: Map<string, list<Page>>
postWords: Map<string, list<Post>>
Admin
blockMember(): bool
unblockMember(): bool
disablePage(): bool
enablePage(): bool
Extends
Person
name: string
address: Address
email: string
phone: string
Account
status: AccountStatus
accountId: string
password: string
resetPassword(): bool
Work
title: string
company: string
location: string
from: date
to: date
description: string
<<interface>>
Search
SearchMember(Name)
SearchGroup(Name)
SearchPage(Name)
PostSearch(Word)
<<uses>>
suggestion
Extends
connection/follows
Member
dateOfMembership: date
name: string
searchMemberSuggestions: map
sendMessage(): bool
createPost(): bool
sendConnectionInvitation(): bool
Profile
profilePicture: blob
coverPhoto: blob
gender: string
Education
school: string
degree: string
fromYear: string
toYear: string
description: string
Photo/Video
title: string
content: blob
storageID: string
Places
name: string
follows
writes
creates
likes
shares
joins
sends
sends
makes
Page
name: string
description: string
type: string
totalMembers: int
addRecommendation(): bool
Group
name: string
description: string
totalMembers: int
addMember(): bool
ConnectionInvitation
memberInvited: Member
status: ConnectionInvitationStatus
dateCreated: date
updateDate: date
acceptConnection(): bool
rejectConnection(): bool
Message
sendTo: Member[]
messageBody: string
media: blob[]
addMember(): bool
Post
text: string
totalLikes: int
totalShares: int
updateText(): bool
PrivacyList
name: string
creationDate: dateTime
addMember(): bool
visible to
likes
creates
creates
creates
creates
Recommendation
rating: int
description: string
createdAt: date
acceptRecommendation(): bool
creates
Notification
send(): bool
notificationId: int
createdOn: date
content: string
Comment
text: string
totalLikes: int
updateText(): bool
creates
Extends
Extends
EmailNotification
email: string
PushNotification
phoneNumber: string