Object-oriented design has always been a critical part of the software engineering interview process. Most of the engineers struggle with the object-oriented design interviews (OODI), partly because of their lack of experience in laying down the design of a complex system, and partly because of the unstructured nature of OODI. Even engineers, who have some sort of experience in building such systems are not comfortable with these interviews. It is mainly because of the open-ended nature of the design problems that don't have a standard answer. This course is a complete guide to master the OODI. It is designed by the hiring managers of Google, Facebook, Microsoft, and Amazon. It not only has a set of carefully handpicked case studies, which have been repeatedly asked at the top tech companies, but also provides a thorough experience to handle different object-oriented design scenarios.

Grokking the Object Oriented Design Interview

By: Design Gurus

Object Oriented Design Case Studies

Contents

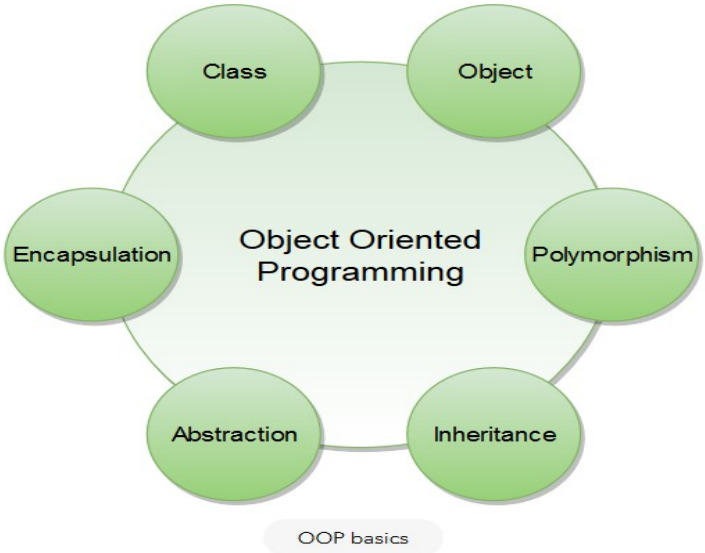
Object-Oriented Design and UML

Object-Oriented Basics

Object-oriented programming (OOP) is a style of programming that focuses on using objects to design and build applications. Contrary to procedure-oriented programming where we design our programs as blocks of statements to manipulate data, OOP organizes the program to combine data and functionality and wrap it inside something called an Object.

If you have never used an object-oriented programming language before, you will need to learn a few basic concepts before you can begin writing any code. This chapter will introduce some of the basic concepts of OOP:

- **Objects:** Objects represent a real-world entity and the basic building block of OOP. For example, an Online Shopping System will have objects such as shopping cart, customer, product item, etc.
- **Class:** Class is the prototype or the blueprint of an object. It is a template definition of the attributes and methods of an object. For example, in the Online Shopping System, the Customer object will have attributes like shipping address, credit card, etc., and methods for placing an order, canceling an order, etc.



OOP basics

The four principles of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism.

- **Encapsulation:** Encapsulation is the mechanism of binding the data together and hiding it from the outside world. Encapsulation is achieved when each object keeps its state private, such that, other objects don't have direct access to its state. Instead, they can access this state only through a set of public functions.
- **Abstraction:** Abstraction can be thought of as a natural extension of encapsulation. It means hiding all but the relevant data about an object in order to reduce the complexity of the system. In a large system, objects talk to each other, which makes it difficult to maintain a large code base, abstraction helps by hiding internal implementation details of objects and only reveal those operations that are relevant to other objects.
- **Inheritance:** Inheritance is the mechanism of creating new classes from existing ones.
- **Polymorphism:** Polymorphism (from Greek, meaning "many forms") is the ability of an object to take different forms and thus to respond to the same message in different ways depending upon the context. Take the example of a chess game; a piece has many forms like bishop, rock or knight - all these pieces will respond differently to the 'move' message.

OO Analysis and Design

OO Analysis and Design is a structured method for analyzing and designing a system by applying the object-orientated concept. This design process consists of an investigation of objects constituting the system. It starts by identifying the objects of the system and then figuring out the interactions between different objects.

The process of OO analysis and design can be described as:

1. Identifying the objects of a system.
2. Defining relationships between objects.
3. Establish an interface of each object.
4. Making a design, which can be converted to executables using OO languages.

We need a standard method/tool to document all this information, for this purpose we use UML. UML can be described 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 becomes essential to understand OO concept.

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

What is UML?

UML stands for Unified Modeling Language and is used to model Object-Oriented Analysis of a software system. UML is a way of visualizing and documenting a software system using a collection of diagrams to help engineers, businesspeople and system architects to understand the behavior and structure of the system being designed.

Benefits of using UML:

1. To develop a quick understanding of software system.
2. UML modeling helps to break a complex system into discrete pieces that can be understood easily.
3. Use UML's graphical notations to communicate design decisions.
4. UML is independent of any specific platform or language or technology, so it is easier to abstract out concepts.
5. Handover the system to a new team becomes easier.

Types of UML Diagrams: The current UML standards call for 14 different kinds of diagrams. These diagrams are organized into two distinct groups: structural diagrams and behavioral or interaction diagrams. As the name suggests, some UML diagrams try to analyze and depict the structure of a system or process, whereas other describe the behavior of the system, its actors, and its building components. The different types are broken down as follows:

Structural UML diagrams

- Class diagram
- Object diagram
- Package diagram
- Component diagram
- Composite structure diagram
- Deployment diagram
- Profile Diagram

Behavioral UML diagrams

- Use case diagram

- Activity diagram
- Sequence diagram
- State diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram

We will be focusing on the following UML diagrams in this course:

- **Use Case Diagram:** This diagram is used to describe a set of user scenarios. It 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:** Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.

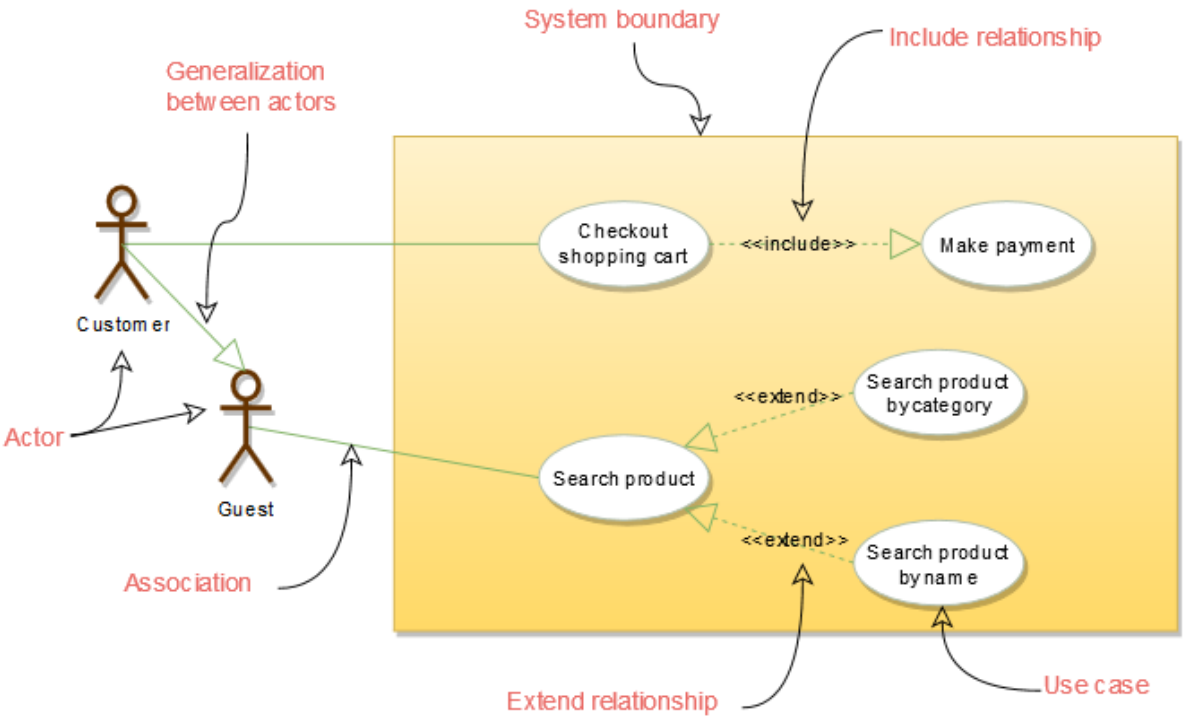
Use Case Diagrams

Use case diagrams describe a set of actions (called use cases) that a system should or can perform in collaboration with one or more external users of the system (called actors). Each use case should provide some observable and valuable result to the actors.

1. Use Case Diagram describes the high-level functional behavior of the system.
2. It answers what system does from the user point of view.
3. Use case answers 'What will the system do?' and at the time tells us 'What will the system NOT do?'.

A use case illustrates a unit of functionality provided by the system. The primary purpose of the use-case diagram is to help the development teams visualize the functional requirements of a system, including the relationship of "actors" to the essential processes, as well as the relationships among different use cases.

To show a use case on a use-case diagram, we draw an oval in the middle of the diagram and put the name of the use case in the center of the oval. To draw an actor (indicating a system user) on a use-case diagram, we draw a stick person to the left or right of the diagram.



Sample use-case diagram for online shopping system

Sample use-case diagram for online shopping system

The following are the different components of the use case diagrams:

- **System boundary:** A system boundary defines the scope and limits of the system. The system boundary is shown as a rectangle that spans all use cases of the system.
- **Actors:** An actor is one of the entities who perform specific actions. These roles are the actual business roles of the users in a given system. An actor interacts with a use case of the system. For example, for a banking system, the customer is one of the actors.
- **Use Case:** A use case represents a business functionality that is distinct. The use case should list the discrete business functionality that is specified in the problem statement. Every business functionality is a potential use case.
- **Include:** Include relationship represents an invocation of one use case by the other. From the coding perspective, it is like one function been called by the other function.
- **Extend:** This relationship signifies that the extending use case will work exactly like the base use case only that some new steps will be inserted in the extended use case.

Class Diagram

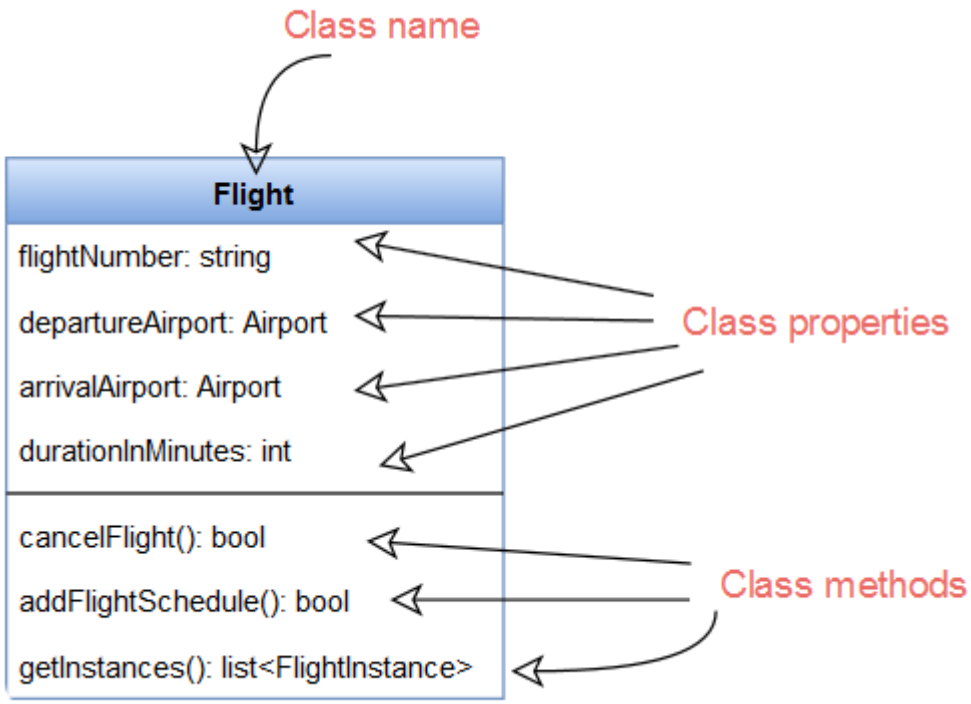
Class diagram is the backbone of object-oriented modeling; it shows how the different entities (people, things, and data) relate to each other; in other words, it shows the static structures of the system.

A class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams, which can be mapped directly to object-oriented languages.

The purpose of the class diagram can be summarized as:

1. Analysis and design of the static view of an application.
2. Describe the responsibilities of a system.
3. Provide a base for component and deployment diagrams.
4. Forward and reverse engineering.

A class is depicted in the class diagram as a rectangle with three horizontal sections, as shown in the below figure. The upper section shows the class's name (Flight), the middle section contains the properties of the class, and the lower section contains the class's operations (or "methods").



These are different types of relationship between classes:

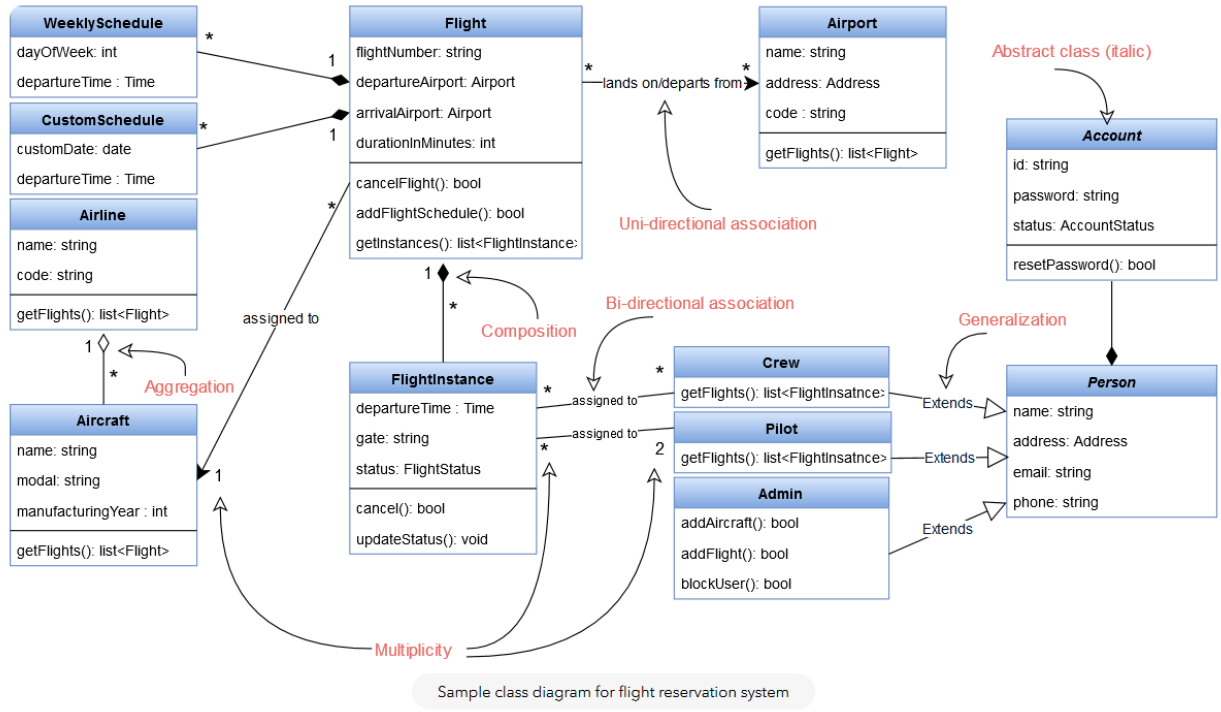
Association: If two classes in a model need to communicate with each other, there must be a link between them, and an association can represent that. Association can be represented by a line between these classes with an arrow indicating the navigation direction.

- 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.

- In contrast, in a uni-directional 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 uni-directional association

What Is 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 the 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 like "0…*" which means "zero to many" or like "2…4" which means "two to four".

We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association. The diagram below, demonstrates that a FlightInstance has exactly two Pilots but a Pilot can have many FlightInstances.



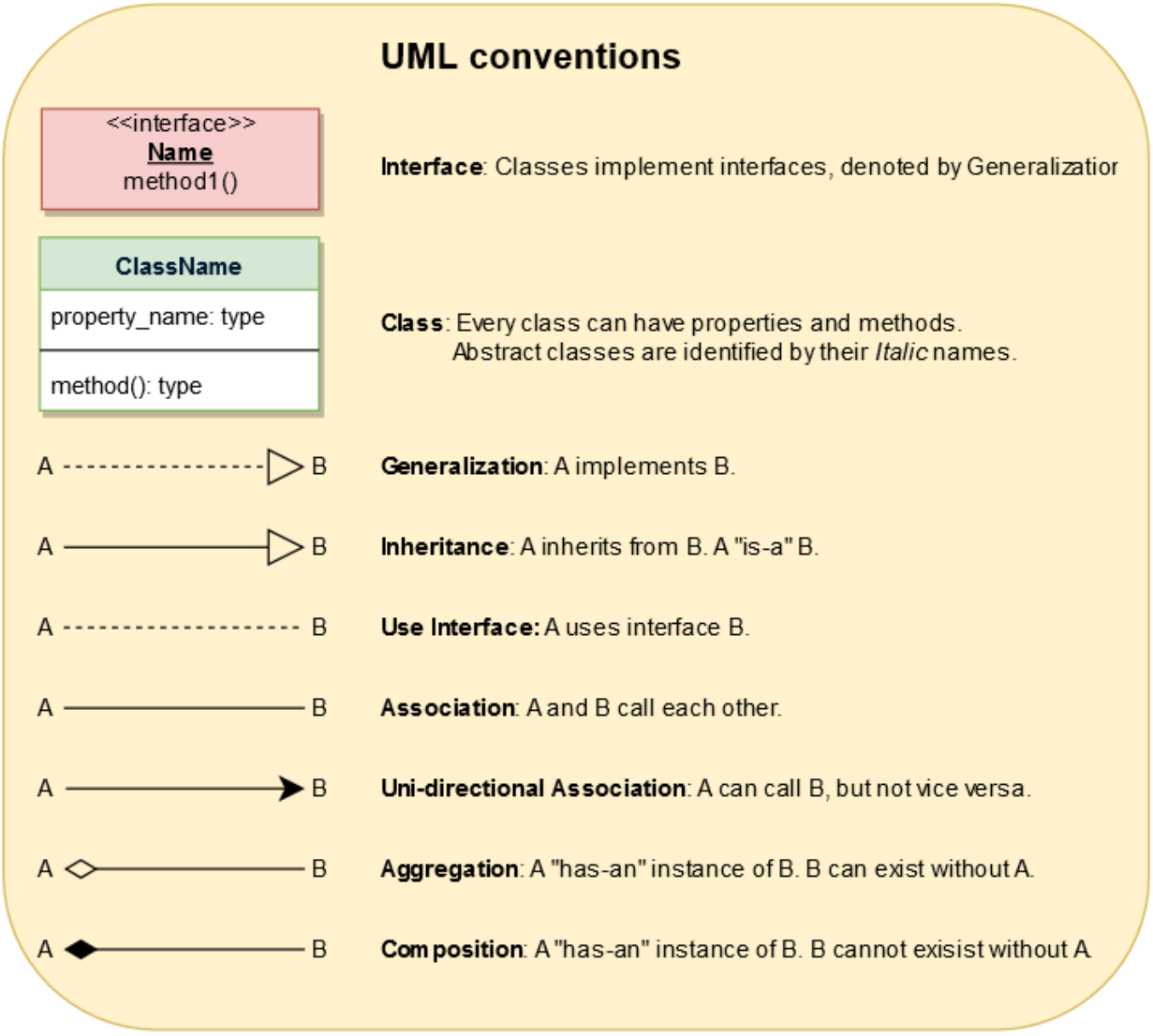
Sample class diagram for flight reservation system

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 gets destroyed automatically.

Generalization: Generalization is a 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.

Abstract class: An abstract class is identified by specifying its name in Italic. In the above diagram, both Person and Account classes are abstract classes.



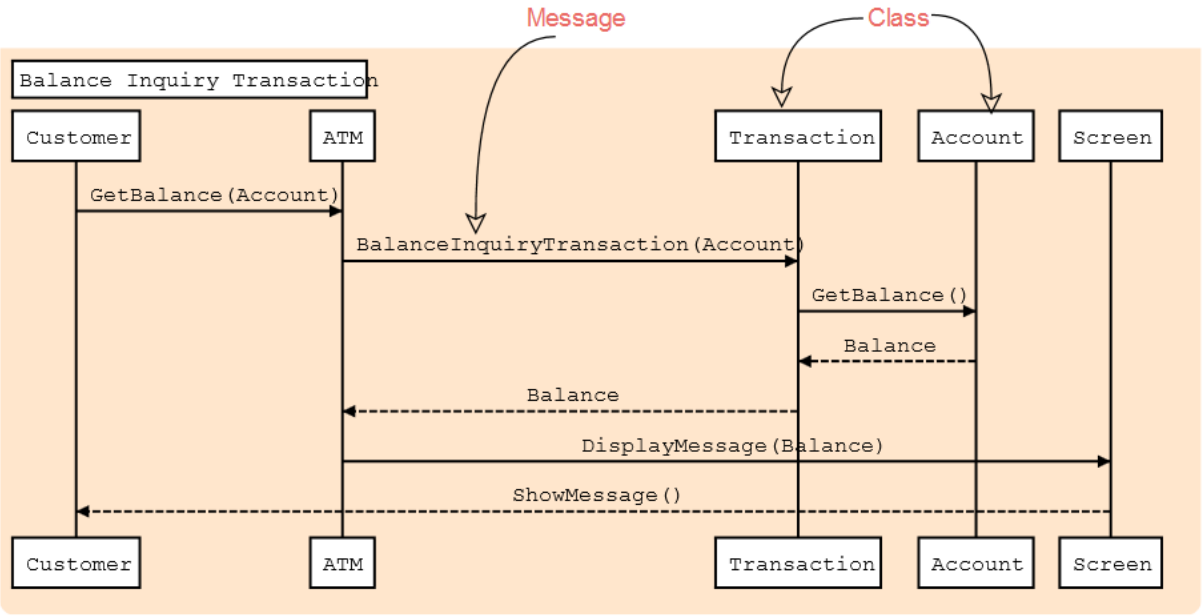
Sequence diagram

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time. This diagram is used to explore the logic of complex operations, function or procedure. In this diagram, the sequence of the interactions between the objects is represented in a step by step manner.

Sequence diagrams show a detailed flow for a specific use case or even just part of a particular use case. They are almost self-explanatory; they show the calls between the

different objects in their sequence and can explain, at a detailed level, different calls to various objects.

A sequence diagram has two dimensions: The vertical dimension shows the sequence of messages in the time order that they occur; the horizontal dimension shows the object instances to which the messages are sent.



Sample sequence diagram for ATM balance inquiry

Sample sequence diagram for ATM balance inquiry

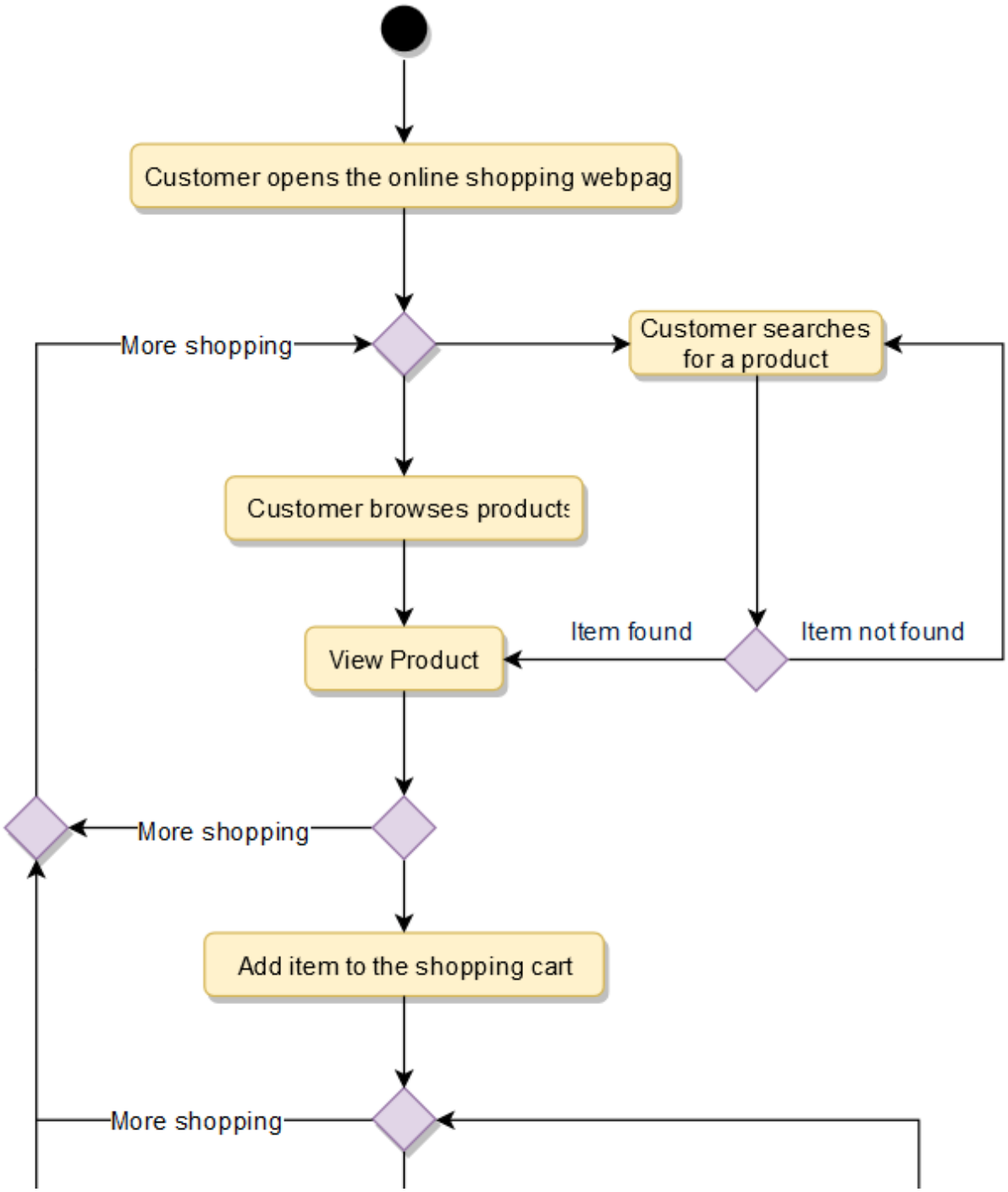
A sequence diagram is straightforward to draw. Across the top of your diagram, identify the class instances (objects) by putting each class instance inside a box (see above figure). If a class instance sends a message to another class instance, draw a line with an open arrowhead pointing to the receiving class instance and place the name of the message above the line. Optionally, for important messages, you can draw a dotted line with an arrowhead pointing back to the originating class instance; label the returned value above the dotted line.

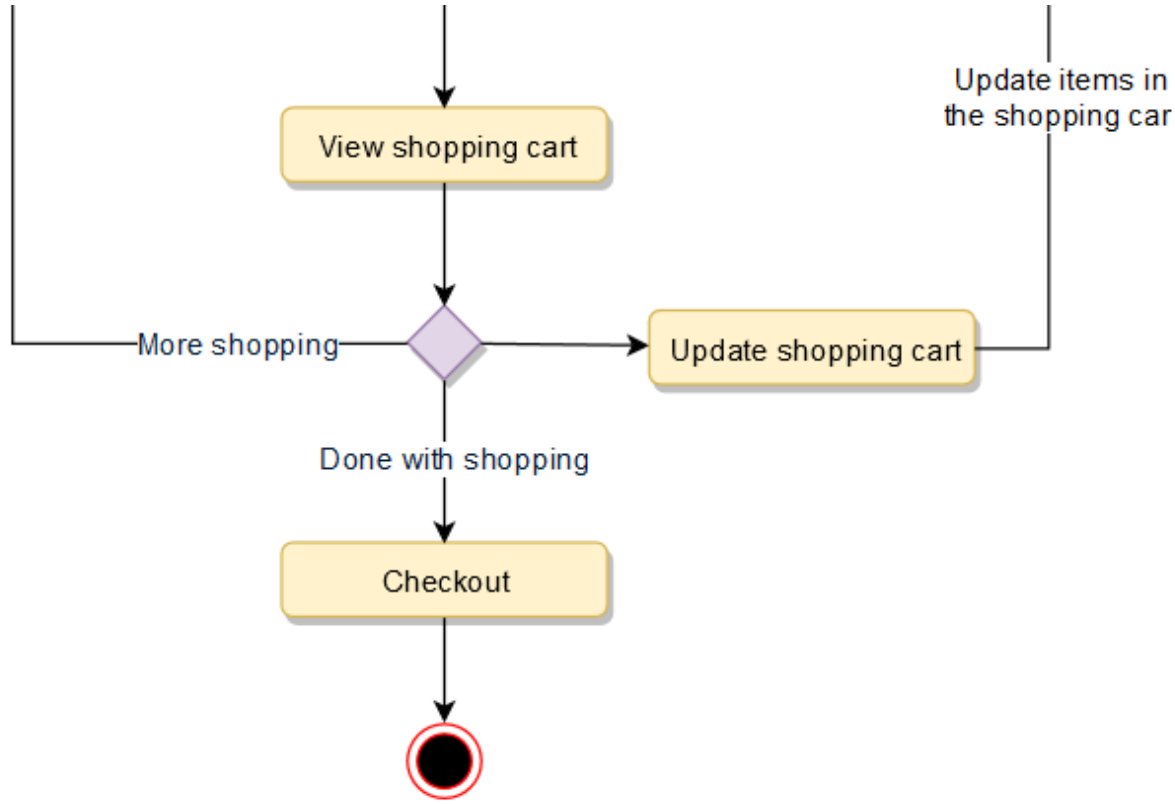
Activity Diagrams

We use Activity Diagrams to illustrate the flow of control in a system. An activity diagram shows the flow of control or object flow with emphasis on the condition of flow and the sequence in which it happens. We can also use an activity diagram to refer to the steps involved in the execution of a use case.

Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.

Following is the activity diagram for a user performing online shopping:


Customer opens the online shopping webpag
More shopping
Customer searches for a product
Customer browses products
Item found
Item not found
View Product
More shopping
Add item to the shopping cart
More shopping



Sample activity diagram for online shopping

What is the difference between Activity diagram and Sequence diagram?

Activity diagram captures the process flow. They are used for functional modeling. A functional model, represents the flow of values from external inputs, through operations and internal data stores, to external outputs.
Sequence diagram tracks the interaction between the objects. They are used for dynamic modeling which is represented by tracking states, transitions between states and the events to trigger these transitions.

Design a Library Management System

A Library Management System is a software build 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

Always clarify requirements at the beginning of the interview. Be sure to ask questions to find the exact scope of the system that the interviewer has in mind.

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

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 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 on how many books a member can check-out.
6. The system should be able to collect fines for books returned after the due date.
7. Members should be able to reserve books that are not currently available.
8. 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.
9. The system will be able to read barcodes from books and members' library cards.

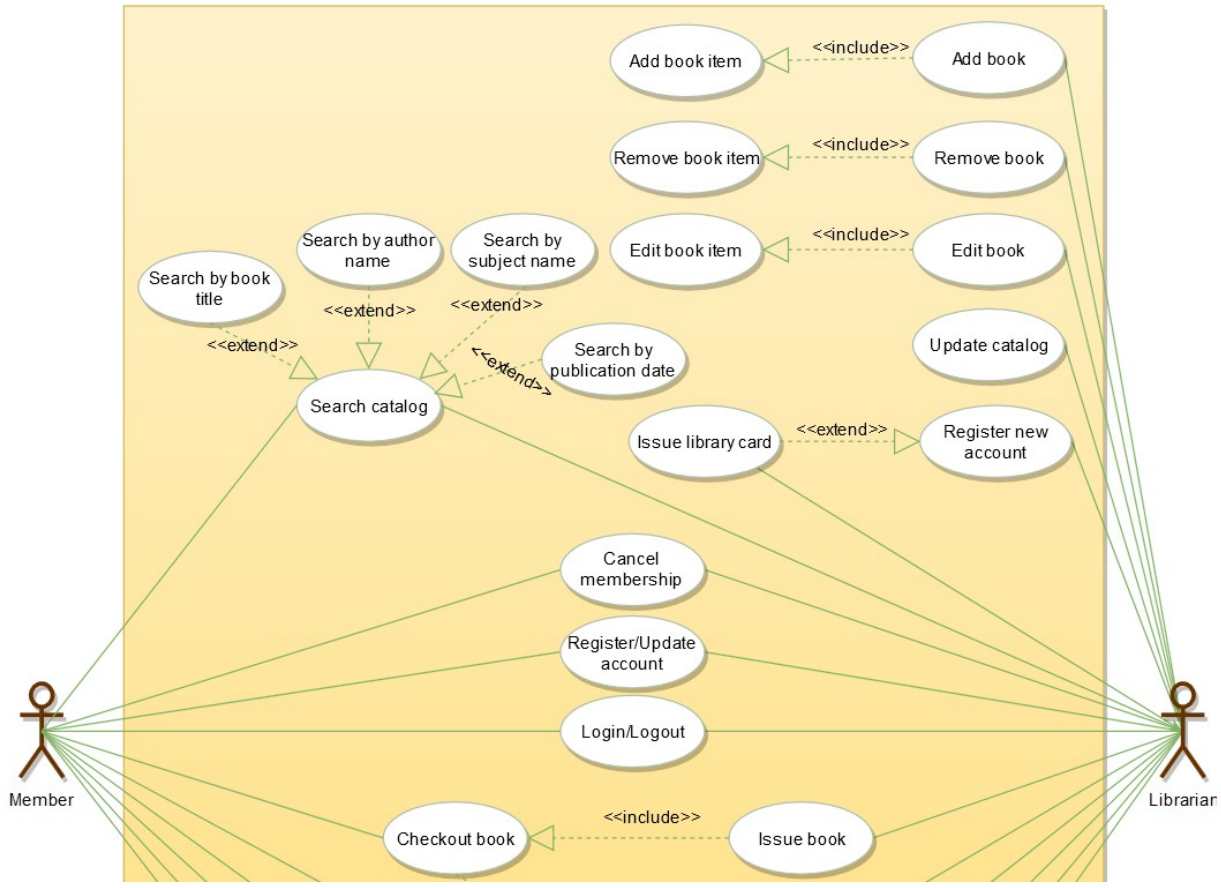
Use case diagram

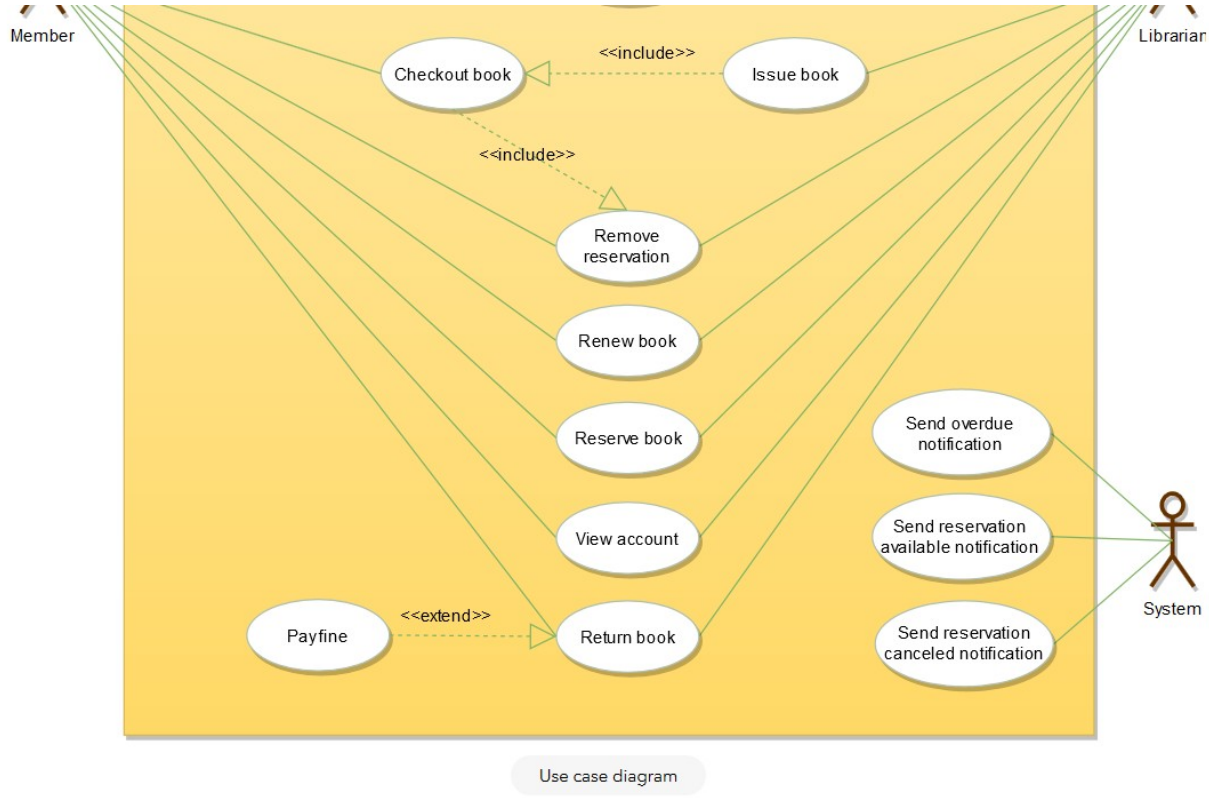
We have three main actors in our system:

- **Librarian:** Mainly responsible for adding and modifying books, book items, and users. Librarian can issue, reserve and return book items too.
- **Member:** All members can search the catalog, as well as checkout, reserve, renew and return a book.
- **System:** Mainly responsible for sending notifications for overdue books, reservation canceled, 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.





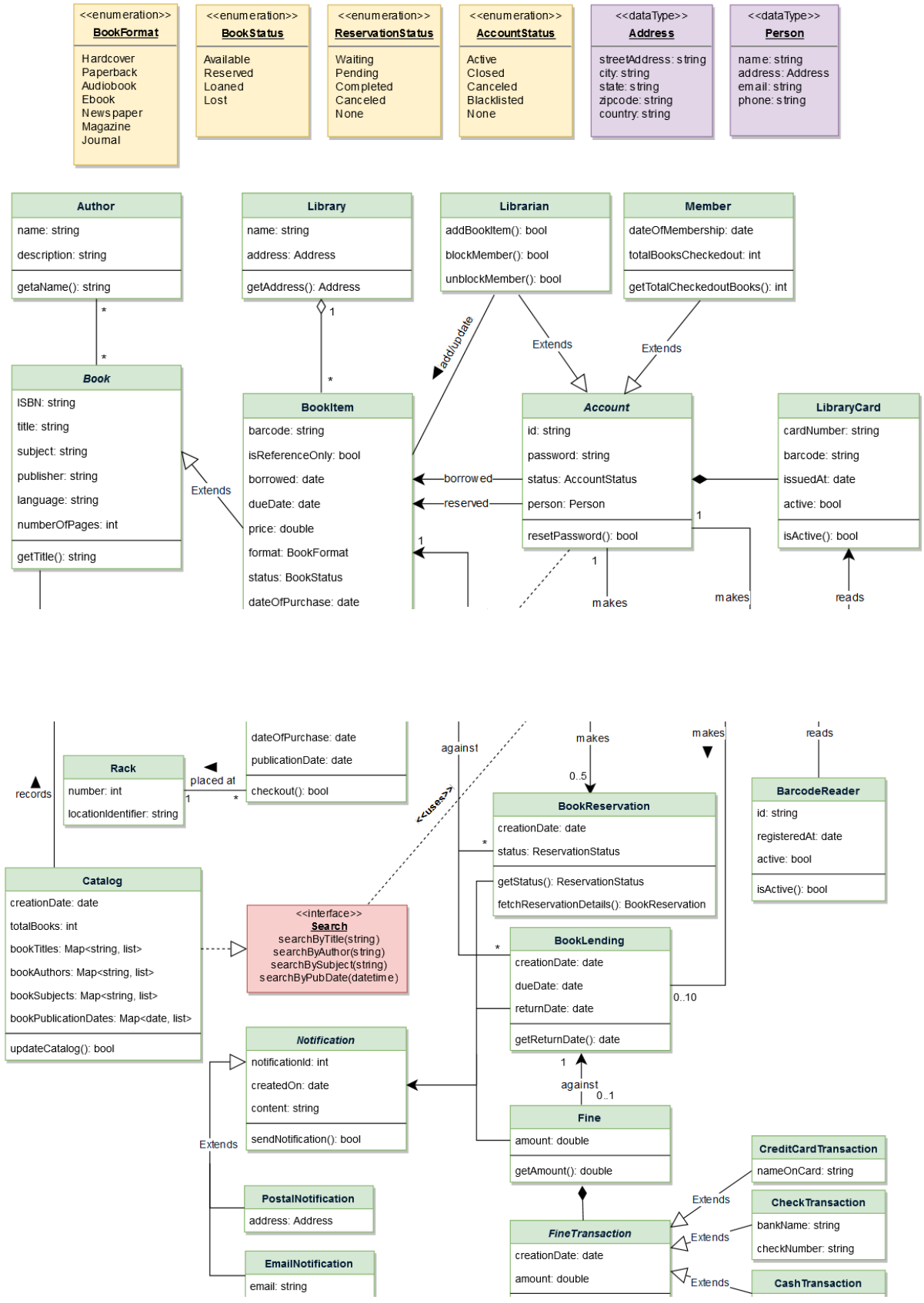
Use case diagram

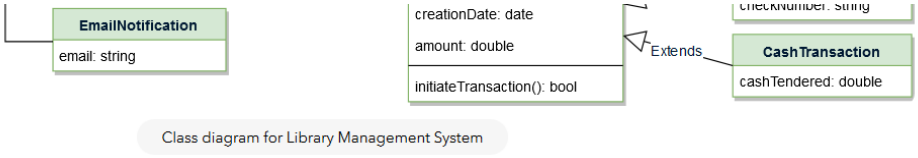
Class diagram

Here are the main classes of our Library Management System:

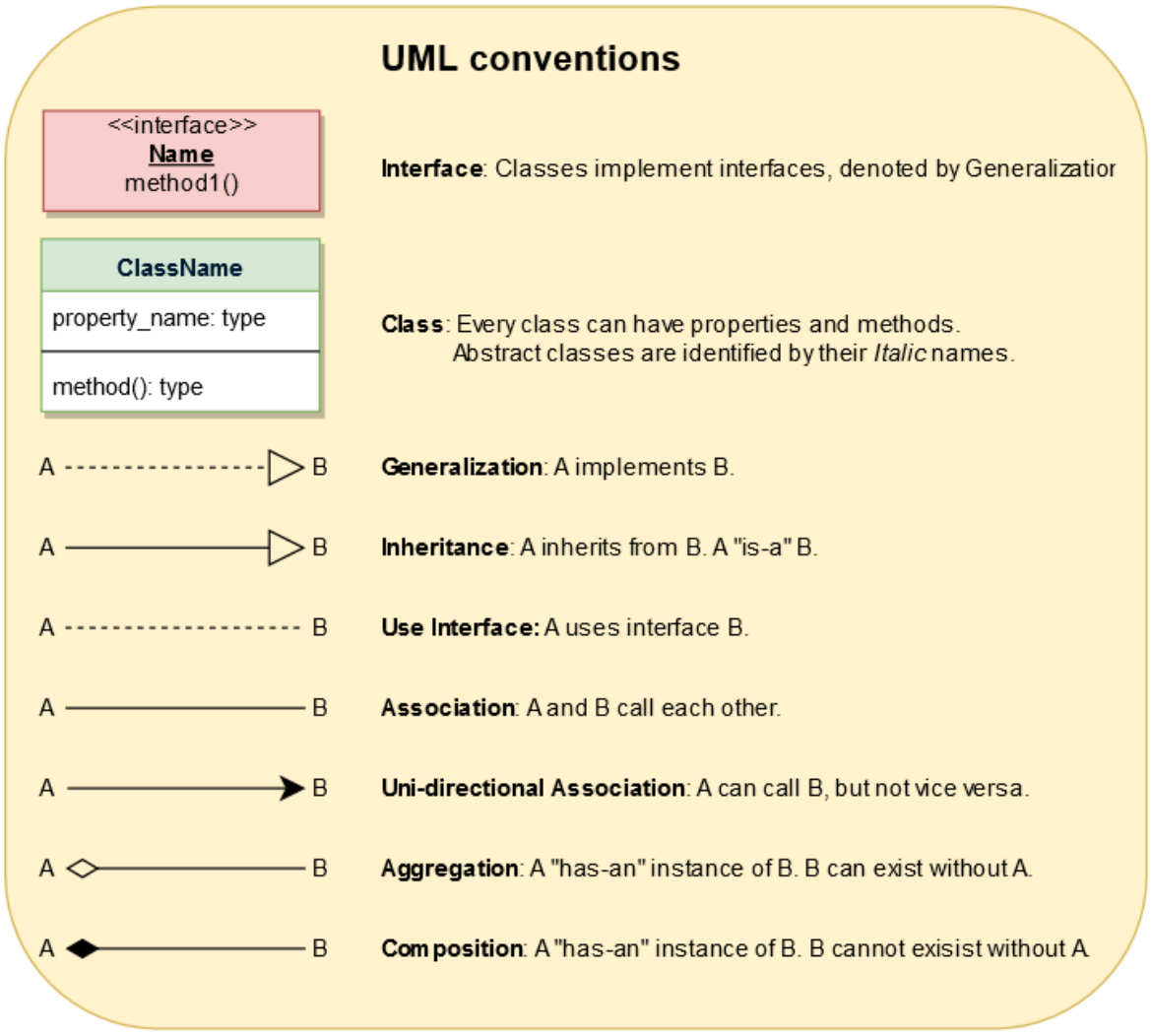
- **Library:** The central part of the organization for which this software has been designed. It has attributes like 'Name' to distinguish it from any other libraries and 'Address' to describe its location.
- **Book:** The basic building block of the system. Every book will have ISBN, Title, Subject, Publishers, etc.
- **BookItem:** Any book can have multiple copies, each copy will be considered a book item in our system.
- **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:** Will be responsible for managing reservations against book items.
- **BookLending:** This class Will manage the checking-out of book items.
- **Catalog:** Our system will support searching through four catalogs: Title, Author, Subject and Publish-date catalogs.
- **Fine:** This class will be responsible for calculating and collecting fines from members.
- **Author:** This class will be encapsulating book authors.
- **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 the rack in the library.

- **Notification:** This class will take care of sending notifications to members.



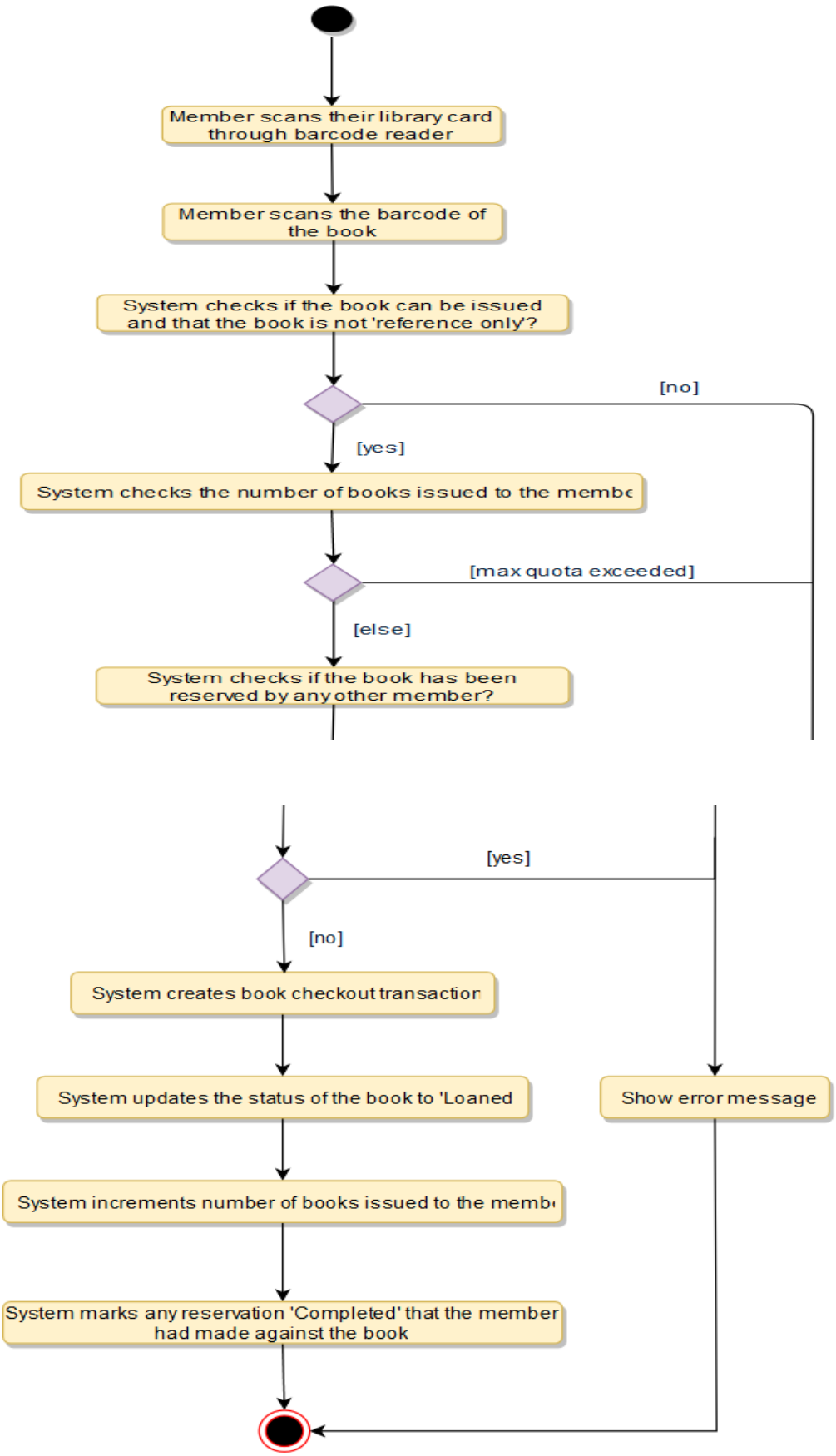


Class diagram for Library Management System

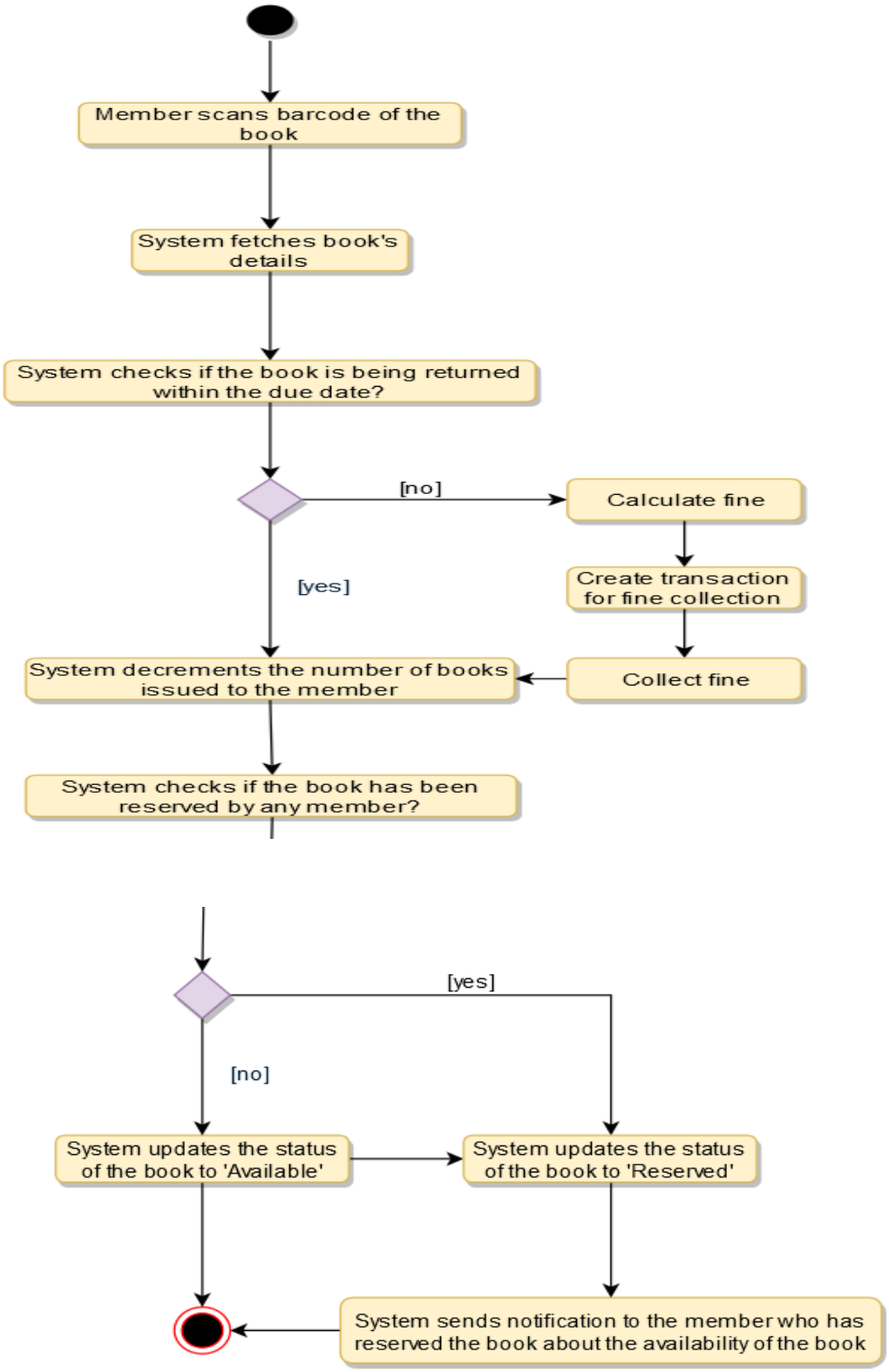


Activity diagrams

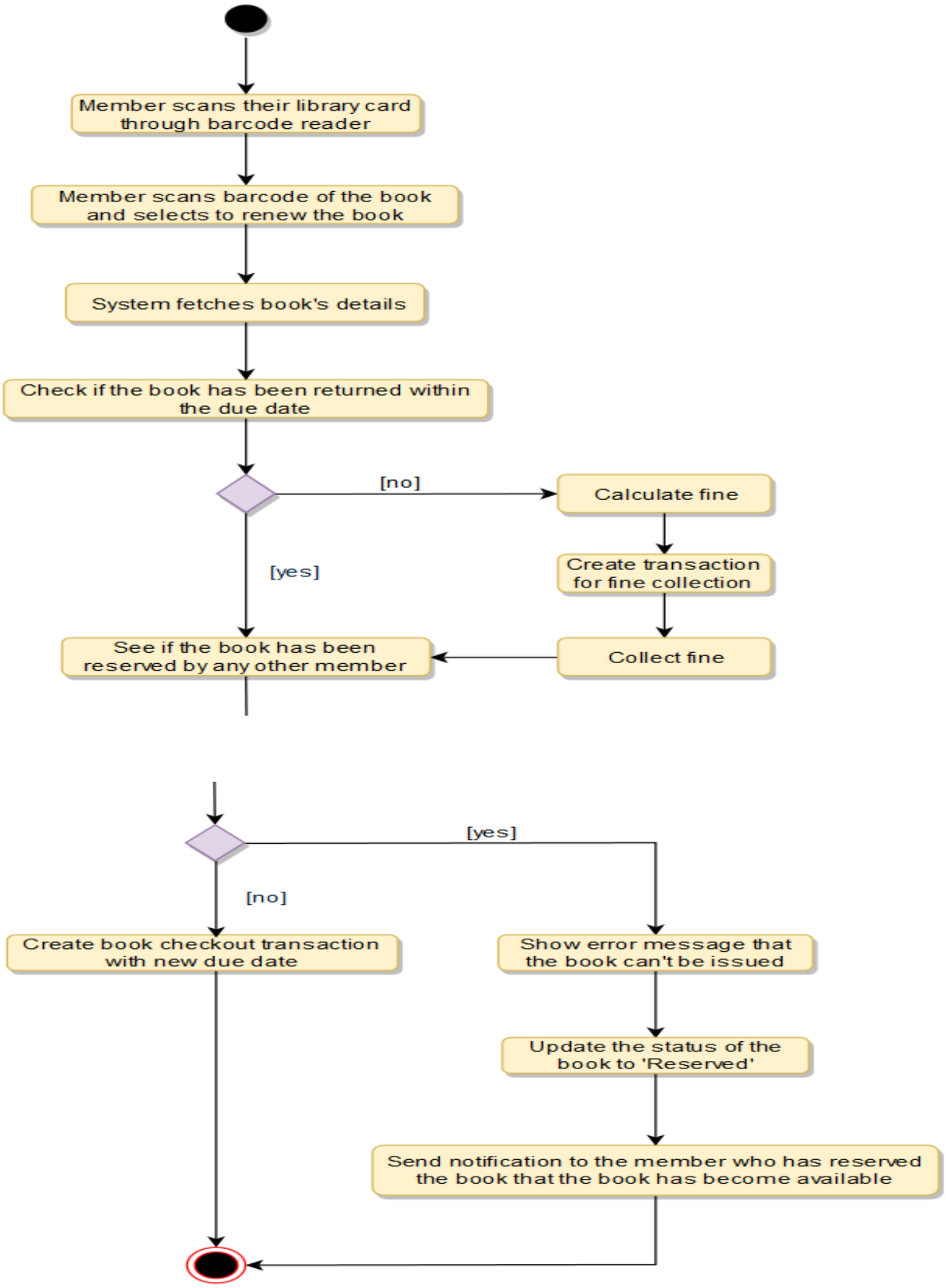
Check-out a book: Any library member or librarian can perform this activity. Here are the set of steps to check-out a book:


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 membe
[max quota exceeded]
[else]
System checks if the book has been reserved by any other member?
[yes]
[no]
System creates book checkout transactior
System updates the status of the book to 'Loaned
Show error message
System increments number of books issued to the membe
System marks any reservation 'Completed' that the member had made against the book

Return a book: Any library member or librarian can perform this activity. The system will collect the fine from members if they are returning the books after the due date. Here are the different steps for returning a book:



Renew a book: While renewing (re-issuing) a book, the system will check for fines and see if any other member has not reserved the same book, in that case the book item cannot be renewed. Here are the different steps for renewing a book:



Code

Here is the code for the use-cases mentioned above 1) Check-out a book, 2) Return a book, and 3) Renew a book.

Note: This code only focuses on the design part of the use-cases. Since you are not supposed to write a fully executable code in an interview, the reader can assume parts of the code to interact with the database and payment system, etc.

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

```
public enum BookFormat {
  HARDCOVER,
  PAPERBACK,
  AUDIO_BOOK,
  EBOOK,
  NEWSPAPER,
  MAGAZINE,
  JOURNAL
}

public enum BookStatus {
  AVAILABLE,
  RESERVED,
  LOANED,
  LOST
}

public enum ReservationStatus{
  WAITING,
  PENDING,
  CANCELED,
  NONE
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  NONE
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}
```

```
public class Person {
  private String name;
  private Address address;
  private String email;
  private String phone;
}

public class Constants {
  public static final int MAX_BOOKS_ISSUED_TO_A_USER = 5;
  public static final int MAX_LENDING_DAYS = 10;
}
```

Account, Member, and Librarian: These classes represent different people that interact with our system:

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

public abstract class Account {
  private String id;
  private String password;
  private AccountStatus status;
  private Person person;

  public boolean resetPassword();
}

public class Librarian extends Account {
  public boolean addBookItem(BookItem bookItem);
  public boolean blockMember(Member member);
  public boolean unBlockMember(Member member);
}

public class Member extends Account {
  private Date dateOfMembership;
  private int totalBooksCheckedout;
  public int getTotalCheckedoutBooks();
  public boolean reserveBookItem(BookItem bookItem);
  private void incrementTotalBooksCheckedout();

  public boolean checkoutBookItem(BookItem bookItem);
  public boolean returnBookItem(BookItem bookItem);
  public boolean renewBookItem(BookItem bookItem);

  public boolean checkoutBookItem(BookItem bookItem) {
    if(this.getTotalBooksCheckedOut() >=
Constants.MAX_BOOKS_ISSUED_TO_A_USER ) {
      ShowError("The user has already checkedout maximum number of books");
      return false;
```

```
    }
    BookReservation bookReservation =
BookReservation.fectchReservationDetails(bookItem.getBarcode());
    if( bookReservation != null &&  bookReservation.getMemberId() != this.getId() ) {
      // book item has a pendening reservation from another user
        ShowError("This book is reserved by another member");
        return false;
      } else if( bookReservation != null ) {
      // book item has a pendening reservation from the give member, update it
      bookReservation.updateStatus(ReservationStatus.COMPLETED);
    }

    if(!bookItem.checkout(this.getId())) {
      return false;
    }

    this.incrementTotalBooksCheckedout();
    return true;
  }

  private void checkForFine(String bookItemBarcode) {
    BookLending bookLending = BookLending.fectchLendingDetails(bookItemBarcode);
    Date dueDate = bookLending.getDueDate();
    Date today = new Date();
    // check if the book has been returned within the due date
    if(todaye.compareTo(dueDate) > 0) {
      long diff = todayDate.getTime() - dueDate.getTime();
      long diffDays = diff / (24 * 60 * 60 * 1000);
      Fine.collectFine(memberId, diffDays);
    }
  }

  public void returnBookItem(BookItem bookItem) {
    this.checkForFine();
    BookReservation bookReservation =
BookReservation.fectchReservationDetails(bookItem.getBarcode());
    if(bookReservation != null) {
      // book item has a pendening reservation
      bookItem.updateBookItemStatus(BookStatus.RESERVED);
      bookReservation.sendBookAvailableNotification();
    }
    bookItem.updateBookItemStatus(BookStatus.AVAILABLE);
  }

  public bool renewBookItem(BookItem bookItem) {
    this.checkForFine();
    BookReservation bookReservation =
BookReservation.fectchReservationDetails(bookItem.getBarcode());
    if( bookReservation != null && bookReservation.getMemberId() !=
member.getMemberId() ) {
```

```
      // book item has a pendening reservation from another member
      ShowError("This book is reserved by another member");
      member.decrementTotalBooksCheckedout();
      bookItem.updateBookItemState(BookStatus.RESERVED);
      bookReservation.sendBookAvailableNotification();
      return false;
    } else if( bookReservation != null ){
      // book item has a pendening reservation from this member
      bookReservation.updateStatus(ReservationStatus.COMPLETED);
    }
    BookLending.lendBook(bookItem.getBarCode(), this.getMemberId());

bookItem.updateDueDate(LocalDate.now().plusDays(Constants.MAX_LENDING_DAYS));
    return true;
  }
}
```

BookReservation, BookLending and Fine: These classes represent a book reservation, lending and fine collection respectively.

```
public class BookReservation {
  private Date creationDate;
  private ReservationStatus status;
  private String bookItemBarcode;
  private String memberId;

  public static BookReservation fectchReservationDetails(String barcode);
}

public class BookLending {
  private Date creationDate;
  private Date dueDate;
  private Date returnDate;
  private String bookItemBarcode;
  private String memberId;

  public static void lendBook(String barcode, String memberId);
  public static BookLending fectchLendingDetails(String barcode);
}

public class Fine {
  private Date creationDate;
  private double bookItemBarcode;
  private String memberId;

  public static void collectFine(String memberId, long days) {}
}
```

BookItem: To encapsulate a book item. This class will be responsible for processing reservation, return and renew of a book item.

```
public abstract class Book {
  private String ISBN;
  private String title;
  private String subject;
  private String publisher;
  private String language;
  private int numberOfPages;
  private List<Author> authors;
}

public class BookItem extends Book {
  private String barcode;
  private boolean isReferenceOnly;
  private Date borrowed;
  private Date dueDate;
  private double price;
  private BookFormat format;
  private BookStatus status;
  private Date dateOfPurchase;
  private Date publicationDate;
  private Rack placedAt;

  public boolean checkout(String memberId) {
    if(bookItem.getIsReferenceOnly()) {
      ShowError("This book is Reference only and can't be issued");
      return false;
    }
    if(!BookLending.lendBook(this.getBarCode(), memberId)){
      return false;
    }
    this.updateBookItemStatus(BookStatus.LOANED);
    return true;
  }
}

public class Rack {
  private int number;
  private String locationIdentifier;
}
```

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

```
public interface Search {
  public List<Book> searchByTitle(String title);
  public List<Book> searchByAuthor(String author);
```

```
  public List<Book> searchBySubject(String subject);
  public List<Book> searchByPubDate(Date publishDate);
}

public class Catalog implements Search {
  private HashMap<String, List<Book>> bookTitles;
  private HashMap<String, List<Book>> bookAuthors;
  private HashMap<String, List<Book>> bookSubjects;
  private HashMap<String, List<Book>> bookPublicationDates;

  public List<Book> searchByTitle(String query) {
    // return all books containing the string query in their title.
    return bookTitltes.get(query);
  }

  public List<Book> searchByAuthor(String query) {
    // return all books containing the string query in their author's name.
    return bookAuthors.get(query);
  }
}
```

Design a Parking Lot

Let's make an object-oriented design for a multi-floor 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 of large area.

A Parking Lot

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 have a parking ticket from the entry points and can pay the parking fee on their way out at the exit points.
4. Customers can pay the tickets at the automated exit panel or to the parking attendant.
5. Customers can pay either cash and through credit cards.
6. Customers should also be able to pay the parking fee at the customer's info portal at each floor. When 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 slots. The system should support multiple types of parking slots like Compact, Large, Handicapped, Mototrbyke, etc.
9. The Parking lot will have some parking lots specified for electric cars. These parking lots can have an electric panel from where customers can pay and charge their vehicles.
10. The system should support parking for different types of vehicles like car, truck, van, motorbike, etc.

11. Each parking floor will have a display board showing a free parking slot for each slot 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.

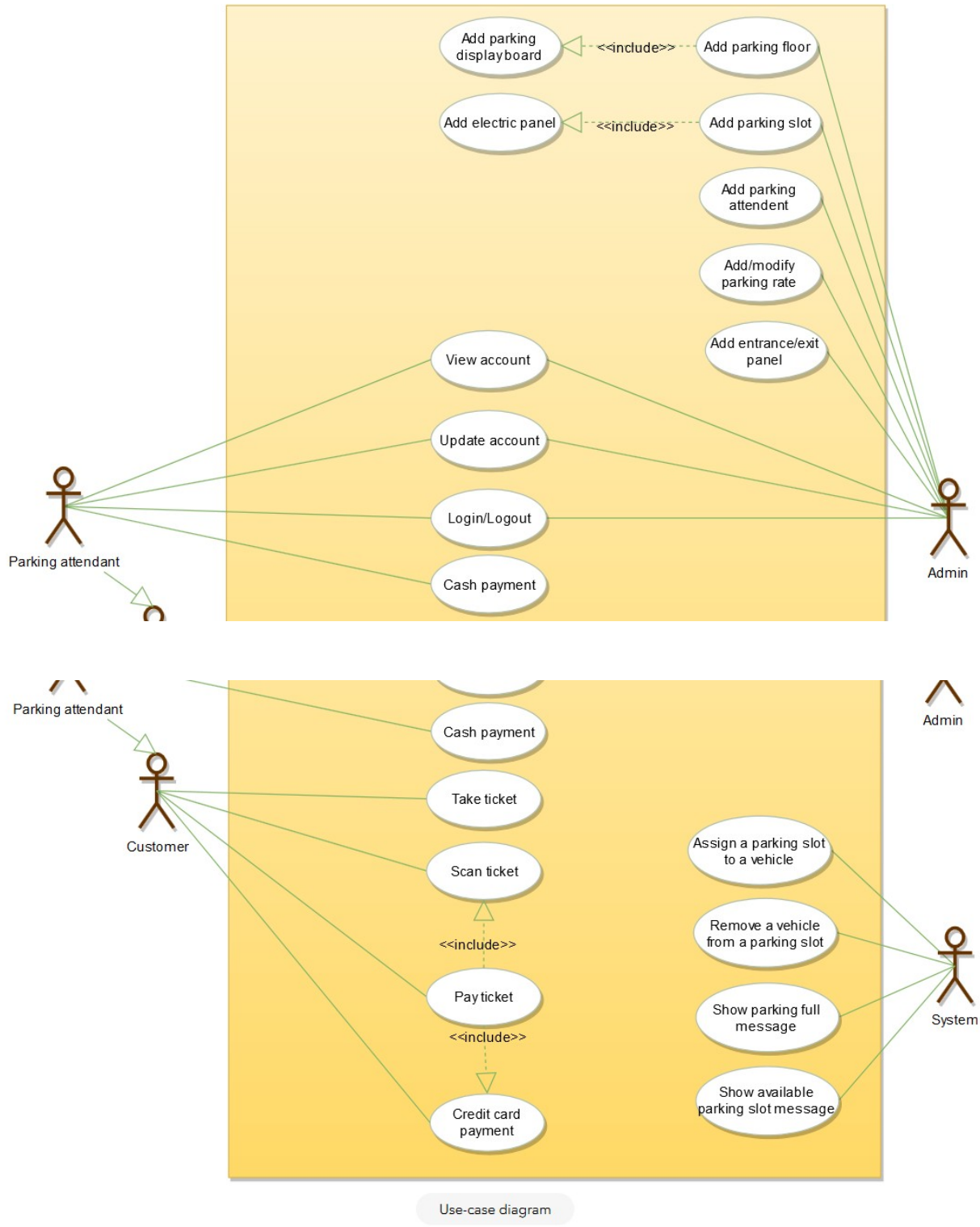
Use case diagram

Here are the main Actors in our system:

- **Admin:** Mainly responsible for adding and modifying parking floors, parking slots, 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 show messages on to different info panels, as well as assigning and removing a vehicle from a parking slot.

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 slots.
- **Add/Remove/Edit parking slot:** To add, remove or modify a parking slot to a parking floor.
- **Add/Remove a parking attendant:** To add or remove a parking attendant from the system.
- **Take ticket:** Customers will take a new parking ticket when entering the parking lot.
- **Scan ticket:** To scan a ticket to see how much it would charge.
- **Credit card payment:** To pay the ticket fee through credit card.
- **Cash payment:** To pay the parking ticket through cash.
- **Add/Modify parking rate:** Admin can add or modify the hourly parking rate.



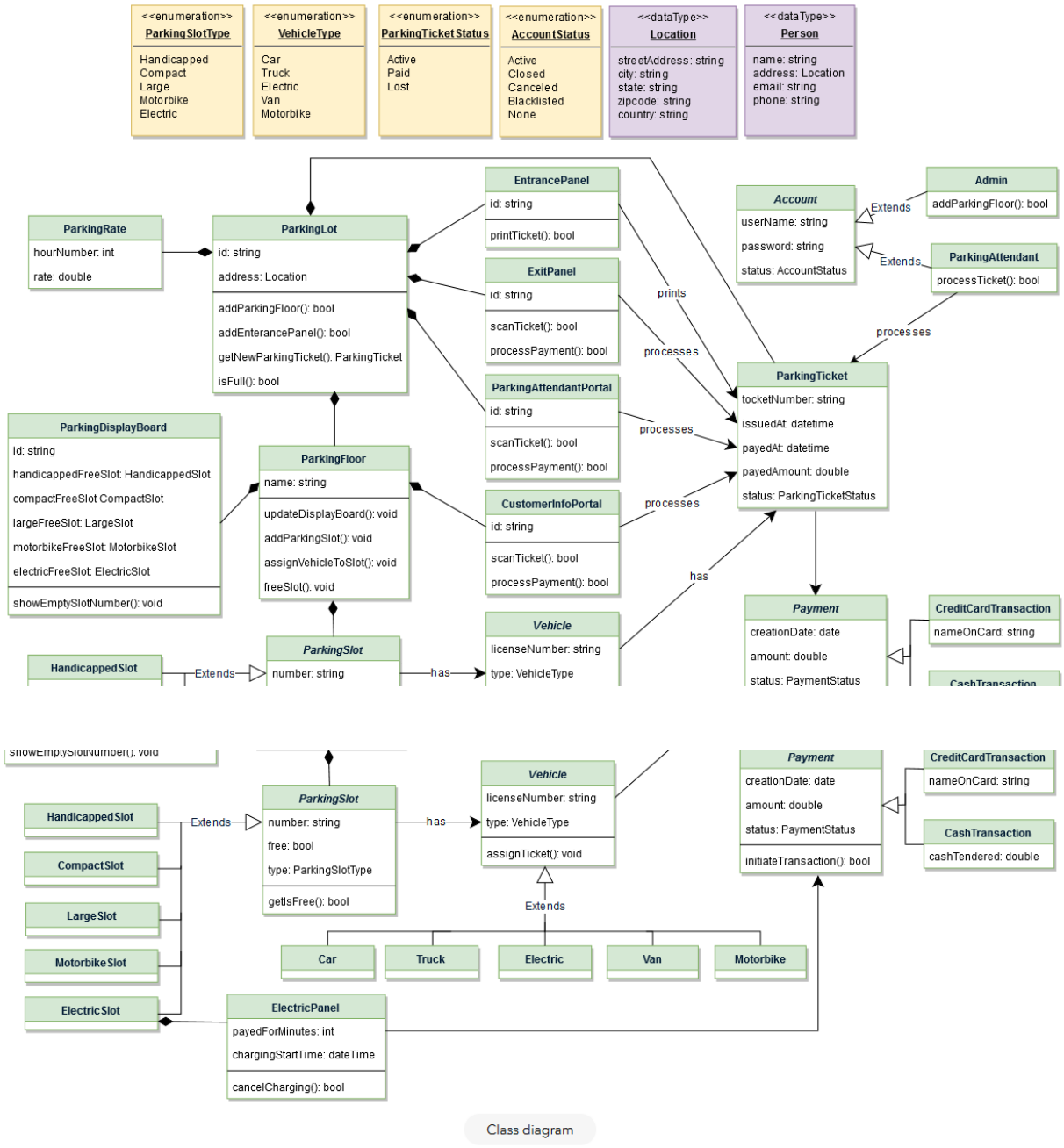
Use-case diagram

Use-case diagram

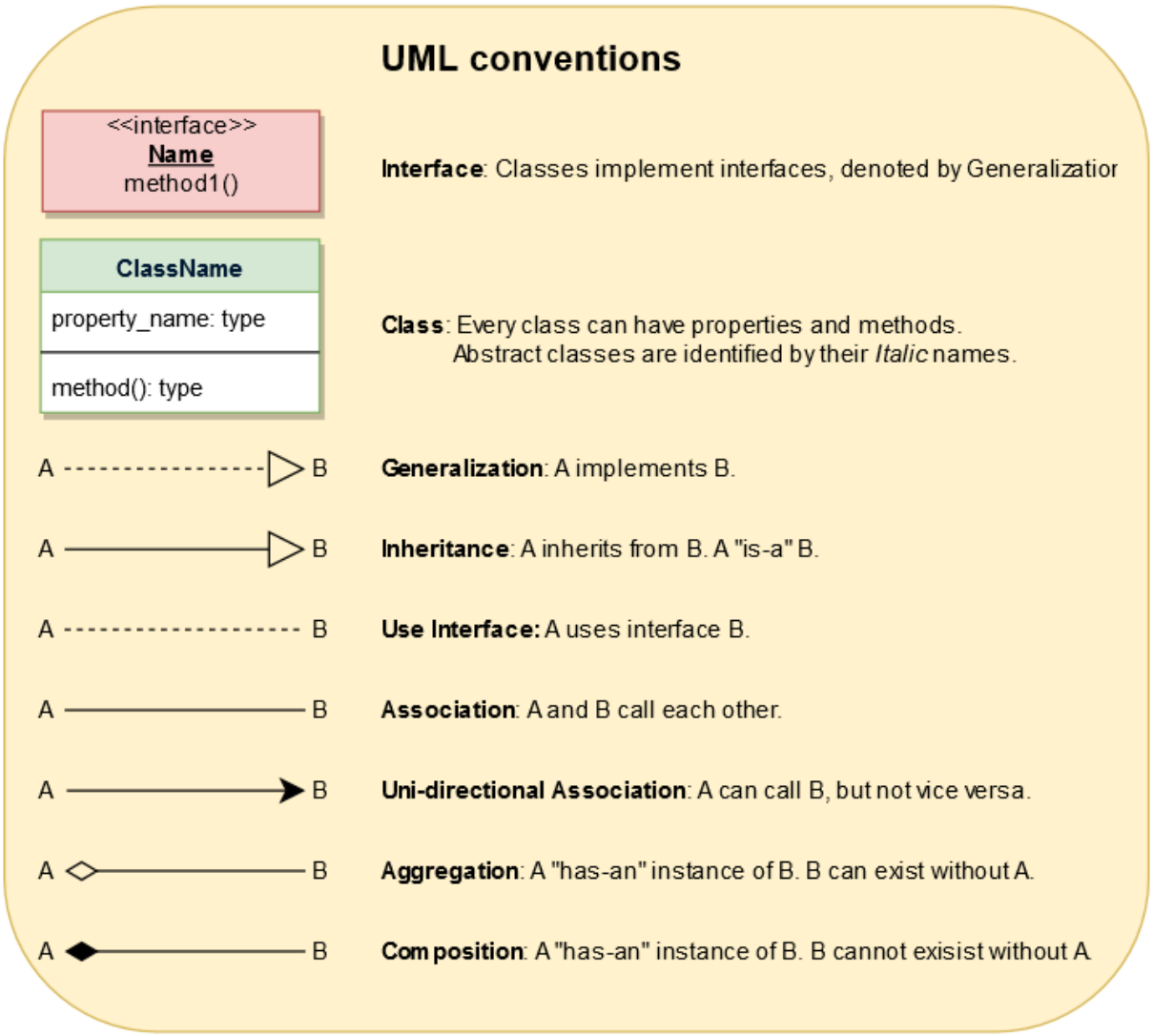
Class diagram

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.
- **ParkingSlot:** Each parking floor has many parking slots. Our system will support different parking slots 1) Handicapped, 2) Compact, 3) Large, 4) Motorbike, and 5) Electric.
- **Account:** We will have two types of accounts in the system, one will be an Admin, and the other will be of a parking attendant.
- **Parking ticket:** This class will be encapsulating a parking ticket. Customers will take a ticket when they enter the parking lot.
- **Vehicle:** Vehicles will be parked in the parking slots. Our system will support different types of vehicles 1) Car, 2) Truck, 3) Electric, 4) Van and 5) Motorbike.
- **EntrancePanel and ExitPanel:** EntrancePanel will be printing tickets, and ExitPanel will facilitate paying the fee against a parking ticket.
- **Payment:** Will be responsible for making a payment transaction. The system will support credit card and cash transactions.
- **ParkingRate:** This class will keep track of the hourly parking rates. For every hour it will specify a dollar amount. For example, for a two hours parking ticket, this class will tell how much would it cost for the first hour and how much for the second hour.
- **ParkingDisplayBoard:** Each parking floor will have a display board to show available parking slots for each slot type. This class will be responsible for displaying the latest state of free parking slots to the customers.
- **ParkingAttendantPortal:** This class will encapsulate all the operations that an attendant can perform, like scanning a ticket and processing the payments.
- **CustomerInfoPortal:** This class will encapsulate the info portal that the customers use to pay for the parking ticket. Once paid the info portal update the ticket to keep track of the payment.
- **ElectricPanel:** Customers use the electric panels to pay and charge their electric vehicles.

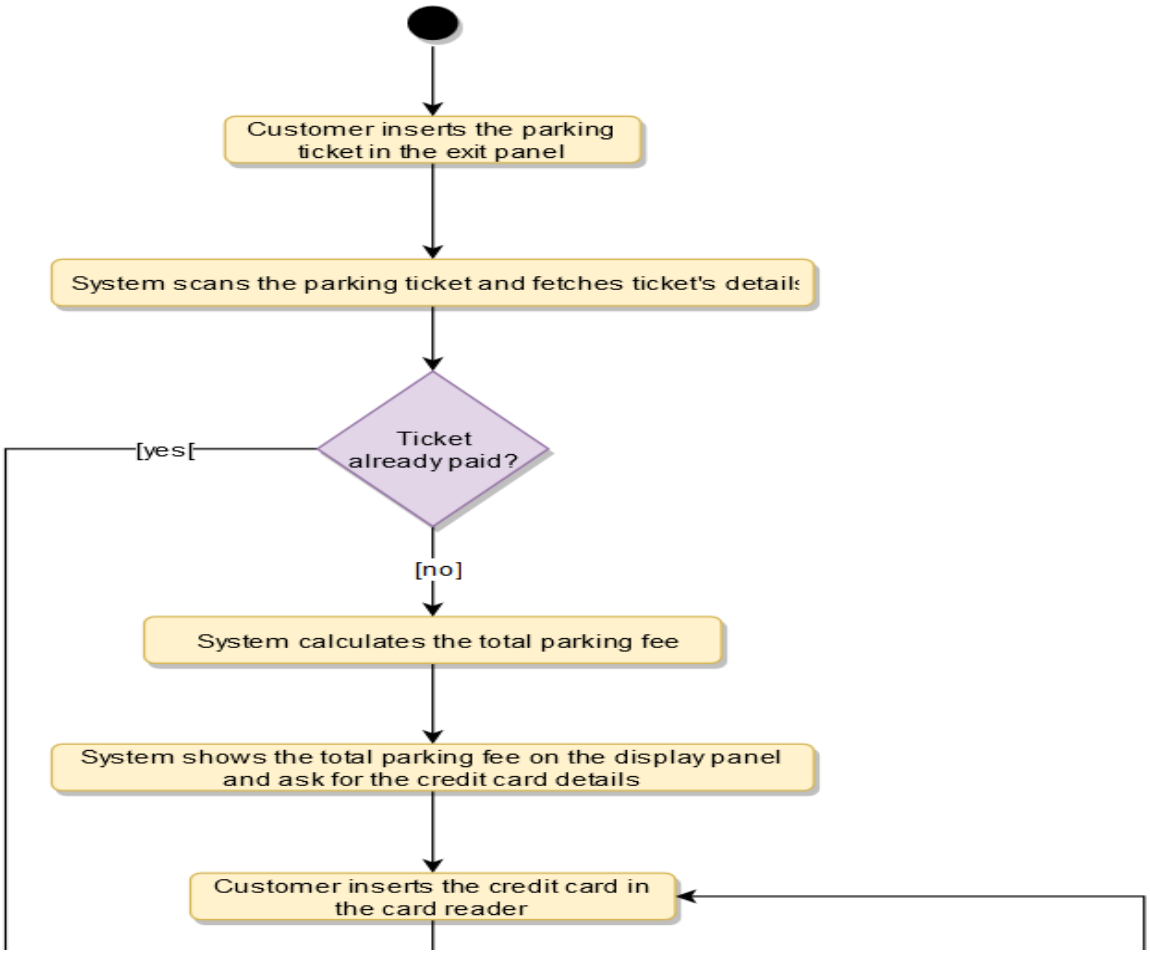


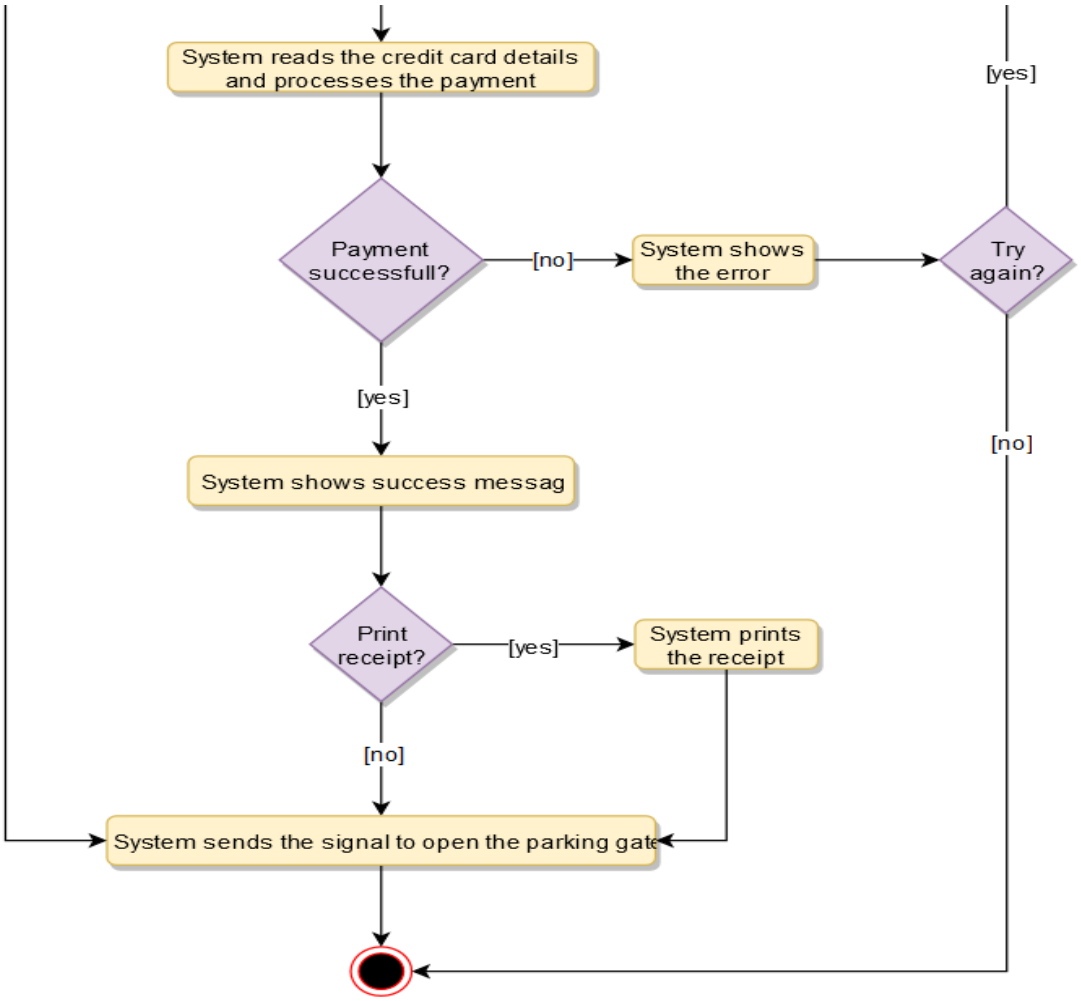
Class diagram



Activity diagrams

Customer paying for parking ticket: Any customer can perform this activity. Here are the set of steps:


Customer inserts the parking ticket in the exit panel
System scans the parking ticket and fetches ticket's details
Ticket already paid?
[yes]
[no]
System calculates the total parking fee
System shows the total parking fee on the display panel and ask for the credit card details
Customer inserts the credit card in the card reader



Code

Following is the skeleton code for our parking lot system:

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

```
public enum VehicleType{
  CAR,
  TRUCK,
  ELECTRIC,
  VAN,
  MOTORBIKE
}

public enum ParkingSlotType{
  HANDICAPPED,
  COMPACT,
  LARGE,
  MOTORBIKE,
  ELECTRIC
}
```

```
public enum AccountStatus{
  ACTIVE,
  BLOCKED,
  BANNED,
  COMPROMIZED,
  ARCHIVED,
  UNKNOWN
}

public enum ParkingTicketStatus{
  ACTIVE,
  PAID,
  LOST
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}

public class Person {
  private String name;
  private Address address;
  private String email;
  private String phone;
}
```

Account, Admin, and ParkingAttendant: These classes represent different people that interact with our system:

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

public abstract class Account {
  private String userName;
  private String password;
  private AccountStatus status;
  private Person person;

  public boolean resetPassword();
}

public class Admin extends Account {
  public bool addParkingFloor(ParkingFloor floor);
  public bool addParkingSlot(String floorName, ParkingSlot slot);
```

```
  public bool addParkingDisplayBoard(String floorName, ParkingDisplayBoard
displayBoard);
  public bool addCustomerInfoPanel(String floorName, CustomerInfoPanel infoPanel);

  public bool addEntrancePanel(EntrancePanel entrancePanel);
  public bool addExitPanel(ExitPanel exitPanel);
}

public class ParkingAttendant extends Account {
  public bool processTicket(string TicketNumber);
}
```

ParkingSlot: Here is the definition of ParkingSlot and all of its children classes:

```
public abstract class ParkingSlot {
  private String number;
  private boolean free;
  private Vehicle vehicle;
  private final ParkingSlotType type;

  public boolean IsFree();

  public ParkingSlot(ParkingSlotType type) {
    this.type = type;
  }

  public boolean assignVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
    free = false;
  }

  public boolean removeVehicle() {
    this.vehicle = null;
    free = true;
  }
}

public class HandicappedSlot extends ParkingSlot {
  public HandicappedSlot() {
    super(ParkingSlotType.HANDICAPPED);
  }
}

public class CompactSlot extends ParkingSlot {
  public CompactSlot() {
    super(ParkingSlotType.COMPACT);
  }
}

public class LargeSlot extends ParkingSlot {
```

```
  public LargeSlot() {
    super(ParkingSlotType.LARGE);
  }
}

public class MotorbikeSlot extends ParkingSlot {
  public MotorbikeSlot() {
    super(ParkingSlotType.MOTORBIKE);
  }
}

public class ElectricSlot extends ParkingSlot {
  public ElectricSlot() {
    super(ParkingSlotType.ELECTRIC);
  }
}
```

Vehicle: Here is the definition for Vehicle and all of its child classes:

```
public abstract class Vehicle {
  private String licenseNumber;
  private final VehicleType type;
  private ParkingTicket ticket;

  public Vehicle(VehicleType type) {
    this.type = type;
  }

  public void assignTicket(ParkingTicket ticket) {
    this.ticket = ticket;
  }
}

public class Car extends Vehicle {
  public Car() {
    super(VehicleType.CAR);
  }
}

public class Van extends Vehicle {
  public Van() {
    super(VehicleType.VAN);
  }
}

public class Truck extends Vehicle {
  public Truck() {
    super(VehicleType.TRUCK);
  }
}
```

ParkingFloor: This class encapsulates a parking floor:

```
public class ParkingFloor {
  private String name;
  private HashMap<String, HandicappedSlot> handicappedSlots;
  private HashMap<String, CompactSlot> compactSlots;
  private HashMap<String, LargeSlot> largeSlots;
  private HashMap<String, MotorbikeSlot> motorbikeSlots;
  private HashMap<String, ElectricSlot> electricSlots;
  private HashMap<String, CustomerInfoPortal> infoPortals;
  private ParkingDisplayBoard displayBoard;

  public ParkingFloor(String name) {
    this.name = name;
  }

  public void addParkingSlot(ParkingSlot slot) {
    switch(slot.getType()) {
      case ParkingSlotType.HANDICAPPED:
        handicappedSlots.put(slot.getNumber(), slot);
        break;
      case ParkingSlotType.COMPACT:
        compactSlots.put(slot.getNumber(), slot);
        break;
      case ParkingSlotType.LARGE:
        largeSlots.put(slot.getNumber(), slot);
        break;
      case ParkingSlotType.MOTORBIKE:
        motorbikeSlots.put(slot.getNumber(), slot);
        break;
      case ParkingSlotType.ELECTRIC:
        electricSlots.put(slot.getNumber(), slot);
        break;
      default: print("Wrong parking slot type!");
    }
  }

  public void assignVehicleToSlot(Vehicle vehicle, ParkingSlot slot){
    slot.assignVehicle(vehicle);
    switch(slot.getType()) {
      case ParkingSlotType.HANDICAPPED:
        updateDisplayBoardForHandicapped(slot); break;
      case ParkingSlotType.COMPACT:
        updateDisplayBoardForCompact(slot); break;
      case ParkingSlotType.LARGE:
        updateDisplayBoardForLarge(slot); break;
      case ParkingSlotType.MOTORBIKE:
        updateDisplayBoardForMotorbike(slot); break;
      case ParkingSlotType.ELECTRIC:
        updateDisplayBoardForElectric(slot); break;
```

```
    default: print("Wrong parking slot type!");
   }
  }

  private void updateDisplayBoardForHandicapped(ParkingSlot slot) {
   if(this.displayBoard.getHandicappedFreeSlot().getNumber() == slot.getNumber()) {
    // find another free handicapped parking and assign to displayBoard
    for (String key : handicappedSlots.keySet()) {
     if(handicappedSlots.get(key).isFree()) {
      this.displayBoard.setHandicappedFreeSlot(handicappedSlots.get(key));
     }
    }
    this.displayBoard.showEmptySlotNumber();
   }
  }

  private void updateDisplayBoardForCompact(ParkingSlot slot) {
   if(this.displayBoard.getCompactFreeSlot().getNumber() == slot.getNumber()) {
    // find another free compact parking and assign to displayBoard
    for (String key : compactSlots.keySet()) {
     if(compactSlots.get(key).isFree()) {
      this.displayBoard.setCompactFreeSlot(compactSlots.get(key));
     }
    }
    this.displayBoard.showEmptySlotNumber();
   }
  }

  public void freeSlot(ParkingSlot slot){
   slot.removeVehicle();
   switch(slot.getType()) {
    case ParkingSlotType.HANDICAPPED: freeelectricSlotCount++; break;
    case ParkingSlotType.COMPACT: freeCompactSlotCount++; break;
    case ParkingSlotType.LARGE: freeLargeSlotCount++; break;
    case ParkingSlotType.MOTORBIKE: freeMotorbikeSlotCount++; break;
    case ParkingSlotType.ELECTRIC: freeelectricSlotCount++; break;
    default: print("Wrong parking slot type!");
   }
  }
}
```

ParkingDisplayBoard: This class encapsulates a parking display board:

```
public class ParkingDisplayBoard {
  private String id;
  private HandicappedSlot handicappedFreeSlot;
  private CompactSlot compactFreeSlot;
  private LargeSlot largeFreeSlot;
  private MotorbikeSlot motorbikeFreeSlot;
  private ElectricSlot electricFreeSlot;
```

```
  public void showEmptySlotNumber() {
    String message = "";
    if(handicappedFreeSlot.IsFree()){
      message += "Free Handicapped: " + handicappedFreeSlot.getNumber();
    } else {
      message += "Handicaped is full";
    }
    message += System.lineSeparator();

    if(compactFreeSlot.IsFree()){
      message += "Free Compact: " + compactFreeSlot.getNumber();
    } else {
      message += "Compact is full";
    }
    message += System.lineSeparator();

    if(largeFreeSlot.IsFree()){
      message += "Free Large: " + largeFreeSlot.getNumber();
    } else {
      message += "Large is full";
    }
    message += System.lineSeparator();

    if(motorbikeFreeSlot.IsFree()){
      message += "Free Motorbike: " + motorbikeFreeSlot.getNumber();
    } else {
      message += "Motorbike is full";
    }
    message += System.lineSeparator();

    if(electricFreeSlot.IsFree()){
      message += "Free Electric: " + electricFreeSlot.getNumber();
    } else {
      message += "Electric is full";
    }

    Show(message);
  }
}
```

ParkingLot: Our system will have only one object of this class. This can be enforced by using the Singleton pattern. In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object.

```
public class ParkingLot {
  private String name;
  private Location addres;
  private ParkingRate parkingRate;
```

```
  private int compactSlotCount;
  private int largeSlotCount;
  private int motorbikeSlotCount;
  private int electricSlotCount;
  private final int maxCompactCount;
  private final int maxLargeCount;
  private final int maxMotorbikeCount;
  private final int maxElectricCount;

  private HashMap<String, EntrancePanel> entrancePanels;
  private HashMap<String, ExitPanel> exitPanels;
  private HashMap<String, ParkingFloor> parkingFloors;

  // all active parking tickets, identified by their ticketNumber
  private HashMap<String, ParkingTicket> activeTickets;

  // singleton ParkingLot to ensure only one object of ParkingLot in the system,
  // all entrance panels will use this object to create new parking ticket:
getNewParkingTicket(),
  // similarly exit panels will also use this object to close parking tickets
  private static ParkingLot parkingLot = null;

  // private constructor to restrict for singleton
  private ParkingLot() {
    // 1. initialize variables: read name, address and parkingRate from database
    // 2. initialize parking floors: read the parking floor map from database,
    //  this map should tell how many parking slots are there on each floor. This
    //  should also initialize max slot counts too.
    // 3. initialize parking slot counts by reading all active tickets from database
    // 4. initialize entrance and exit panels: read from database
  }

  // static method to get the singleton instance of StockExchange
  public static ParkingLot getInstance()
  {
    if(parkingLot == null) {
      parkingLot = new ParkingLot();
    }
    return parkingLot;
  }

  // note that the following method is 'synchronized' to allow multiple entrances
  // panels to issue a new parking ticket without interfering with each other
  public synchronized ParkingTicket getNewParkingTicket(Vehicle vehicle)
    throws ParkingFullException {
    if(this.isFull(vehicle.getType())) {
      throw new ParkingFullException();
    }
    ParkingTicket ticket = new ParkingTicket();
    vehicle.assignTicket(ticket);
```

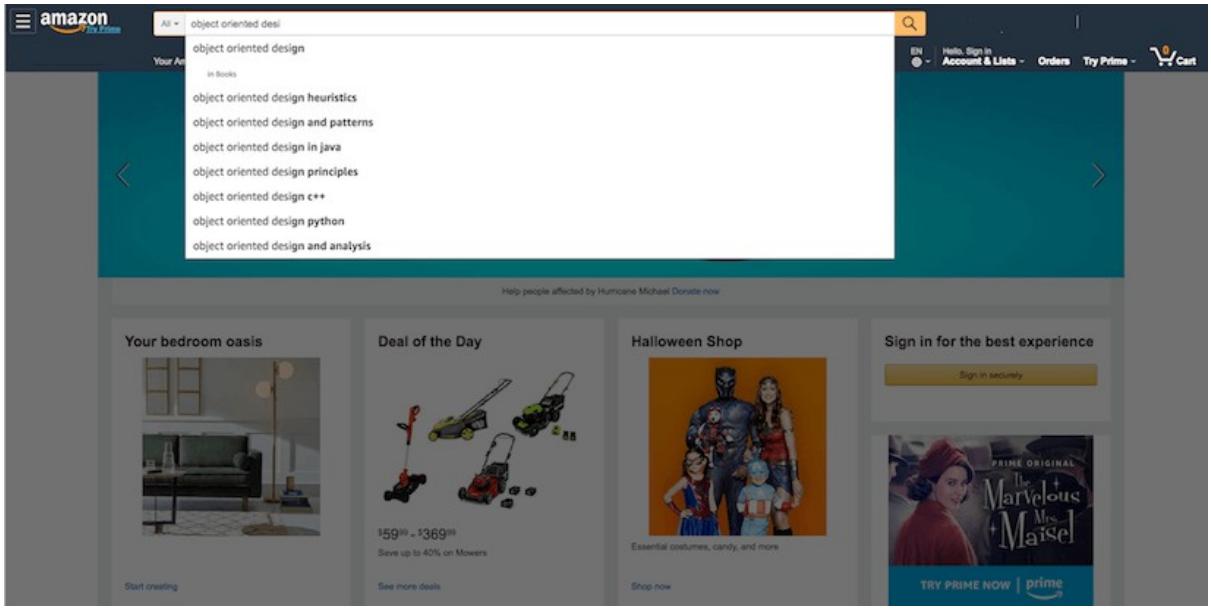
```
    ticket.saveInDB();
    // if the ticket is successfully saved in the database, we can increment the parking slot
count
    this.incrementSlotCount(vehicle.getType());
    this.activeTickets.put(ticket.getTicketNumber(), ticket);
    return ticket;
  }

  public boolean isFull(VehicleType type) {
    // trucks and vans can only be parked in LargeSlot
    if(type == VehicleType.Truck || type == VehicleType.Van) {
      return largeSlotCount >= maxLargeCount;
    }

    // moterbikes can only be parked at motorbike slots
    if(type == VehicleType.Moterbike) {
      return motorbikeSlotCount >= maxMotorbikeCount;
    }

    // cars can be parked at compact or large slots
    if(type == VehicleType.Car) {
      return (compactSlotCount + largeSlotCount) >= (maxCompactCount + maxLargeCount);
    }

    // electric car can be parked at compact, large or electric slots
    return (compactSlotCount + largeSlotCount + electricSlotCount)
      >= (maxCompactCount + maxLargeCount + maxElectricCount);
  }

  // increment the parking slot count based on the vehicle type
  private boolean incrementSlotCount(VehicleType type) {
    if(type == VehicleType.Truck || type == VehicleType.Van) {
      largeSlotCount++;
    } else if (type == VehicleType.Moterbike) {
      motorbikeSlotCount++;
    } else if (type == VehicleType.Car) {
      if(compactSlotCount < maxCompactCount) {
        compactSlotCount++;
      } else {
        largeSlotCount++;
      }
     } else { // electric car
      if(electricSlotCount < maxElectricCount) {
        electricSlotCount++;
      } else if(compactSlotCount < maxCompactCount) {
          compactSlotCount++;
      } else {
        largeSlotCount++;
      }
    }
```

```
  }

  public boolean isFull() {
    for (String key : parkingFloors.keySet()) {
      if(!parkingFloors.get(key).isFull()) {
        return false;
      }
    }
    return true;
  }

  public void addParkingFloor(ParkingFloor floor) { /* stores in database */ }
  public void addEntrancePanel(EntrancePanel entrancePanel) { /* stores in database */ }
  public void addExitPanel(ExitPanel exitPanel) { /* stores in database */ }
}
```

Design Amazon - Online Shopping System

Let's design and online retail store.

Amazon (amazon.com) is the world's largest online retailer. The company was originally a bookseller but has expanded to sell a wide variety of consumer goods and digital media. For the sake of this problem, we will focus on their online retail business where users can sell/buy their products.



Requirements and Goals of the System

We will be designing a system with the following requirements:

1. Users should be able to add new products for selling.
2. Users should be able to search products by their name or category.

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

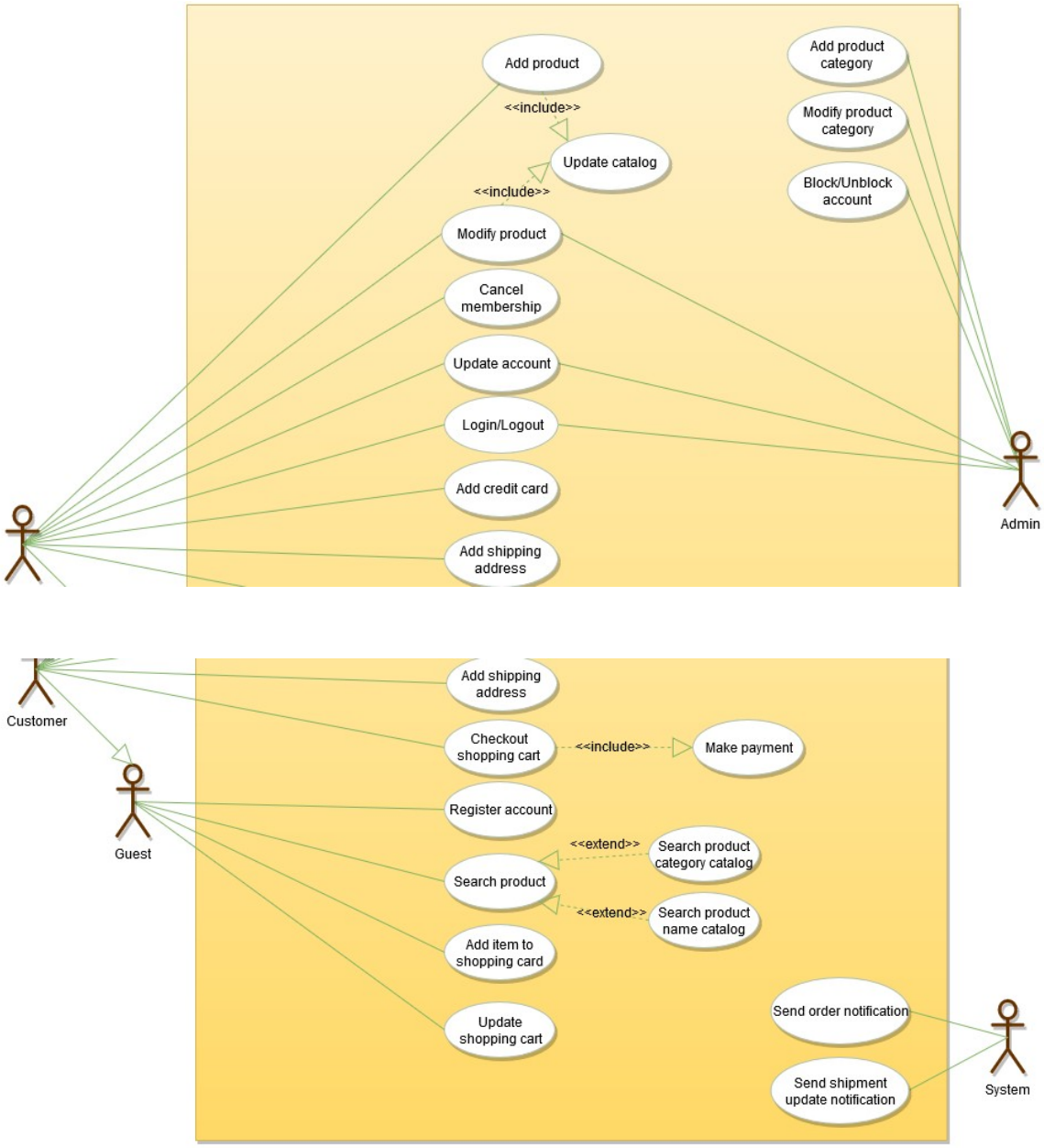
Use-case Diagram

We have four main Actors in our system:

- **Admin:** Mainly responsible for adding and modifying new product categories and account management.
- **Guest:** All guests can search the catalog, as well as add/remove items to the shopping cart, and become a registered member.
- **Member:** Members can perform all activities that guests can do, in addition to that customers can place orders and add new products for selling.
- **System:** Mainly responsible for sending notifications for order 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 be updating the catalog too.
2. Search products by their name or category.
3. Add/remove product items to the shopping cart.
4. Checkout to buy product items in the shopping cart.
5. Make payment to place order.
6. Add a new product category.
7. Send notification to members for shipment updates.

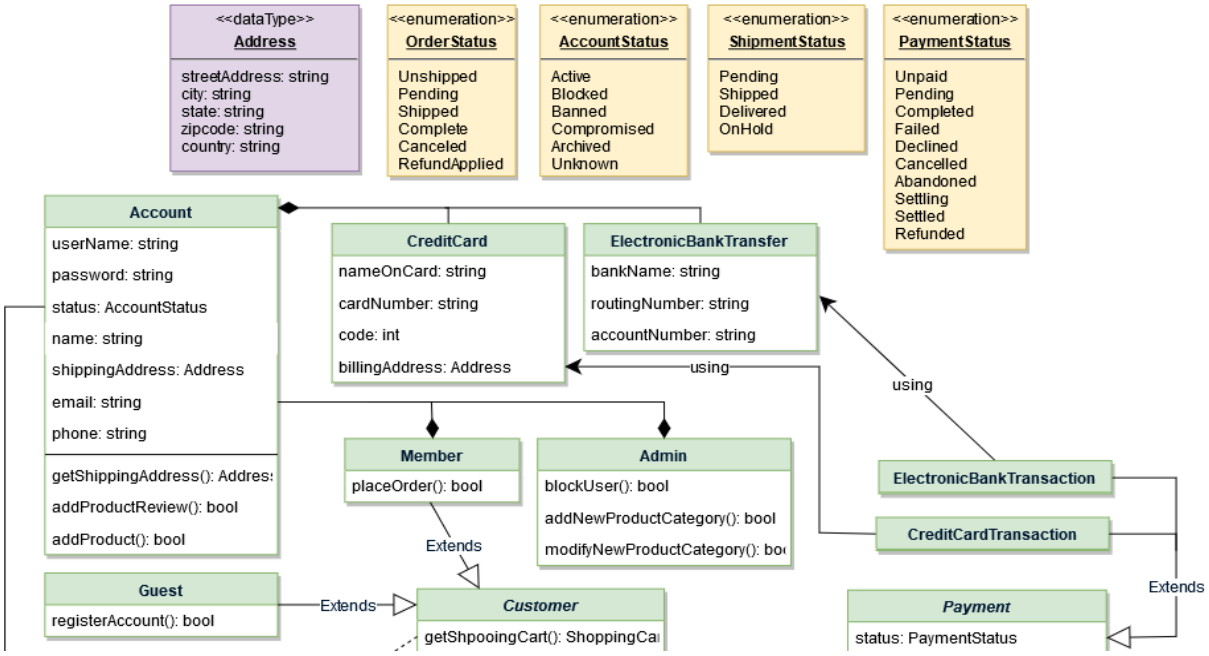


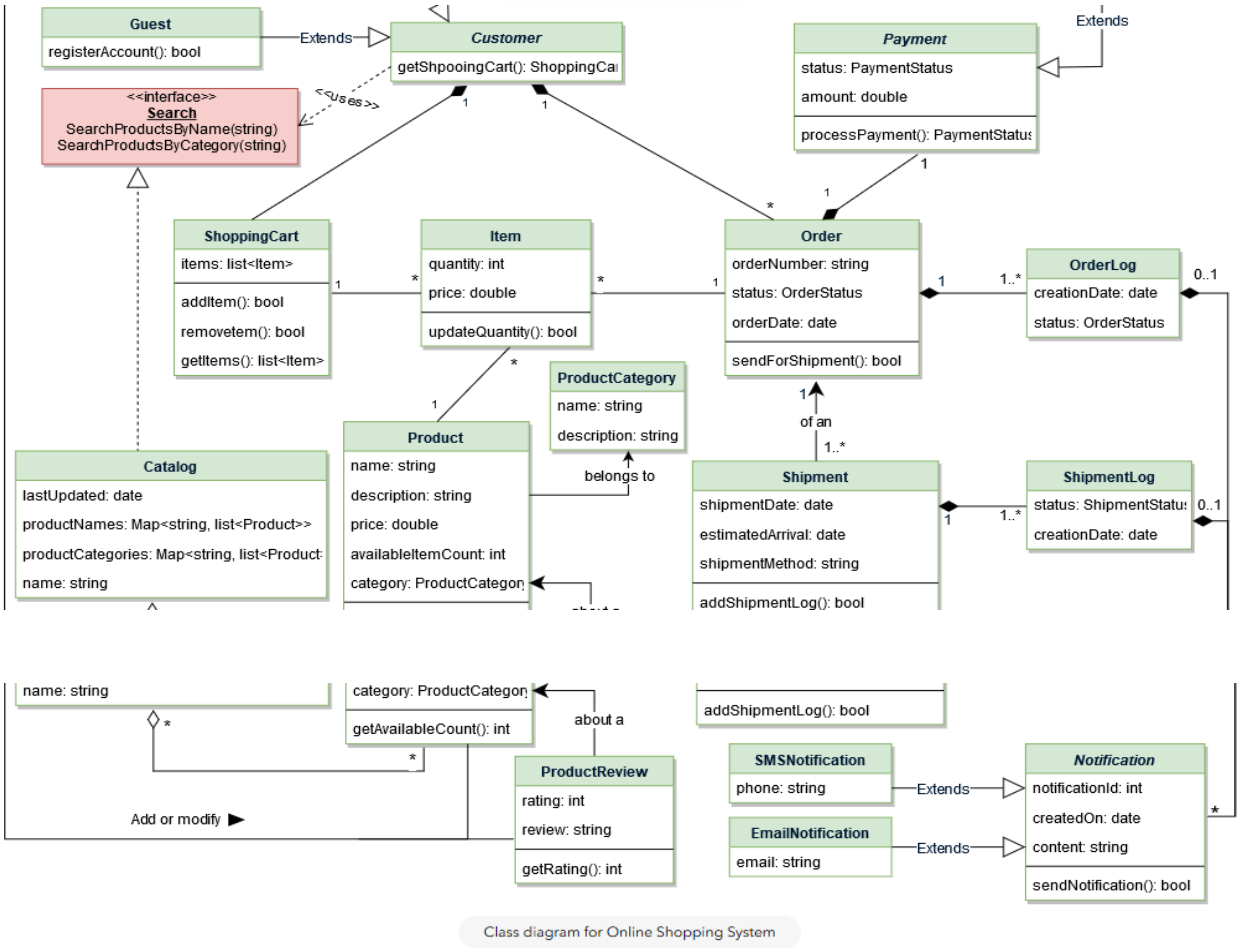
Class diagram

Here is the description of the different classes of our Online Shopping System:

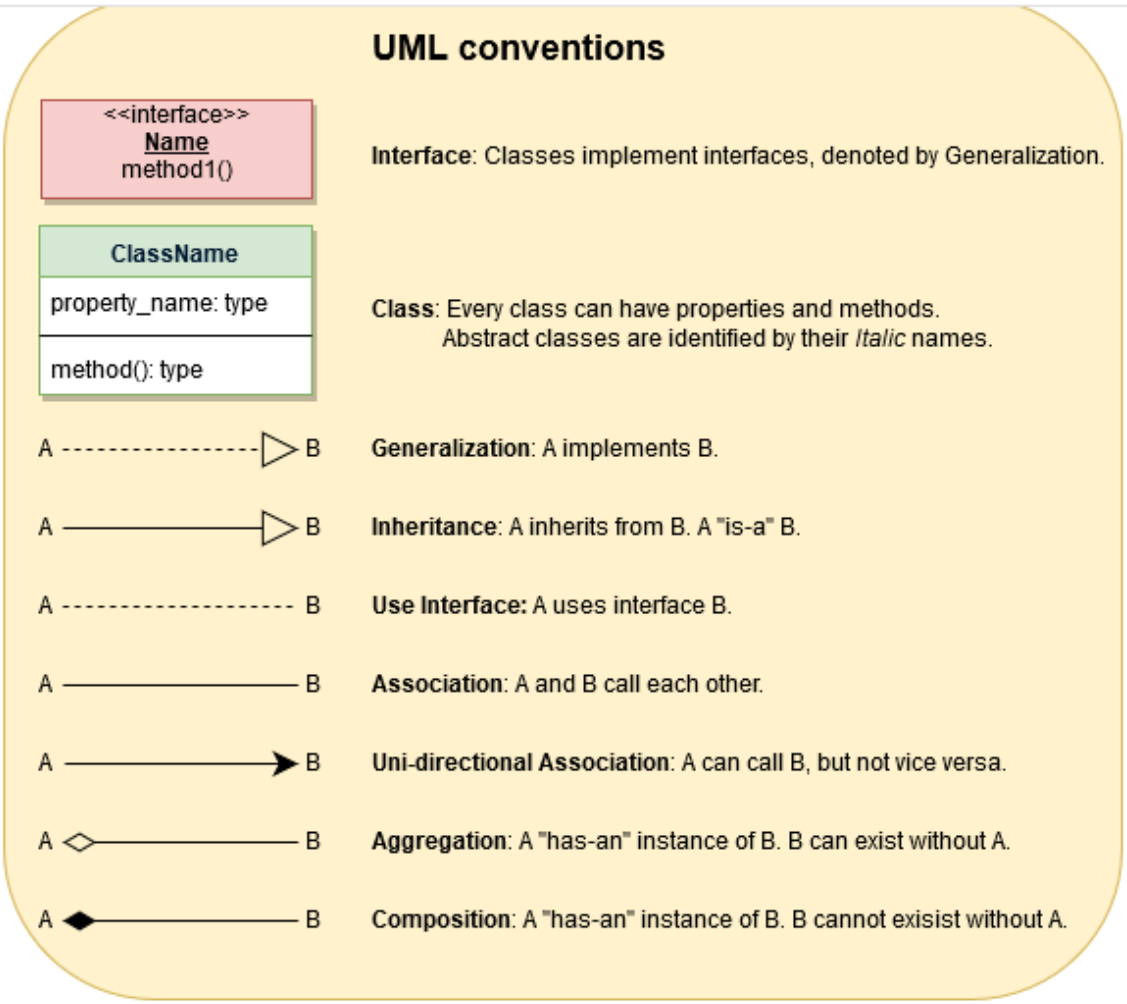
- **Account:** There are two types of registered accounts in the system, one will be an Admin who is responsible for adding new product categories and block/unblock members. The other account, a Member, can buy/sell products.
- **Guest:** Guests can search and view products and add them in the shopping cart. To place an order they have to become a registered member.
- **Catalog:** Users of our system can search products by their name or category. This class will keep an index of products for faster search.
- **ProductCategory:** To encapsulate the different categories of products like books, electronics, etc.

- **Product:** This class will encapsulate the entity that the users of our system will be selling and buying. 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 in the shopping cart.
- **Item:** To encapsulate a product item that the users will be buying and placing in the shopping cart.
- **Order:** To encapsulate a buying order to buy everything in the shopping cart.
- **OrderLog:** To keep a track of different statuses of the order like unshipped, pending, complete, canceled, etc.
- **ShipmentLog:** To keep a track of different statuses of the shipment like pending, shipped, delivered, etc.
- **Notification:** This class will take care of sending notifications to customers.
- **Payment:** This class will encapsulate the payment against an order. Members can pay through credit card or electronic bank transfer.



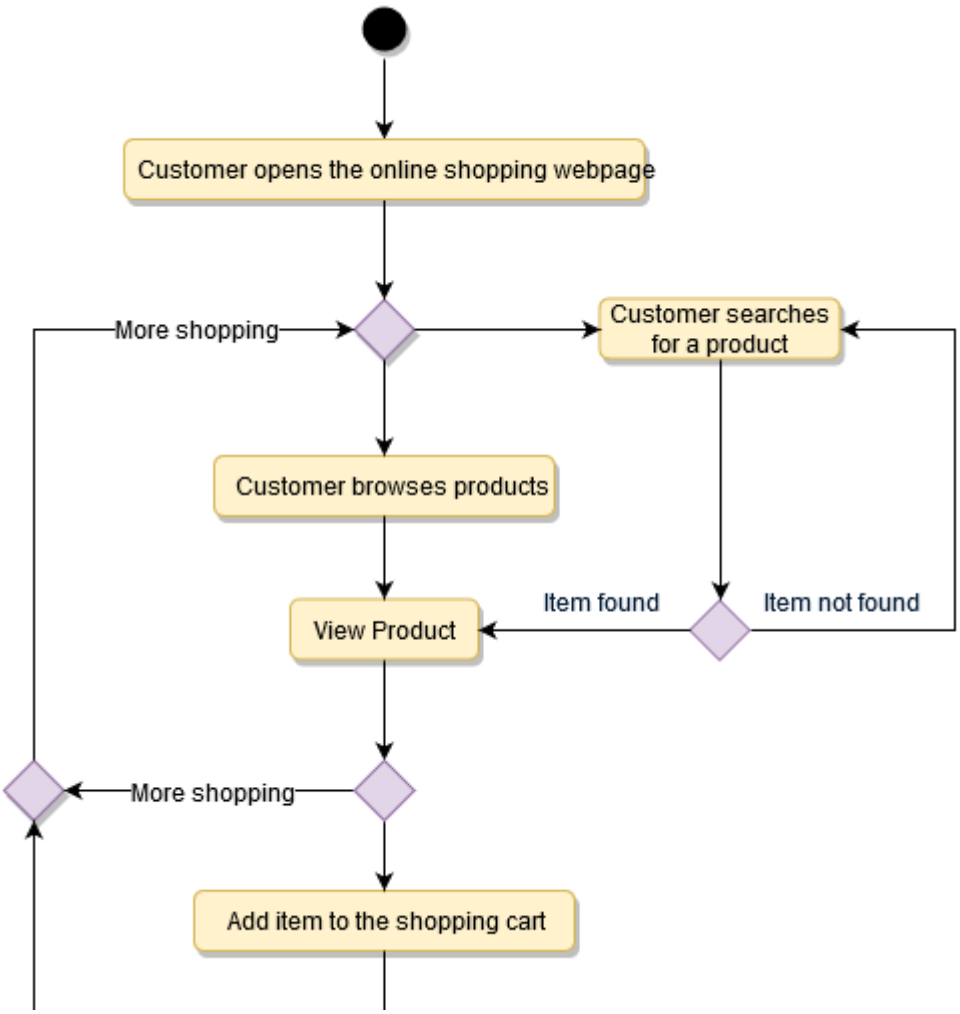


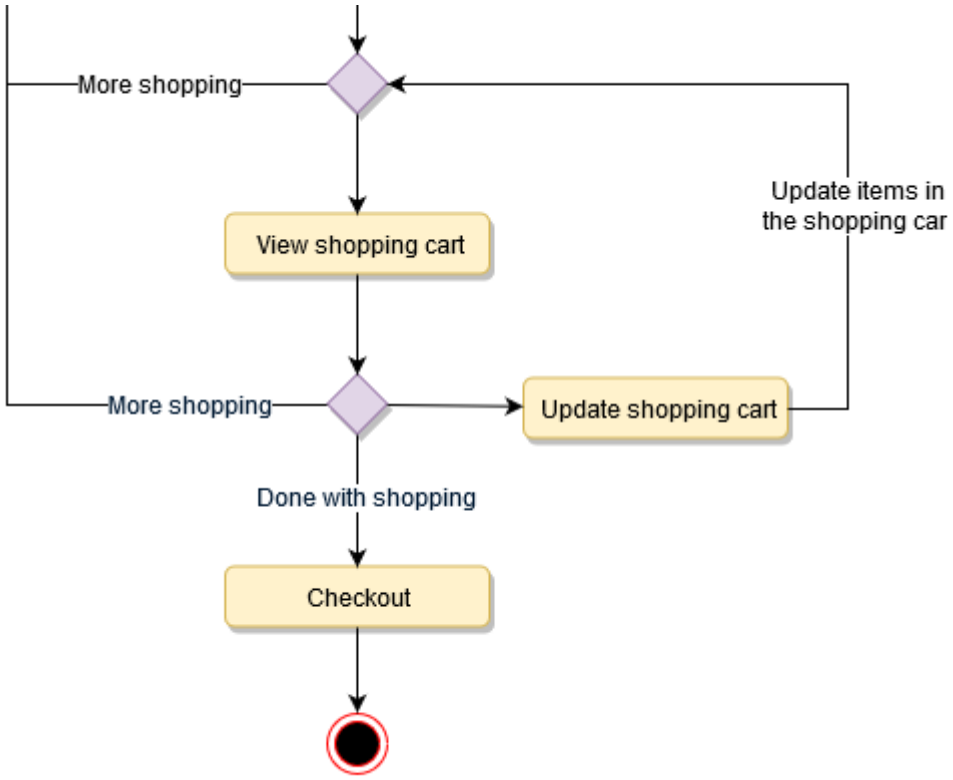
Class diagram for Online Shopping System



Activity Diagram

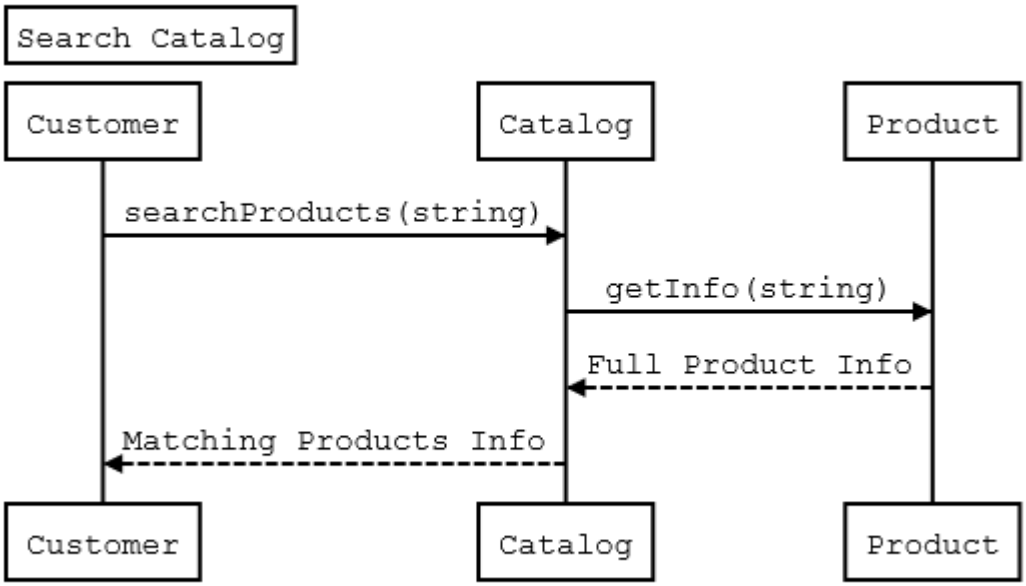
Following is the activity diagram for a user performing online shopping:


Customer opens the online shopping webpage
More shopping
Customer searches for a product
Customer browses products
Item found
Item not found
View Product
More shopping
Add item to the shopping cart

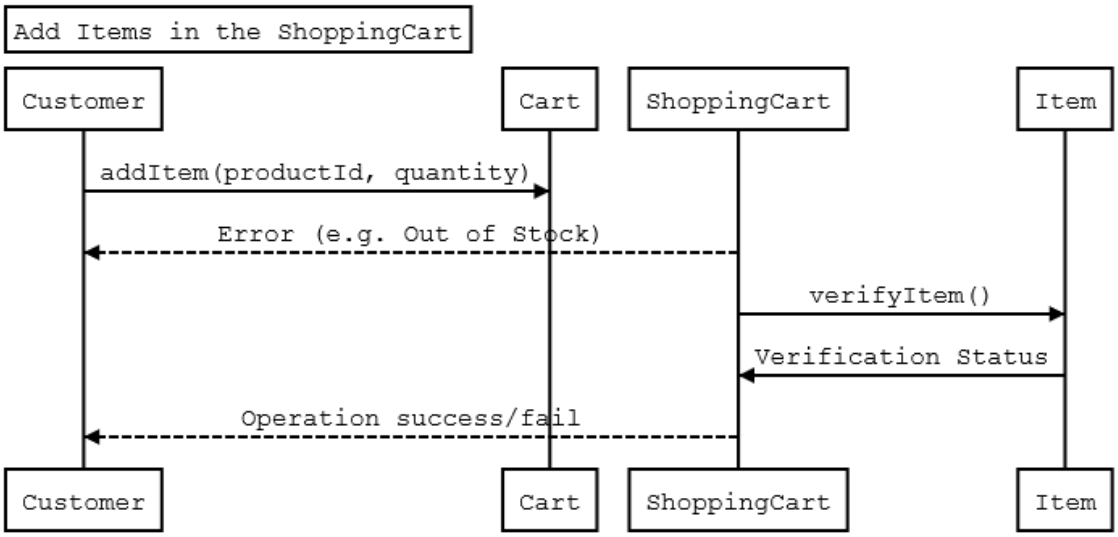


Sequence Diagram

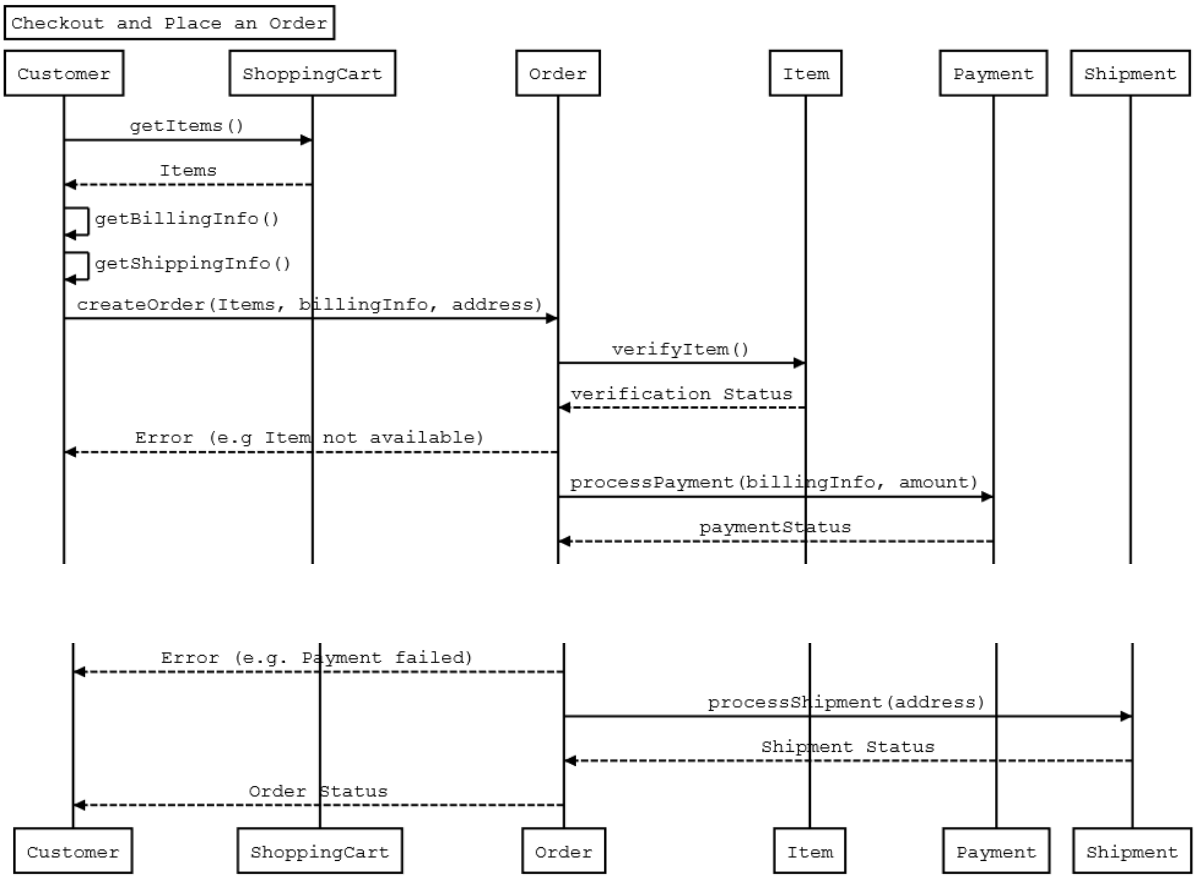
1. Here is the sequence diagram for searching from the catalog:



2. Here is the sequence diagram for adding an item to the shopping cart:



3. Here is the sequence diagram for checking-out to place an order:



Code

Here is the high-level definition for the classes described above.

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

```
public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}

public enum OrderStatus{
  UNSHIPPED,
  PENDING,
  SHIPPED,
  COMPLETED,
  CANCELED,
  REFUND_APPLIED
}

public enum AccountStatus{
  ACTIVE,
  BLOCKED,
  BANNED,
  COMPROMIZED,
  ARCHIVED,
  UNKNOWN
}

public enum ShipmentStatus{
  PENDING,
  SHIPPED,
  DELIVERED,
  ON_HOLD,
}

public enum PaymentStatus{
  UNPAID,
  PENDING,
  COMPLETED,
  FILLED,
  DECLINED,
  CANCELLED,
  ABONDENED,
  SETTLING,
  SETTLED,
  REFUNDED
}
```

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

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

public class Account {
  private String userName;
  private String password;
  private AccountStatus status;
  private String name;
  private Address shippingAddress;
  private String email;
  private String phone;

  private List<CreditCard> creditCards;
  private List<ElectronicBankTransfer> bankAccounts;

  public boolean addProduct(Product product);
  public boolean addProductReview(ProductReview review);
  public boolean resetPassword();
}

public abstract class Customer {
  private ShoppingCart cart;
  private Order order;

  public ShoppingCart getShoppingCart();
  public bool addItemToCart(Item item);
  public bool removeItemFromCart(Item item);
}

public class Guest extends Customer {
  public bool registerAccount();
}

public class Mamber extends Customer {
  private Account account;
  public OrderStatus placeOrder(Order order);
}
```

ProductCategory, Product, and ProductReview: Here are the classes related to a product:

```
public class ProductCategory {
  private String name;
  private String description;
}

public class ProductReview {
  private int rating;
  private String review;

  private Member reviewer;
```

```
}

public class Product {
  private String name;
  private String description;
  private double price;
  private ProductCategory category;
  private int availableItemCount;

  private Account seller;

  public int getAvailableCount();
  public boolean updatePrice(double newPrice);
}
```

ShoppingCart, Item, Order, and OrderLog: Users will put items in shopping cart and place order to buy all items in the cart.

```
public class Item {
  private int quantity;
  private double price;

  public boolean updateQuantity(int quantity);
}

public class ShoppingCart {
  private List<Items> items;

  public boolean addItem(Item item);
  public boolean removeItem(Item item);
  public boolean updateItemQuantity(Item item, int quantity);
  public List<Item> getItems();
  public boolean checkout();
}

public class OrderLog {
  private String orderNumber;
  private Date creationDate;
  private OrderStatus status;
}

public class Order {
  private String orderNumber;
  private OrderStatus status;
  private Date orderDate;
  private List<OrderLog> orderLog;

  public boolean sendForShipment();
```

```
  public boolean makePayment(Payment payment);
  public boolean addOrderLog(OrderLog orderLog);
}
```

Shipment, ShipmentLog, and Notification: After successfully placing an order, a shipment record will be created:

```
public class ShipmentLog {
  private String shipmentNumber;
  private ShipmentStatus status;
  private Date creationDate;
}

public class Shipment {
  private String shipmentNumber;
  private Date shipmentDate;
  private Date estimatedArrival;
  private String shipmentMethod;

  public boolean addShipmentLog(ShipmentLog shipmentLog);
}

public abstract class Notification {
  private int notificationId;
  private Date createdOn;
  private String content;

  public boolean sendNotification(Account account);
}
```

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

```
public interface Search {
  public List<Product> searchProductsByName(String name);
  public List<Product> searchProductsByCategory(String category);
}

public class Catalog implements Search {
   HashMap<String, List<Product>> productNames;
   HashMap<String, List<Product>> productCategories;

  public List<Product> searchProductsByName(String name) {
    return productNames.get(name);
  }

  public List<Product> searchProductsByCategory(String category) {
    return productCategories.get(category);
  }
}
```

Design Stack Overflow

Stack Overflow is one of the largest online community 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 and can receive badges for their valued contributions. Higher reputation let users unlock new privileges like the ability to vote, comment, and even edit other people's posts.

Requirements and Goals of the System

We will be designing a system with the following requirements:

1. Any non-member (guest) can search and view questions. 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 an extremely off-topic or of very low-quality questions. Moderator can delete/un-delete any question.
12. Moderators can close a question or undelete an already deleted question.
13. The system should also be able to tell most frequently used tags in the questions.

Use-case Diagram

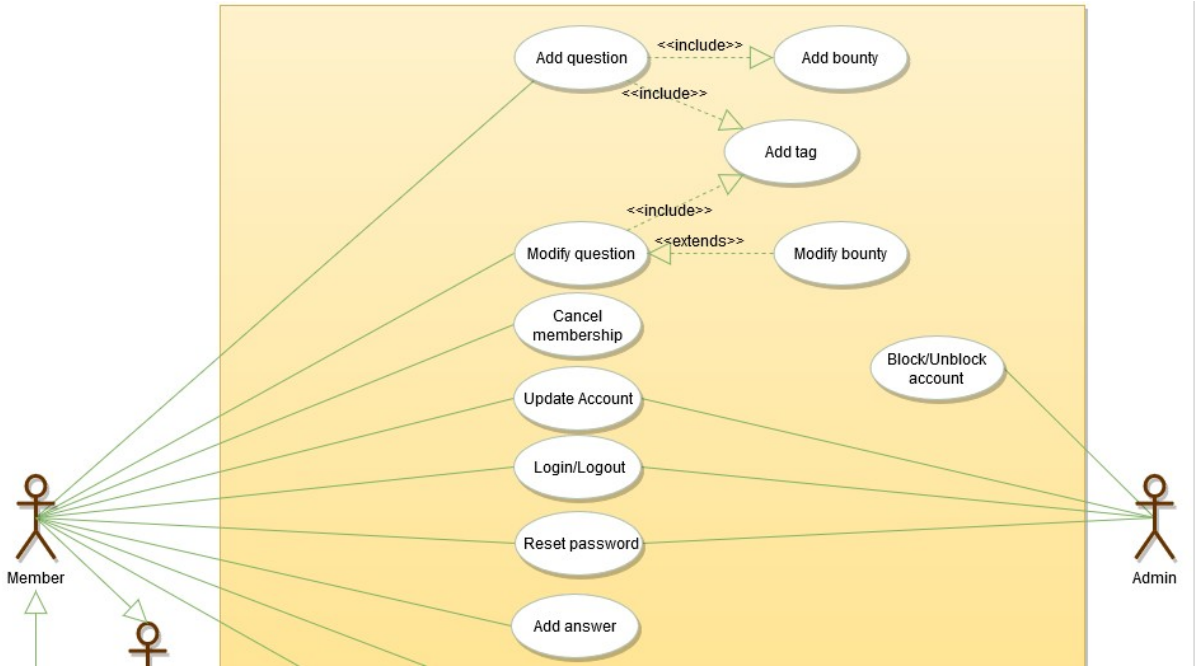
We have five main actors in our system:

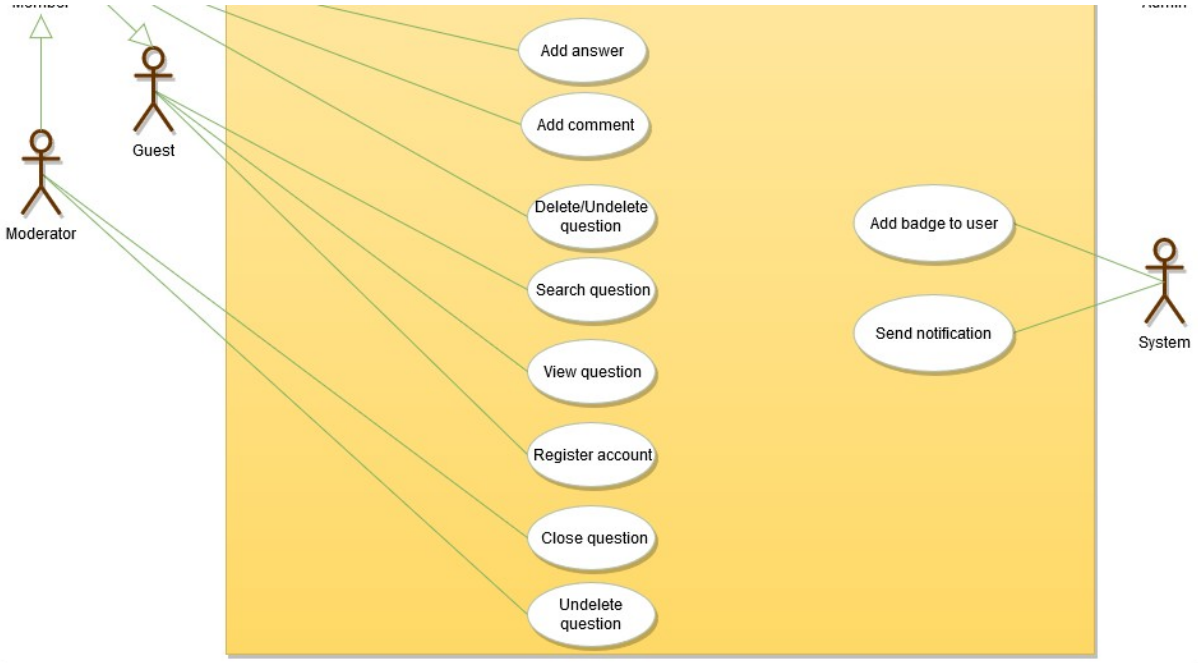
- **Admin:** Mainly responsible for blocking or unblocking member.

- **Guest:** All guests can search and view questions.
- **Member:** All members can perform all activities that guests can do, in addition to that customers can add/remove questions, answers, and comments. Members can delete and un-delete their questions, answers or comments.
- **Moderator:** In addition to all 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.



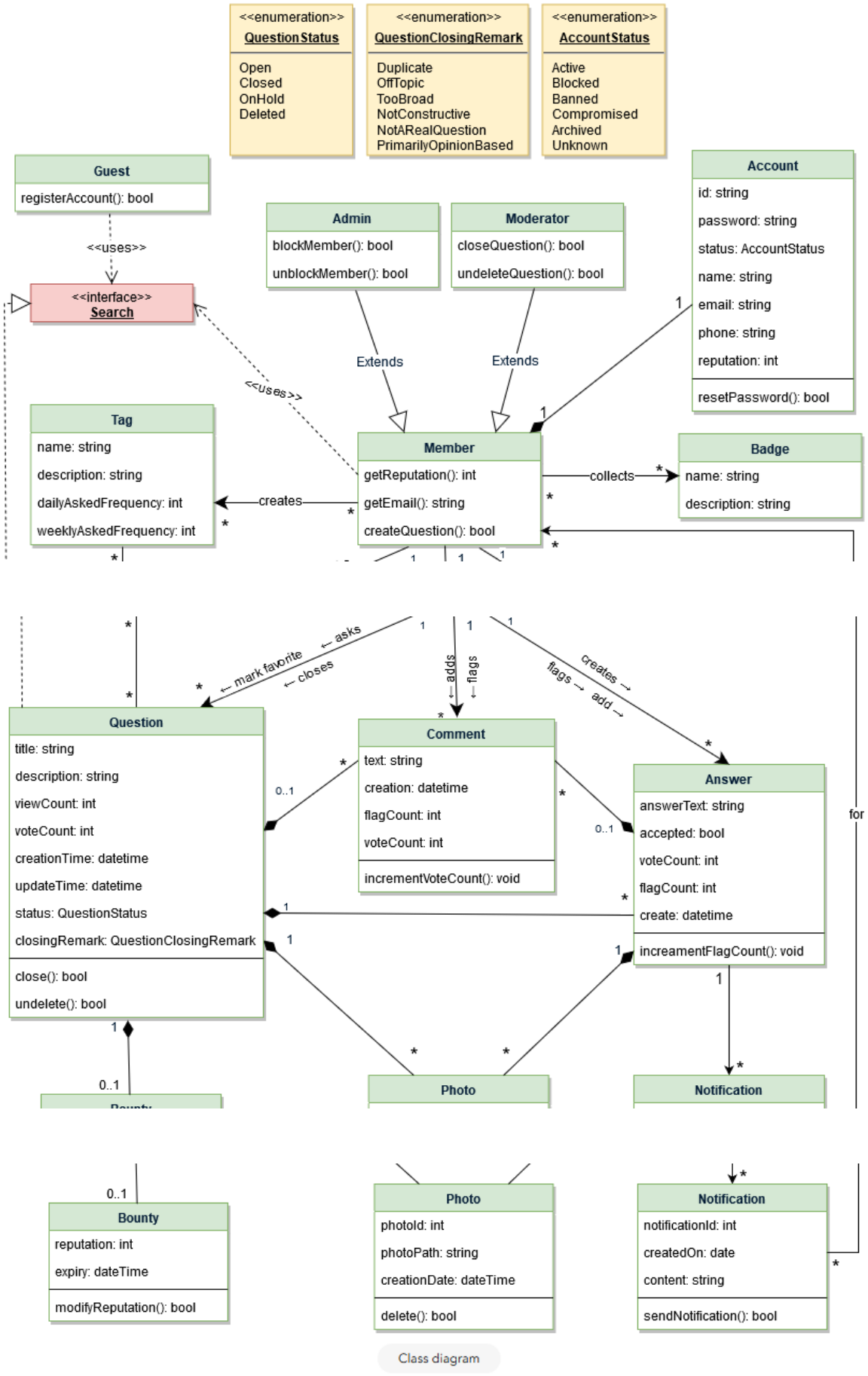


Use case diagram

Class diagram

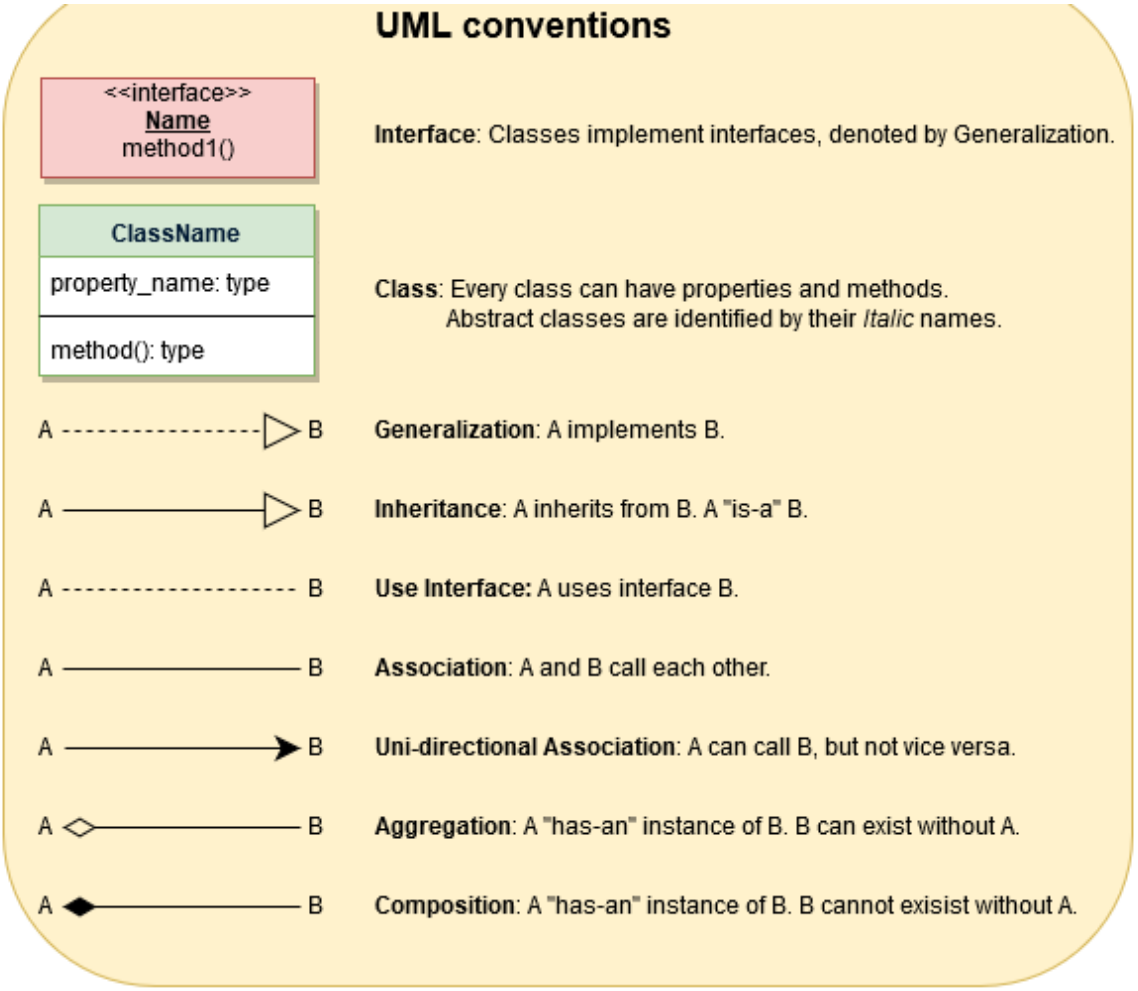
Here are the main classes of Stack Overflow System:

- **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 be tracking how many times the question has been viewed or voted. We should also be tracking the status of the question as well as closing remark 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 be tracking how many times an answer is voted 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 the 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:** Question or answers can have photos.
- **Bounty:** Each member while asking a question can put 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 take care of sending notifications to members and assigning badges to members based on their reputations.



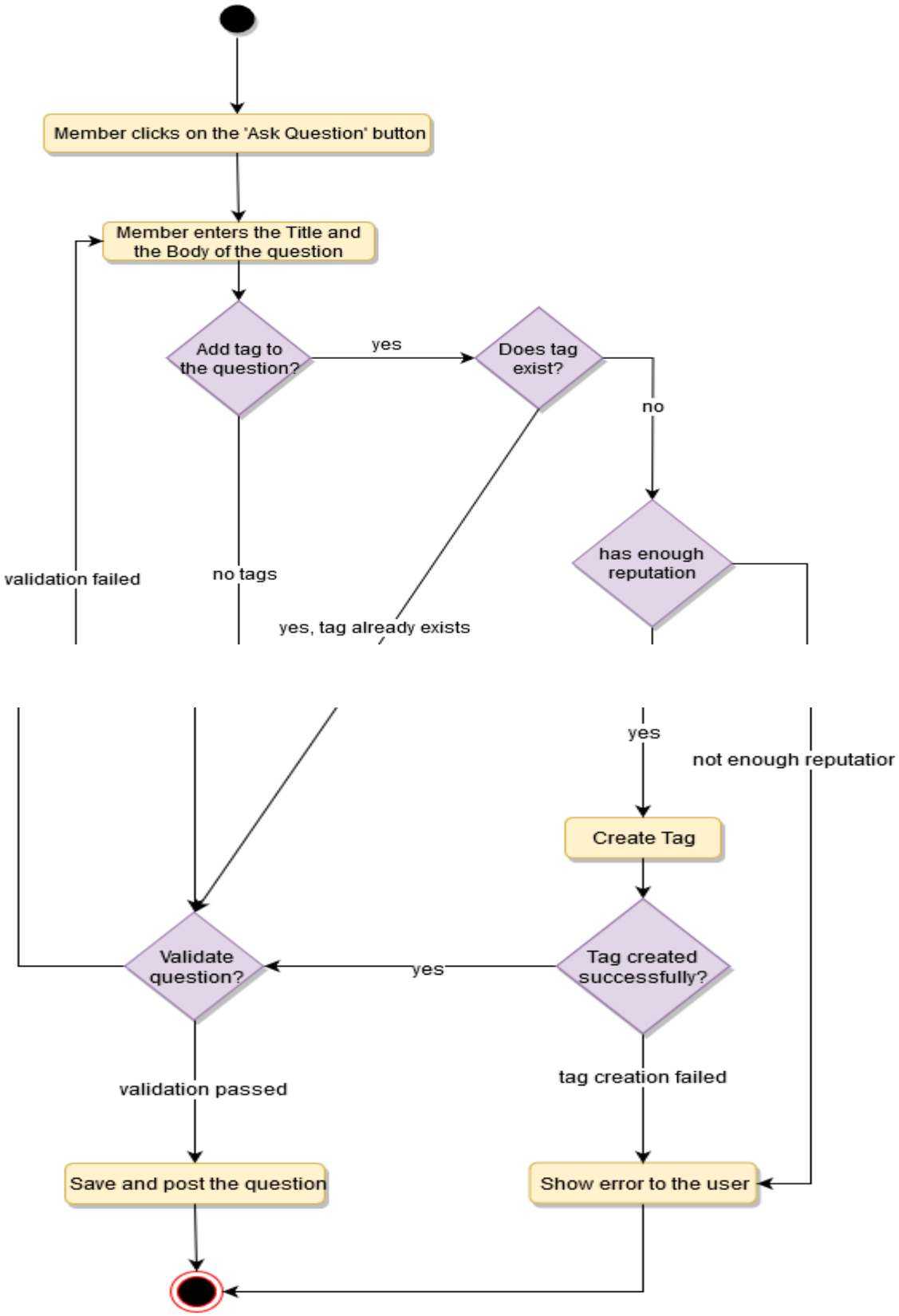
Class diagram

Class diagram



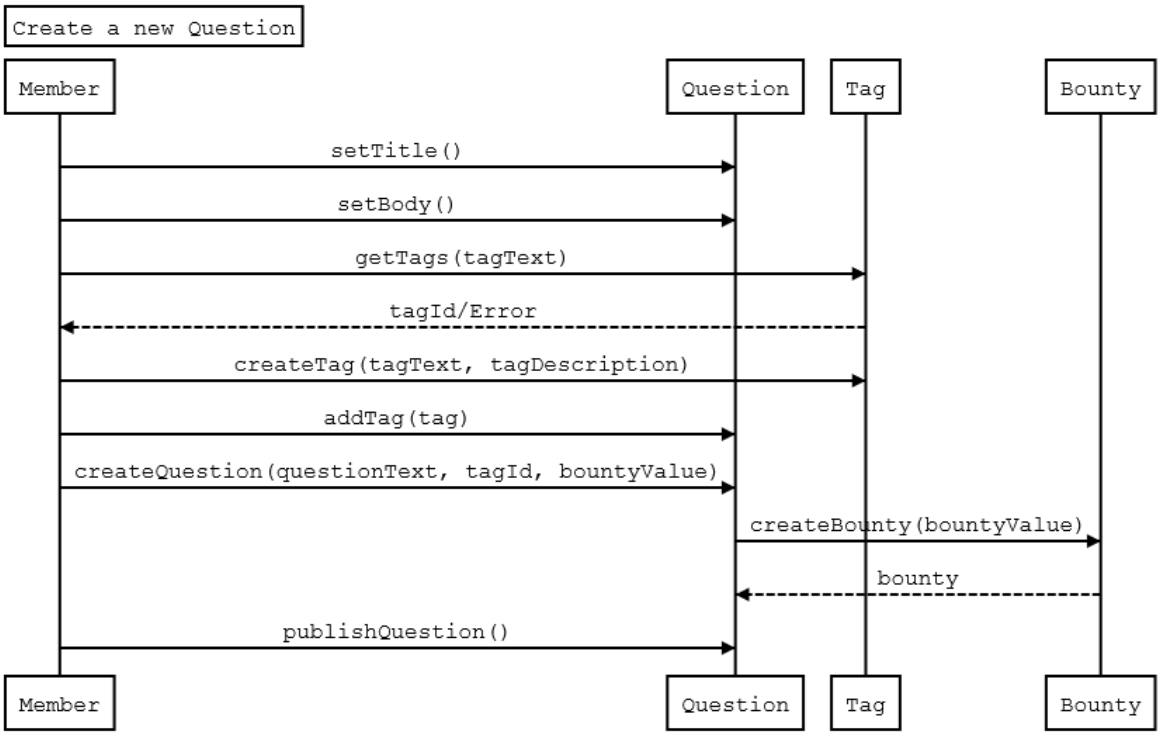
Activity diagrams

Post a new question: Any member or moderator can perform this activity. Here are the set of steps to place an order:



Sequence Diagram

Following is the sequence diagram for creating a new question:



Code

Here is the high-level definition for the classes described above.

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

```
public enum QuestionStatus{
  OPEN,
  CLOSED,
  ON_HOLD,
  DELETED
}

public enum QuestionClosingRemark{
  DUPLICATE,
  OFF_TOPIC,
  TOO_BROAD,
  NOT_CONSTRUCTIVE,
  NOT_A_REAL_QUESTION,
  PRIMARILY_OPINION_BASED
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
```

```
  BLOCKED
}
```

Account, Member, Admin, and Moderator: These classes represent different people that interact with our system:

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

public class Account {
  private String id;
  private String password;
  private AccountStatus status;
  private String name;
  private Address address;
  private String email;
  private String phone;
  private int reputation;

  public boolean resetPassword();
}

public class Member {
  private Account account;
  private List<Badge> badges;

  public int getReputation();
  public String getEmail();
  public boolean createQuestion(Question question);
  public boolean createTag(Tag tag);
}

public class Admin extends Member {
  public boolean blockMember(Member member);
  public boolean unblockMember(Member member);
}

public class Moderator extends Member {
  public boolean closeQuestion(Question question);
  public boolean undeleteQuestion(Question question);
}
```

Badge, Tag, and Notification: Members have badges, questions have tags and notifications:

```
public class Badge {
  private String name;
  private String description;
}

public class Tag {
  private String name;
```

```
  private String description;
  private long dailyAskedFrequency;
  private long weeklyAskedFrequency;
}

public class Notification {
  private int notificationId;
  private Date createdOn;
  private String content;

  public boolean sendNotification();
}
```

Photo and Bounty: Members can put bounties on questions. Answers and Questions can have multiple photos:

```
public class Photo {
  private int photoId;
  private String photoPath;
  private Date creationDate;

  private Member creatingMember;

  public boolean delete();
}

public class Bounty {
  private int reputation;
  private Date expiry;

  public boolean modifyReputation(int reputation);
}
```

Question, Comment and Answer: Members can ask questions, as well as add an answer to any question. All members can add comments to all open questions or answers:

```
public interface Search {
  public static List<Question> search(String query);
}

public class Question implements Search {
  private String title;
  private String description;
  private int viewCount;
  private int voteCount;
  private Date creationTime;
  private Date updateTime;
  private QuestionStatus status;
  private QuestionClosingRemark closingRemark;

  private Member askingMember;
  private Bounty bounty;
  private List<Photo> photos;
```

```
  private List<Comment> comments;
  private List<Answer> answers;

  public boolean close();
  public boolean undelete();
  public boolean addComment(Comment comment);
  public boolean addBounty(Bounty bounty);

  public static List<Question> search(String query) {
    // return all questions containing the string query in their title or description.
  }
}

public class Comment {
  private String text;
  private Date creationTime;
  private int flagCount;
  private int voteCount;

  private Member askingMember;

  public boolean incrementVoteCount();
}

public class Answer {
  private String answerText;
  private boolean accepted;
  private int voteCount;
  private int flagCount;
  private Date creationTime;

  private Member creatingMember;
  private List<Photo> photos;

  public boolean incrementVoteCount();
}
```

Design a Movie Ticket Booking System

A movie ticket booking system provides its customers the ability to purchase theatre seats online. E-ticketing systems allow the customers to browse through movies currently being played and to book seats, anywhere and anytime.

Requirements and Goals of the System

Our ticket booking service should meet the following requirements:

1. Our ticket booking service should be able to list down different cities where its affiliate cinemas are located.
2. Each cinema can have multiple halls and each hall can run a 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 user selects the movie, the service should display the cinemas running that movie and its available shows.
6. The user should be able to select the show at a particular cinema and book their tickets.
7. The service should be able to show the user the seating arrangement of the cinema hall. The user should be able to select multiple seats according to their preference.
8. The user should be able to distinguish between available seats and booked ones.
9. The system should be able to 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 through credit cards or cash.
11. The system should ensure that no two users can reserve the same seat.
12. Customers should be able to add a discount coupon to their payment.

Usecase diagram

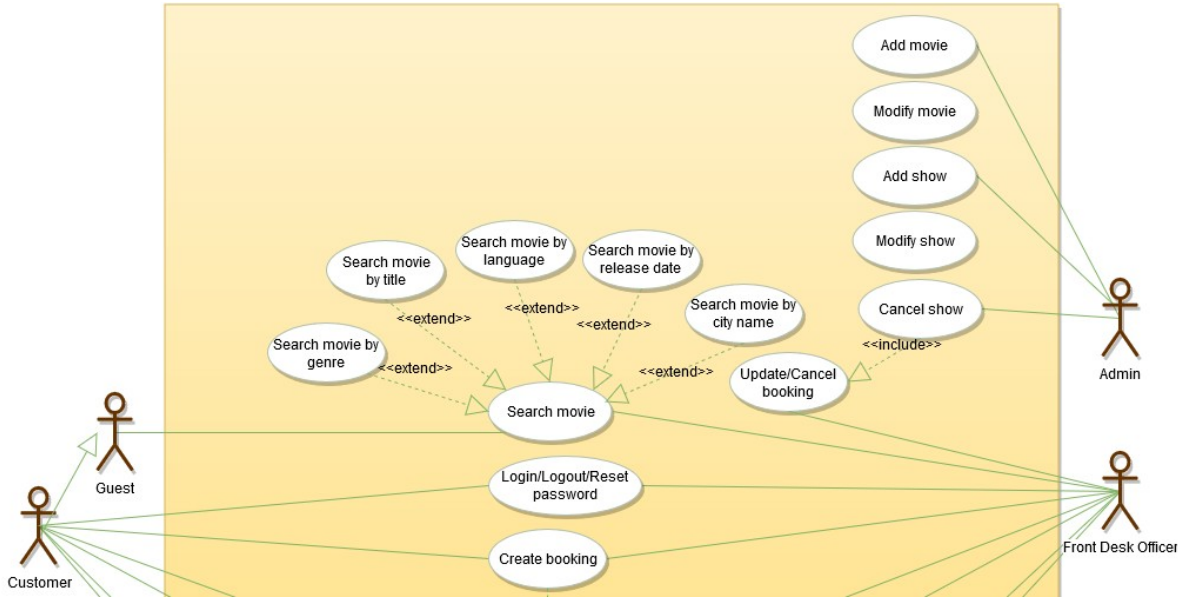
We have five main Actors in our system:

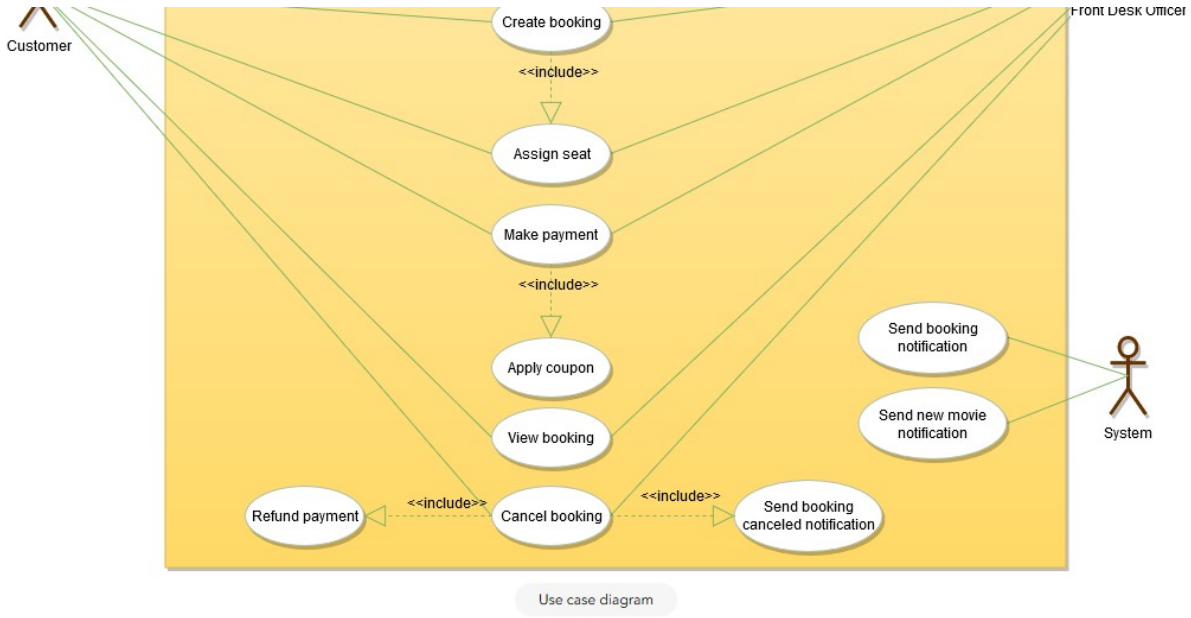
- **Admin:** Responsible for adding new movies and their shows, canceling any movie or show, blocking/unblocking users, etc.

- **FrontDeskOfficer:** Will be able to 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, booking, cancellations, etc.

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

- **Search movies:** Customers can 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 detail about the show.
- **Make payment for booking:** To pay for the booking.
- **Add a coupon to the payment:** Customers can add a discount coupon to their 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 amount if the cancellation is done within the allowed time frame.





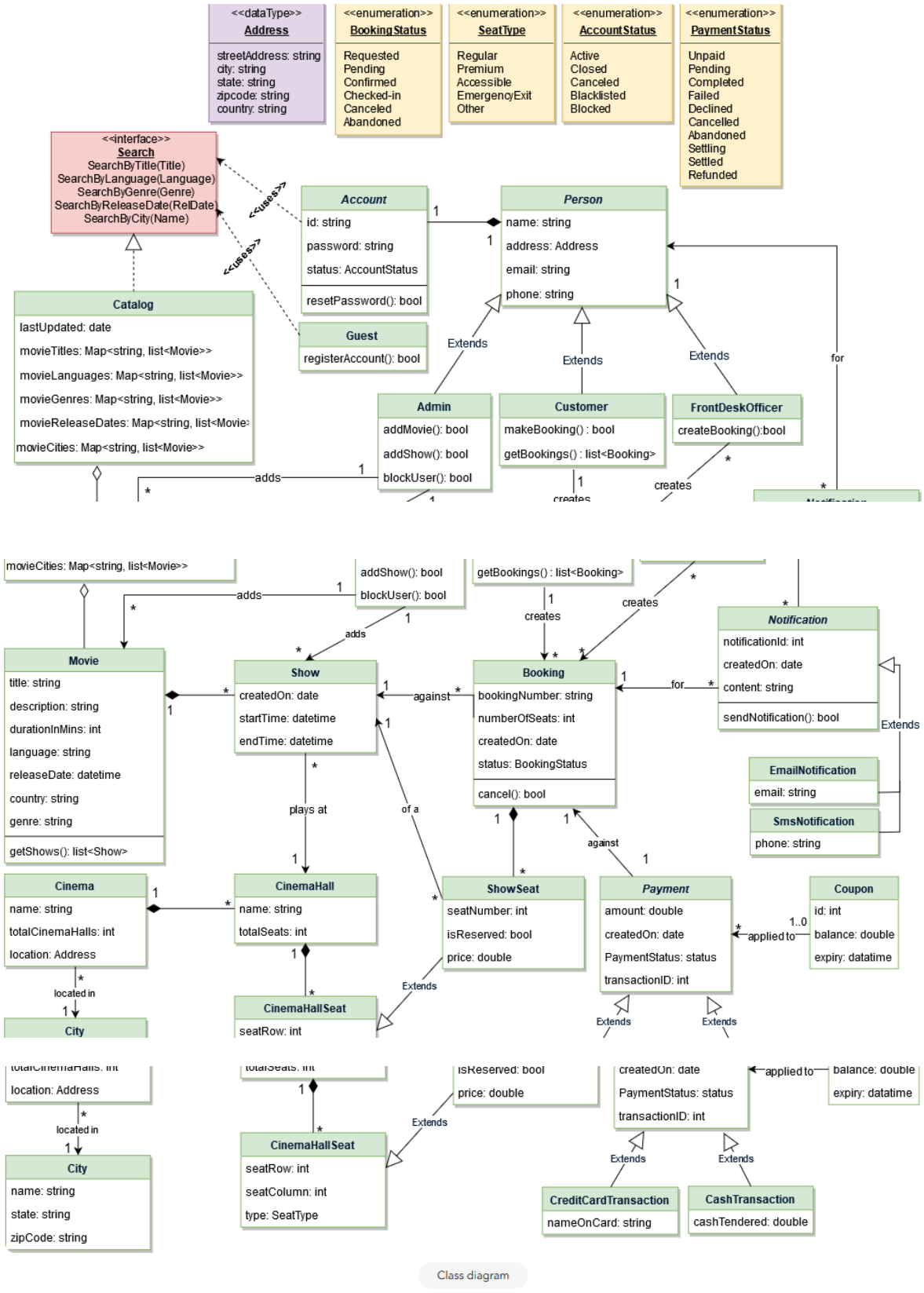
Use case diagram

Class diagram

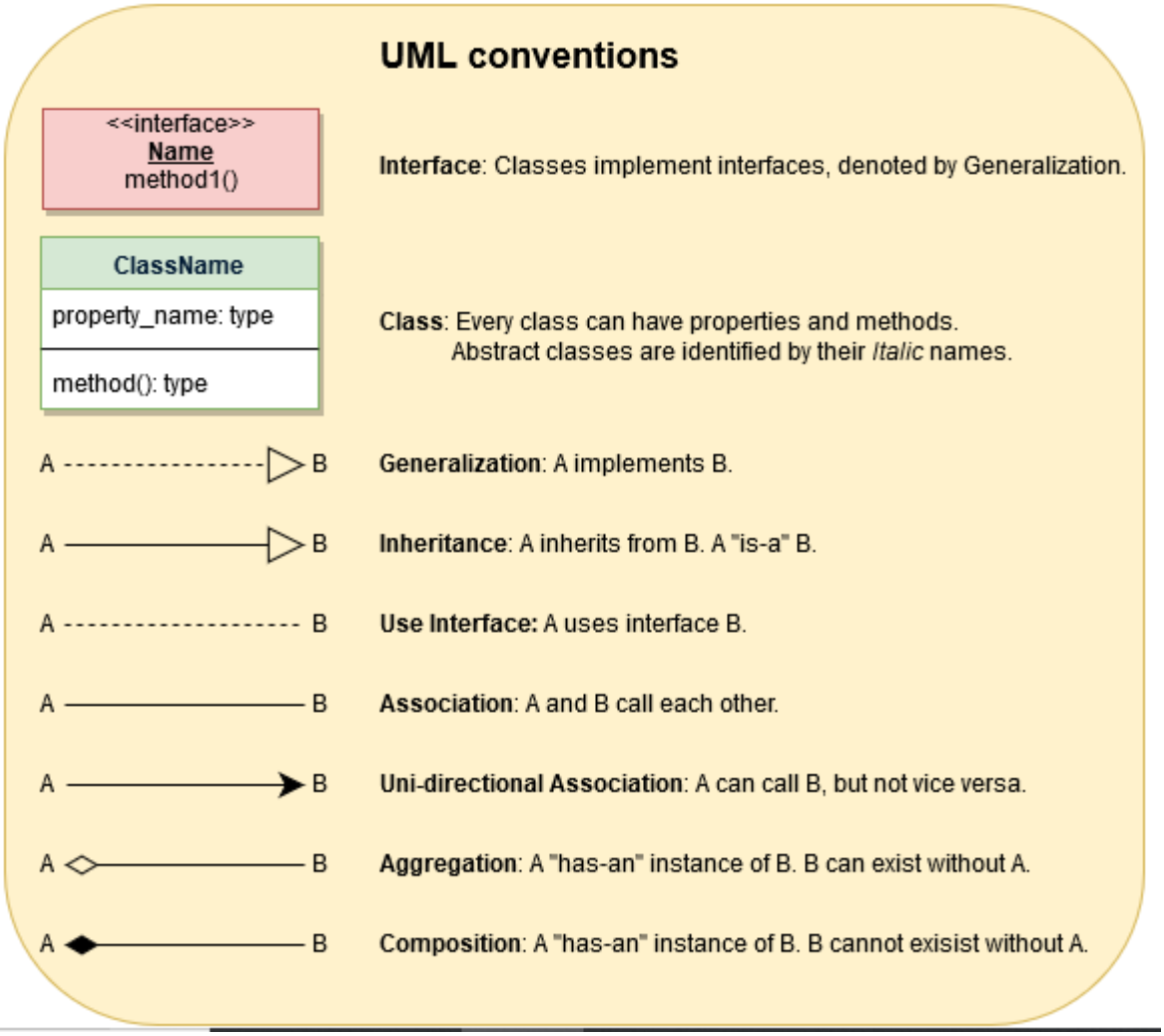
Here are the main classes of our Airline Management System:

- **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 movies for shows.
- **Guest:** Guests can search and view movies descriptions. To make a booking for a show they have to become a registered member.
- **Cinema:** The main part of the organization for which this software has been designed. It has attributes like 'name' to distinguish it from other cinemas.
- **CinemaHall:** Each cinema will have multiple halls containing multiple seats.
- **City:** Each City can have multiple Cinemas.
- **Movie:** The main entity of the system. Movies have attributes like title, description, language, genre, release date, city name, etc.
- **Show:** Any 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:** Will be responsible for collecting payments from customers.
- **Notification:** Will take care of sending notifications to customers.

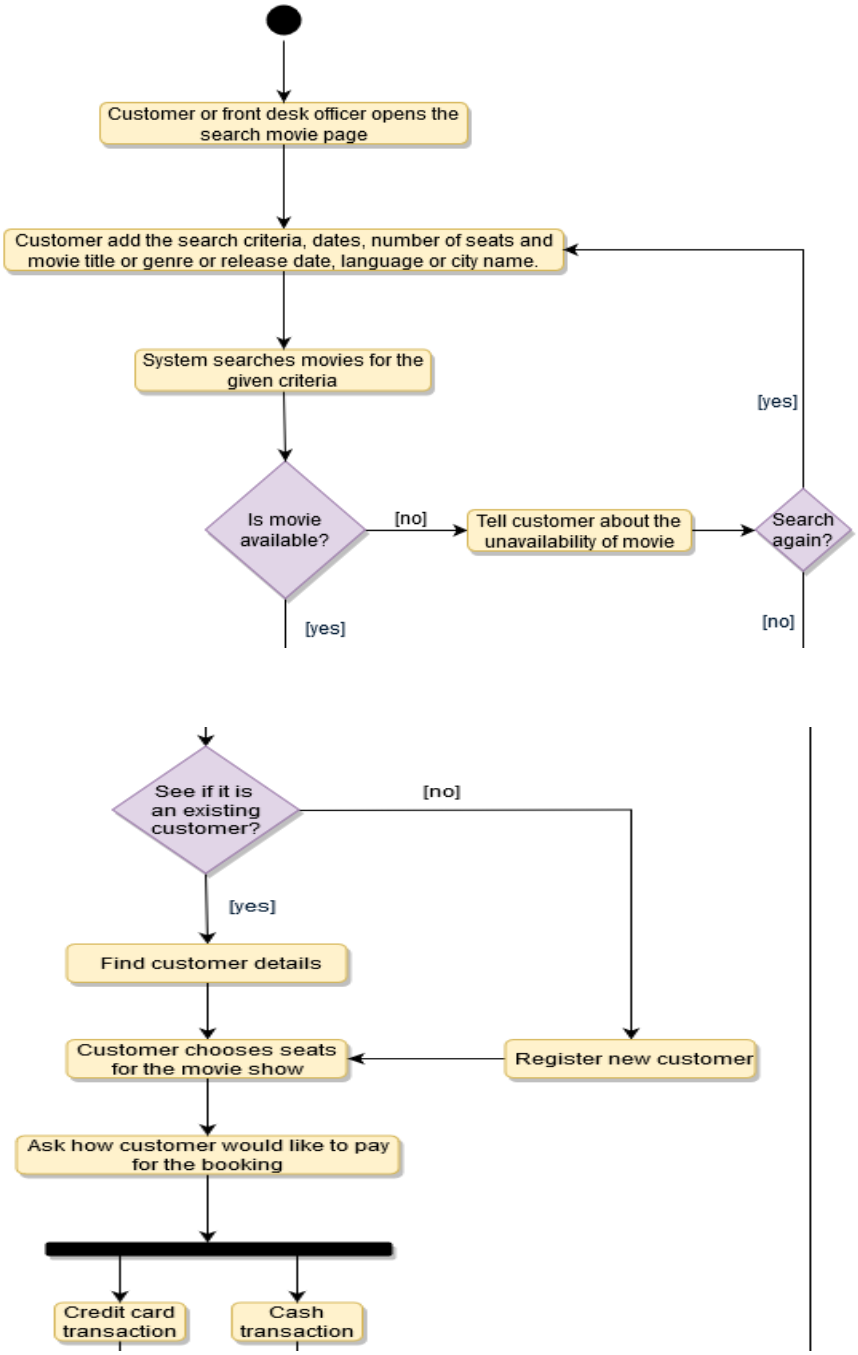


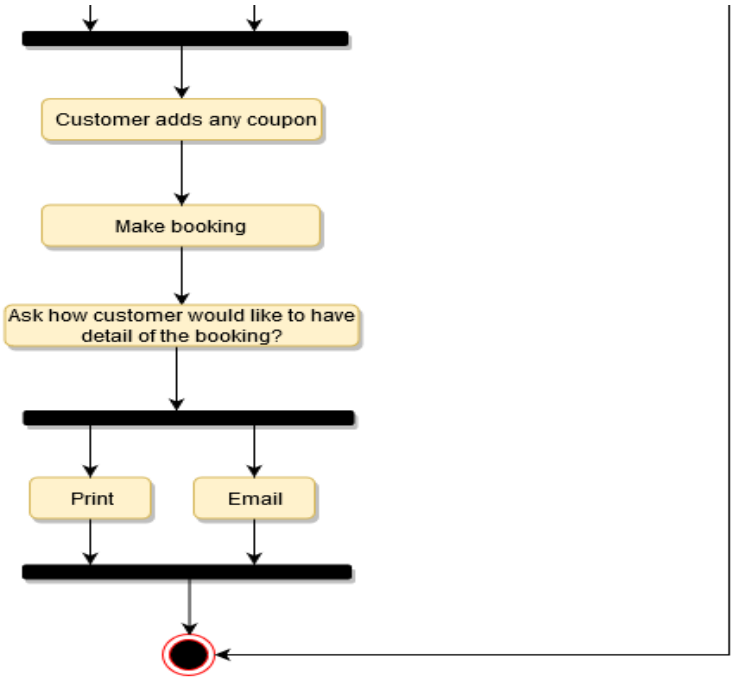
Class diagram



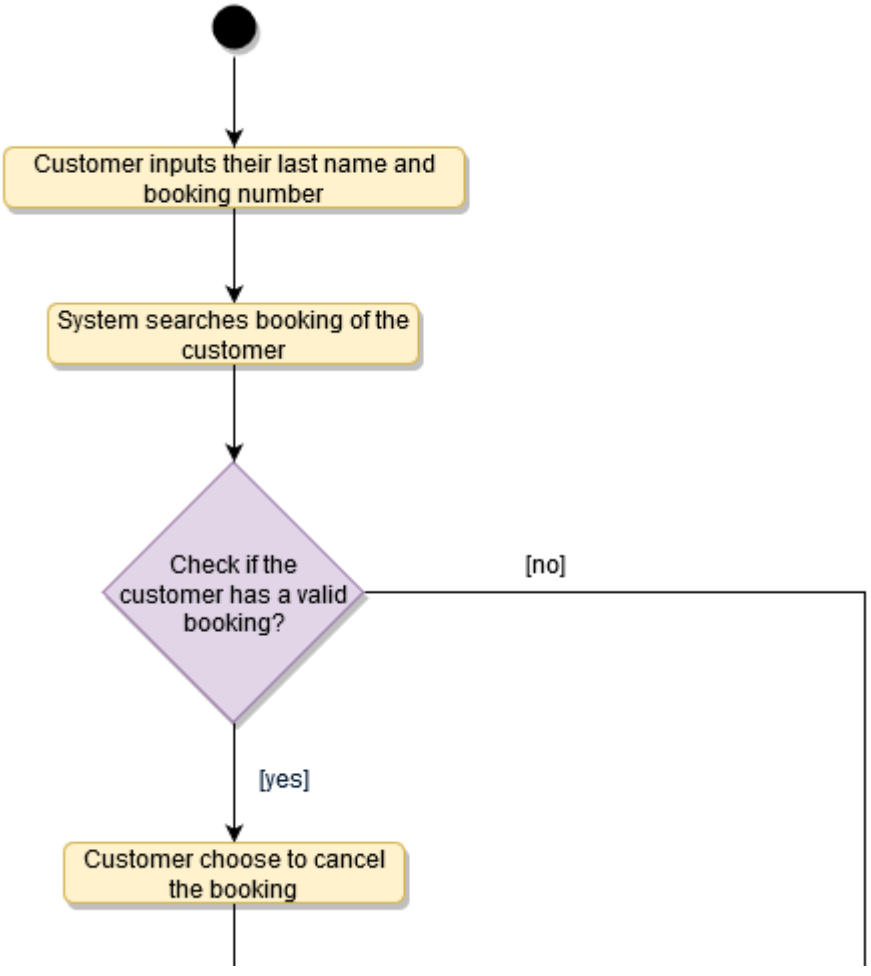
Activity Diagram

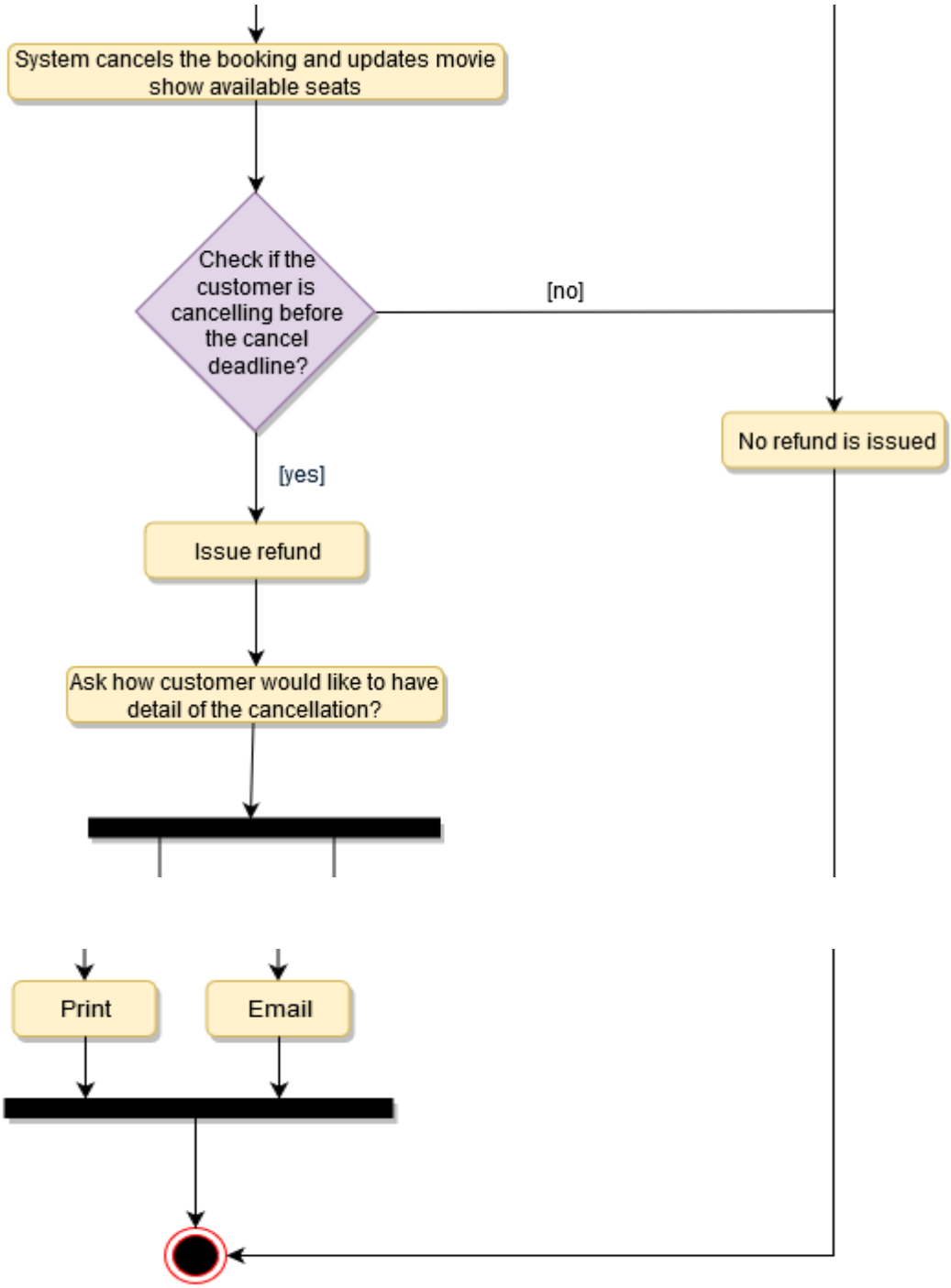
- **Make a booking:** Any customer can perform this activity. Here are the set of steps to book a ticket for a show:


Customer or front desk officer opens the search movie page
Customer add the search criteria, dates, number of seats and movie title or genre or release date, language or city name.
System searches movies for the given criteria
Is movie available?
[no]
Tell customer about the unavailability of movie
Search again?
[yes]
[no]
[yes]
See if it is an existing customer?
[no]
[yes]
Find customer details
Customer chooses seats for the movie show
Register new customer
Ask how customer would like to pay for the booking
Credit card transaction
Cash transaction



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





Code

Here is the high-level definition for the classes described above.

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

```
public enum BookingStatus{
  REQUESTED,
  PENDING,
  CONFIRMED,
```

```
  CHECKED_IN,
  CANCELED,
  ABONDENDED
}

public enum SeatType{
  REGULAR,
  PREMIUM,
  ACCESSIBLE,
  SHIPPED,
  EMERGENCY_EXIT,
  OTHER
}

public enum AccountStatus{
  ACTIVE,
  BLOCKED,
  BANNED,
  COMPROMIZED,
  ARCHIVED,
  UNKNOWN
}

public enum PaymentStatus{
  UNPAID,
  PENDING,
  COMPLETED,
  FILLED,
  DECLINED,
  CANCELED,
  ABONDENED,
  SETTLING,
  SETTLED,
  REFUNDED
}
```

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

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

public abstract class Account {
  private String id;
  private String password;
  private AccountStatus status;

  public boolean resetPassword();
}
```

```
public abstract class Person extends Account{
  private String name;
  private Address address;
  private String email;
  private String phone;
}

public class Customer extends Person {
  public boolean makeBooking(Booking booking);
  public List<Booking> getBookings();
}

public class Admin extends Person {
  public boolean addMovie(Movie movie);
  public boolean addShow(Show show);
  public boolean blockUser(Customer customer);
}

public class FrontDeskOfficer extends Person {
  public boolean createBooking(Booking booking);
}

public class Guest {
  public bool registerAccount();
}
```

Show and Movie: A movie will have many shows:

```
public class Show {
  private int showId;
  private Date createdOn;
  private Date startTime;
  private Date endTime;
  private CinemaHall playedAt;
  private Movie movie;
}

public class Movie {
  private String title;
  private String description;
  private int durationInMins;
  private String language;
  private Date releaseDate;
  private String country;
  private String genre;
  private Admin movieAddedBy;

  private List<Show> shows;
  public List<Show> getShows();
}
```

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

```
public class Booking {
  private String bookingNumber;
  private int numberOfSeats;
  private Date createdOn;
  private BookingStatus status;

  private Show show;
  private List<ShowSeat> seats;
  private Payment payment;

  public boolean makePayment(Payment payment);
  public boolean cancel();
  public boolean assignSeats(List<ShowSeat> seats);
}

public class ShowSeat extends CinemaHallSeat{
  private int showSeatId;
  private boolean isReserved;
  private double price;
}

public class Payment {
  private double amount;
  private Date createdOn;
  private int transactionId;
  private PaymentStatus status;
}
```

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

```
public class City {
  private String name;
  private String state;
  private String zipCode;
}

public class Cinema {
  private String name;
  private int totalCinemaHalls;
  private Address location;

  private List<CinemaHall> halls;
}

public class CinemaHall {
  private String name;
  private int totalSeats;
```

```
  private List<CinemaGallSeat> seats;
  private List<Show> shows;
}
```

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

```
public interface Search {
  public List<Movie> searchByTitle(String title);
  public List<Movie> searchByLanguage(String language);
  public List<Movie> searchByGenre(String genre);
  public List<Movie> searchByRekeaseDate(Date delDate);
  public List<Movie> searchByCity(String cityName);
}

public class Catalog implements Search {
   HashMap<String, List<Movie>> movieTitles;
   HashMap<String, List<Movie>> movieLanguages;
   HashMap<String, List<Movie>> movieGenres;
   HashMap<Date, List<Movie>> movieReleaseDates;
   HashMap<String, List<Movie>> movieCities;

  public List<Movie> searchByTitle(String title) {
    return movieTitles.get(name);
  }

  public List<Movie> searchByLanguage(String language) {
    return movieLanguages.get(language);
  }

  //...

  public List<Movie> searchByCity(String Date) {
    return movieCities.get(Date);
  }
}
```

Concurrency

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

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;


BEGIN TRANSACTION;
```

-- Suppose we intend to reserve three seats (IDs: 54, 55, 56) for ShowID=99

Select * From ShowSeat where ShowID=99 && ShowSeatID in (54, 55, 56) && isReserved=0

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

update ShowSeat table...

update Booking table ...

COMMIT TRANSACTION;

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

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

Here is the sample Java code:

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

public class Customer extends Person {

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

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

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

```
      PreparedStatement preparedStatement = dbConnection.prepareStatement(selectSQL);
      preparedStatement.setInt(1, booking.getShow().getShowId());
      Array array = dbConnection.createArrayOf("INTEGER", seatIds);
      preparedStatement.setArray(2, array);

      ResultSet rs = preparedStatement.executeQuery();
      // With TRANSACTION_SERIALIZABLE all the read rows will have the write lock, so
we can
      // safely assume that no one else is modifying them.
      if (rs.next()) {
        rs.last(); // move to the last row, to calculate the row count
        int rowCount = rs.getRow();
        // check if we have expected number of rows, if not, this means another process is
        // trying to process at least one of the same row, if that is the case we
        // should not process this booking.
        if(rowCount == seats.size()) {
          // update ShowSeat table...
          // update Booking table ...
          dbConnection.commit();
          return true;
        }
      }
    } catch (SQLException e) {
      dbConnection.rollback();
      System.out.println(e.getMessage());
    }
    return false;
  }
}
```

Read JDBC Transaction Isolation Levels for details.

Design an ATM

ATM is a part of our life, which helps us in daily transactions and business. 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. At this time, the ATM provides the people with good services especially the people can get money at any time. We need the ATM system because not all the bank branches are open all days of the week, and some of the customers may not be in a situation to visit the bank every time they want to withdraw money or deposit money.

ATM

Requirements and Goals of the System

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

1. **Card reader:** to read the users ATM-cards.
2. **Keypad:** to enter the information to the ATM e.g. PIN. cards.
3. **Screen:** to display the messages to the users.
4. **Cash dispenser:** for dispensing cash.
5. **Deposit slot:** to deposit cash or checks from the users.
6. **Printer:** for printing the 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 account 1) Checking and 2) Saving, and should be able to perform following five transactions on the ATM:

1. **Balance inquiry:** To see how what funds are in each account.
2. **Deposit cash:** To deposit some cash.
3. **Deposit check:** To deposit a check.
4. **Withdraw cash** To withdraw some money from the checking account.
5. **Transfer funds:** To transfer funds to another account.

How ATM works?

The ATM will be managed by an operator, who operates the ATM and refills it with cash and receipts. The ATM will be serving one customer at a time and while serving it should not shut down. To start any transaction in the ATM, the user should insert the ATM card which contains user's account information, after that the user should enter the Personal

Identification Number (PIN) for authentication. After this, the ATM will send the user information to the bank to gain the authentication from the bank; without authentication, the user cannot perform any transaction/service.

The user's ATM card will be kept in the ATM until the user requisites against that, e.g., the user can press the cancel button at any time, and the ATM Card will be ejected. The ATM will maintain an internal log of transactions that contains information about hardware failures; this log will be used by the ATM operator to resolve any issues.

1. Identify the system user through the PIN number.
2. In the case of depositing checks the amount of this check will not be added instantly to the user account, it is subject to manual verification and bank approval.
3. It is assumed that the Bank manager will have access to ATM system information that stored in the Bank database, and this information is updated automatically.
4. It is assumed that the user deposits will not be added to their account immediately because it will be subject to verification by the bank.
5. It is assumed the ATM card is the main player when it comes to security; users will authenticate themselves with their debit card and security pin.

Use Cases

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

Operator: operator will be responsible for the following operations:

1. Turning the ATM in ON/OFF status using the designated Key-Switch.
2. To refill the ATM with cash.
3. Refill ATM's printer with receipts.
4. Assumed to refill ATM's printer with INK.
5. Takeout deposited cash and checks.

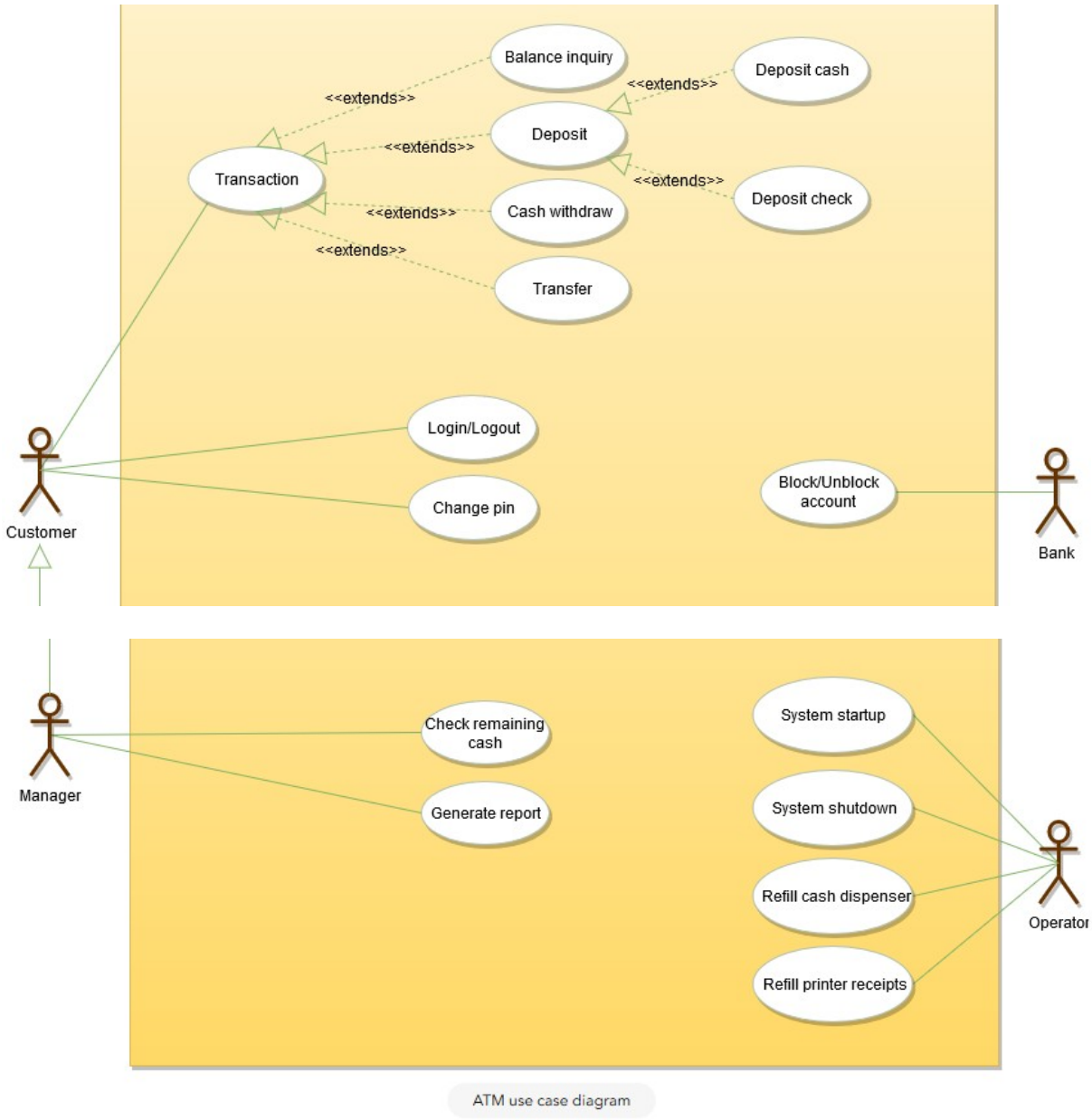
Customer: The ATM customer can perform the following operations:

1. Balance inquiry: the user can view his/her account balance.
2. Cash withdraw: 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 the other accounts.

Bank Manager:

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

Here is the use case diagram of our ATM system:



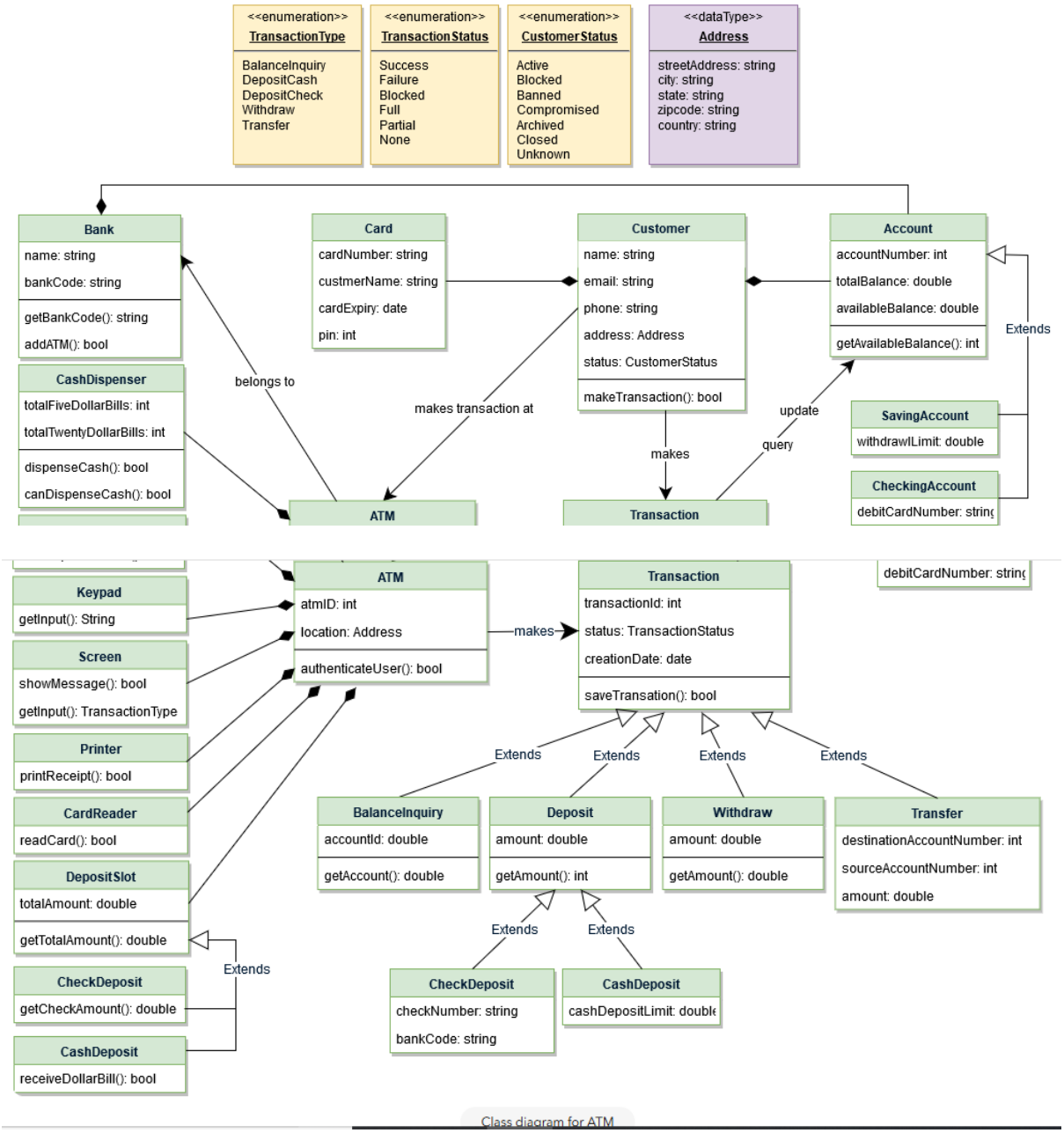
ATM use case diagram

Class diagram

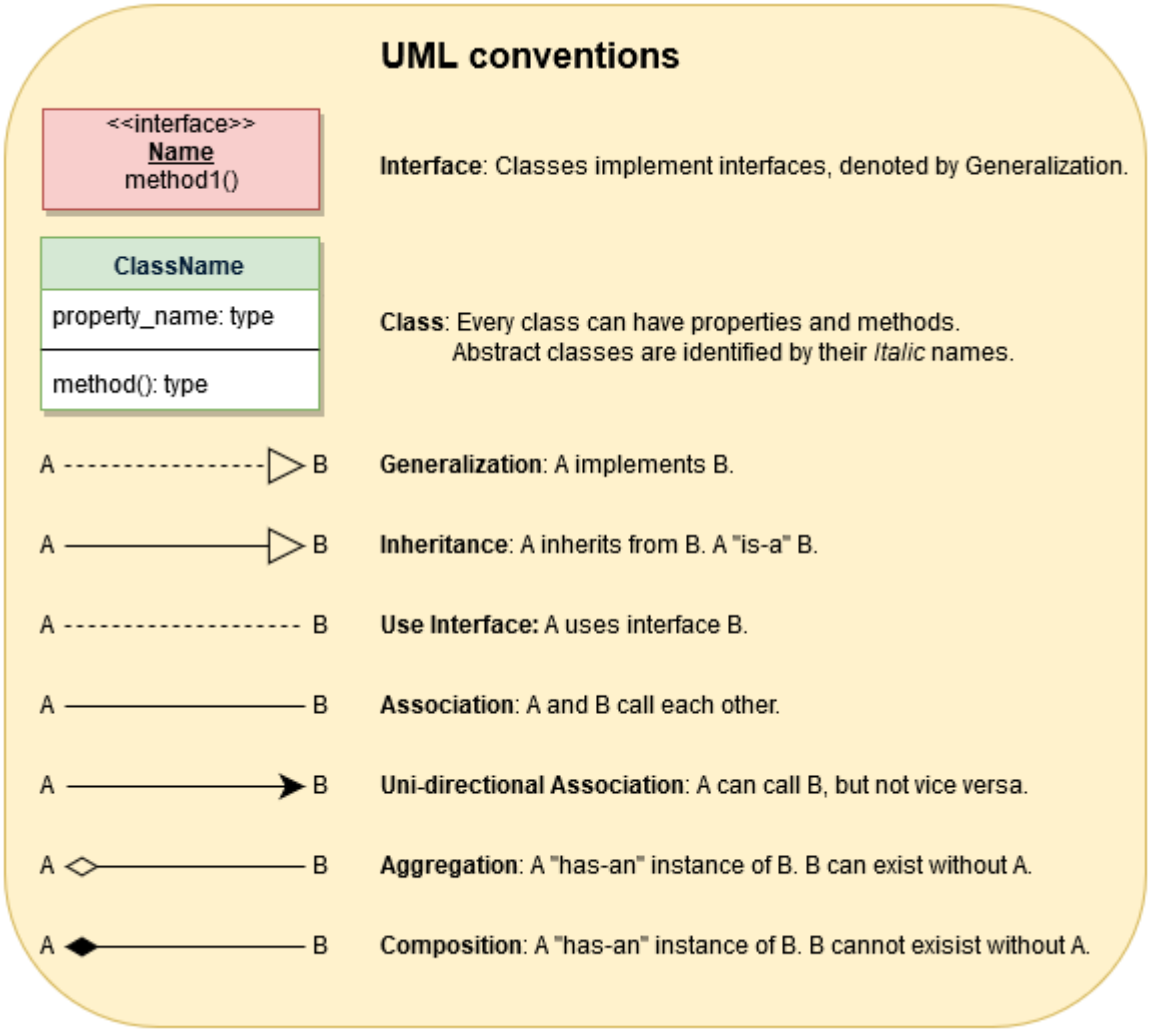
Here are the main classes of the ATM System:

- **ATM:** The main part of the software for which this software has been designed. It has attributes like 'atmID' to distinguish it from any other ATMs available and 'location' which defines the physical address of the address.
- **CashReader:** To encapsulate the ATM's card reader to be used for user authentication.
- **CashDispenser:** To encapsulate the ATM component which will be dispensing cash.
- **Keypad:** User will use the ATM's keypad to enter pin or amounts.
- **Screen:** Users will be shown all messages on the screen and they will be selecting different transaction by touching the screen.
- **Printer:** To print receipts.

- **DepositSlot:** User can deposit checks or cash through the deposit slot.
- **Bank:** To encapsulate the bank who owns the ATM. The bank will hold all the account information and the ATM will communicate with the bank to perform different customer's transactions.
- **Account:** We'll have two types of accounts in the system, 1)Checking and 2)Saving.
- **Customer:** The class will be encapsulating ATM's customer, it will have customer's basic information like name, email, etc.
- **Card:** Encapsulating ATM card that the customer will be using to authenticate themselves. Each customer can have one card.
- **Transaction:** Encapsulating all transactions that the customer can perform on the ATM, like BalancaInquiry, Deposit, Withdraw, etc.

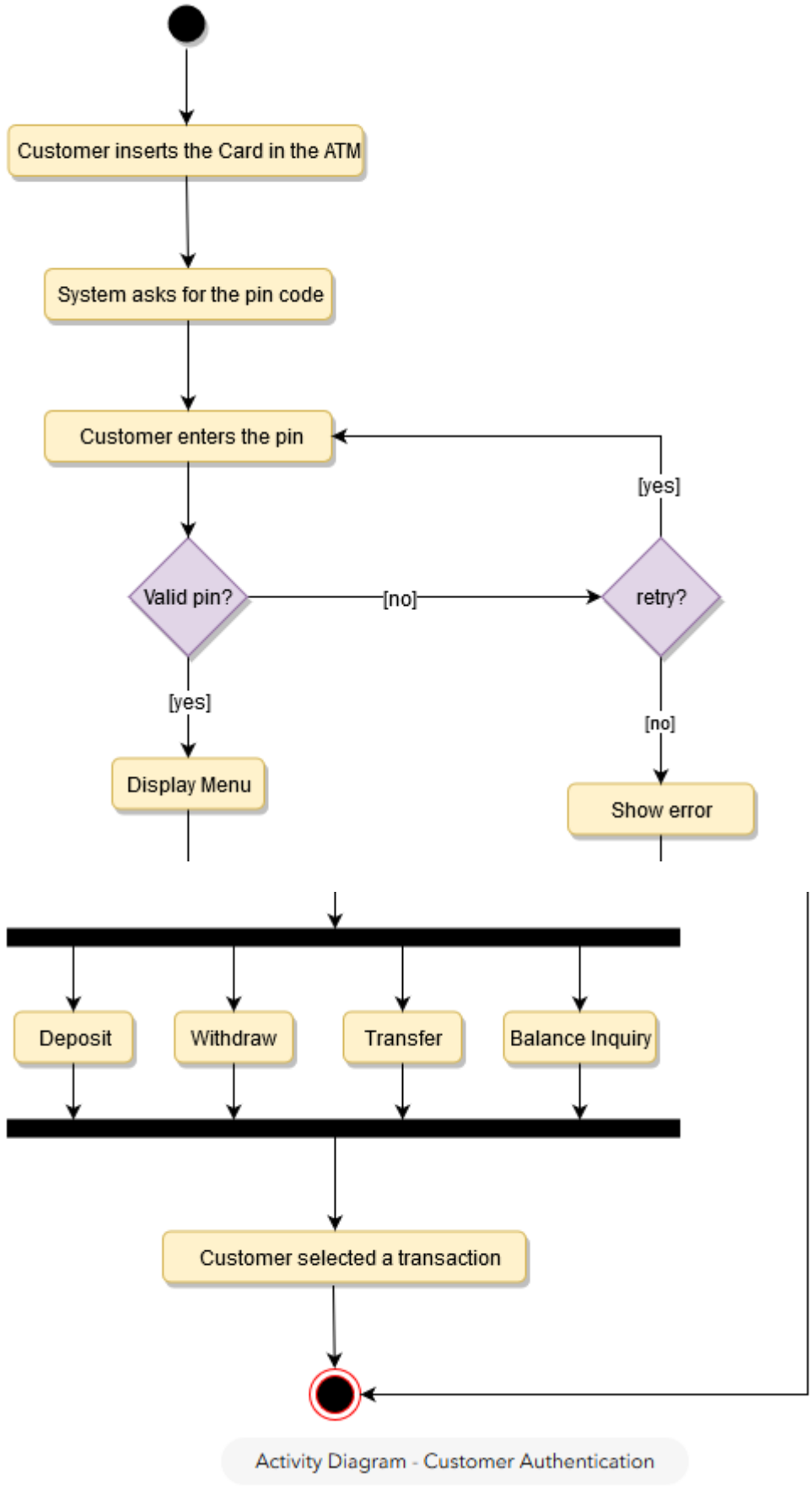


Class diagram for ATM



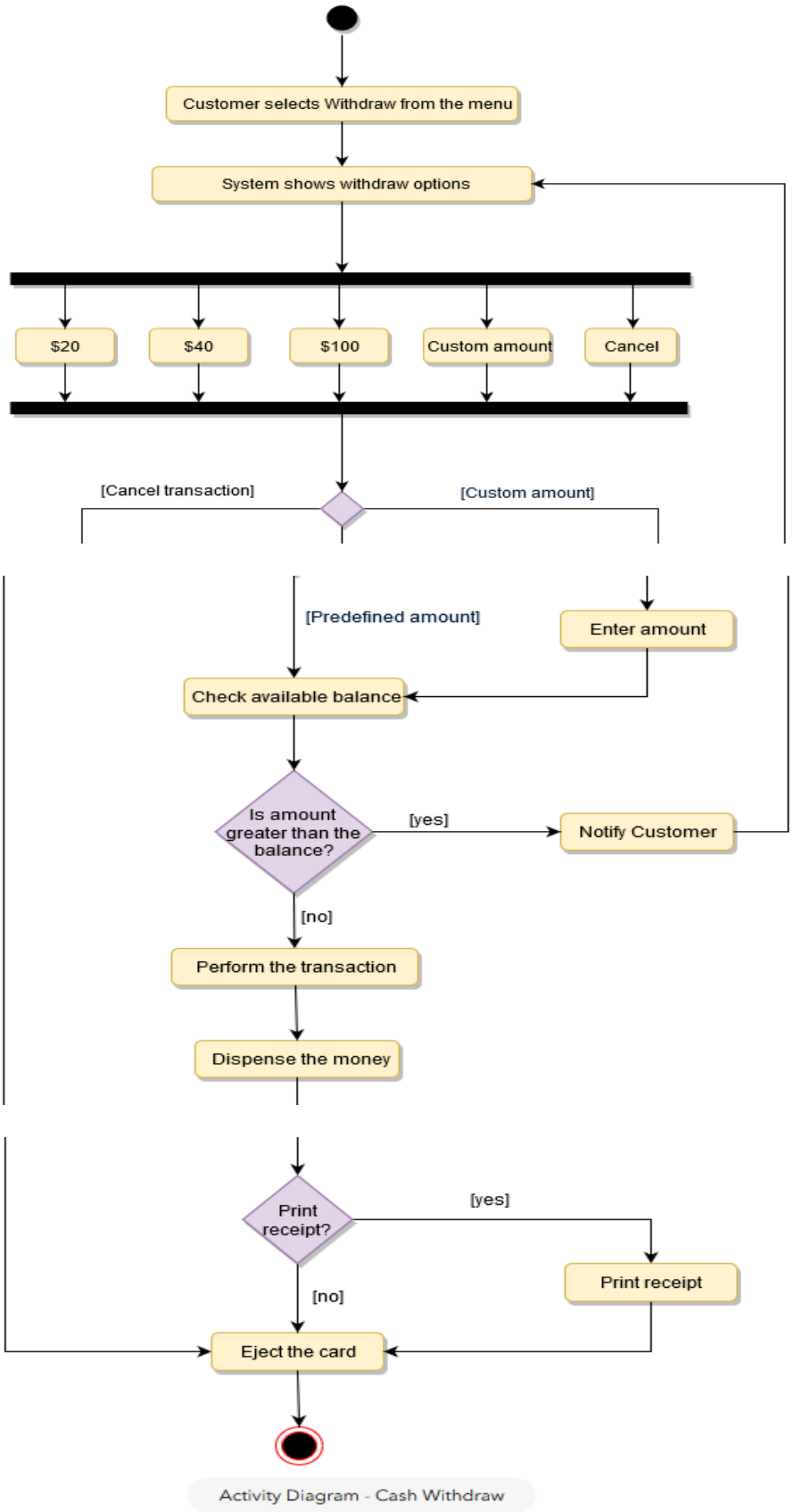
Activity Diagram

Customer authentication: Following is the activity diagram for a customer authenticating themselves to perform an ATM transaction:



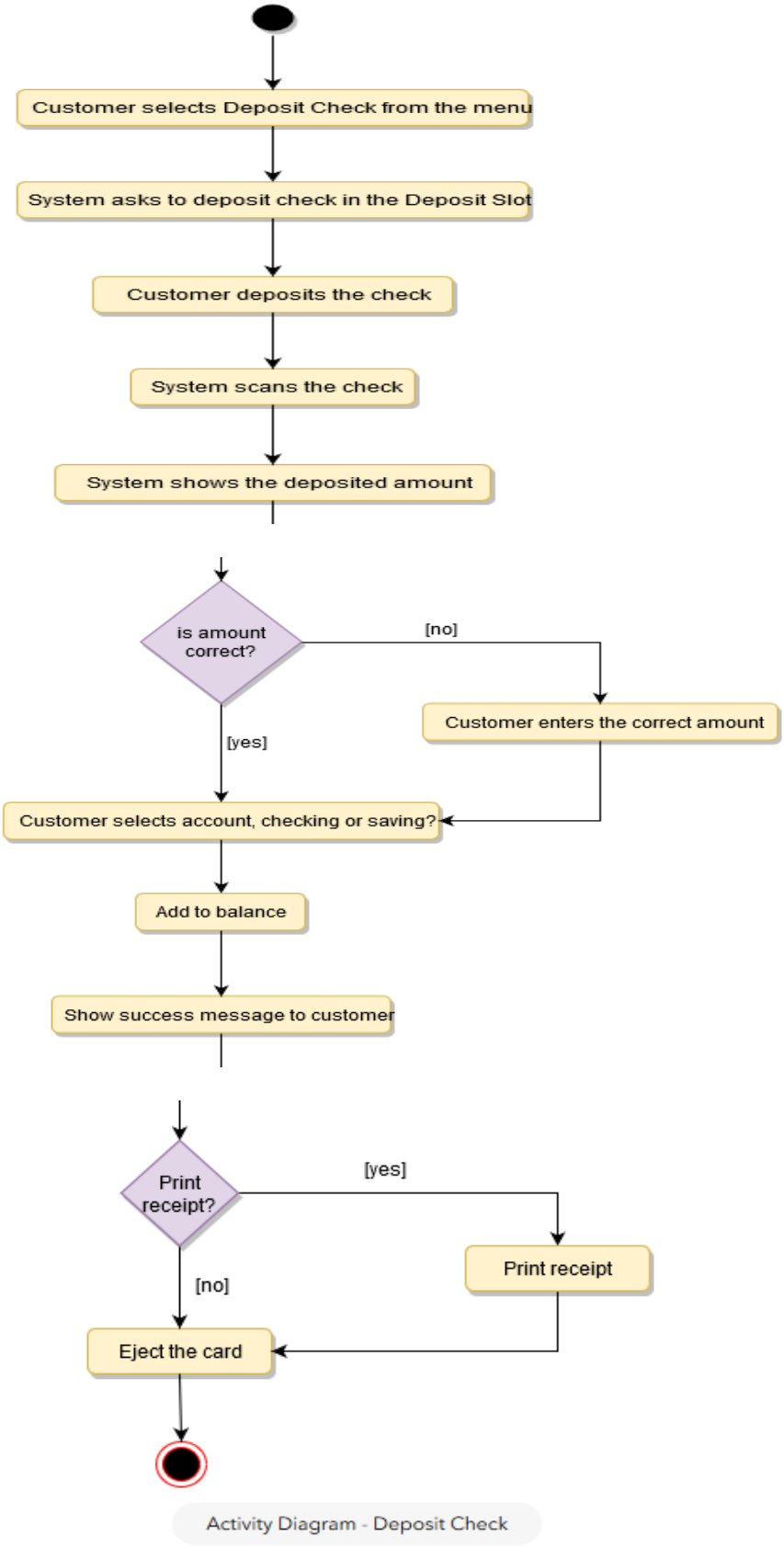
Activity Diagram - Customer Authentication

Withdraw: Following is the activity diagram for a user doing a cash withdral:



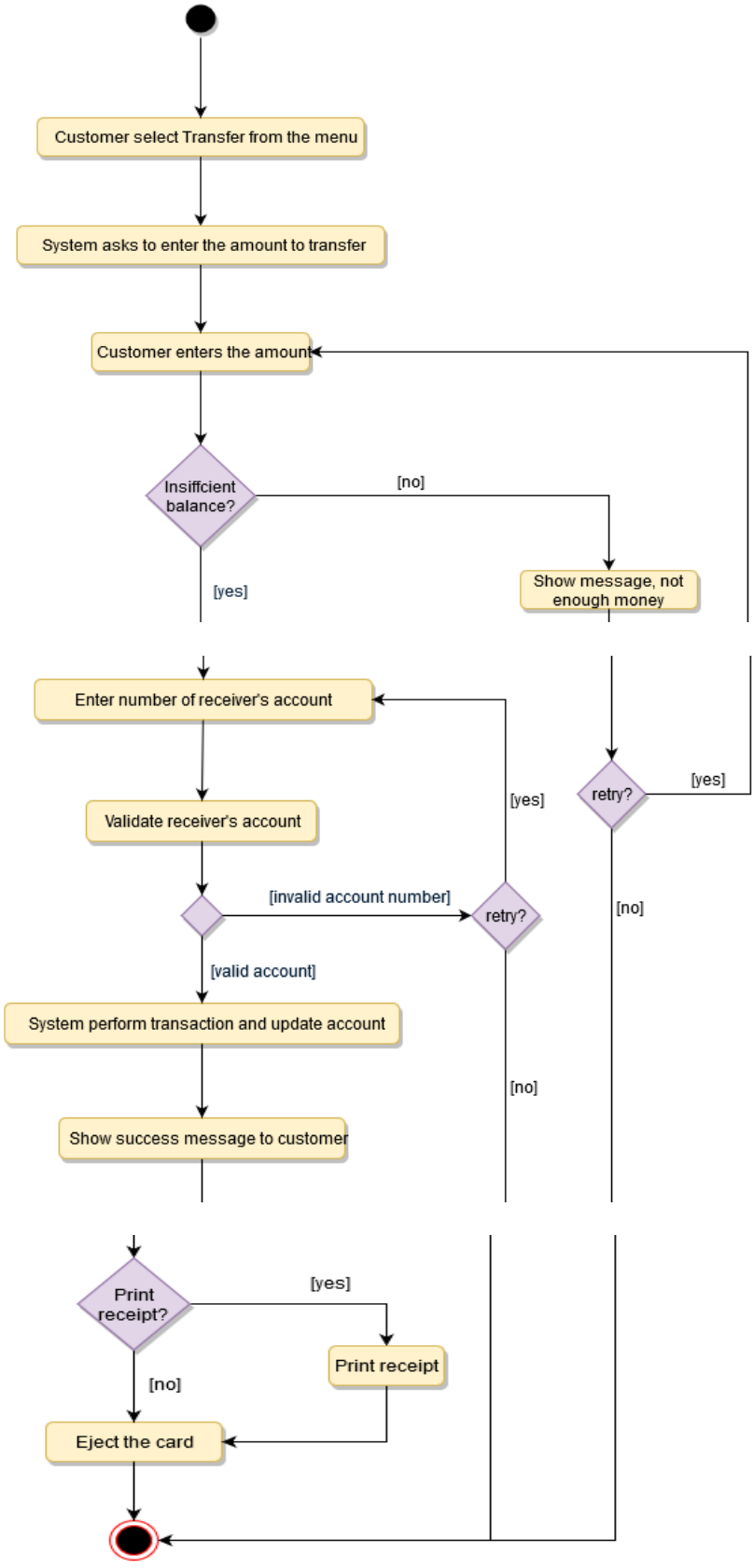
Activity Diagram - Cash Withdraw

Deposit check: Following is the activity diagram for the customer depositing a check:



Activity Diagram - Deposit Check

Transfer: Following is the activity diagram for a user transferring funds to another account:

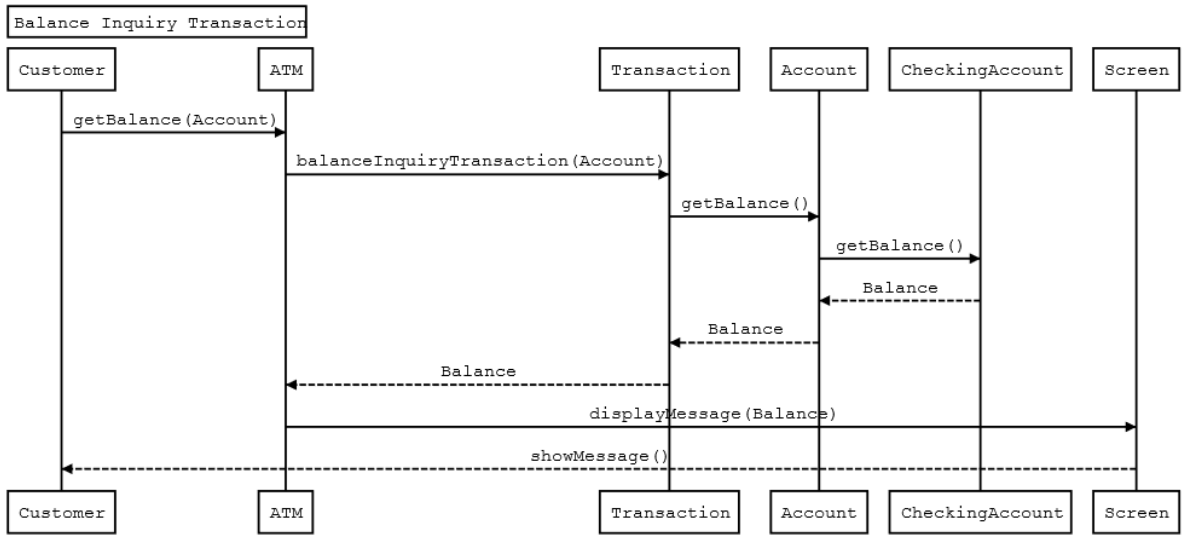


Activity Diagram - Transfer funds

Activity Diagram - Transfer funds

Sequence Diagram

Here is the sequence diagram for balance inquiry transaction:



Code

Here is the code for the use-cases mentioned above 1) Check-out a book, 2) Return a book, and 3) Renew a book.

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

```
public enum TransactionType{
  BALANCE_INQUIRT,
  DEPOSIT_CASH,
  DEPOSIT_CHECK,
  WITHDRAW,
  TRANSFER
}

public enum TranscationStatus{
  SUCCESS,
  FAILURE,
  BLOCKED,
  FULL,
  PARTIAL,
  NONE
}

public enum CustomerStatus{
  ACTIVE,
  BLOCKED,
```

```
  BANNED,
  COMPROMIZED,
  ARCHIVED,
  CLOSED,
  UNKNOWN
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}
```

Customer, Card, and Account: Customer encapsulates the ATM user, card the ATM card and Account can be of two type checking and saving:

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

public class Customer {
  private String name;
  private String email;
  private String phone;
  private Address address;
  private CustomerStatus status;

  private Card card;
  private Account account;

  public boolean makeTransaction(Transactiopn transaction);
  public Address getBillingAddress();
}

public class Card {
  private String cardNumber;
  private String customerName;
  private Date cardExpiry;
  private int pin;

  public Address getBillingAddress();
}

public abstract class Account {
  private int accountNumber;
  private double totalBalance;
  private double availableBalance;

  public double getAvailableBalance();
```

```
}

public class SavingAccount extends Account {
  private double withdrawLimit;
}

public class CheckingAccount extends Account {
  private String debitCardNumber;
}
```

Bank, ATM, CashDispenser, Keypad, Screen, Printer and DepositSlot: ATM will have different components like keypad, screen, etc.

```
public class Bank {
  private String name;
  private String bankCode;

  public String getBankCode();
  public boolean addATM();
}

public class ATM {
  private int atmID;
  private Address location;

  private CashDispenser cashDispenser;
  private Keypad keypad;
  private Screen screen;
  private Printer printer;
  private CheckDeposit checkDeposit;
  private CashDeposit cashDeposit;

  public boolean authenticateUser();
  public boolean makeTransaction(Customer customer, Transactiopn transaction);
}

public class CashDispenser {
  private int totalFiveDollarBills;
  private int totalTwentyDollarBills;

  public boolean dispenseCash(double amount);
  public boolean canDispenseCash();
}

public class Keypad {
  public String getInput();
}

public class Screen {
  public boolean showMessage(String message);
  public TransactionType getInput();
```

```
}

public class Printer {
  public boolean printReciept(Transaction transaction);
}

public abstract class DepositSlot {
  private double totalAmount;
  public double getTotalAmount();
}

public class CheckDeposit extends DepositSlot {
  public double getCheckAmount();
}

public class CashDeposit extends DepositSlot {
  public double receiveDollarBill();
}
```

Transaction and its subclasses: Customers can perform different transactions on the ATM, these classes encapsulate them:

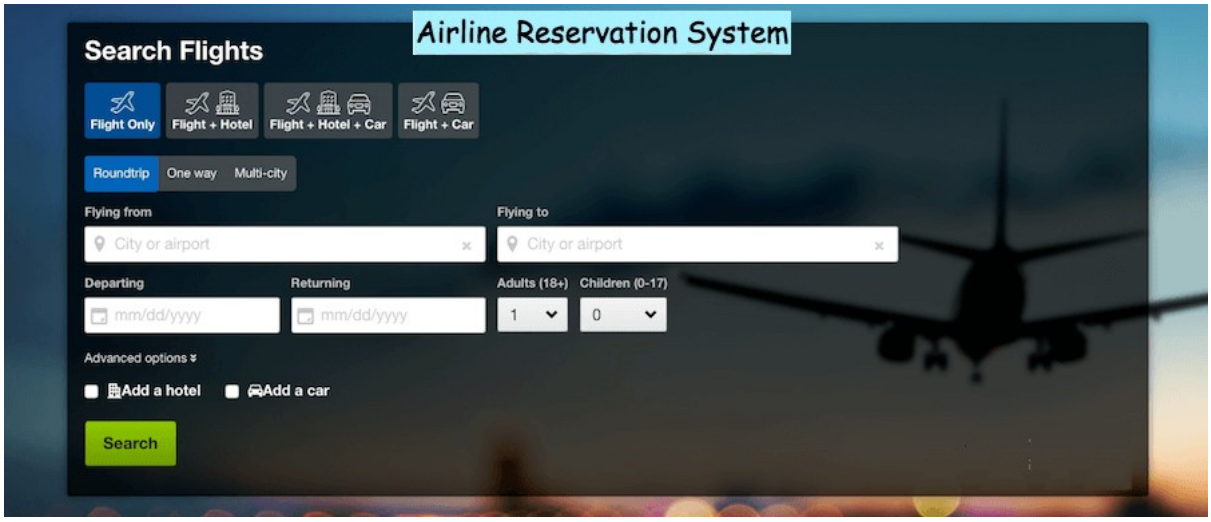
```
public abstract class Transaction {
  private int transactionId;
  private Date creationTime;
  private TransactionStatus status;
  public boolean makeTransation();
}

public class BalanceInquiry extends Transaction {
  private int accountId;
  public double getAccountId();
}

public abstract class Deposit extends Transaction {
  private double amount;
  public double getAmount();
}

public class CheckDeposit extends Deposit {
  private String checkNumber;
  private String bankCode;

  public String getCheckNumber();
}

public class CashDeposit extends Deposit {
  private double cashDepositLimit;
}

public class Withdrwa extends Transaction {
```

```
  private double amount;
  public double getAmount();
}

public class Transfer extends Transaction {
  private int destinationAccountNumber;
  public int getDestinationAccount();
}
```

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 its passengers. They carry or hire aircraft for this purpose. All operations for airlines are controlled by their airline management system.

This system involves scheduling of flights, air ticket reservation, flight cancellation, customer support, and staff management. Daily flights updates can also be retrieved by using the system.



System Requirements

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

1. Customers should be able to search flight for a given date and source/destination airport.
2. Customers should be able to reserve an air ticket for any scheduled flight. Customers can build a multi-flight itinerary.
3. Users of the system can check flight schedule, their departure time, available seats, arrival time and other details about flights.
4. Customers can make a reservation for multiple passengers under one itinerary.
5. Only admin of the system can add new aircraft, flight, and flight schedule. Admin can cancel any pre-scheduled flight. (notification will be sent to all stakeholders)

6. Customers can also 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.

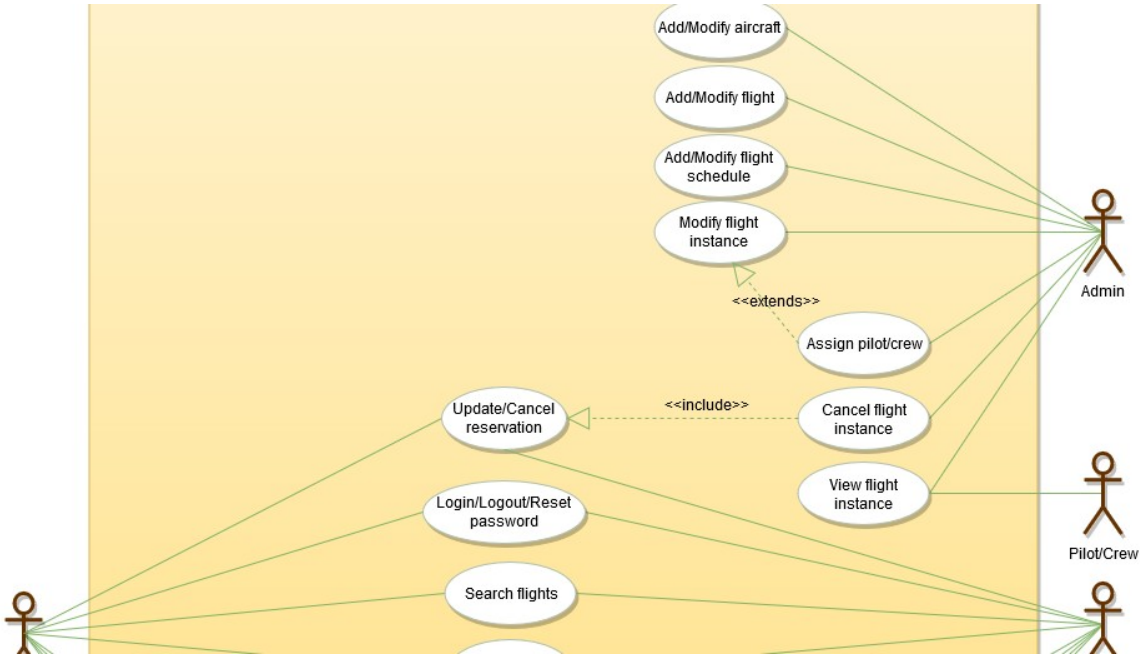
Usecase diagram

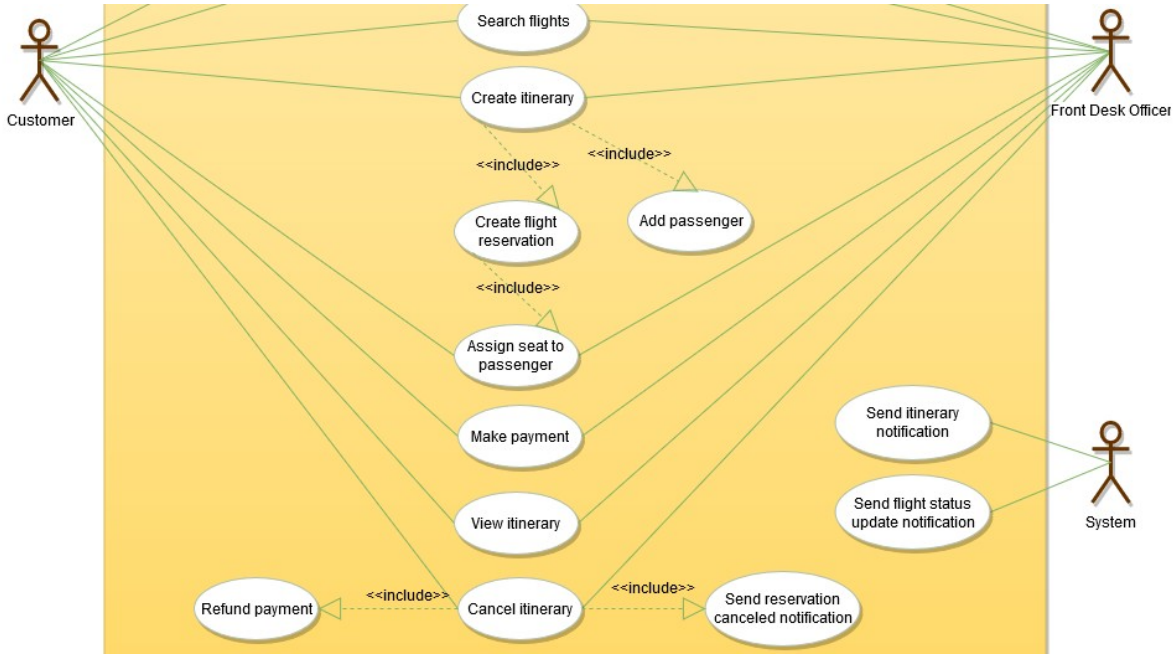
We have five main Actors in our system:

- **Admin:** Responsible for adding new flights and their schedule, canceling any flight, maintaining staff related works.
- **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 for itinerary changes, flight status updates, etc.

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

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



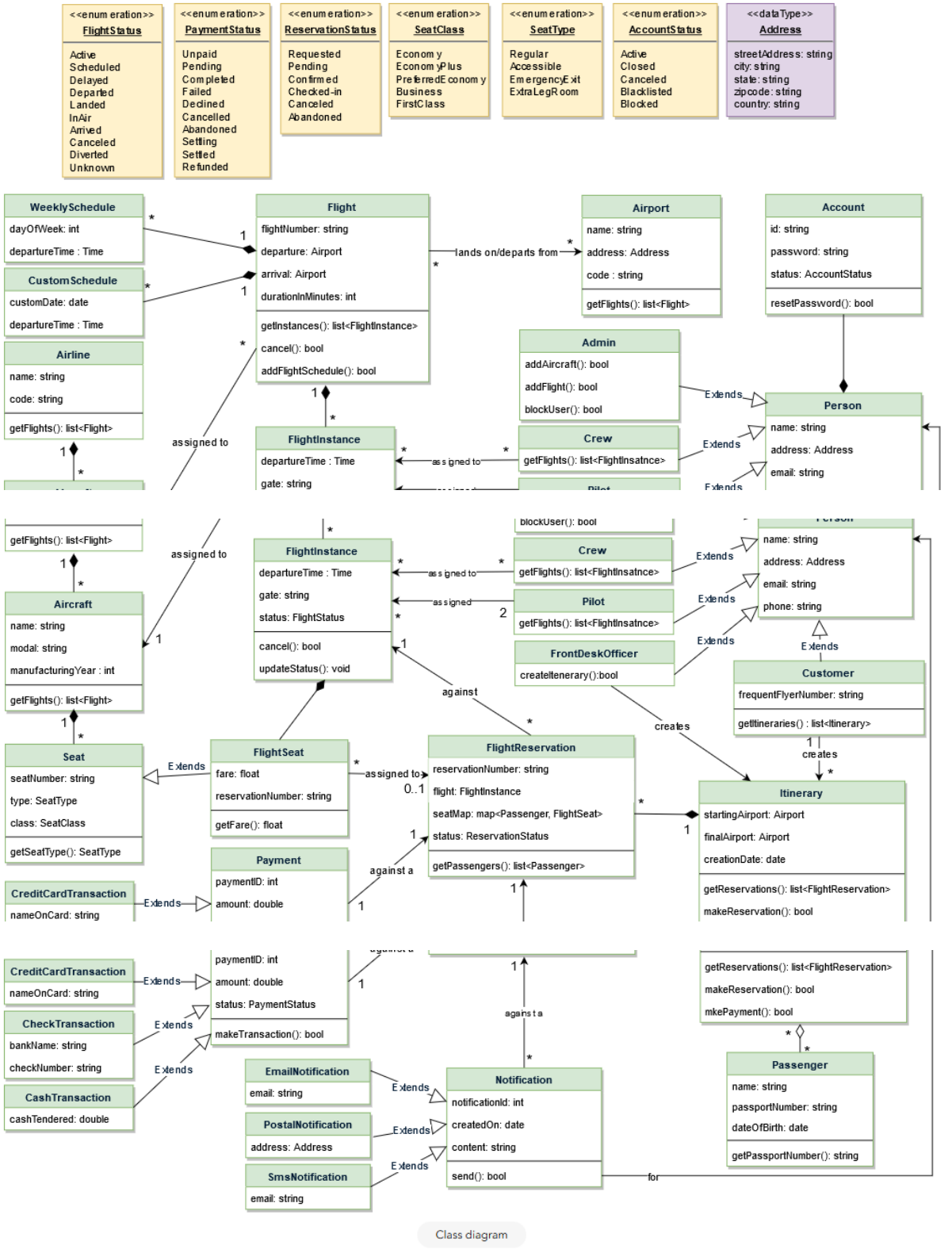


Class diagram

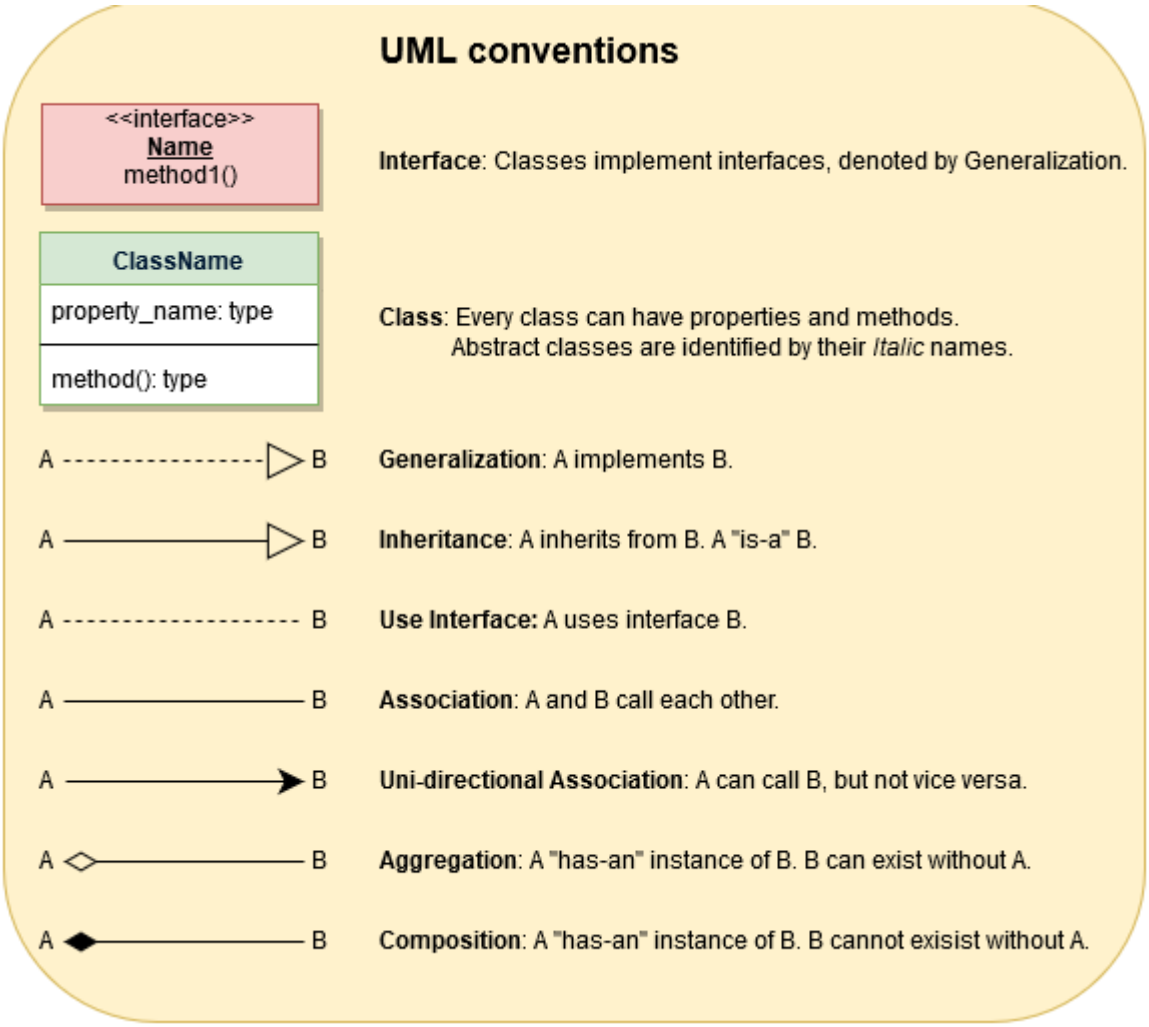
Here are the main classes of our Airline Management System:

- **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 purposes.
- **Airport:** Each airline operates from different airports. Each airport has a name, address, and a unique code.
- **Aircraft:** Airlines own or hire aircraft to carry out their flights. Aircraft has attributes like name, model, manufacturing year, etc.
- **Flight:** The main entity of the system. Every flight will have flight number, departure and arrival airport, assigned aircraft, etc.
- **FlightInstance:** Any flight can have multiple occurrences; each occurrence will be considered a flight instance in our system, e.g., there could be a British Airways flight from London to Tokyo (flight number: BA212). Assume if this flight occurs twice a week, both 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, passengers list, 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 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.

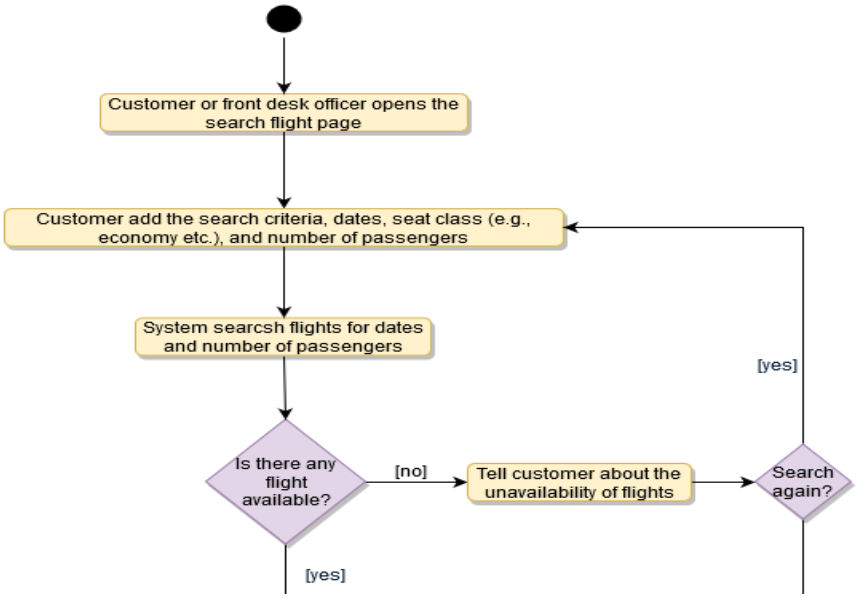


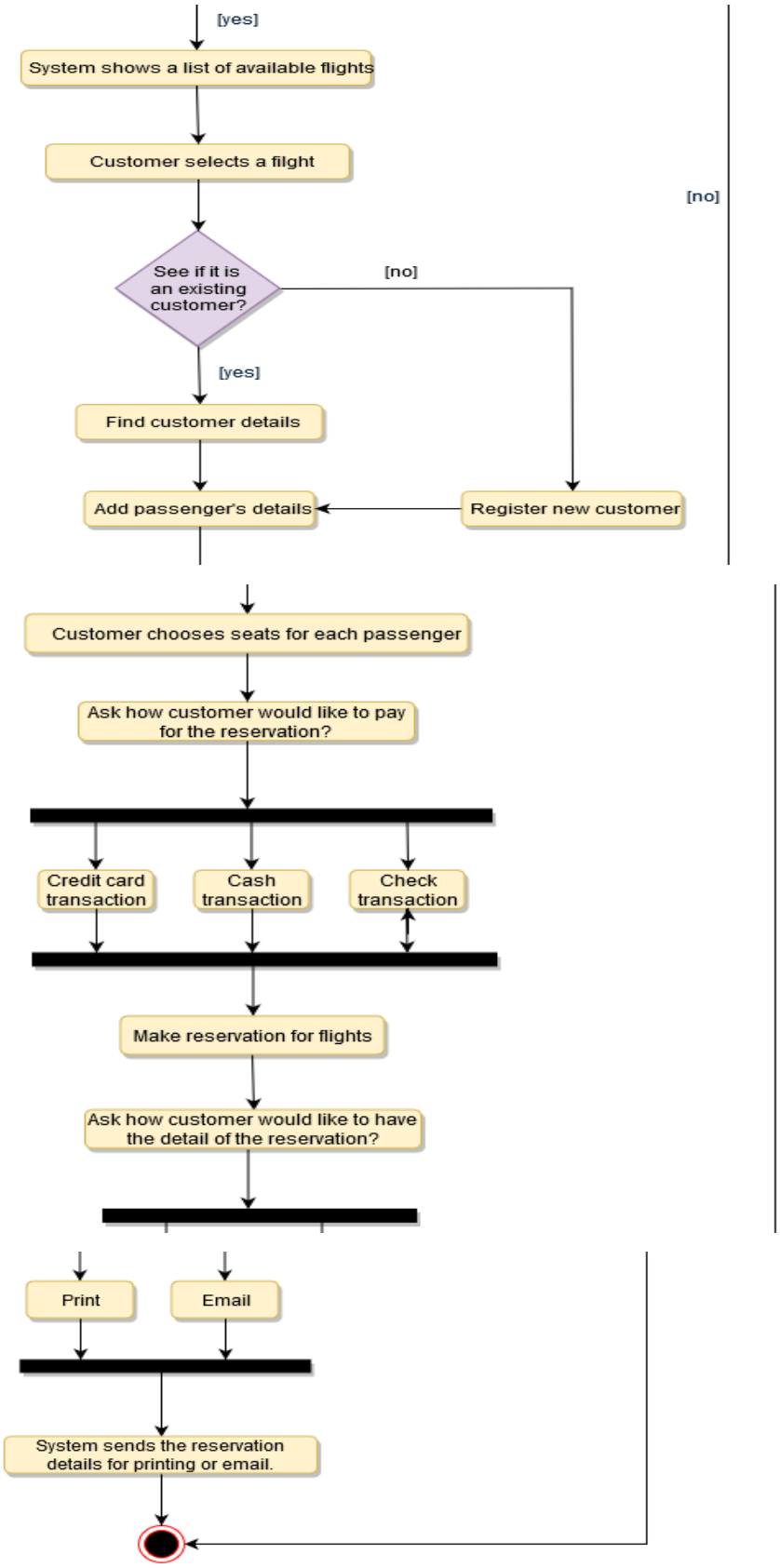
Class diagram



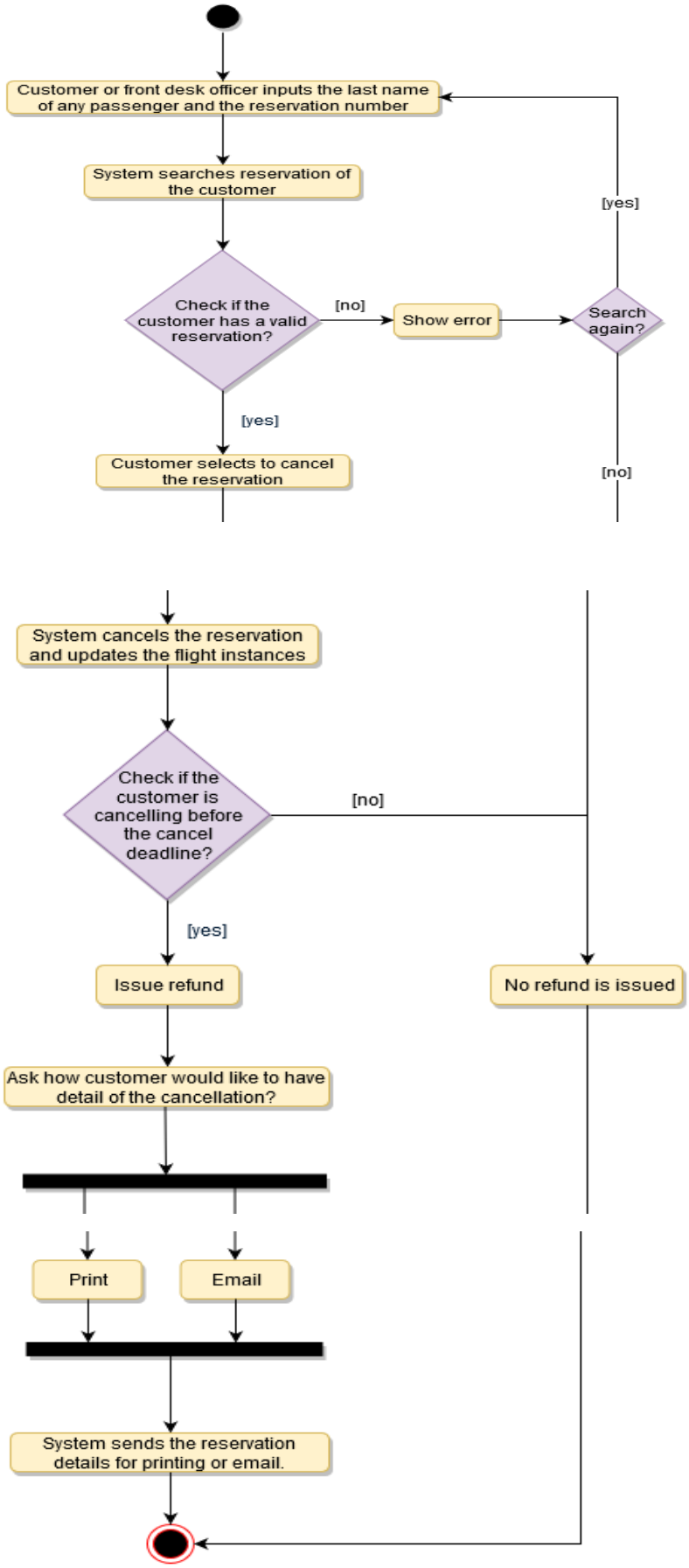
Activity diagrams

- **Reserve a ticket:** Any customer can perform this activity. Here are the set of steps to reserve a ticket:





- **Cancel a reservation:** Customer can perform this activity to cancel their reservation. Here are the set of steps to cancel a reservation:


Customer or front desk officer inputs the last name of any passenger and the reservation number
System searches reservation of the customer
Check if the customer has a valid reservation?
[no]
Show error
Search again?
[yes]
[yes]
[no]
Customer selects to cancel the reservation
System cancels the reservation and updates the flight instances
Check if the customer is cancelling before the cancel deadline?
[no]
[yes]
Issue refund
No refund is issued
Ask how customer would like to have detail of the cancellation?
Print
Email
System sends the reservation details for printing or email.

Code

Here is the code for major classes.

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

```
public enum FlightStatus{
  ACTIVE,
  SCHEDULED,
  DELAYED,
  DEPARTED,
  LANDED,
  IN_AIR,
  ARRIVED,
  CANCELLED,
  DIVERTED,
  UNKNOWN
}

public enum PaymentStatus{
  UNPAID,
  PENDING,
  COMPLETED,
  FILLED,
  DECLINED,
  CANCELLED,
  ABONDENED,
  SETTLING,
  SETTLED,
  REFUNDED
}

public enum ReservationStatus{
  REQUESTED,
  PENDING,
  CONFIRMED,
  CHECKED_IN,
  CANCELLED,
  ABONDONED
}

public enum SeatClass {
  ECONOMY,
  ECONOMY_PLUS,
  PREFFERRED_ECONOMY,
  BUSINESS,
  FIRST_CLASS
}

public enum SeatType {
```

```
  REGULAR,
  ACCESSIBLE,
  EMERGENCY_EXIT,
  EXTR_LEG_ROOM
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  BLOCKED
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}
```

Account, Person, Customer and Passenger:

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

public class Account {
  private String id;
  private String password;
  private AccountStatus status;

  public boolean resetPassword();
}

public class Person {
  private String name;
  private Address address;
  private String email;
  private String phone;

  private Account account;
}

public class Customer extends Person {
  private String frequentFlyerNumber;

  public List<Itinerary> getItineraries();
}

public class Passenger {
```

```
  private String name;
  private String passportNumber;
  private Date dateOfBirth;

  public String getPassportNumber() {
    return this.passportNumber;
  }
}
```

Airport, Aircraft, Seat and FlightSeat:

```
public class Airport {
  private String name;
  private Address address;
  private String code;

  public List<FlightInstance> getFlights();
}

public class Aircraft {
  private String name;
  private String model;
  private int manufacturingYear;
  private List<Seat> seats;

  public List<FlightInstance> getFlights();
}

public class Seat {
  private String seatNumber;
  private SeatType type;
  private SeatClass class;
}

public class FlightSeat extends Seat {
  private double fare;
  public double getFare();
}
```

Flight Schedule classes, Flight, FlightInstance, FlightReservation, Itinerary:

```
public class WeeklySchedule {
  private int dayOfWeek;
  private Time departureTime;
}

public class CustomSchedule {
  private Date customDate;
  private Time departureTime;
}

public class Flight {
```

```
  private String flightNumber;
  private Airport departure;
  private Airport arrival;
  private int durationInMinutes;
  private Aircraft aircraft;

  private List<WeeklySchdules> weeklySchedules;
  private List<CustomSchdules> customSchedules;
  private List<FlighInstance> flightInstances;
}

public class FlightInstance {
  private Date departureTime;
  private String gate;
  private FlightStatus status;

  public bool cancel();
  public void updateStatus(FlightStatus status);
}

public class FlightReservation {
  private String reservationNumber;
  private FlightInstance flight;
  private Map<Passenger, FlightSeat> seatMap;
  private Date creationDate;
  private ReservationStatus status;

  public static FlightReservation fectchReservationDetails(String reservationNumber);
  public List<Passenger> getPassengers();
}

public class Itinerary {
  private String customerId;
  private Airport startingAirport;
  private Airport finalAirport;
  private Date creationDate;
  private List<FlightReservation> reservations;

  public List<FlightReservation> getReservations();
  public boolean makeReservation();
  public boolean makePayment();
}
```

Design Blackjack and a Deck of Cards

Let's design a game of Blackjack.

Blackjack is the most widely played casino game in the world. It falls under the category of comparing-card game and is usually played between several players and a dealer. Each

player, in turn, competes against the dealer, but players do not play against each other. In Blackjack, all players and the dealer are trying to build a hand that totals 21 points without going over. The hand closest to 21 wins.

System Requirements

Blackjack is played with one or more standard 52-card decks. The standard deck has 13 ranks in 4 suits.

Background

- To start with, the player and the dealer are dealt separate hands. A hand has two cards in it.
- The dealer has one card exposed (the up card) and one card concealed (the hole card), leaving the player with incomplete information about the state of the game.
- The player's objective is to make a hand that has more points than the dealer, but less than or equal to 21 points.
- The player is responsible for placing bets when they are offered and taking additional cards to complete their hand.
- The dealer will draw additional cards according to a simple rule: when the dealer's hand is 16 or less, they will draw cards (called a hit), when it is 17 or more, they will not draw additional cards (or stand pat).

Points calculation

Blackjack has a different point values for the cards:

- The number cards (2-10) have the expected point values.
- The face cards (Jack, Queen, and King) all have a value of 10 points.
- The Ace can count as one point or eleven points. Because of this, an Ace and a 10 or face card totals 21. This two-card winner is called “blackjack”.
- When the points include an ace counting as 11, the total is called soft-total; when the ace counts as 1, the total is called hard-total. For example, A-5 is called a soft 16 because it could be considered a hard 6.

Gameplay

1. The player places an initial bet.
2. The player and dealer are each dealt a pair of cards.
3. Both of the player’s cards are face up, the dealer has one card up and one card down.
4. If the dealer’s card is an ace, the player is offered insurance.

Initially, the player has a number of choices:

- If the two cards are the same rank, the player can elect to split into two hands.
- The player can double their bet and take just one more card.
- The more typical scenario is for the player to take additional cards (a hit) until either their hand totals more than 21 (they bust), or their hand totals exactly 21, or they elect to stand.

If the player’s hand is over 21, their bet is resolved immediately as a loss. If the player’s hand is 21 or less, it will be compared to the dealer’s hand for resolution.

Dealer has an Ace. If the dealer’s up card is an ace, the player is offered an insurance bet. This is an additional proposition that pays 2:1 if the dealer’s hand is exactly 21. If this insurance bet wins, it will, in effect, cancel the loss of the initial bet. After offering insurance to the player, the dealer will check their hole card and resolve the insurance bets. If the hole card is a 10-point card, the dealer has blackjack, the card is revealed, and insurance bets are paid. If the hole card is not a 10-point card, the insurance bets are lost, but the card is not revealed.

Split Hands. When dealt two cards of the same rank, the player can split the cards to create two hands. This requires an additional bet on the new hand. The dealer will deal an additional card to each new hand, and the hands are played independently. Generally, the typical scenario described above applies to each of these hands.

Bets

- Ante. The initial bet, it is mandatory to play.
- Insurance. This bet is offered only when the dealer shows an ace. The amount must be half the ante.
- Split. This can be thought of as a bet that is offered only when the player’s hand has two cards are of equal rank. The amount of the bet must match the original ante.
- Double. This can be thought of as a bet that is offered instead of a taking an ordinary hit. The amount of the bet must match the original ante.

Usecase diagram

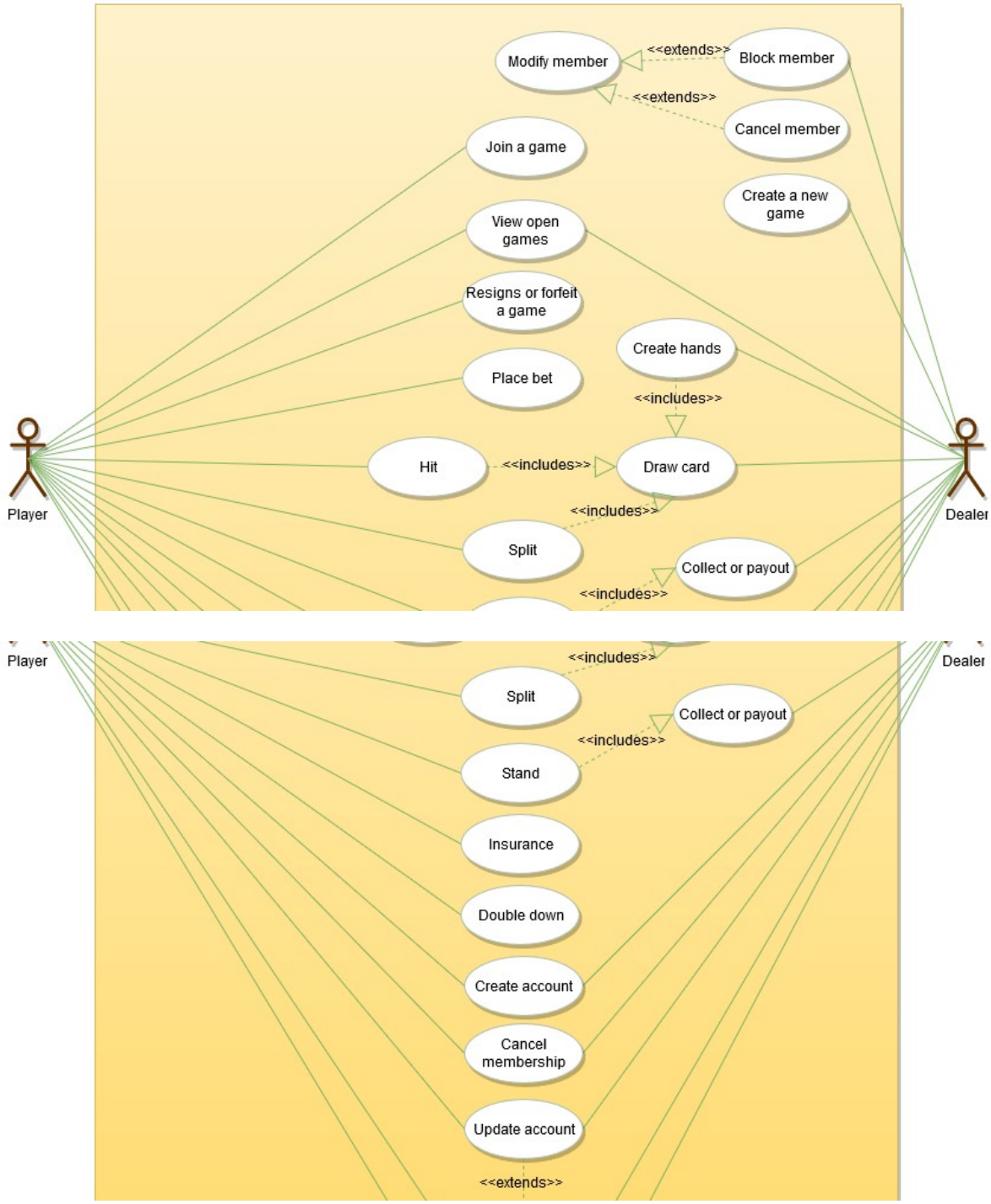
We have two main Actors in our system:

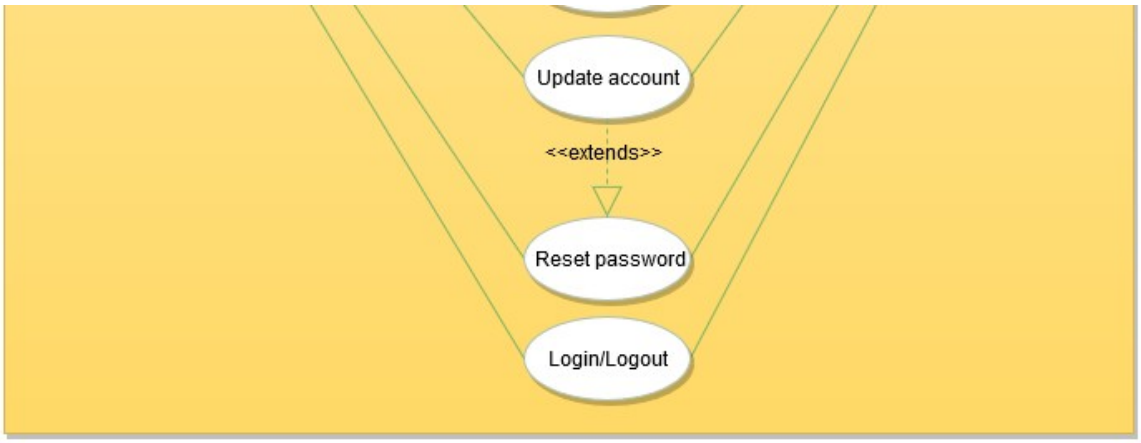
- **Dealer:** Mainly responsible for dealing cards and game resolution.
- **Player:** Places the initial bets, accepts or declines additional bets, including insurance, and split. Accepts or rejects the offered resolution, including even money. Chooses among hit, double and stand pat options.

Typical Blackjack Game Usecases

Here are the top use cases of the Blackjack game:

- **Create Hands:** Initially both the player and the dealer are given two cards each. The player has both the cards visible whereas only one card of the dealer's hand is visible to the player.
- **Place Bet:** To start the game, the player has to place a bet.
- **Player plays the hand:** If the hand is under 21 points, the player has three options:
    - Hit: The hand gets an additional card and this process repeats.
    - Double Down: The player creates an additional bet, and the hand gets one more card and play is done.
    - Stands Pat: If the hand is 21 points or over, or the player chooses to stand pat, the game is over.
    - Resolve Bust. If a hand is over 21, it is resolved as a loser.
- **Dealer plays the hand:** The dealer keeps getting a new card if total point value of the hand is 16 or less and stops dealing cards on the point value of 17 or more.
    - Dealer Bust: If the dealer's hand is over 21, the player's wins the game. Player Hands with two cards totaling 21 ("blackjack") are paid 3:2, all other hands are paid 1:1.
- **Insurance:** If the dealer's up card is an Ace, then the player is offered insurance:
    - Offer Even Money. If the player's hand totals to a soft 21, a blackjack; the player is offered an even money resolution. If the player accepts, the entire game is resolved at this point. The ante is paid at even money; there is no insurance bet.
    - Offer Insurance. The player is offered insurance, who can accept by creating a bet. For players with blackjack, this is the second offer after even money is declined. If the player declines, there are no further insurance considerations.
    - Examine Hole Card. The dealer's hole card is examined, if it is a 10-point value, the insurance bet is resolved as a winner, and the game is over. Otherwise, the insurance is resolved as a loser, the hole card is not revealed, and play will continue.
- **Split:** If the player's hand has both the cards of equal rank, the player is offered a split. The player accepts by creating an additional Bet. The original hand is removed; The two original cards are split and then deals two extra cards to create two new Hands. There will not be any further splitting.
- **Game Resolution** The Player's Hand is compared against the Dealer's Hand, the hand with higher point value wins. In the case of a tie, the bet is returned. When the player wins, a winning hand with two cards totaling 21 ("blackjack") is paid 3:2, any other winning hand is paid 1:1.


Modify member
<<extends>>
Block member
<<extends>>
Cancel member
Join a game
Create a new game
View open games
Resigns or forfeit a game
Create hands
Place bet
<<includes>>
Player
Hit
<<includes>>
Draw card
Dealer
<<includes>>
Split
Collect or payout
<<includes>>
Player
<<includes>>
Split
Collect or payout
<<includes>>
Stand
Insurance
Double down
Create account
Cancel membership
Update account
<<extends>>
Dealer



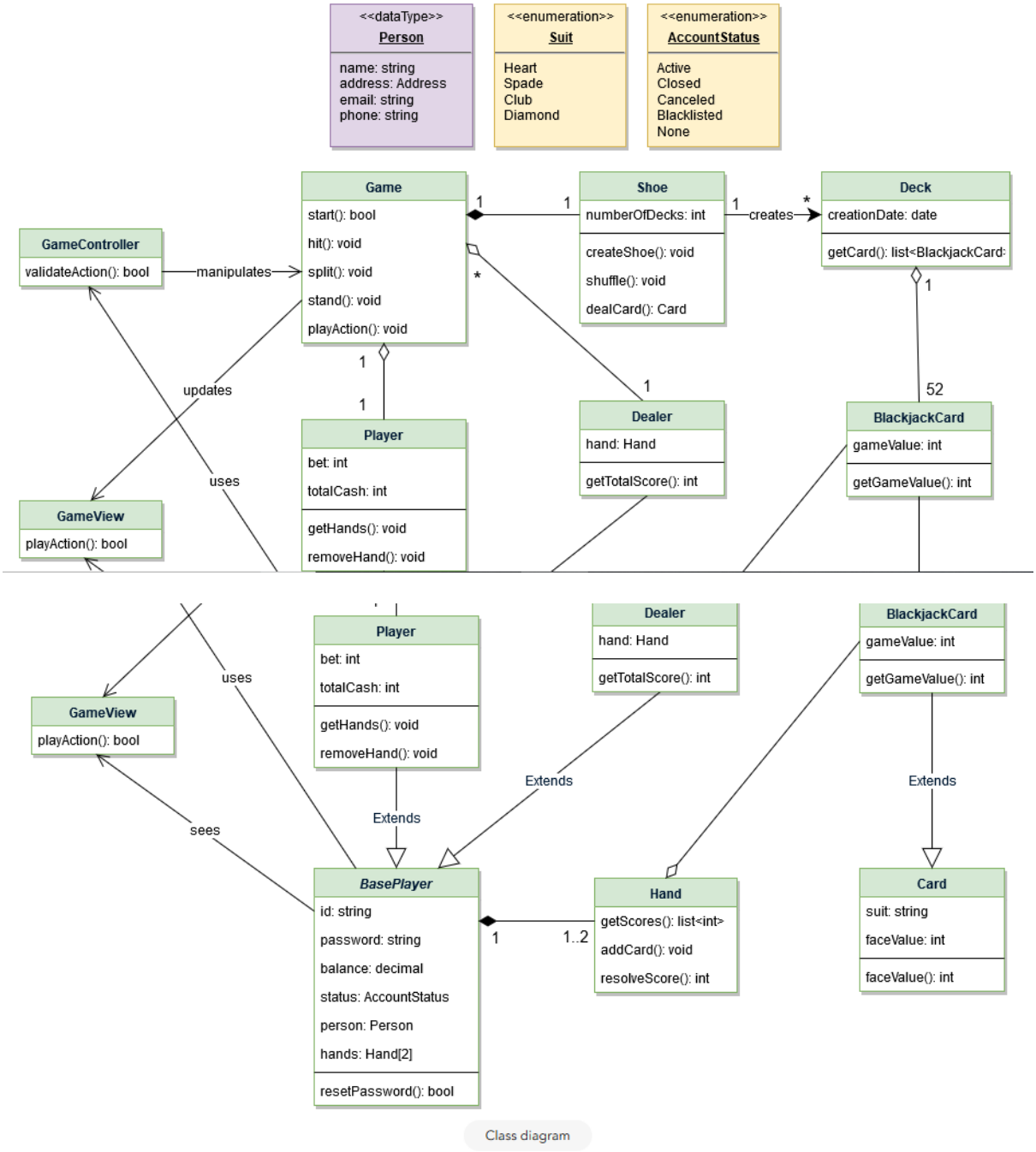
Use case diagram

Use case diagram

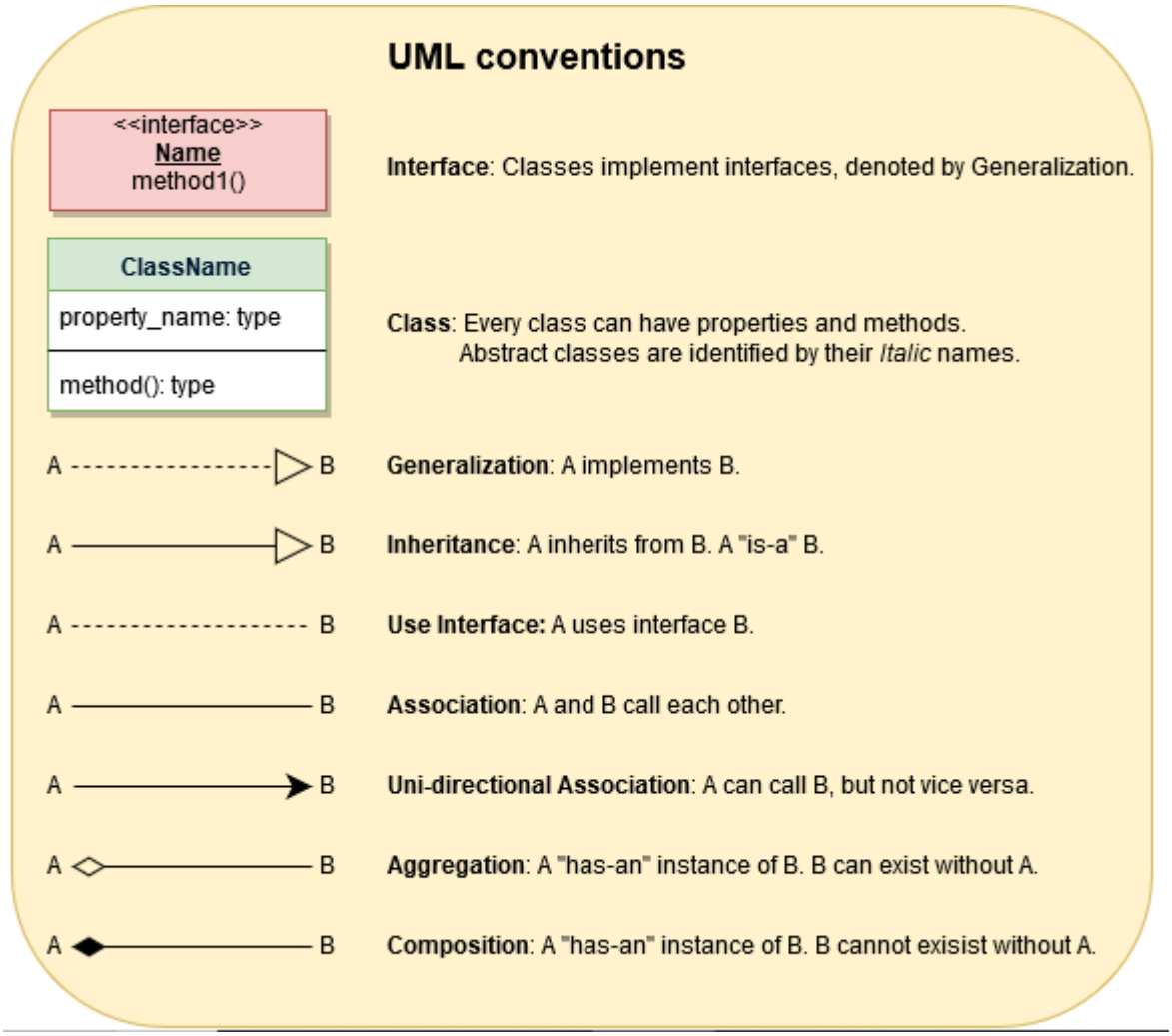
Class diagram

Here are the main classes of our Blackjack game:

- **Card:** A standard playing card has a suit and point value from 1 to 11.
- **BlackjackCard:** In blackjack cards have different face values, e.g., jack, queen and king, all have a face value of 10. An ace can be counted as either 1 or 11.
- **Deck:** A standard playing deck has 52 cards and 4 suits.
- **Shoe:** Contains a set of decks. In casinos, a dealer's shoe is a gaming device to hold multiple decks of playing cards.
- **Hand:** A collection of cards with one or two point values: a hard value (when an ace counts as 1) and a soft value (when an ace counts as 11).
- **Player:** Places the initial bets, updates the stake with amounts won and lost. Accepts or declines offered additional bets, including insurance, and split. Accepts or declines offered resolution, including even money. Chooses among hit, double and stand options.
- **Game:** This class encapsulates the basic sequence of play. It runs the game, offers bets to Player, deals the cards from the Shoe to Hands, updates the state of the game, collects losing bets, pays winning bets. Splits Hands. Responds to player choices of a hit, double and stand.

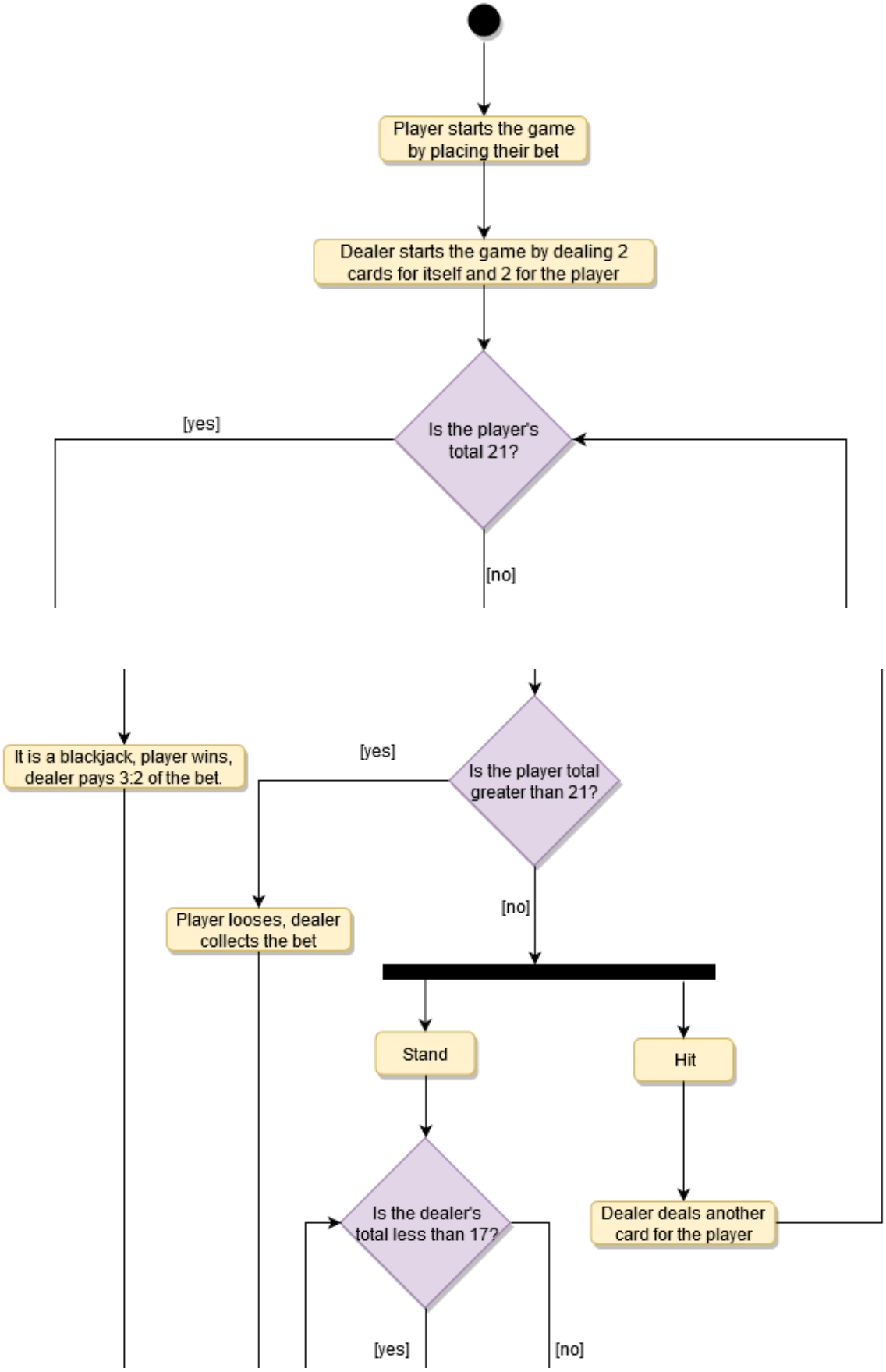


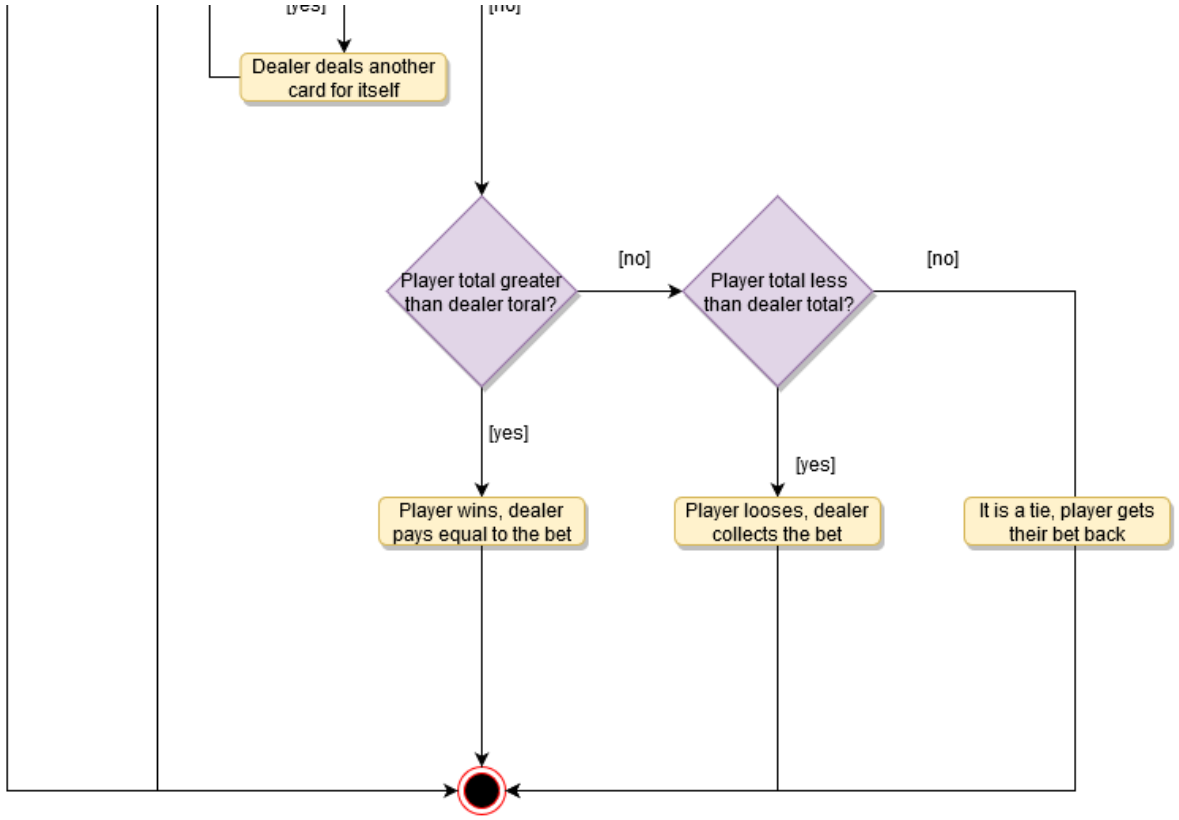
Class diagram



Activity diagrams

Blackjack hit or stand: Here are the set of steps to play blackjack with hit or stand:


Player starts the game by placing their bet
Dealer starts the game by dealing 2 cards for itself and 2 for the player
Is the player's total 21?
[yes]
[no]
It is a blackjack, player wins, dealer pays 3:2 of the bet.
Is the player total greater than 21?
[yes]
[no]
Player looses, dealer collects the bet
Stand
Hit
Is the dealer's total less than 17?
Dealer deals another card for the player
[yes]
[no]



Code

Enums: Here are the required enums:

```
public enum SUIT {
  HEART,
  SPADE,
  CLUB,
  DIAMOND
}
```

Card: Following class encapsulates a playing card:

```
public class Card {
  private SUIT suit;
  private int faceValue;

 public SUIT getSuit() {
    return suit;
  }

  public int getFaceValue() {
    return faceValue;
  }

  Card(SUIT suit, int faceValue) {
    this.suit = suit;
```

```
    this.faceVale = faceValue;
  }
}
```

BlackjackCard: BlackjackCard extends from Card class represent a blackjack card:

```
public class BlackjackCard extends Card {
  private int gameValue;

  public int getGameValue() {
    return gameValue;
  }

  public BlackjackCard(SUIT suit, int faceValue) {
    super(suit, faceValue);
    this.gameValue = faceValue;
    if(this.gameValue > 10) {
      this.gameValue = 10;
    }
  }
}
```

Deck and Shoe: Shoe contains cards from multiple decks:

```
public class Deck {
  private List<BlackjackCard> cards;
  private Date creationDate;

  public Deck() {
    this.creationDate = new Date();
    this.cards = new ArrayList<BlackjackCard>();
    for(int value = 1 ; value <= 13 ; value++){
      for(SUIT suit : SUIT.values()){
        cardDeck.add(new BlackjackCard(suit, value));
      }
    }
  }

  public List<BlackjackCard> getCards() {
    return cards;
}

public class Shoe {
  private List<BlackjackCard> cards;
  private int numberOfDecks;

  private void createShoe() {
    this.cards = new ArrayList<BlackjackCard>();
    for(int decks = 0 ; decks < numberOfDecks ; decks++){
      cards.add(new Deck().getCards());
    }
  }
```

```
  public Shoe(int numberOfDecks) {
    this.numberOfDecks = numberOfDecks;
    createShoe();
    shuffle();
  }

  public void shuffle() {
    int cardCount = cards.size();
    Random r = new Random();
    for (int i = 0; i < cardCount ; i++){
      int index = r.nextInt(cardCount-i-1);
      swap(i, index);
    }
  }

  public void swap(int i, int j) {
    BlackjackCard temp = cards[i];
    cards[i] = cards[j];
    cards[j] = temp;
  }

  //Get the next card from the shoe
  public BlackjackCard dealCard() {
    if(cards.size() == 0 ){
      createShoe();
    }
    return cards.remove(0);
  }
}
```

Hand: Hand class encapsulates a blackjack hand which can contain multiple cards:

```
public class Hand {
  private ArrayList<BlackjackCard> cards;

  private List<int> getScores() {
    List<int> totals = new ArrayList();
    total.add(0);

    for(BlackjackCard card : cards){
      List<int> newTotals = new ArrayList();
      for(int score: totals){
        newTotals.add(card.faceValue() + score);
        if(card.faceValue() == 1) {
          newTotals.add(11 + score);
        }
      }
      totals = newTotals;
    }
    return totals;
  }
```

```
 public Hand(BlackjackCard c1, BlackjackCard c2) {
   this.cards = new ArrayList<BlackjackCard>();
   this.cards.add(c1);
   this.cards.add(c2);
  }

  public void addCard(BlackjackCard card) {
   cards.add(card);
  }

  // get highest score which is less than or equal to 21
  public int resolveScore(){
   List<int> scores = getScores();
   int bestScore = 0;
   for(int score: scores) {
    if(score <=  21 && score > bestScore){
     bestScore = score;
    }
   }
   return bestScore;
  }
}
```

Player: Player class extends from BasePlayer:

```
public abstract class BasePlayer {
  private String id;
  private String passwords;
  private double balance;
  private AccountStatus status;
  private Person person;
  private List<Hand> hands;

  public boolean resetPassword();

  public List<Hand> getHands() {
   return hands;
  }

  public void addHand(Hand hand) {
   return hands.add(hand);
  }

  public void removeHand(Hand hand) {
   return hands.remove(hand);
  }
}

public class Player extends BasePlayer {
  private int bet;
  private int totalCash;
```

```
  public Player(Hand hand) {
    this.hands = new ArrayList<Hand>();
    hands.add(hand);
  }
}
```

Game: This class encapsulates a blackjack game:

```
public class Game {
  private Player player;
  private Dealer dealer;
  private Shoe shoe;
  private final int MAX_NUM_OF_DECKS = 3;

  private void playAction(string action, Hand hand) {
    switch(action) {
      case "hit": hit(hand); break;
      case "split": split(hand); break;
      case "stand pat": break; //do nothing
      case "stand": stand(); break;
      default: print("Wrong input");
    }
  }

  private void hit(Hand hand) {
    hand.addCard(shoe.dealCard());
  }

  private void stand() {
    int dealerScore = dealer.getTotalScore();
    List<Hand> hands = player.getHands();
    for(Hand hand : hands) {
      int bestScore = hand.resolveScore();
      if(playerScore == 21){
        //blackjack, pay 3:2 of the bet
      } else if (playerScore > dealerScore) {
        // pay player equal to the bet
      } else if (playerScore < dealerScore) {
        // collect the bet from the player
      } else { //tie
        // bet goes back to player
      }
    }
  }

  private void split(Hand hand) {
    Cards cards = hand.getCards();
    player.addHand(new Hand(cards[0], shoe.dealCard()));
    player.addHand(new Hand(cards[1], shoe.dealCard()));
    player.removeHand(hand);
  }
```

```
 public Game(Player player, Dealer dealer) {
   this.player = player;
   this.dealer = dealeer;
   Shoe shoe= new Shoe(MAX_NUM_OF_DECKS);
  }

  public void start() {
   player.placeBet(getBetFromUI());

   Hand playerHand = new Hand(shoe.dealCard(), shoe.dealCard());
   player.addToHand(playerHand);

   Hand dealerHand = new Hand(shoe.dealCard(), shoe.dealCard());
   dealer.addToHand(dealerHand);

   while(true){
    List<Hand> hands = player.getHands();
    for(Hand hand : hands) {
     string action = getUserAction(hand);
     playAction(action, hand);
     if(action.equals("stand")) {
      break;
     }
    }
   }
  }
}

public class Main {
  public static void main(String args[]) {
   Player player = new Player();
   Dealer dealer = new Dealer();
   Game game = new Game(player, dealer);
   game.start();
  }
}
```

Design a Hotel Management System

Let's design a hotel management system.

A Hotel Management System is a software build to handle all hotel activities online 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 available rooms in the system as well as booking of rooms and bill generation.

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 room inventory and book any available room.
3. The system should be able to retrieve information like who book a particular room or what are the rooms booked by a specific customer.
4. The system should allow customers to cancel their booking. Full refund if the cancelation is done before 24 hours of check-in date.
5. The system should be able to send notifications whenever the booking is near 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.

Usecase diagram

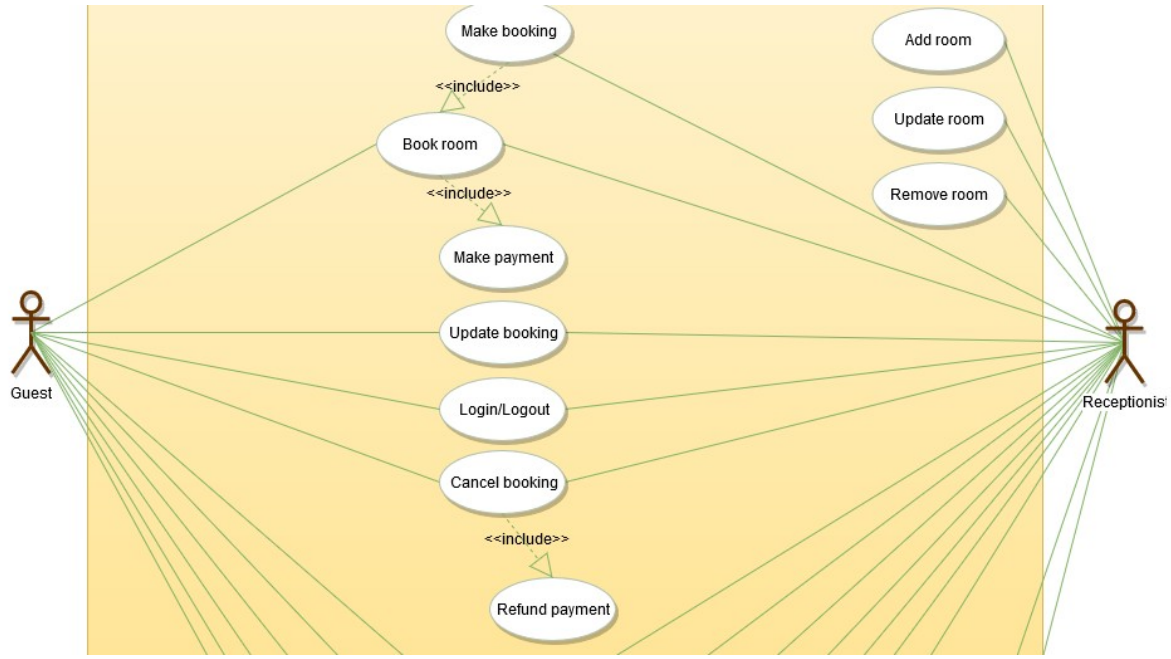
Here are the main Actors in our system:

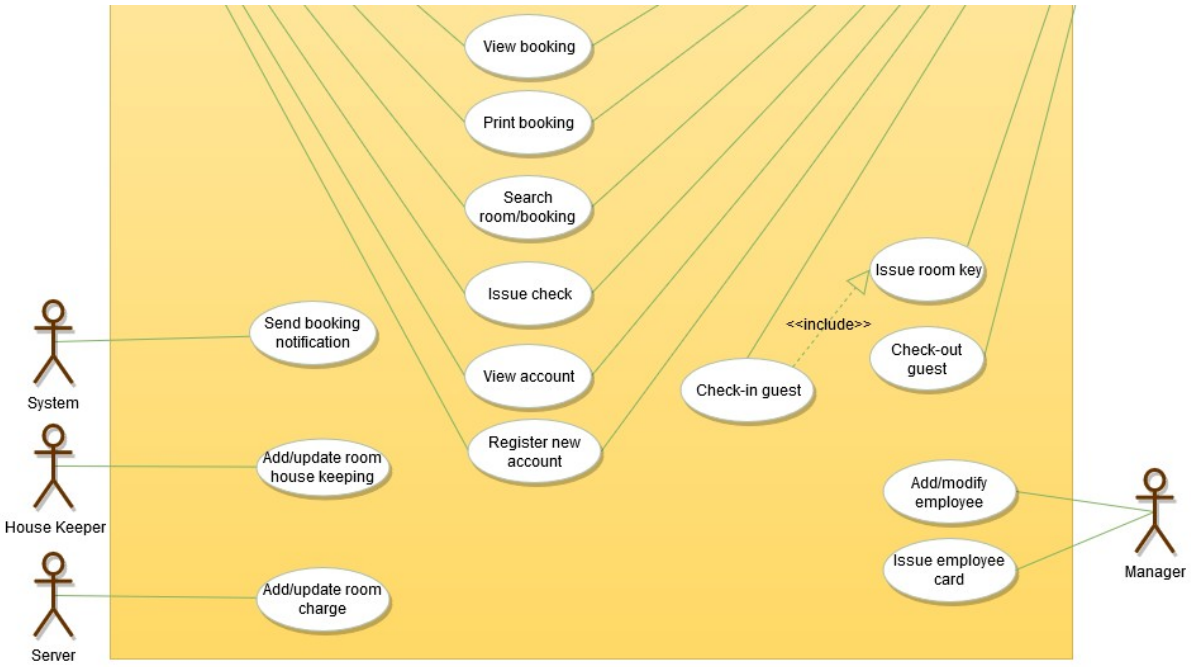
- **Guest:** All guests can search the available rooms, as well as make a booking.
- **Receptionist:** Mainly responsible for adding and modifying rooms, creating room bookings, check-in, and check-out customers.

- **System:** Mainly responsible for sending notifications for room booking, cancellation, etc.
- **Manager:** Mainly responsible for adding new workers.
- **Housekeeper:** To add/modify housekeeping record of rooms.
- **Server:** To add/modify room service record of rooms.

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

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

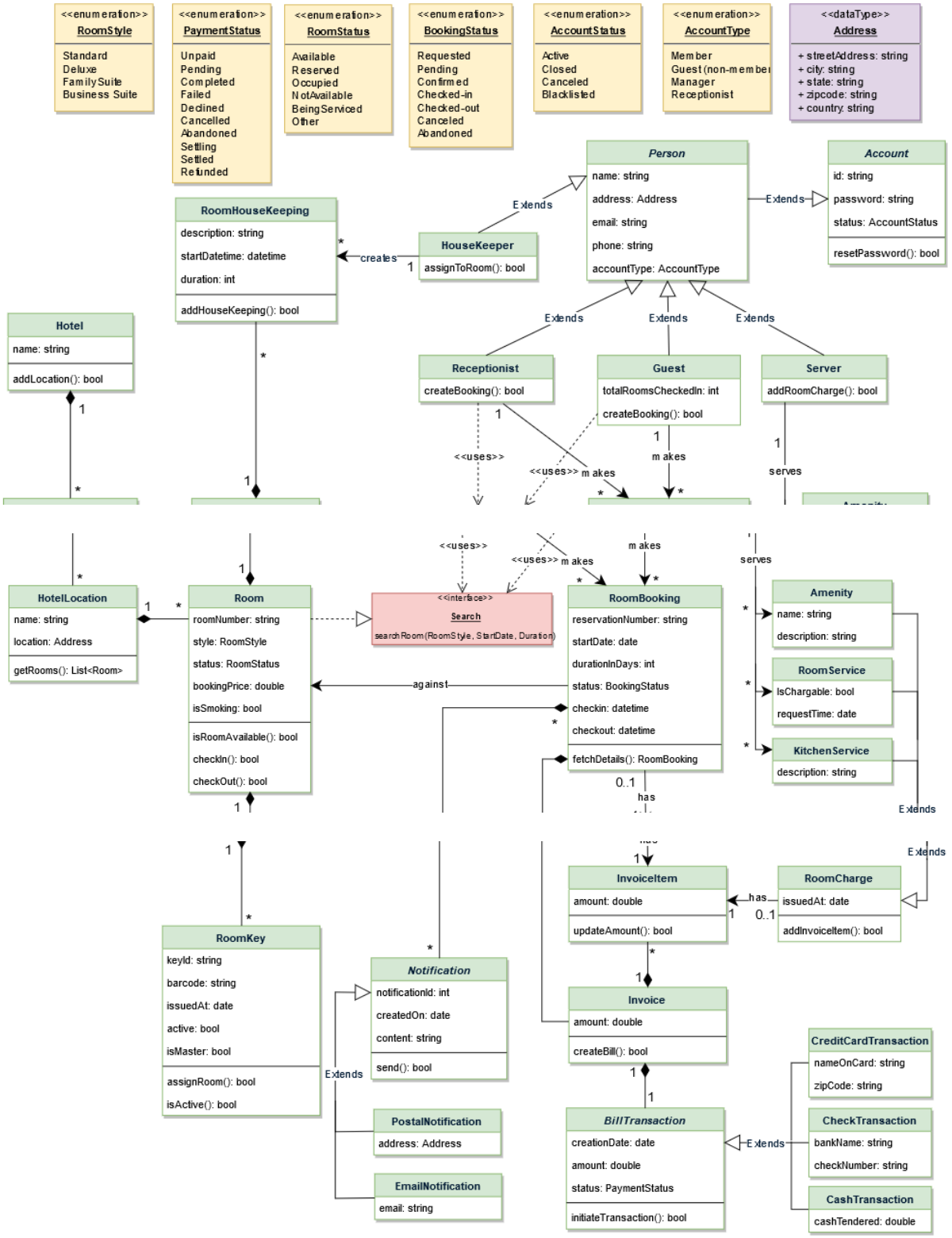




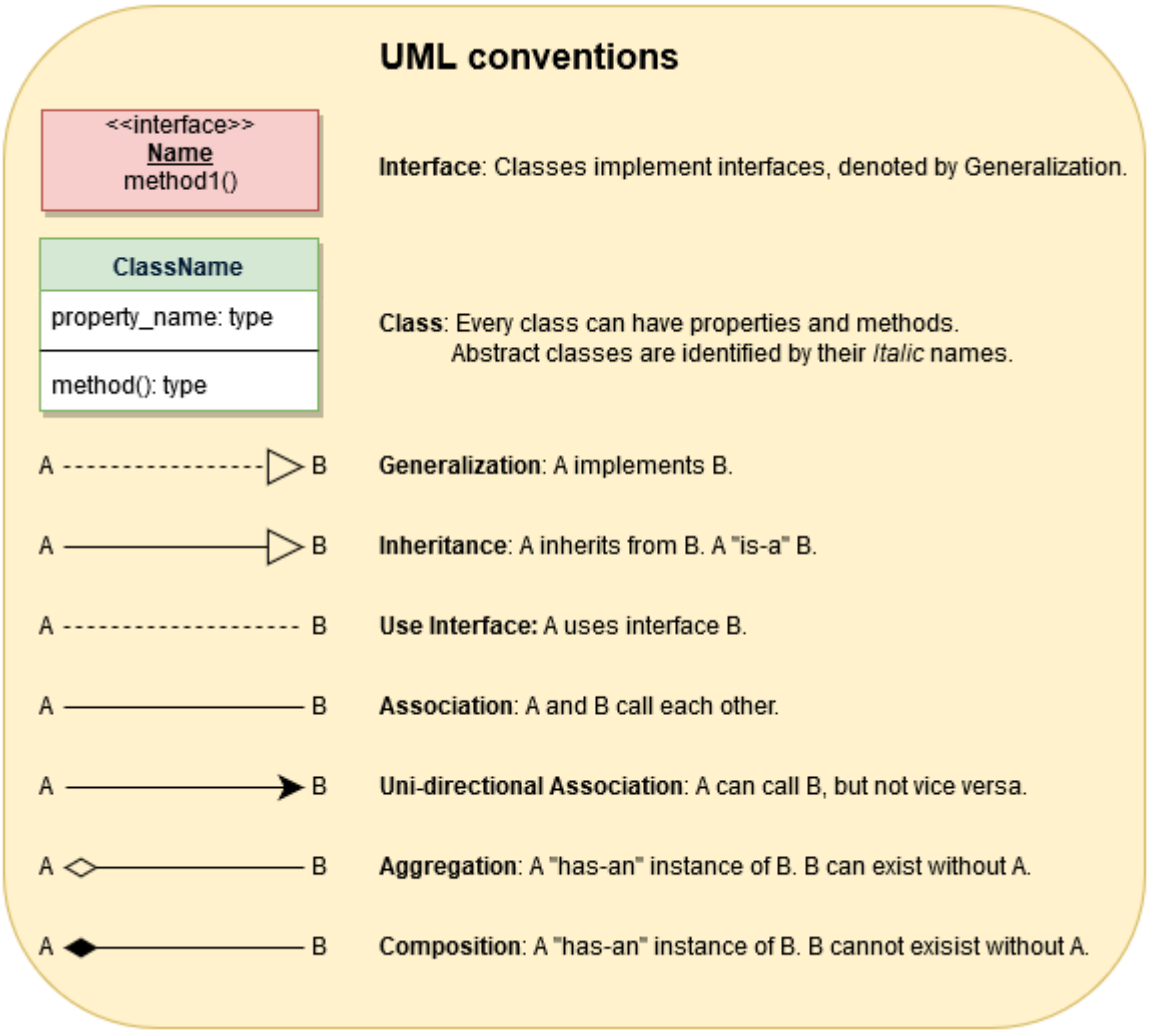
Class diagram

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 housekeeping records of a room, and a Server will handle room service.
- **RoomBooking:** This class will be responsible for managing bookings against a room.
- **Notification:** Will take care of sending notifications to guests.
- **RoomHouseKeeping:** To keep track of all housekeeping record for rooms.
- **RoomCharge:** Encapsulates the details about different types of room services that the guests have requested.
- **Invoice:** Contains different invoice-items for every charge against the room.
- **RoomKey:** Each room can be assigned an electronic card keys. Keys will have a barcode and will be uniquely identified by a key-id.

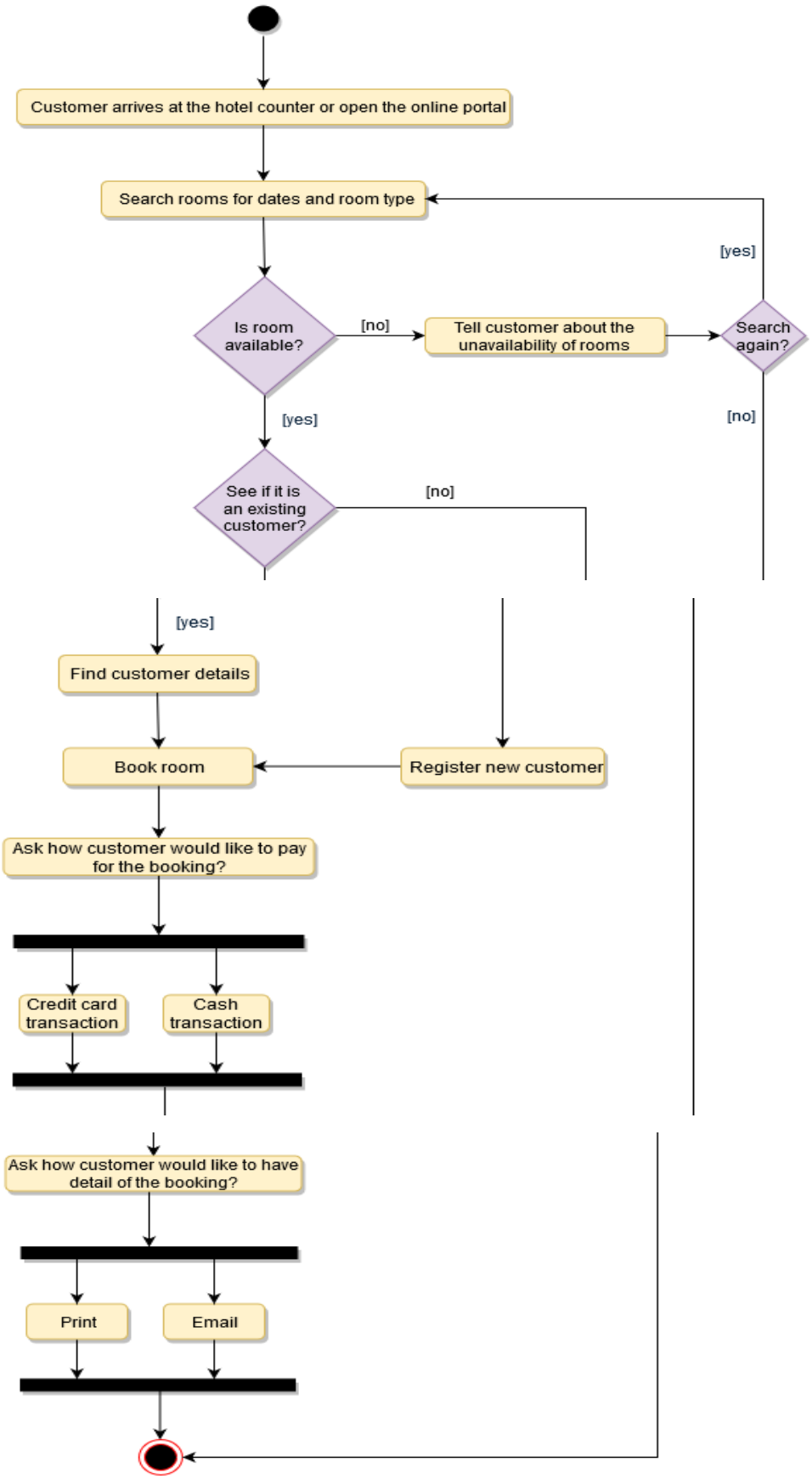


Class diagram

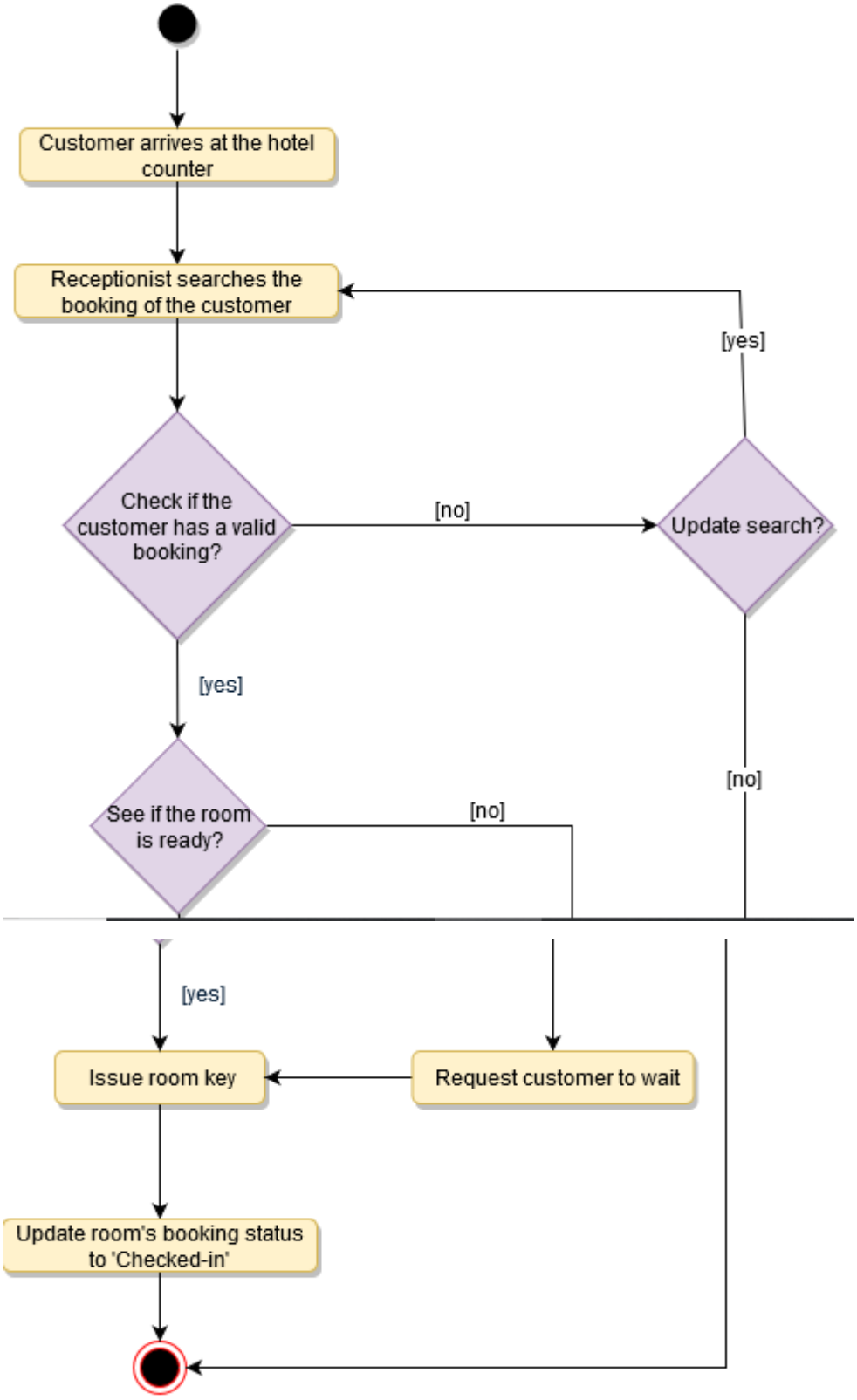


Activity diagrams

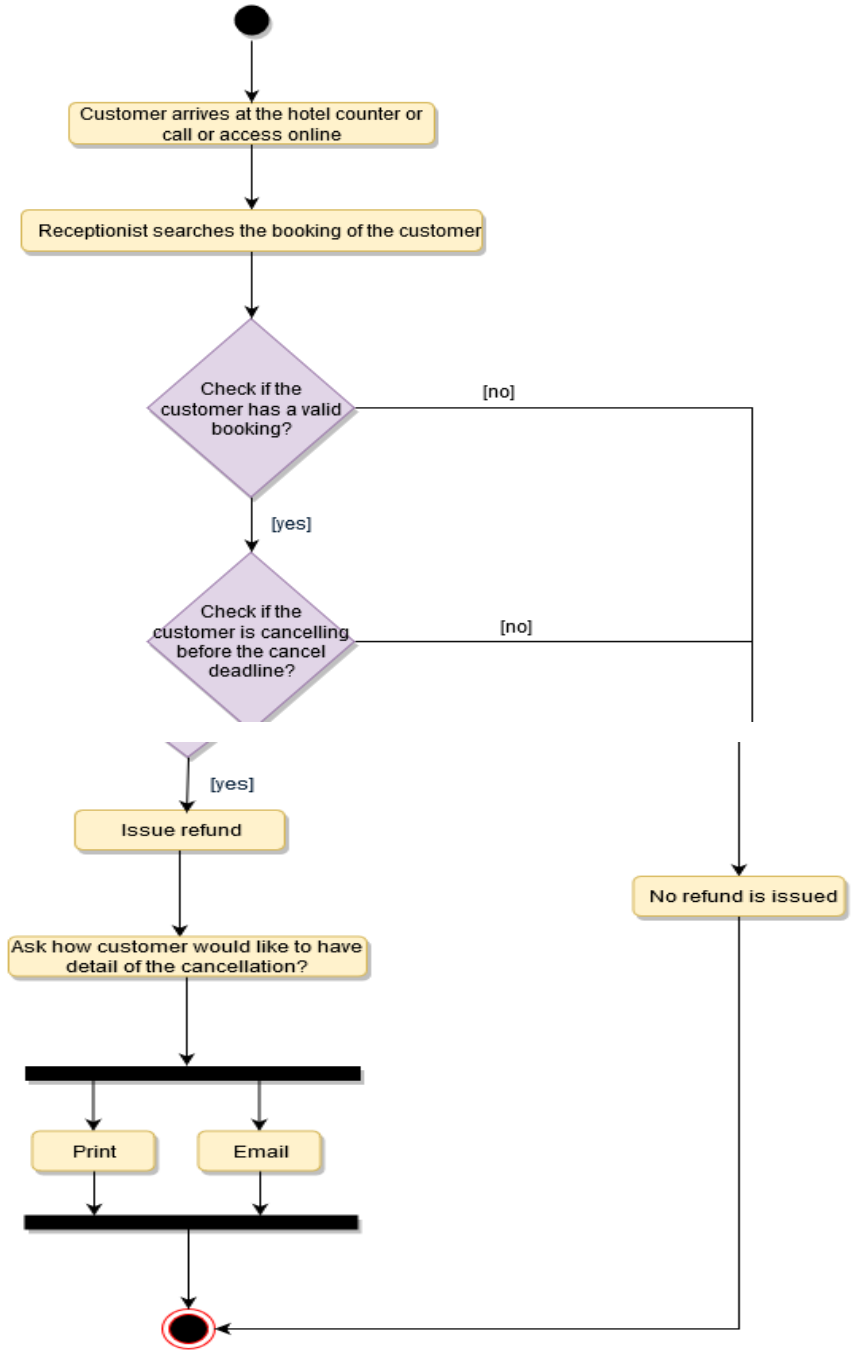
Make a room booing: Any guest or receptionist can perform this activity. Here are the set of steps to book a room:


Customer arrives at the hotel counter or open the online portal
Search rooms for dates and room type
Is room available?
[no]
Tell customer about the unavailability of rooms
Search again?
[yes]
[no]
[yes]
See if it is an existing customer?
[no]
[yes]
Find customer details
Book room
Register new customer
Ask how customer would like to pay for the booking?
Credit card transaction
Cash transaction
Ask how customer would like to have detail of the booking?
Print
Email

Check-in: Guest will check-in against their booking. Receptionist can perform this activity. Here are the different steps of this activity:



Cancel a booking: Guest can cancel their booking. Receptionist can perform this activity. Here are the different steps of this activity:



Code

Here is the high-level definition for the classes described above.

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

```
public enum RoomStyle{
  STANDARD,
  DELUXE,
  FAMILY_SUITE,
  BUSINESS_SUITE
}
```

```
public enum RoomStatus{
  AVAILABLE,
  RESERVED,
  OCCUPIED,
  NOT_AVAILABLE,
  BEING_SERVICED,
  OTHER
}

public enum BookingStatus{
  REQUESTED.
  PENDING,
  CONFIRMED,
  CHECKED_IN,
  CHECKED_OUT,
  CANCELLED,
  ABONDENDED
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  BLOCKED
}

public enum AccountType{
  MEMBER,
  GUEST,
  MANAGER,
  RECEPTIONIST
}

public enum PaymentStatus{
  UNPAID,
  PENDING,
  COMPLETED,
  FILLED,
  DECLINED,
  CANCELLED,
  ABONDENED,
  SETTLING,
  SETTLED,
  REFUNDED
}

public class Address {
  private String streetAddress;
```

```
  private String city;
  private String state;
  private String zipCode;
  private String country;
}
```

Account, Person, Guest, Receptionist, and Additional DriverServer: These classes represent different people that interact with our system:

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

public abstract class Account {
  private String id;
  private String password;
  private AccountStatus status;

  public boolean resetPassword();
}

public abstract class Person extends Account {
  private String name;
  private Address address;
  private String email;
  private String phone;
}


public class Guest extends Person {
  private int totalRoomsCheckedIn;

  public List<RoomBooking> getBookings();
}

public class Receptionist extends Person {
  public List<Member> searchMember(String name);
  public boolean createBooking();
}

public class Server extends Person {
  public boolean addRoomCharge(Room room, RoomCharge roomCharge);
}
```

Hotel and HotelLocation: These classes represent the top level classes of the system:

```
public class HotelLocation {
  private String name;
  private Address location;

  public Address geRooms();
}
```

```
public class Hotel {
  private String name;
  private List<HotelLocation> locations;

  public boolean addLocation(HotelLocation location);
}
```

Room, RoomKey, and RoomHouseKeeping: To encapsulate a room, room key, and house keeping:

```
public interface Search {
  public static List<Room> search(RoomStyle style, Date startDate, int duration);
}


public class Room implements Search {
  private String roomNumber;
  private RoomStyle style;
  private RoomStatus status;
  private double bookingPrice;
  private boolean isSmoking;

  private List<RoomKey> keys;
  private List<RoomHouseKeeping> houseKeepingLog;

  public boolean isRoomAvailable();
  public boolean checkIn();
  public boolean checkOut();

  public static List<Room> search(RoomStyle style, Date startDate, int duration) {
    // return all rooms with the given style and availability
  }
}

public class RoomKey {
  private String keyId;
  private String barcode;
  private Date issuedAt;
  private boolean active;
  private boolean isMaster;

  public boolean assignRoom(Room room);
  public boolean isActive();
}

public class RoomHouseKeeping
 {
  private String description;
  private Date startDatetime;
  private int duration;
  private HouseKeeper houseKeeper;
```

```
  public boolean addHouseKeeping(Room room);
}
```

RoomBooking and RoomCharge: To encapsulate a booking and different charges against a booking:

```
public class RoomBooking {
  private String reservationNumber;
  private Date startDate;
  private int durationInDays;
  private BookingStatus status;
  private Date checkin;
  private Date checkout;

  private int guestID;
  private Room room;
  private Invoice invoce;
  private List<Notification> notifications;

  public static RoomBooking fectchDetails(String reservationNumber);
}

public abstract class RoomCharge {
  public Date issueAt;
  public boolean addInvoiceItem(Invoice invoice);
}

public class Amenity extends RoomCharge {
  public String name;
  public String description;
}

public class RoomService extends RoomCharge {
  public boolean isChargable;
  public Date requestTime;
}

public class KitchenService extends RoomCharge {
  public String description;
}
```

Design a Restaurant Management system

A Restaurant Management System is a software build to handle all restaurant activities easily and safely. 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 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 different seats.
5. Each meal can have multiple meal items corresponding to the menu items.
6. The system should be able to retrieve information about what tables are currently available to place walk-in customers.
7. The system should support the reservation of tables.
8. The receptionist should be able to search available tables for data/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.

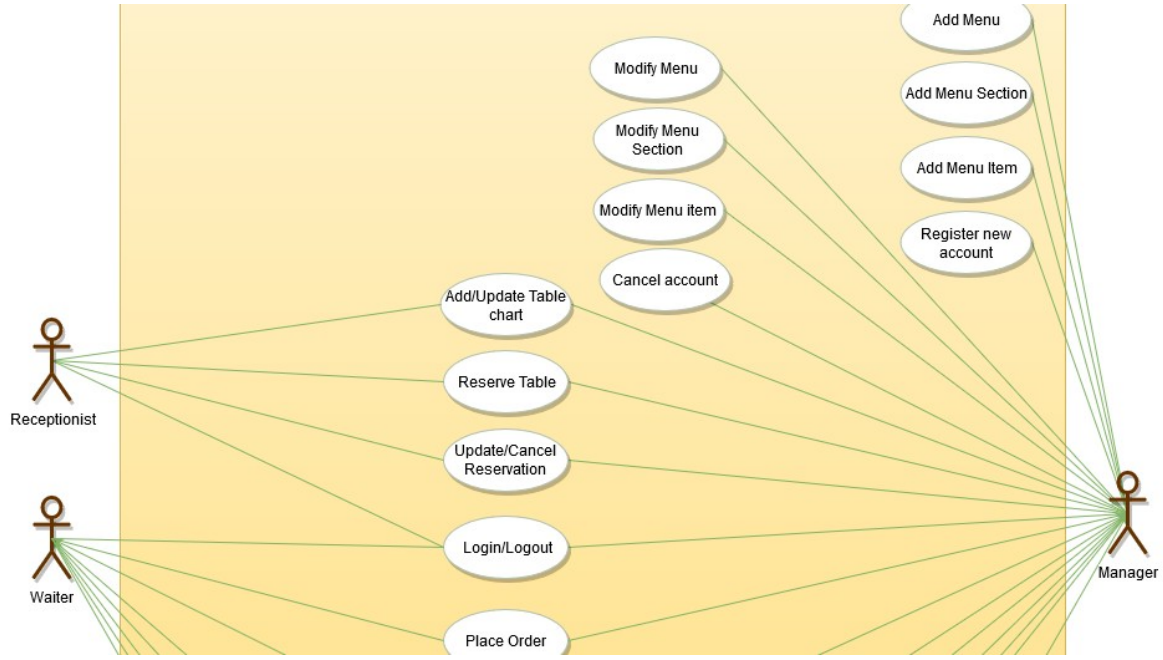
Usecase diagram

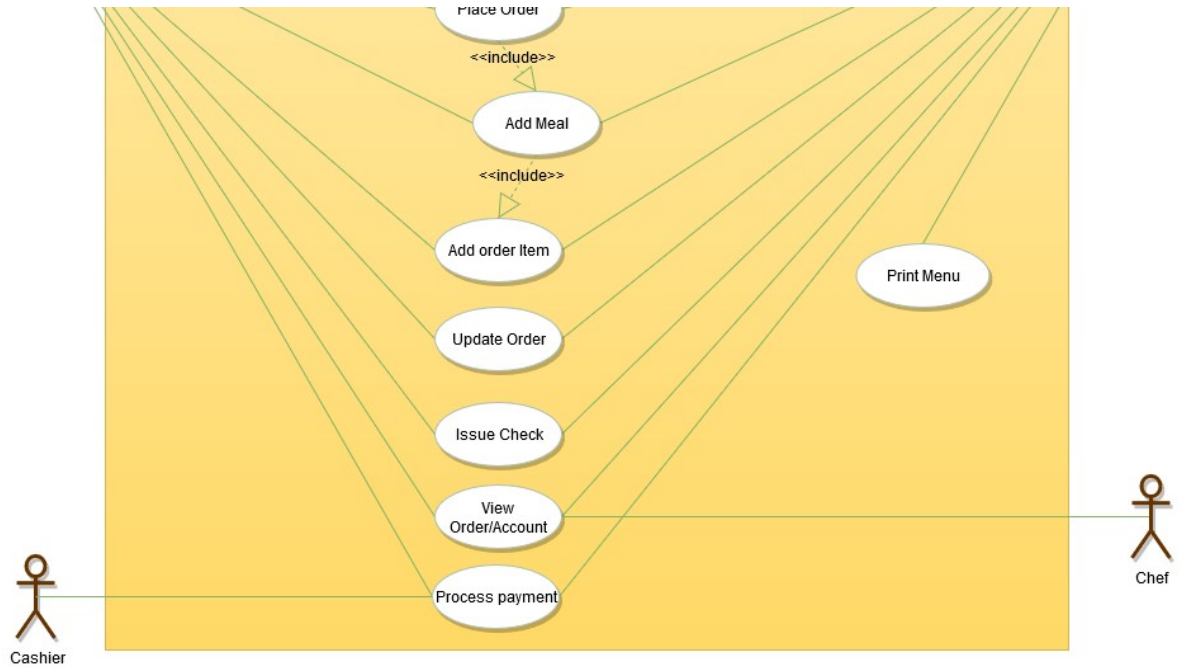
Here are the main Actors in our system:

- **Receptionist:** Mainly responsible for adding and modifying tables and their layout, creating and canceling table reservations.
- **Waiter:** To take/modify orders.
- **Manager:** Mainly responsible for adding new workers and modifying menu.
- **Chef:** To view and work on an order.
- **Cashier:** To create bill/check and process payments.
- **System:** Mainly responsible for sending notifications for table reservations, cancellation, etc.

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

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



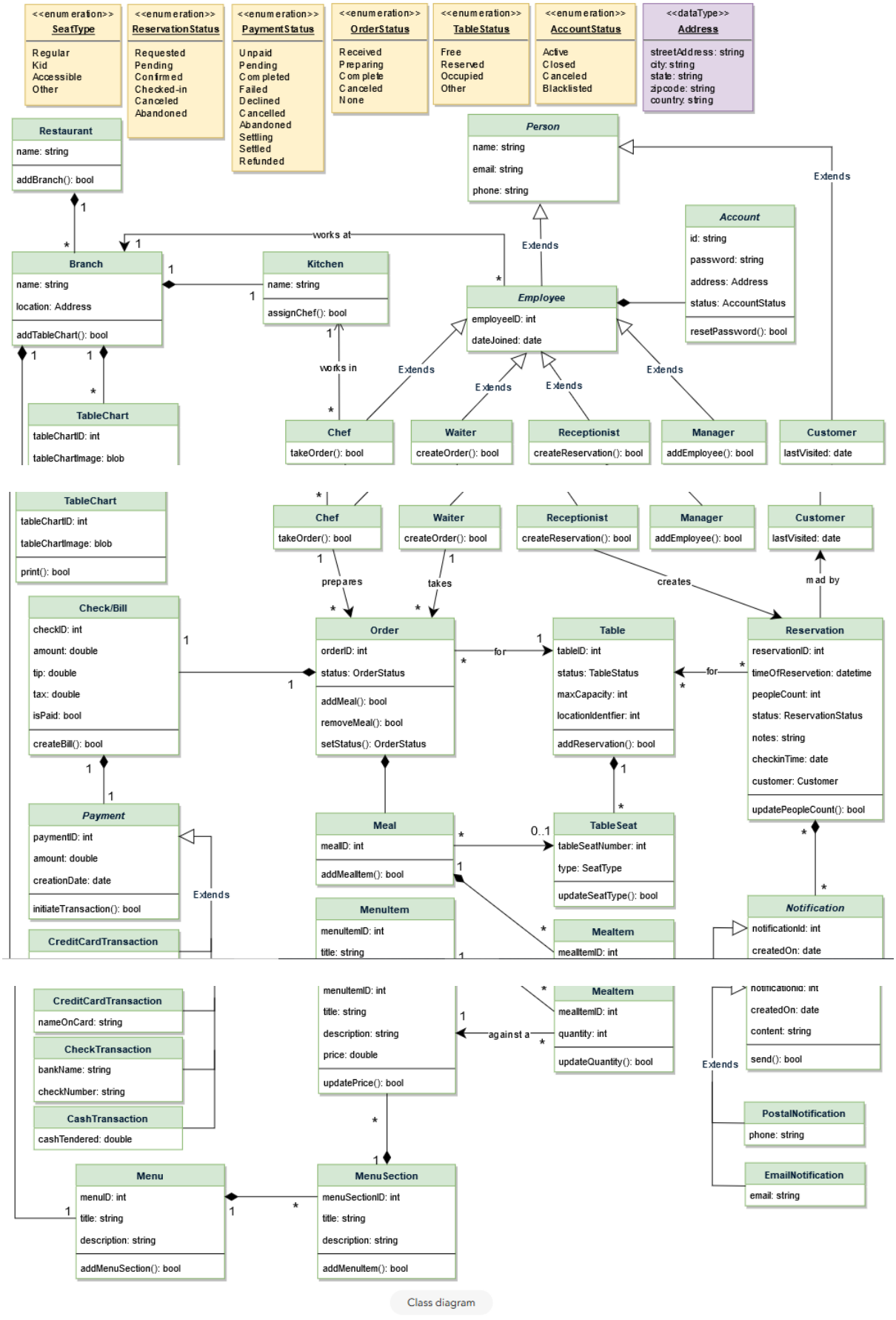


Use case diagram

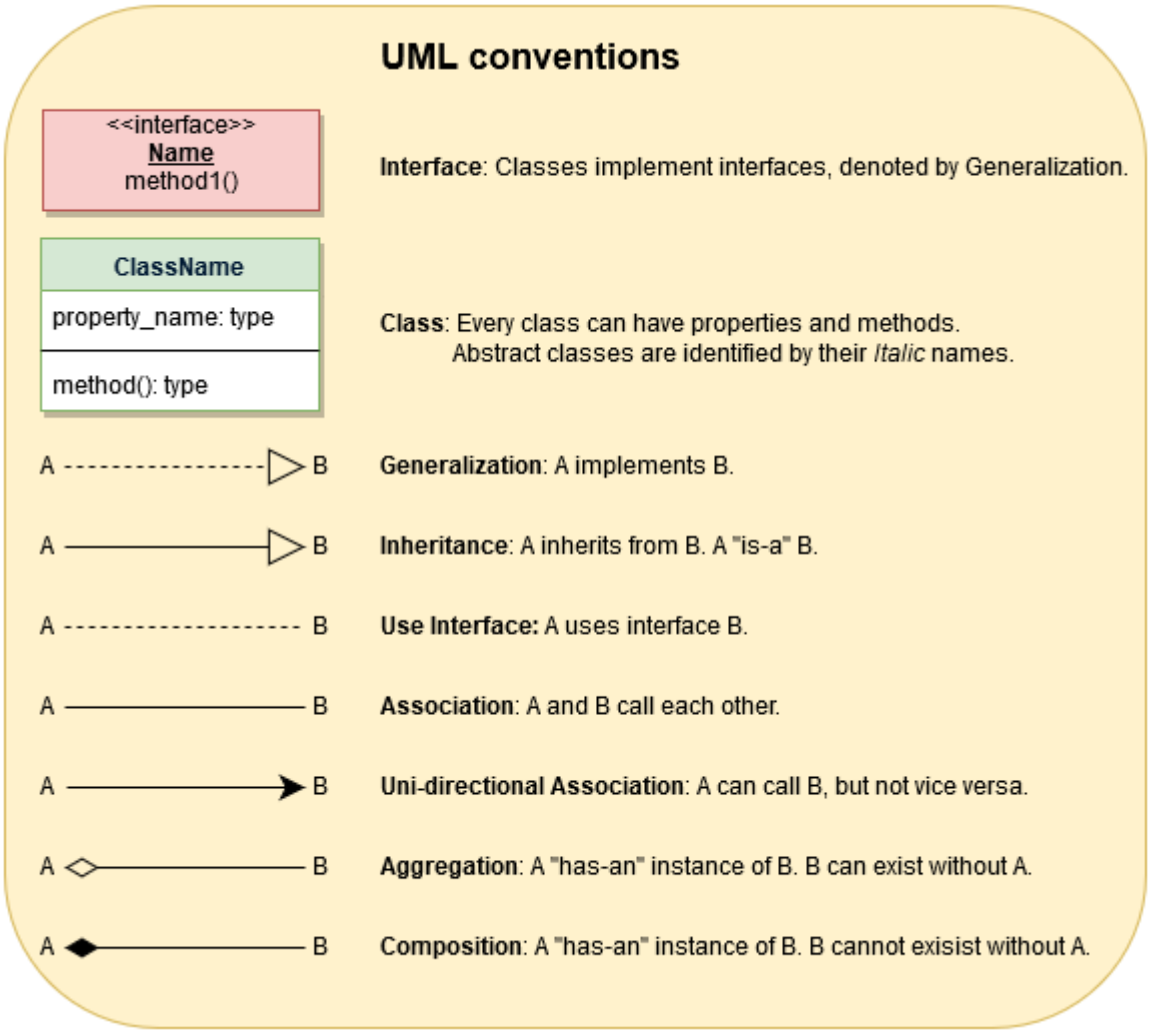
Class diagram

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

- **Restaurant:** This class represents a restaurant. Each restaurant has registered employees. These employees are part of the restaurant because if a restaurant becomes inactive all its employees will get inactive automatically.
- **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 sitting 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. The waiter will place orders in the system.
- **Notification:** Will take care of sending notifications to customers.
- **Bill:** Contains different bill-items for every charge against the room.

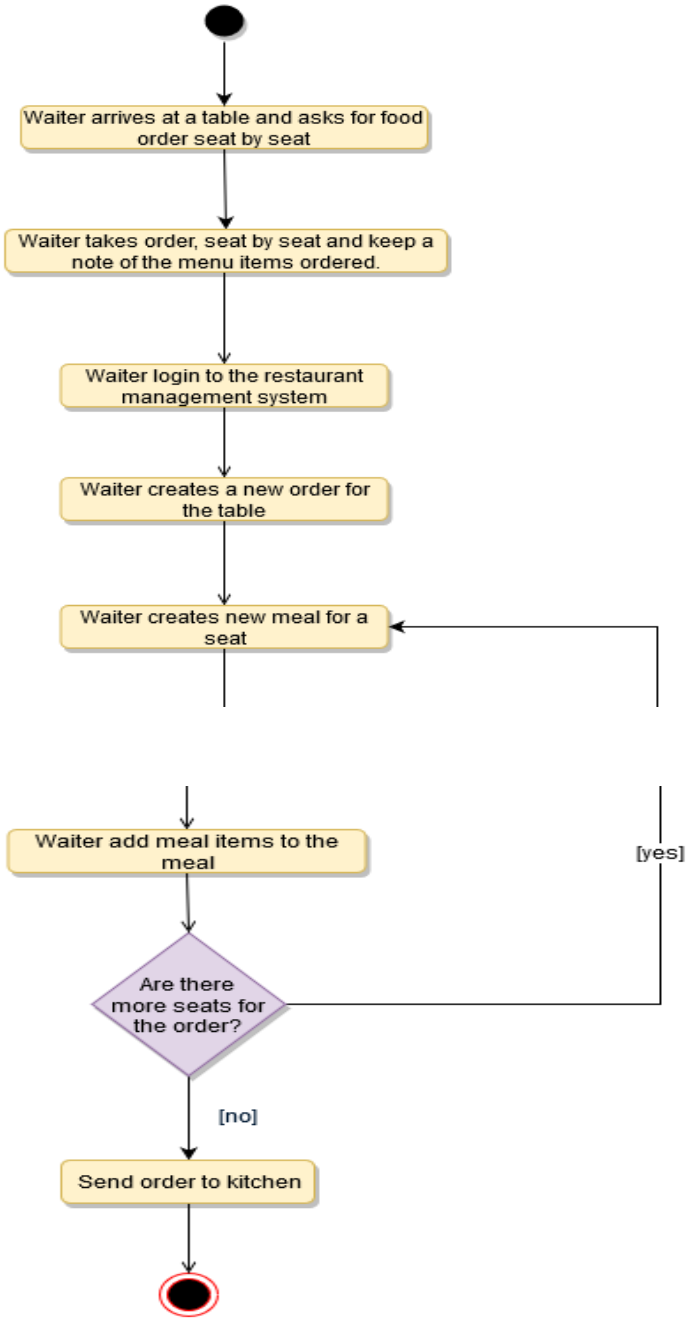


Class diagram

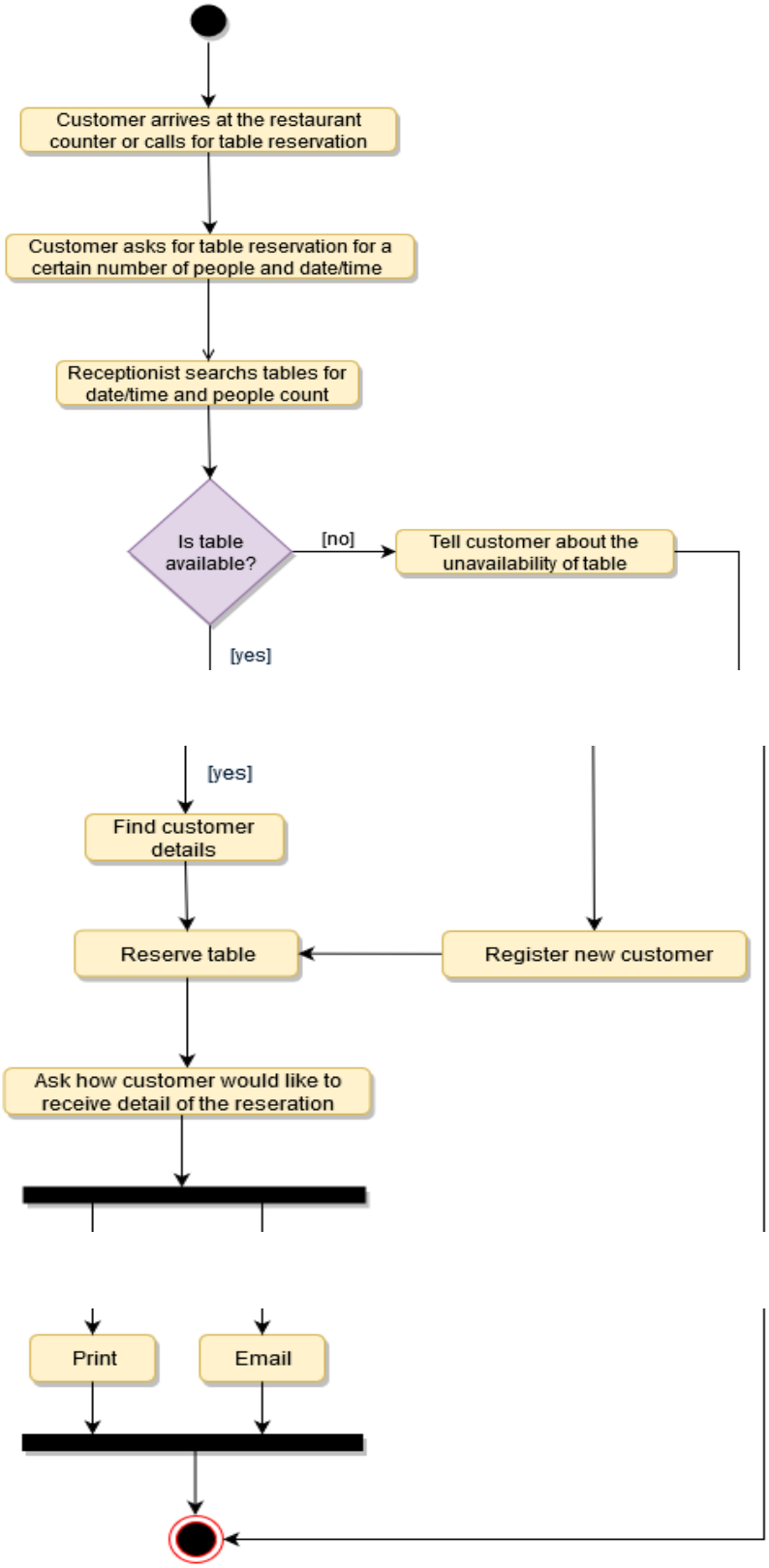


Activity diagrams

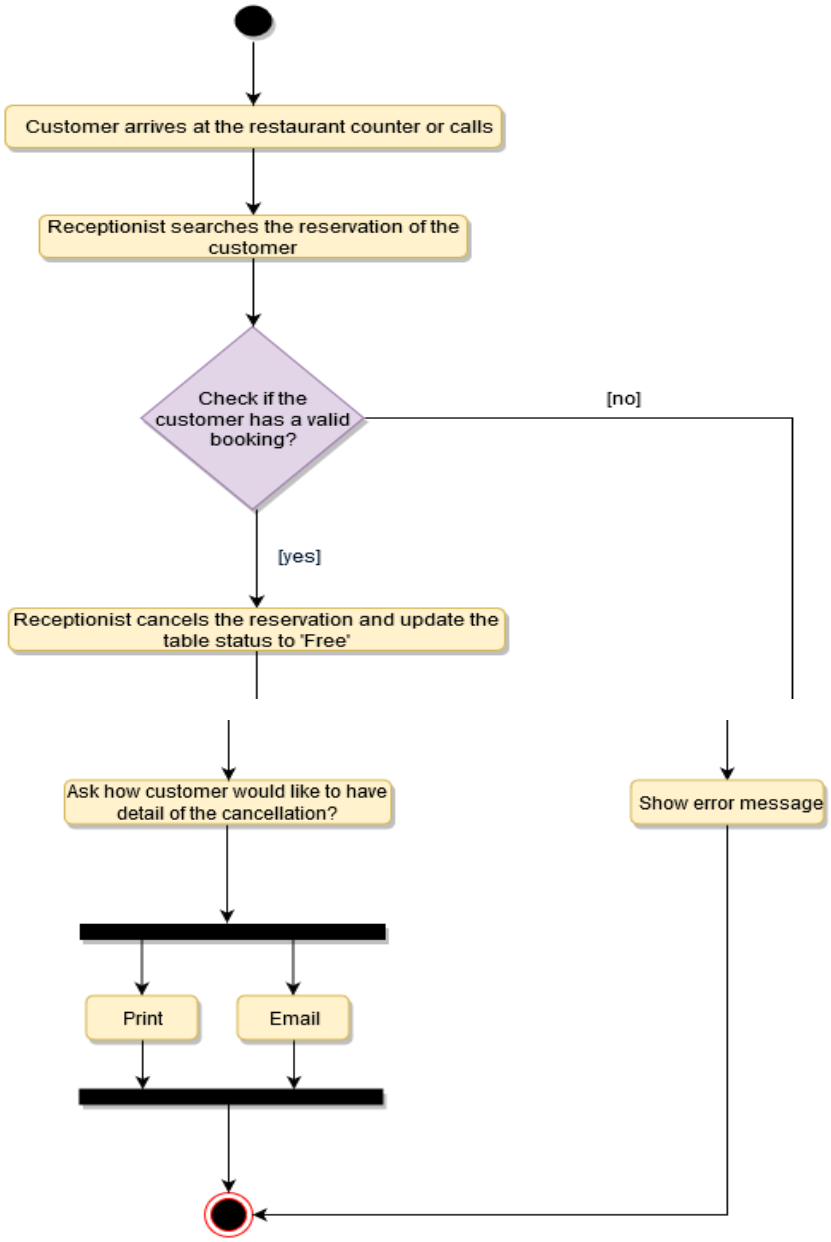
Place order: Any waiter can perform this activity. Here are the set of steps to place an order:



Make a reservation: Any receptionist can perform this activity. Here are the set of steps to place an order:



Cancel a reservation: Any receptionist can perform this activity. Here are the set of steps to place an order:



Code

Here is the high-level definition for the classes described above.

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

```
public enum ReservationStatus{
  REQUESTED,
  PENDING,
  CONFIRMED,
  CHECKED_IN,
  CANCELED,
  ABANDONED
}
```

```
public enum SeatType{
  REGULAR,
  KID,
  ACCESSIBLE,
  Other
}

public enum OrderStatus{
  RECEIVED,
  PREPARING,
  COMPLETED,
  CANCELED,
  NONE
}

public enum TableStatus{
  FREE,
  RESERVED,
  OCCUPIED,
  OTHER
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  BLOCKED
}

public enum PaymentStatus{
  UNPAID,
  PENDING,
  COMPLETED,
  FILLED,
  DECLINED,
  CANCELLED,
  ABONDENED,
  SETTLING,
  SETTLED,
  REFUNDED
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
```

```
}
```

Account, Person, Employee, Receptionist, Manager, and Chef: These classes represent different people that interact with our system:

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

public abstract class Account {
  private String id;
  private String password;
  private Address address;
  private AccountStatus status;

  public boolean resetPassword();
}

public abstract class Person extends Account {
  private String name;
  private String email;
  private String phone;
}


public class Employee extends Person {
  private int employeeID;
  private Date dateJoined;
}

public class Receptionist extends Employee {
  public boolean createReservation();
  public List<Customer> searchCustomer(String name);
}

public class Manager extends Employee {
  public boolean addEmployee();
}

public class Chef extends Employee {
  public boolean takeOrder();
}
```

Restaurant, Branch, Kitchen, TableChart: These classes represent the top-level classes of the system:

```
public class Kitchen {
  private String name;
  private Chef[] chefs;

  private boolean assignChef();
}
```

```
public class Branch {
  private String name;
  private Address location;
  private Kitchen kitchen;

  public Address addTableChart();
}

public class Restaurant {
  private String name;
  private List<Branch> branches;

  public boolean addBranch(Branch branch);
}

public class TableChart {
  private int tableChartID;
  private byte[] tableChartImage;

  public bool print();
}
```

Table, TableSeat, and Reservation: Each table can have multiple seats and customers can make reservations for tables:

```
public class Table {
  private int tableID;
  private TableStatus status;
  private int maxCapacity;
  private int locationIdentifier;

  private List<TableSeat> seats;

  public boolean isTableFree();
  public boolean addReservation();

  public static List<Room> search(int capacity, Date startDate, int duration) {
    // return all rooms with the given capacity and availability
  }
}

public class TableSeat {
  private int tableSeatNumer;
  private SeatType type;

  public boolean updateSeatType(SeatType type);
}

public class Reservation {
  private int reservationID;
  private Date timeOfReservation;
```

```
  private int peopleCount;
  private ReservationStatus status;
  private String notes;
  private Date checkinTime;
  private Customer customer;

  private Table[] tables;
  private List<Notification> notifications;
  public boolean updatePeopleCount(int count);
}
```

Menu, MenuSection, and MenuItem: Each restaurant branch will have its own menu, each menu will have multiple menu sections, which contains menu items:

```
public class Menu {
  private int menuID;
  private String title;
  private String description;
  private List<MenuSection> menuSections;

  public boolean addMenuSection(MenuSection menuSection);
  public boolean print();
}


public class MenuSection {
  private int menuSectionID;
  private String title;
  private String description;
  private List<MenuItem> menuItems;

  public boolean addMenuItem(MenuItem menuItem);
}

public class MenuItem {
  private int menuItemID;
  private String title;
  private String description;
  private double price;

  public boolean updatePrice(double price);
}
```

Order, Meal, and MealItem: Each order will have meals for table seats:

```
public class Meal {
  private int mealID;
  private TableSeat seat;
  private List<MenuItem> menuItems;

  public boolean addMealItem(MealItem mealItem);
}
```

```
public class MealItem {
  private int mealItemID;
  private int quantity;
  private MenuItem menuteItem;

  public boolean updateQuantity(int quantity);
}

public class Order {
  private int OrderID;
  private OrderStatus status;
  private Date creationTime;

  private Meal[] meals;
  private Table table;
  private Check check;
  private Waiter waiter;
  private Chef chef;

  public boolean addMeal(Meal meal);
  public boolean removeMeal(Meal meal);
  public OrderStatus getStatus();
  public boolean setStatus(OrderStatus status);
}
```

Design Chess

Chess is a two-player strategy board game played on a chessboard, a checkered gameboard with 64 squares arranged in an 8×8 grid. There are a few versions of game types for chess that people play all around in the world. In this design problem, we are going to focus on designing a two-player online chess game.

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 the moves of both players.
7. Each side will start will 8 pawns, 2 rocks, 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.

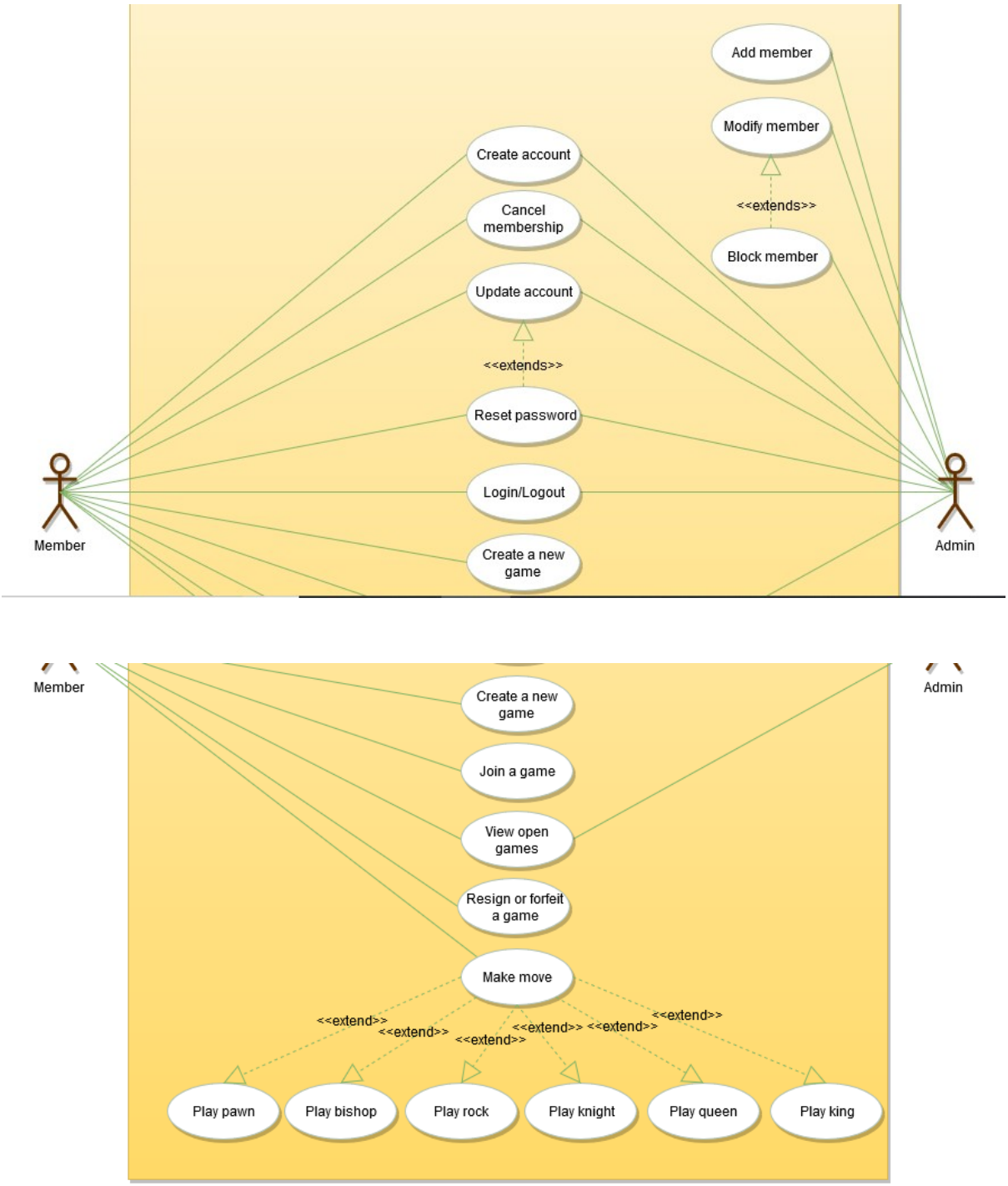
Usecase diagram

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 from a game:** A player resigns/forfeit from 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 game log.

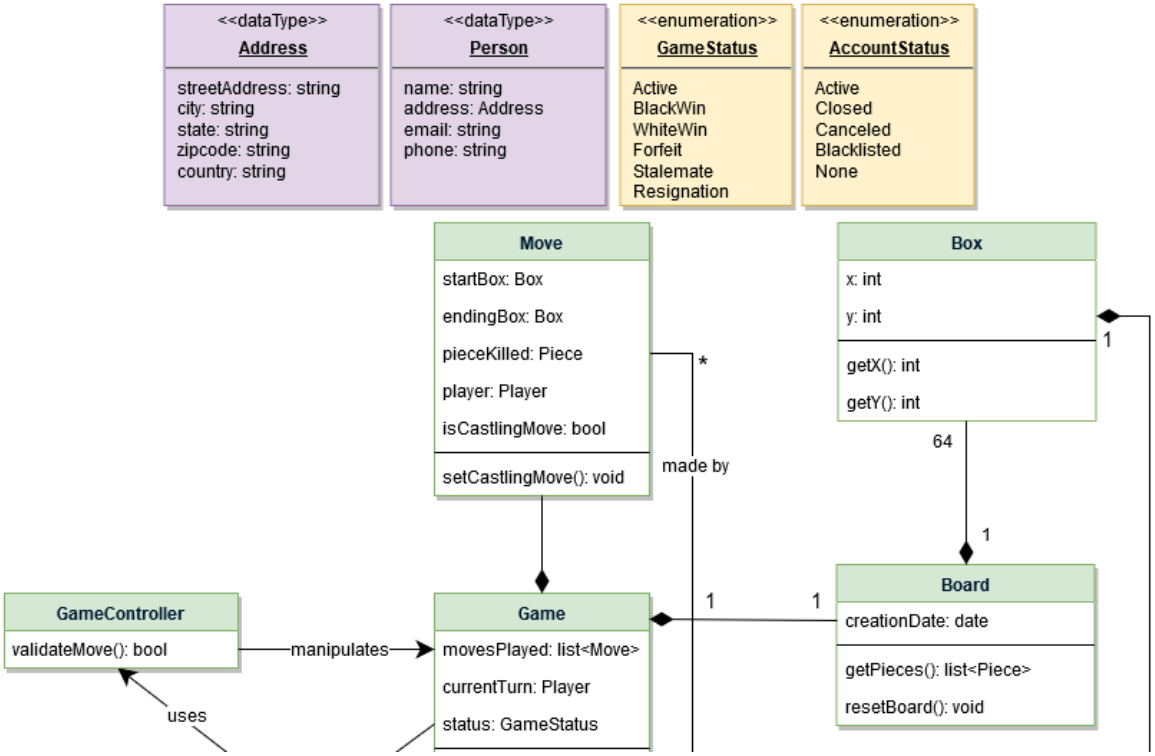


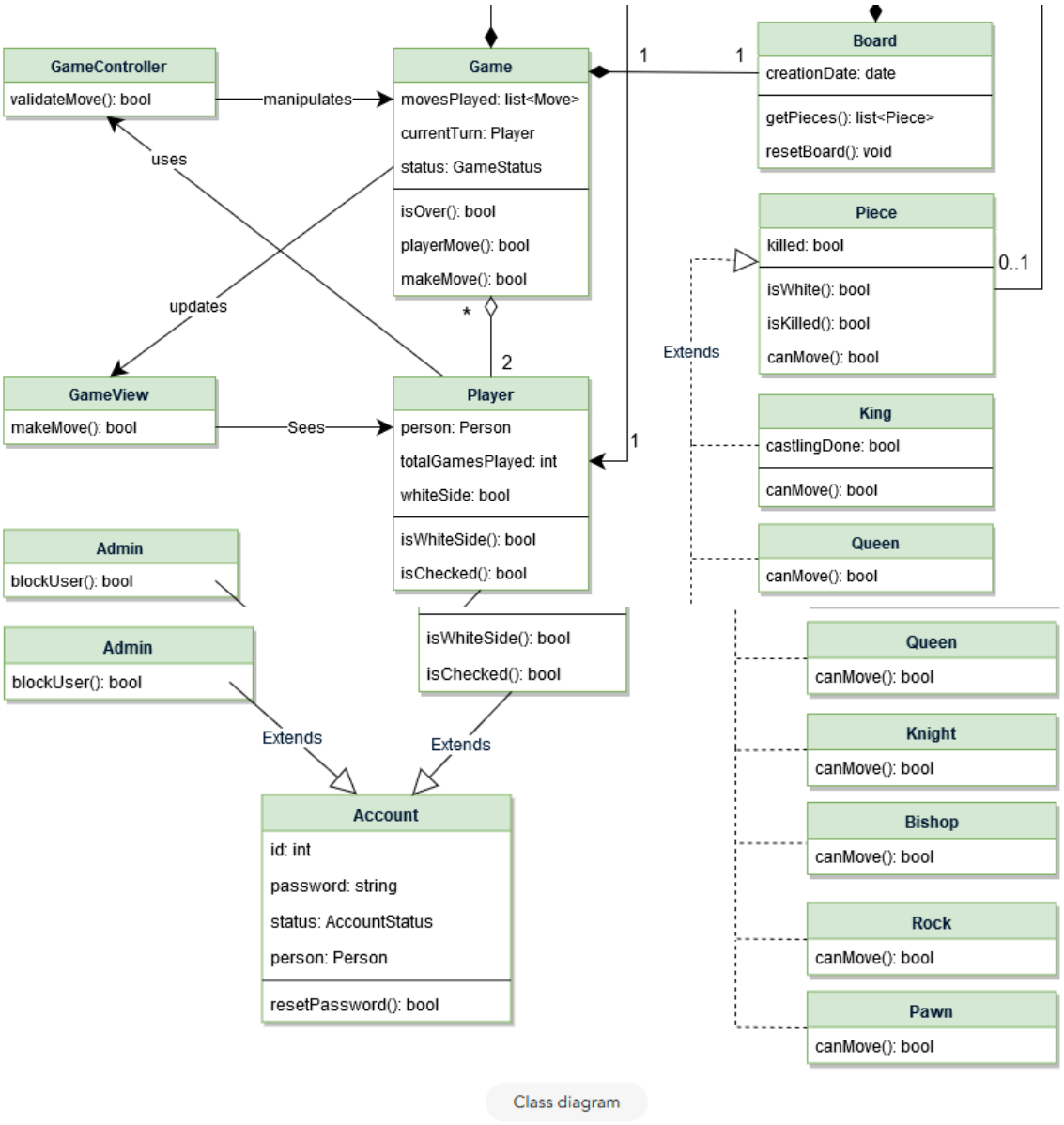
Class diagram

Here are the main classes for chess:

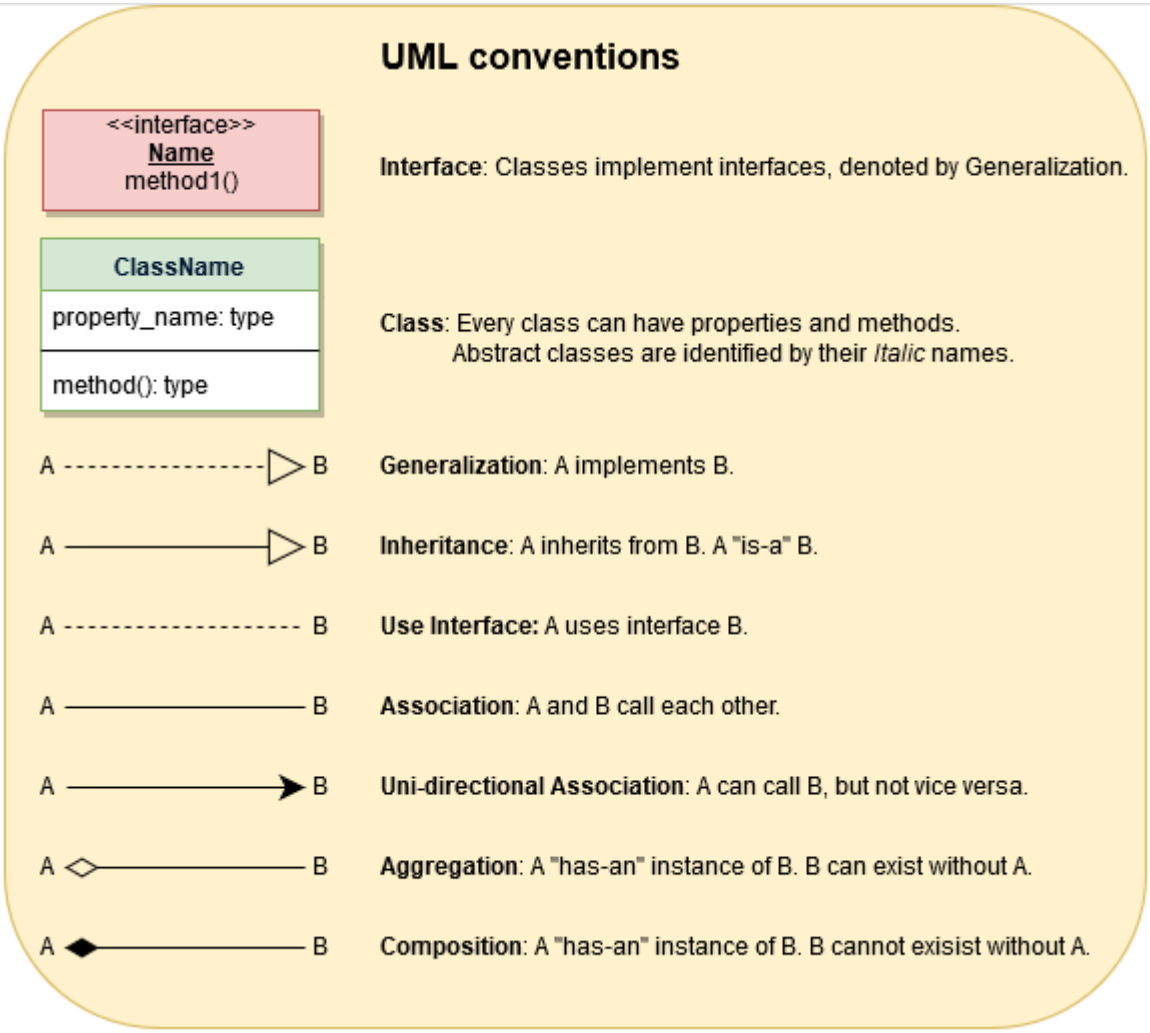
- **Player:** Player class represents one of the participants playing the game. It keeps track of which side (black or white) the player is playing.
- **Account:** We'll have two types of accounts in the system, one will be a player, and the other will be an admin.

- **Game:** This class controls the flow of a game. It keeps track of all the game moves, which player has the current turn, and what will be 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. It contains which color the piece represents and if the piece is currently killed or not. This would be an abstract class, all game pieces will extend it.
- **Move:** Represents a game move, containing starting and ending box. 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 a piece capture.
- **GameController:** Player uses GameControler to make moves.
- **GameView:** Game class updates the GameView to show changes to Players.



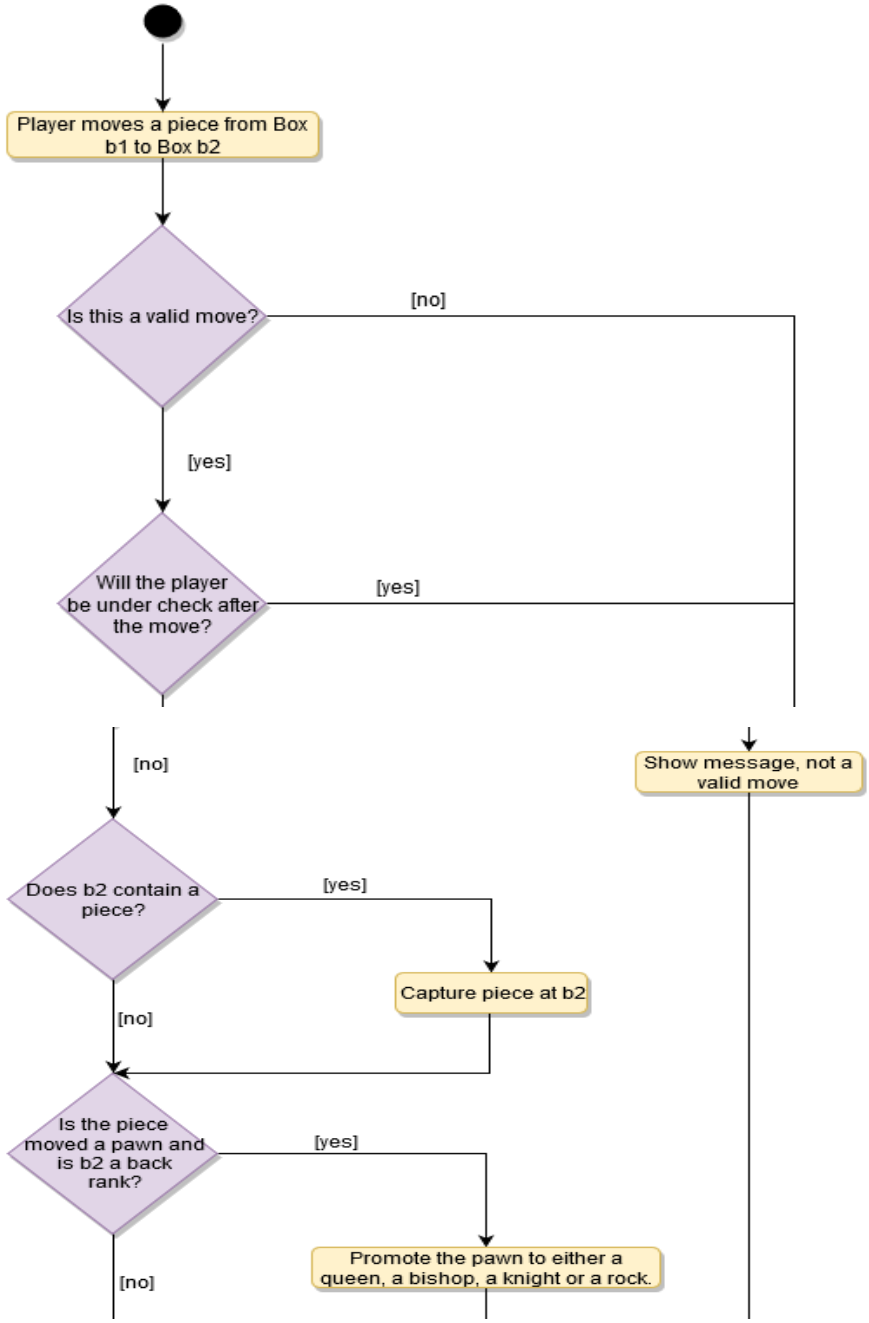


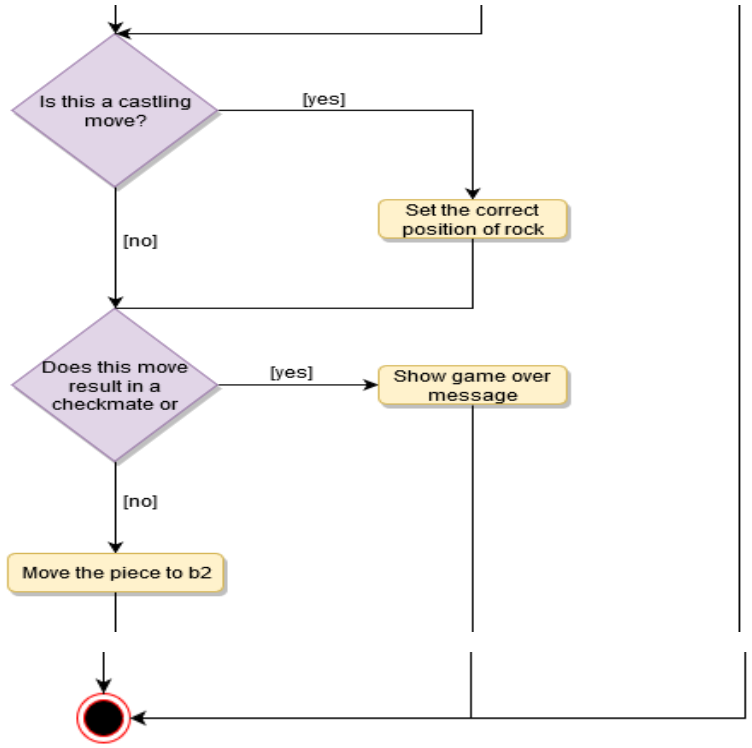
Class diagram



Activity diagrams

Make move: Any Player can perform this activity. Here are the set of steps to make a move:


Player moves a piece from Box b1 to Box b2
Is this a valid move?
[no]
[yes]
Will the player be under check after the move?
[yes]
[no]
Show message, not a valid move
Does b2 contain a piece?
[yes]
Capture piece at b2
[no]
Is the piece moved a pawn and is b2 a back rank?
[yes]
Promote the pawn to either a queen, a bishop, a knight or a rock.
[no]



Code

Here is the code for the top use-cases.

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

```
public enum GameStatus {
  ACTIVE,
  BLACK_WIN,
  WHITE_WIN,
  FORFIET,
  STALEMATE,
  RESIGNATION
}

public enum AccountStatus{
    ACTIVE,
    CLOSED,
    CANCELED,
    BLACKLISTED,
    None;
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
```

```
  private String country;
}

public class Person {
  private String name;
  private Address address;
  private String email;
  private String phone;
}
```

Box: To encapsulate a cell on the chess board:

```
public class Box {
  private Piece piece;
  private int x;
  private int y;

  public Box(int x, int y, Piece piece) {
    this.setPiece(piece);
    this.setX(x);
    this.setY(y);
  }

  public Piece getPiece() {
    return this.piece;
  }

  public void setPiece(Piece p) {
    this.piece = p;
  }

  public int getX() {
    return this.x;
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getY() {
    return this.y;
  }

  public void setY(int y) {
    this.y = y;
  }
}
```

Piece: An abstract class to encapsulate common functionality of all chess pieces:

```
public abstract class Piece {
```

```
  private boolean killed = false;
  private boolean white = false;

  public Piece(boolean white) {
    this.setWhite(white);
  }

  public boolean isWhite() {
    return this.white == true;
  }

  public void setWhite(boolean white) {
    this.white = white;
  }

  public boolean isKilled() {
    return this.killed == true;
  }

  public void setKilled(boolean killed) {
    this.killed = killed;
  }

  public abstract boolean canMove(Board board, Box start, Box end);
}
```

King: To encapsulate King as a chess piece:

```
public class King extends Piece {
  private boolean castlingDone = false;

  public King(boolean white) {
    super(white);
  }

  public boolean isCastlingDone() {
    return this.castlingDone == true;
  }

  public void setCastlingDone(boolean castlingDone) {
    this.castlingDone = castlingDone;
  }

  @Override
  public boolean canMove(Board board, Box start, Box end) {
    // we can't move the piece to a box that has a piece of the same color
    if(end.getPiece().isWhite() == this.isWhite()) {
      return false;
    }

    int x = Math.abs(start.getX() - end.getX());
```

```
    int y = Math.abs(start.getY() - end.getY());
    if(x + y == 1) {
      // check if this move will not result in king being attacked, if so return true
      return true;
    }

    return this.isValidCastling(board, start, end);
  }

  private boolean isValidCastling(Board board, Box start, Box end) {

    if(this.isCastlingDone()) {
      return false;
    }

    // check for the white king castling
    if(this.isWhite()
      && start.getX() == 0 && start.getY() == 4 && end.getY() == 0) {
        // confirm if white king moved to the correct ending box
        if(Math.abs(move.getEnd().getY() - move.getStart().getY()) == 2) {
          // check if there the Rock is in the correct position
          // check if there is no piece between Rock and the King
          // check if the king or rock has not moved before
          // check if this move will not result in king being attacked
          //...
          this.setCastlingDone(true);
          return true;
        }
    } else {
      // check for the black king castling
      this.setCastlingDone(true);
      return true;
    }

    return false;
  }

  public boolean isCastlingMove(Box start, Box end) {
    // check if the starting and ending position are correct
  }
}
```

Knight: To encapsulate knight as a chess piece:

```
public class Knight extends Piece {
  public Knight(boolean white) {
    super(white);
  }

  @Override
  public boolean canMove(Board board, Box start, Box end) {
```

```
    // we can't move the piece to a box that has a piece of the same color
    if(end.getPiece().isWhite() == this.isWhite()) {
      return false;
    }

    int x = Math.abs(start.getX() - end.getX());
    int y = Math.abs(start.getY() - end.getY());
    return x * y == 2;
  }
}
```

Board: To encapsulate a chess board:

```
public class Board {
  Box[][] boxes;

  public Board() {
    this.resetBoard();
  }

  public Box getBox(int x, int y) {

    if(x < 0 || x > 7 || y < 0 || y  > 7) {
      throw new Exception("Index out of bound");
    }

    return boxes[x][y];
  }

  public void resetBoard() {
    // initialize white pieces
    boxes[0][0] = new Box(0, 0, new Rook(true));
    boxes[0][1] = new Box(0, 1, new Knight(true));
    boxes[0][2] = new Box(0, 2, new Bishop(true));
    //...
    boxes[1][0] = new Box(1, 0, new Pawn(true));
    boxes[1][1] = new Box(1, 1, new Pawn(true));
    //...

    // initialize black pieces
    boxes[7][0] = new Box(7, 0, new Rook(false));
    boxes[7][1] = new Box(7, 1, new Knight(false));
    boxes[7][2] = new Box(7, 2, new Bishop(false));
    //...
    boxes[6][0] = new Box(6, 0, new Pawn(false));
    boxes[6][1] = new Box(6, 1, new Pawn(false));
    //...

    // initialize remaining boxes without any piece
    for(int i=2; i < 6; i++) {
      for(int j=0; j < 8; j++) {
```

```
        boxex[i][j] = new Box(i, j, null);
      }
    }
  }
}
```

Player: To encapsulate a chess player:

```
public class Player {
  private Person person;
  private boolean whiteSide = false;

  public Player(Person person, boolean whiteSide){
    this.person = person;
    this.whiteSide = whiteSide;
  }

  public boolean isWhiteSide() {
    return this.whiteSide == true;
  }
}
```

Move: To encapsulate a chess move:

```
public class Move {
  private Player player;
  private Box start;
  private Box end;
  private Piece pieceMoved;
  private Piece pieceKilled;
  private boolean castlingMove = false;

  public Move(Player player, Box start, Box end){
    this.player = player;
    this.start = start;
    this.end = end;
    this.pieceMoved = start.getPiece();
  }

  public boolean isCastlingMove() {
    return this.castlingMove == true;
  }

  public void setCastlingMove(boolean castlingMove) {
    this.castlingMove = castlingMove;
  }
}
```

Game: To encapsulate a chess game:

```
public class Game {
  private Player[] players;
  private Board board;
  private Player currentTurn;
```

```
private GameStatus status;
private List<Move> movesPlayed;

private void initialize(Player p1, Player p2) {
  players[0] = p1;
  players[1] = p2;

  board.resetBoard();

  if(p1.isWhiteSide()) {
    this.currentTurn = p1;
  } else {
    this.currentTurn = p2;
  }

  movesPlayed.clear();
}

public boolean isEnd() {
  return this.getStatus() != GameStatus.ACTIVE;
}

public boolean getStatus() {
  return this.status;
}

public void setStatus(GameStatus status) {
  this.status = status;
}

public boolean playerMove(Player player, int startX, int startY, int endX, int endY) {
    Box startBox = board.getBox(startX, startY);
    Box endBox = board.getBox(startY, endY);
    Move move = new Move(player, startBox, endBox);
    return this.makeMove(move, player);
}

private boolean makeMove(Move move, Player player) {
  Piece sourcePiece = move.getStart().getPiece();
  if (sourcePiece == null) {
    return false;
  }

  // valid player
  if (player != currentTurn) {
    return false;
  }

  if (sourcePiece.isWhite() != player.isWhiteSide()) {
    return false;
```

```
    }

    // valid move?
    if (!sourcePiece.canMove(board, move.getStart(), move.getEnd())){
      return false;
    }

    // kill?
    Piece destPiece = move.getStart().getPiece();
    if (destPiece != null) {
      destPiece.setKilled(true);
      move.setPieceKilled(destPiece);
    }

    // castling?
    if (sourcePiece != null && sourcePiece instanceof King
        && sourcePiece.isCastlingMove()) {
      move.setCastlingMove(true);
    }

    // store the move
    movesPlayed.add(move);

    // move piece from the stat box to end box
    move.getEnd().setPiece(move.getStart().getPiece());
    move.getStart.setPiece(null);

    if (destPiece != null && destPiece instanceof King) {
      if(player.isWhiteSide()) {
        this.setStatus(GameStatus.WHITE_WIN);
      } else {
        this.setStatus(GameStatus.BLACK_WIN);
      }
    }

    // set the current turn to the other player
    if(this.currentTurn == players[0]) {
      this.currentTurn = players[1];
    } else {
      this.currentTurn = players[0];
    }

    return true;
  }
}
```

Design an Online Stock Brokerage System

Let's design an Online Stock Brokerage System.

The online stock brokerage system facilitates the users, i.e. individual investors, to trade (buy and sell) the stocks online. It allows clients to keep track of, and execute their transactions, and shows performance charts of different stocks in their portfolios. It also provides security for their transactions and alerts them to pre-defined levels of changes in stocks, without the use of any middlemen.

Online stock brokerage system automates the 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 following types: 1) market, 2) limit, 3) stop loss and, 4) stop limit order.
4. Users can have multiple ‘lots’ of a stock, which means 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 statement.
6. Users should be able to deposit and withdraw money either through check, wire or electronic bank transfer.
7. The system should be able to send notifications whenever the trade orders are executed.

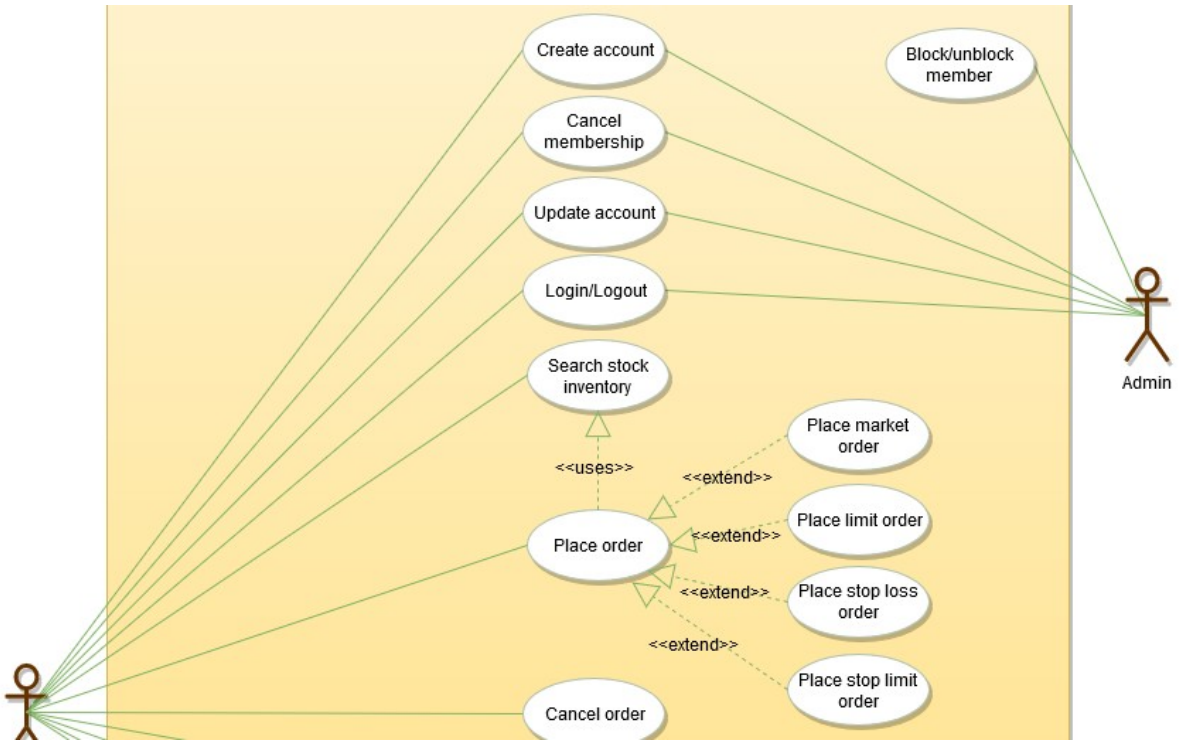
Usecase diagram

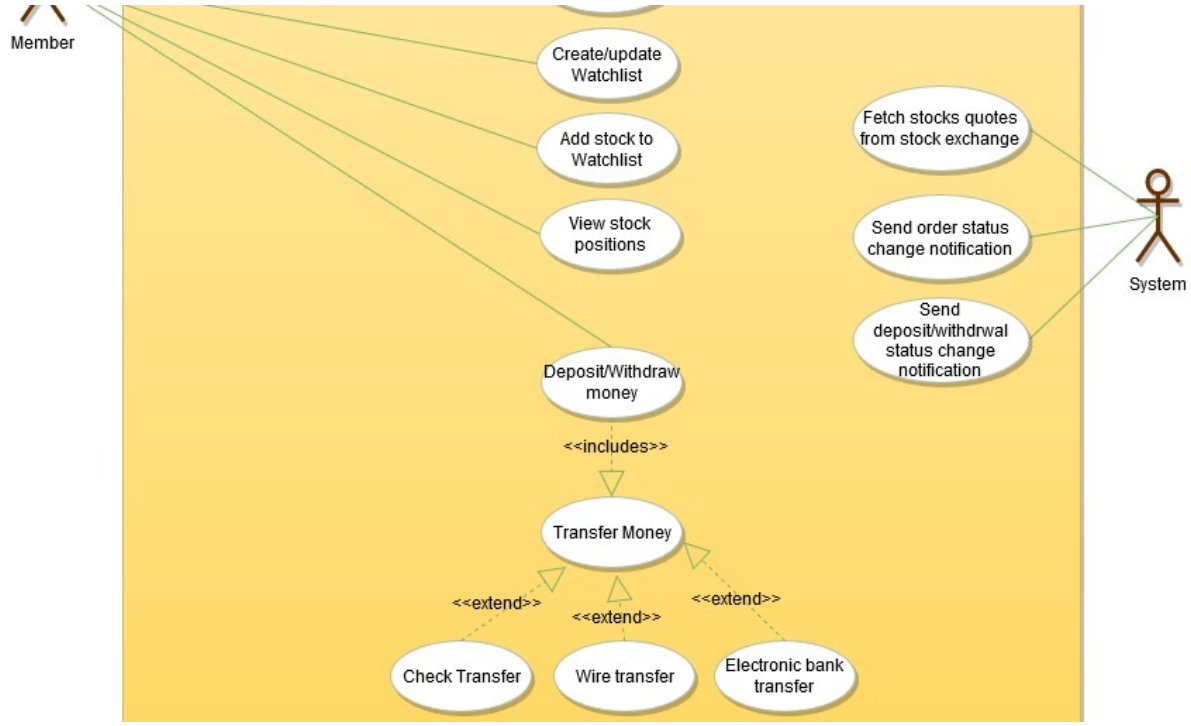
We have three main Actors in our system:

- **Admin:** Mainly responsible for administration 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 the 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 or remove or modify a watchlist.
- **Search stock inventory:** To search stock 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 through check, wire or electronic bank transfer.





Use case diagram for Stock Brokerage System

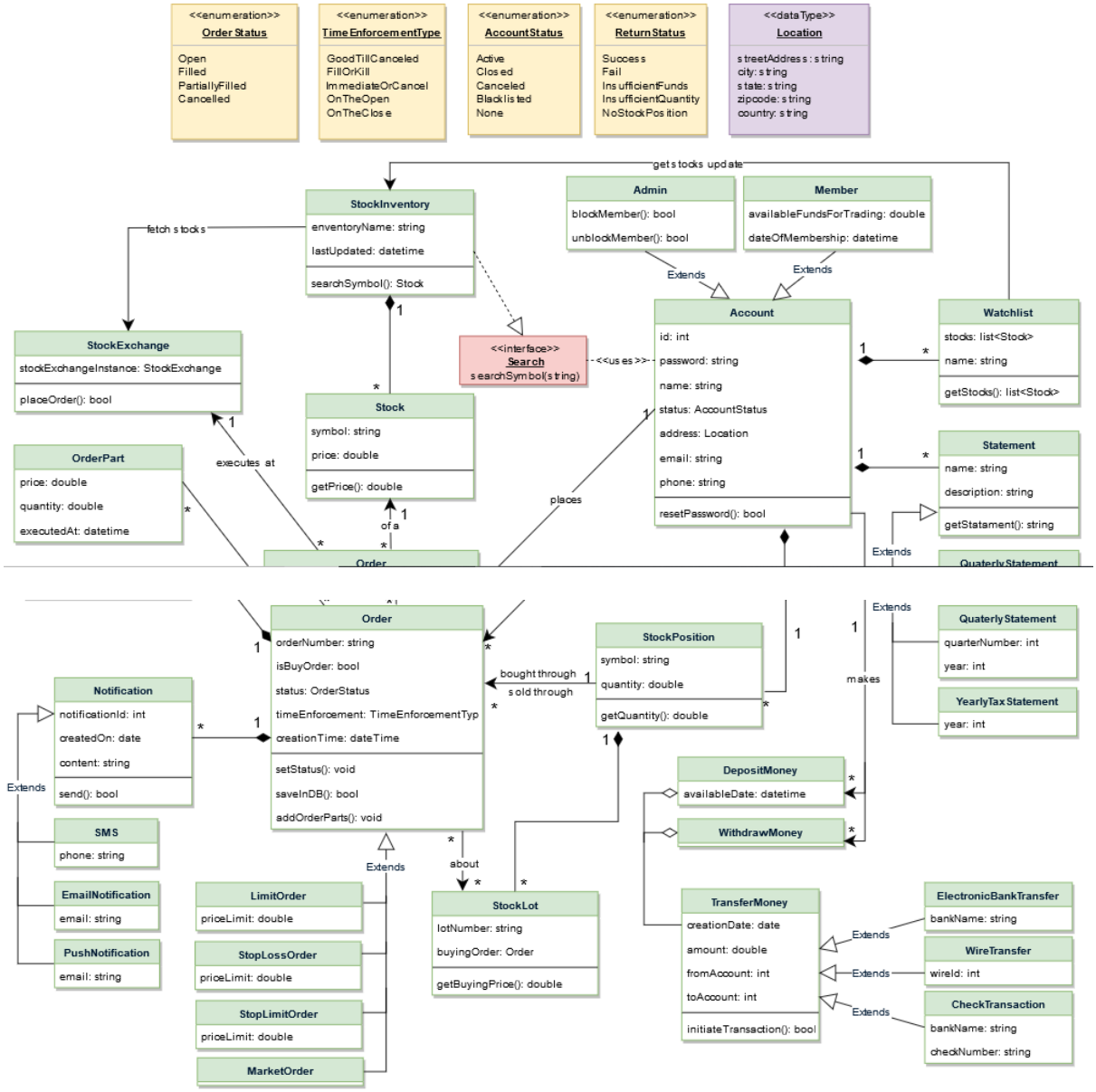
Class diagram

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

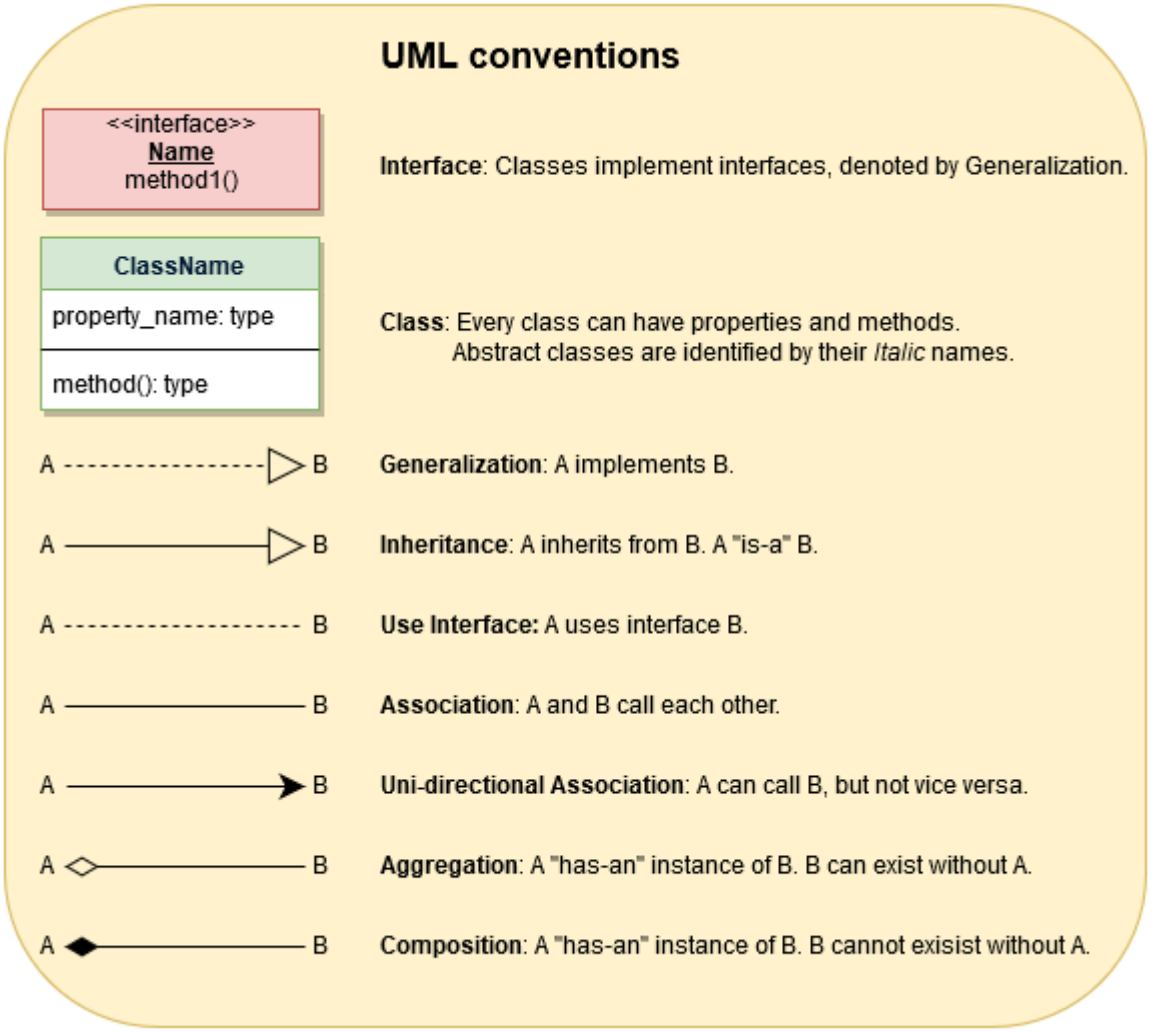
- **Account** Consists of member's name, address, e-mail, phone, total funds, and the funds that are available for trading, etc. We'll have two types of accounts in the system, one will be a general member, and the other will be an Admin. The Account class will also contain all the stocks the member 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 interaction 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 set of stocks that the member wants to follow.
- **Order:** User 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, e.g., 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 what lots they want to sell first.
- **StockPosition:** This class will contain all the stocks that the user holds.
- **Statment:** All members will have reports for quarterly updates and yearly tax statement.
- **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.

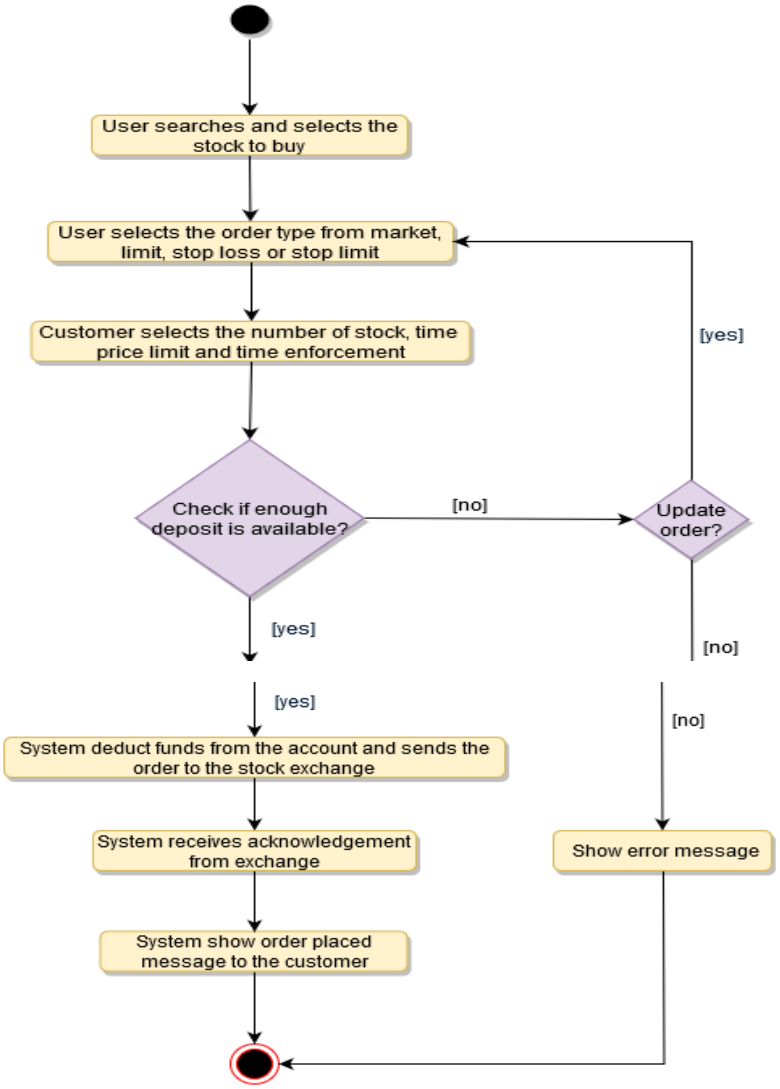


Class diagram

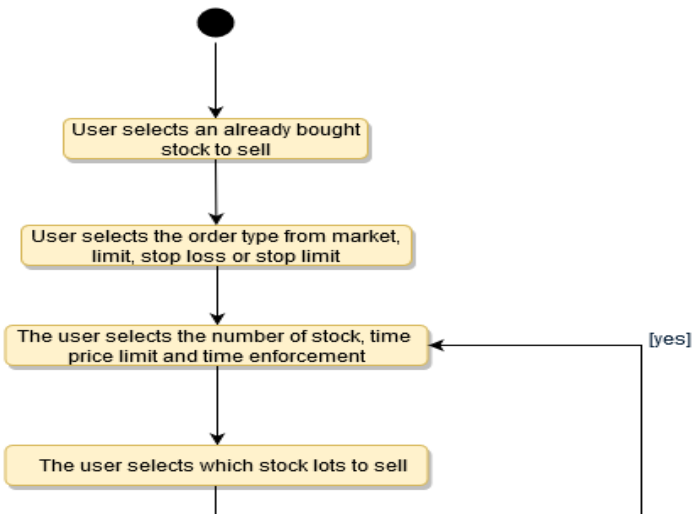


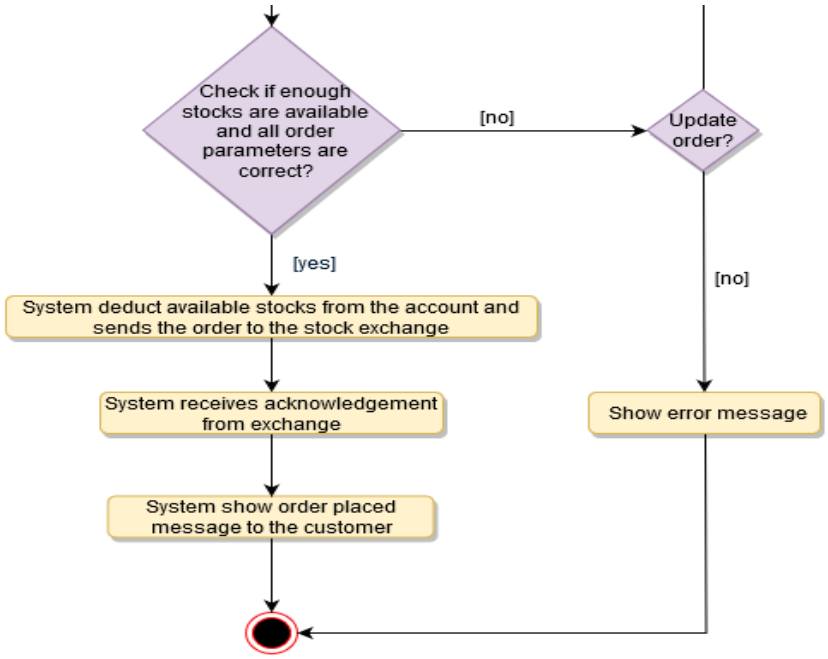
Activity diagrams

Place a buy order: Any system user can perform this activity. Here are the set of steps to place a buy order:



Place a sell order: Any system user can perform this activity. Here are the set of steps to place a buy order:





Code

Here is the code for the top use-cases.

Enums and Constants: Here are the required enums and constants:

```
public enum ReturnStatus {
  SUCCESS,
  FAIL,
  INSUFFICIENT_FUNDS,
  INSUFFICIENT_QUANTITY,
  NO_STOCK_POSITION
}

public enum OrderStatus {
  OPEN,
  FILLED,
  PARTIALLY_FILLED,
  CANCELLED
}

public enum TimeEnforcementType {
  GOOD_TILL_CANCELLED,
  FILL_OR_KILL,
  IMMEDIATE_OR_CANCEL,
  ON_THE_OPEN,
  ON_THE_CLOSE
}

public enum AccountStatus{
    ACTIVE,
```

```
    CLOSED,
    CANCELED,
    BLACKLISTED,
    None
}

public class Location {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}

public staic class Constants {
    public static const MONEY_TRANSFER_LIMIT = 100,000;
}
```

StockExchange: To encapsulate all the interaction with the stock exchange:

```
public class StockExchange {

  private static StockExchange stockExchangeInstance = null;

  // private constructor to restrict for singleton
  private StockExchange() { }

  // static method to get the singleton instance of StockExchange
  public static StockExchange getInstance()
  {
    if(stockExchangeInstance == null) {
      stockExchangeInstance = new StockExchange();
    }
    return stockExchangeInstance;
  }

  public static boolean placeOrder(Order order) {
    boolean returnStatus = getInstance().submitOrder(Order);
    return returnStatus;
  }
}
```

Order: To encapsulate all buy or sell order:

```
public abstract class Order {
  private String orderNumber;
  public boolean isBuyOrder;
  private OrderStatus status;
  private TimeEnforcementType timeEnforcement;
  private Date creationTime;

  private HashMap<int, OrderPart> parts;
```

```
  public void setStatus(OrderStatus status){
    this.status = status;
  }

  public bool saveInDB() {
    // save in the database
  }

  public void addOrderParts(OrderParts parts){
    for(OrderPart part: parts){
      this.parts.put(part.id, part);
    }
  }
}

public class LimitOrder extends Order {
  private double priceLimit;
}
```

Member: Members will be buying and selling stocks:

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

public abstract class Account {
  private String id;
  private String password;
  private String name;
  private AccountStatus status;
  private Location address;
  private String email;
  private String phone;

  public boolean resetPassword();
}

public class Member extends Account {
  private double availableFundsForTraiding;
  private Date dateOfMembership;

  private HashMap<string, StockPosition> stockPositions;
  private HashMap<int, Order> activeOrders;

  public ErrorCode placeSellLimitOrder(
    string stockId,
    float quantity,
    int limitPrice,
    TimeEnforcementType enfocementType )
  {
    // check if member has this stock position
    if(!stockPositions.containsKey(stockId)){
```

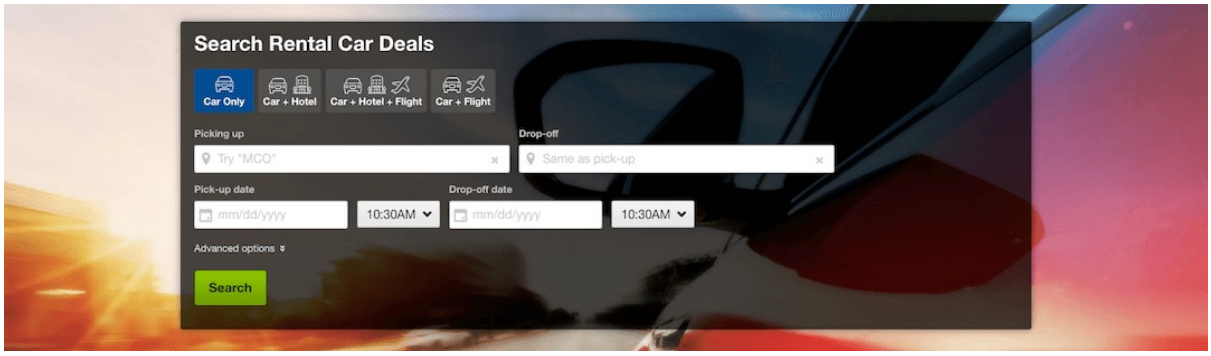
```
    return NO_STOCK_POSITION;
  }

  StockPosition stockPosition = stockPosition.get(stockId);
  // check if the member has enough quantity available to sell
  if(stockPoistion.getQuantity() < quantity){
    return INSUFFICIENT_QUANTITY;
  }

  LimitOrder order =
    new LimitOrder(stockId, quantity, limitPrice, enforcementType);
  order.isBuyOrder = false;
  order.saveInDB();
  boolean success = StockExchange::placeOrder(order);
  if(!success){
    order.setStatus(OrderStatus::FAILED);
    order.saveInDB();
  } else {
    activeOrders.add(orderId, order);
  }
  return success;
}

public ErrorCode placeBuyLimitOrder(
  string stockId,
  float quantity,
  int limitPrice,
  TimeEnforcementType enfocementType)
{
  // check if the member has enough funds to buy this stock
  if(availeFundsForTraiding < quantity * limitPrice ){
    return INSUFFICIENT_FUNDS;
  }

  LimitOrder order =
    new LimitOrder(stockId, quantity, limitPrice, enforcementType);
  order.isBuyOrder = true;
  order.saveInDB();
  boolean success = StockExchange::placeOrder(order);
  if(!success){
    order.setStatus(OrderStatus::FAILED);
    order.saveInDB();
  } else {
    activeOrders.add(orderId, order);
  }
  return success;
}

// this function will be invoked whenever there is an update from
// stock exchange against an order
```

```
  public void callbackStockExchange(
    int orderId,
    List<OrderPart> orderParts,
    OrderStatus status )
  {
    Order order = activeOrders.get(orderId);
    order.addOrderParts(orderParts);
    order.setStatus(status);
    order.updateInDB();

    if(status == OrderStatus::FILLED || status == OrderStatus::CANCELLEd){
      activeOrders.remove(orderId);
    }
  }
}
```

Design a Car Rental System

Let's design a car rental system where customers can rent different vehicles.

A Car Rental System is a software build to handle 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 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 car, truck, SUV, van, and motorbike.
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 who took a particular vehicle or what are the vehicles 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 vehicle inventory and reserve any available vehicle.

6. The system should be able to send notifications whenever the reservation is near pick-up date, as well as when the vehicle is near 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 keep track of all things happened to a vehicle.
10. Members can add rental insurance to their reservation.
11. Members can rent additional equipment like navigation, child seat, ski rack, etc.
12. Any member can add additional services to their reservation like roadside assistance, drive, wifi, etc.

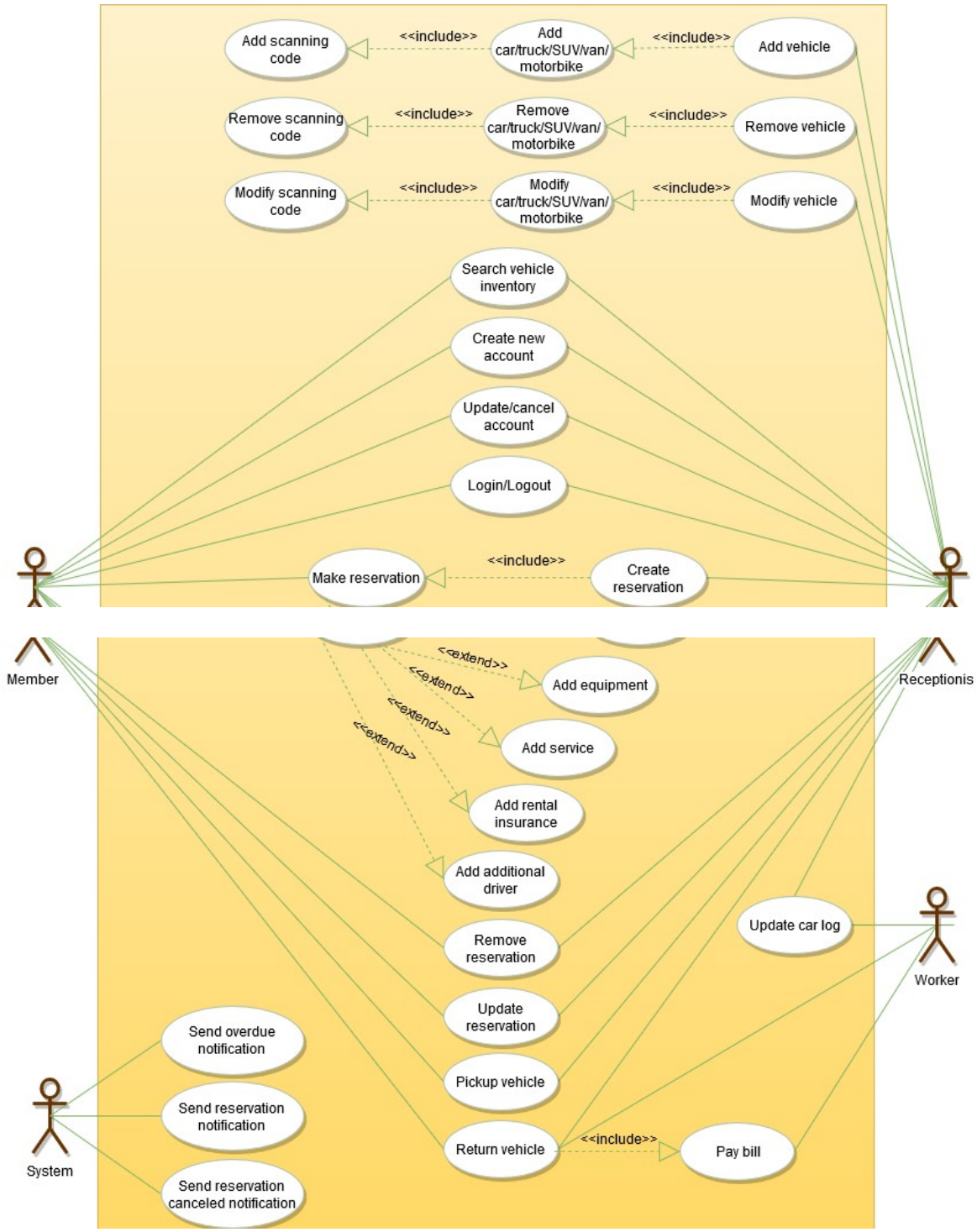
Use-case diagram

We have four main Actors in our system:

- **Receptionist:** Mainly responsible for adding and modifying vehicles and workers. Receptionists can 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 for vehicle overdue, reservation canceled, etc.
- **Worker:** Mainly responsible for taking care of a returning vehicle and update the vehicle log.

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

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



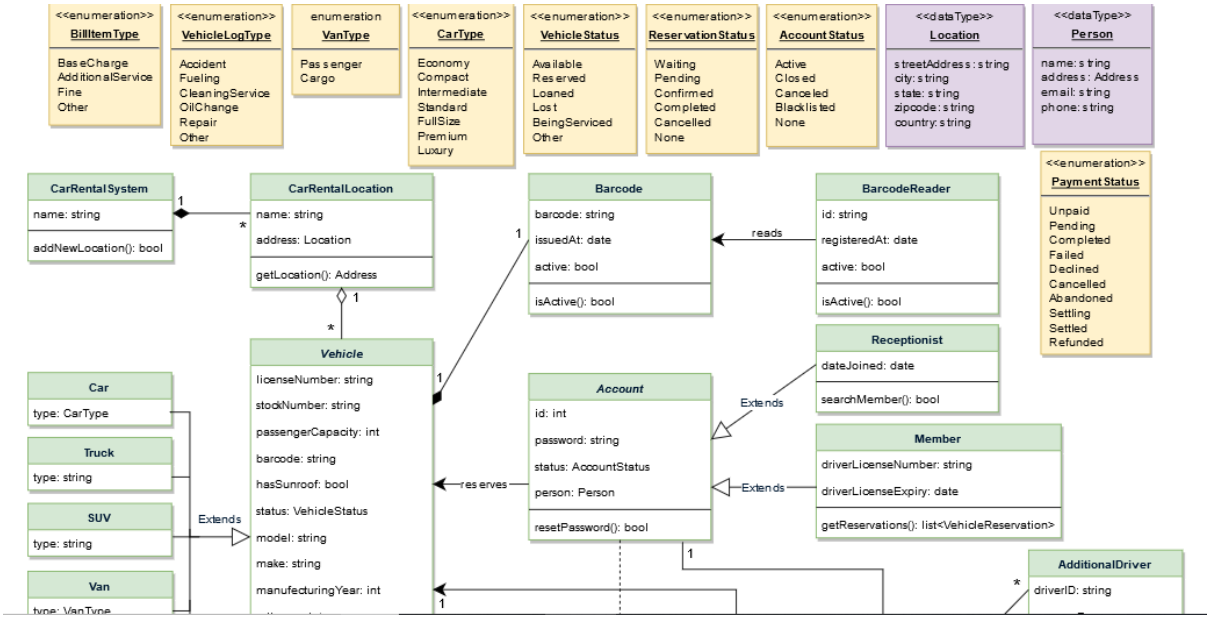
Use case diagram

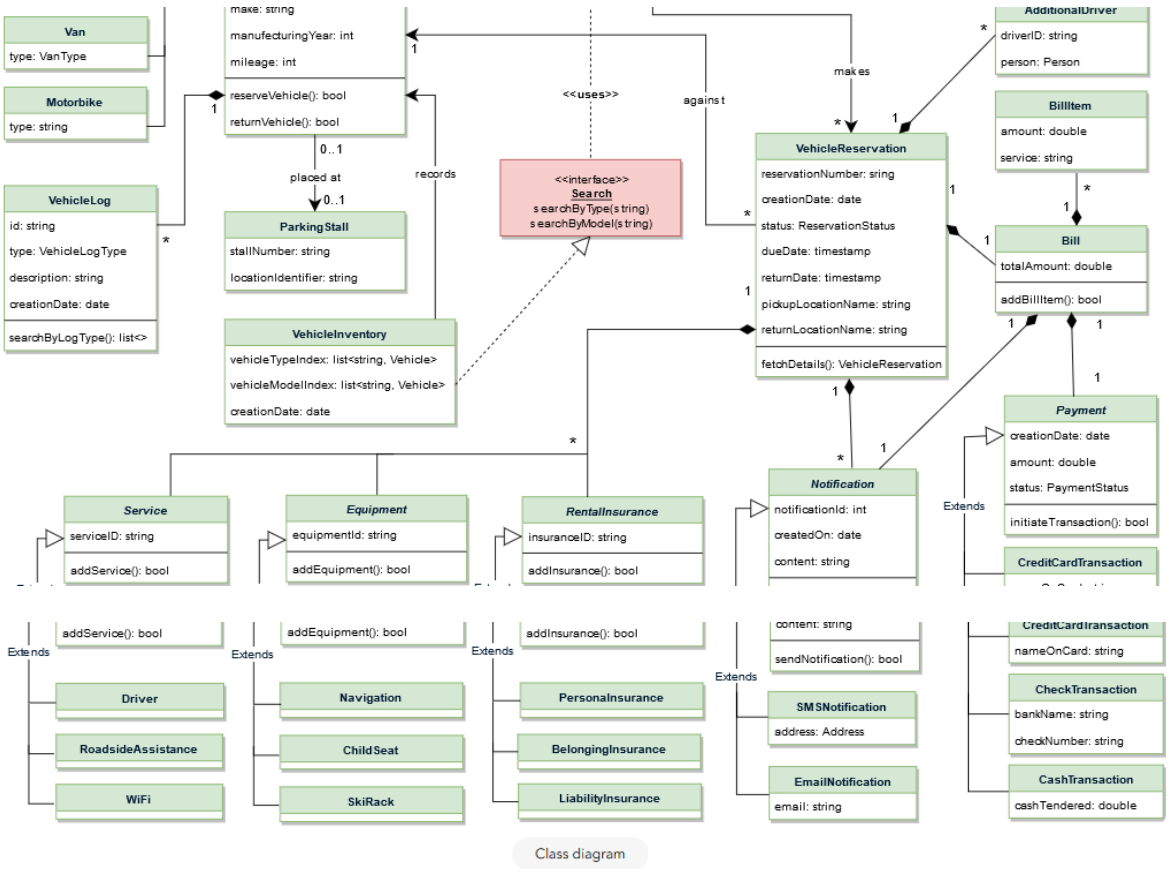
Class diagram

Here are the main classes of our Car Rental System:

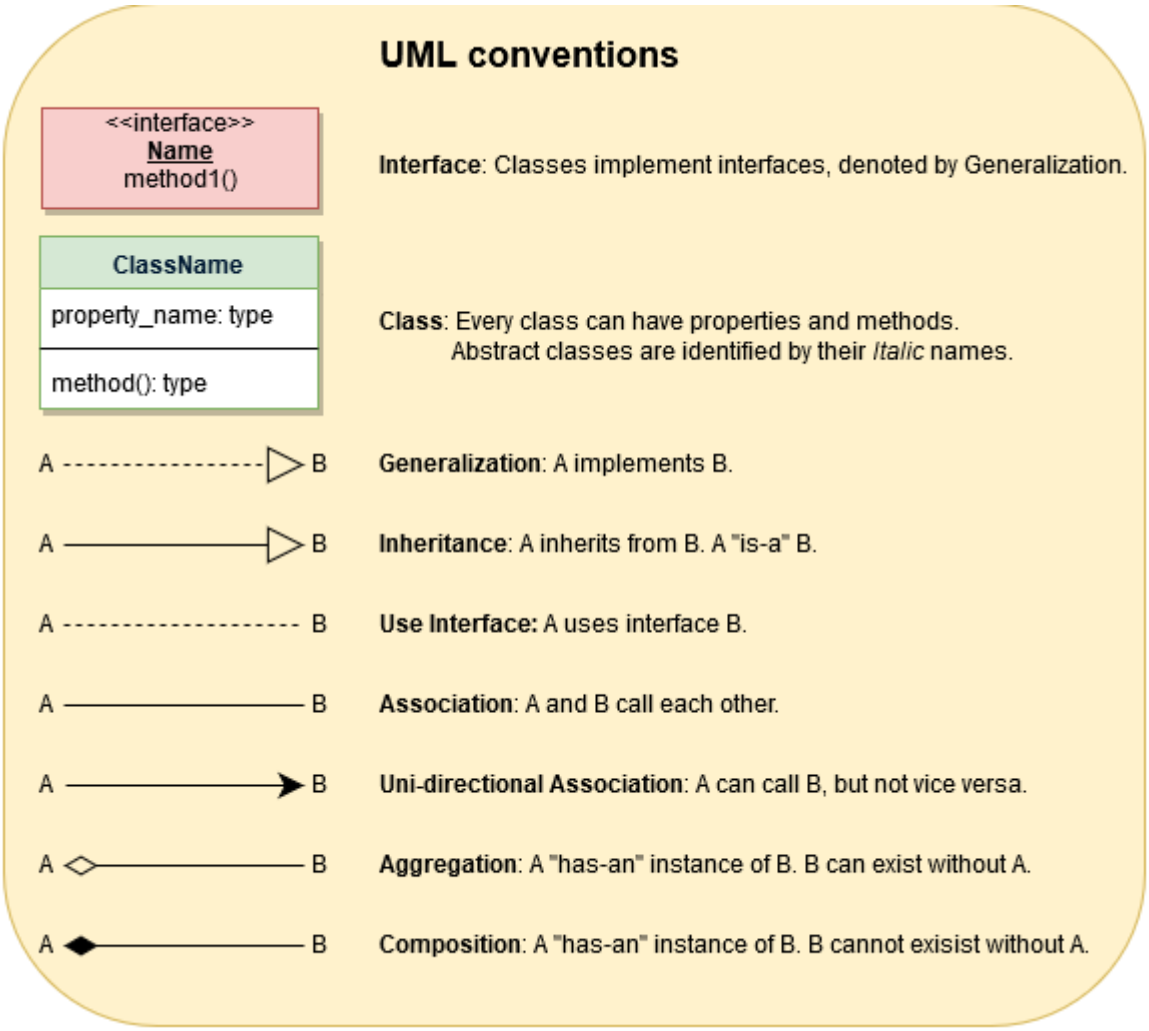
- **CarRentalSystem:** The main part of the organization for which this software has been designed.

- **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 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 returning vehicle.
- **VehicleReservation:** This class will be responsible for managing reservations against a vehicle.
- **Notification:** Will take care of sending notifications to members.
- **VehicleLog:** To keep a track of all things happened to a vehicle.
- **RentalInsurance:** Stores the details about different rental insurances members can add to their reservation.
- **Equipment:** Stores the details about different types of equipment members can add to their reservation.
- **Service:** Stores the details about different types of service members can add to their reservation like additional drivers, roadside assistance, etc.
- **Bill:** Contains different bill-items for every charge against the reservation.



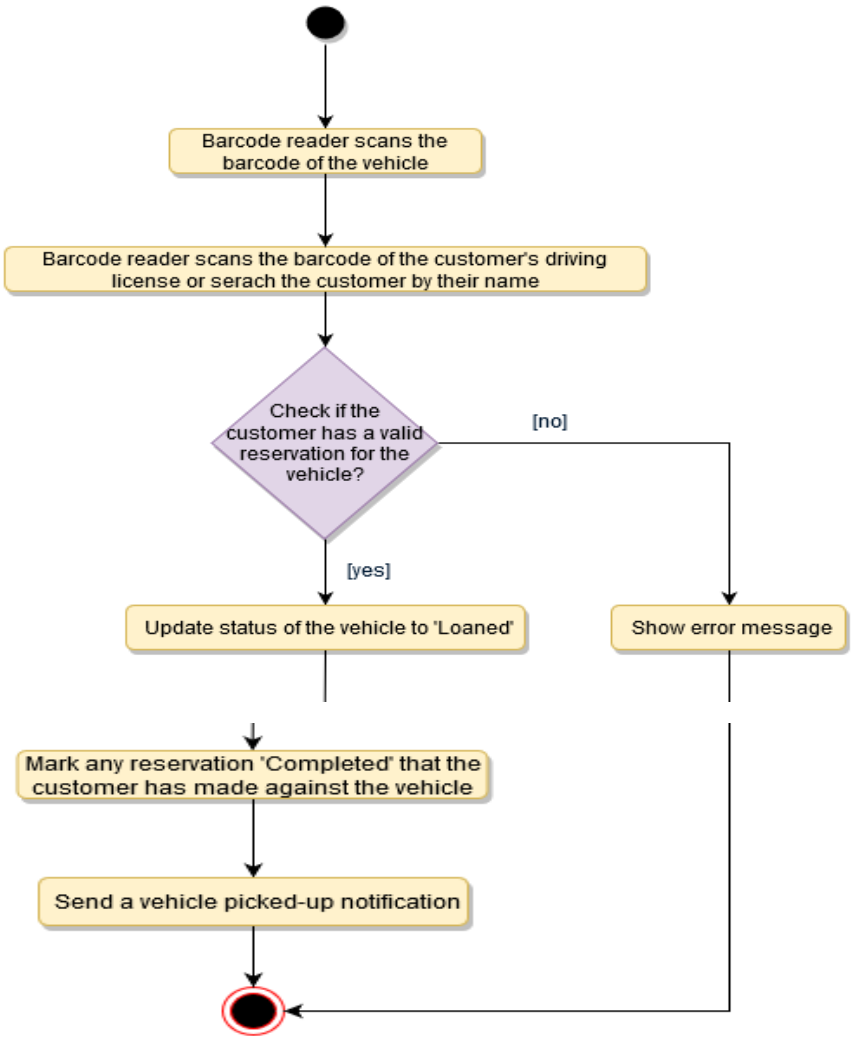


Class diagram

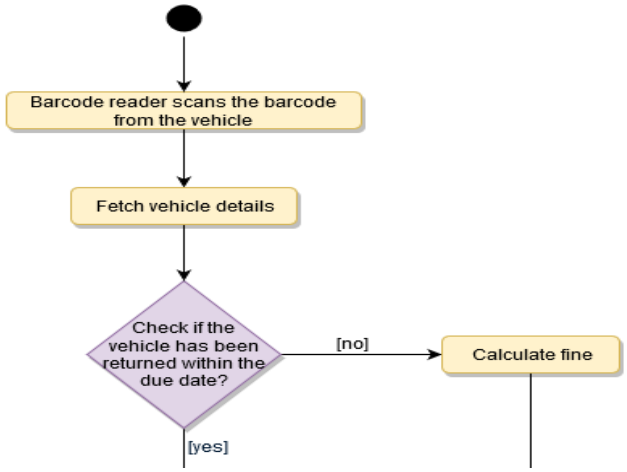


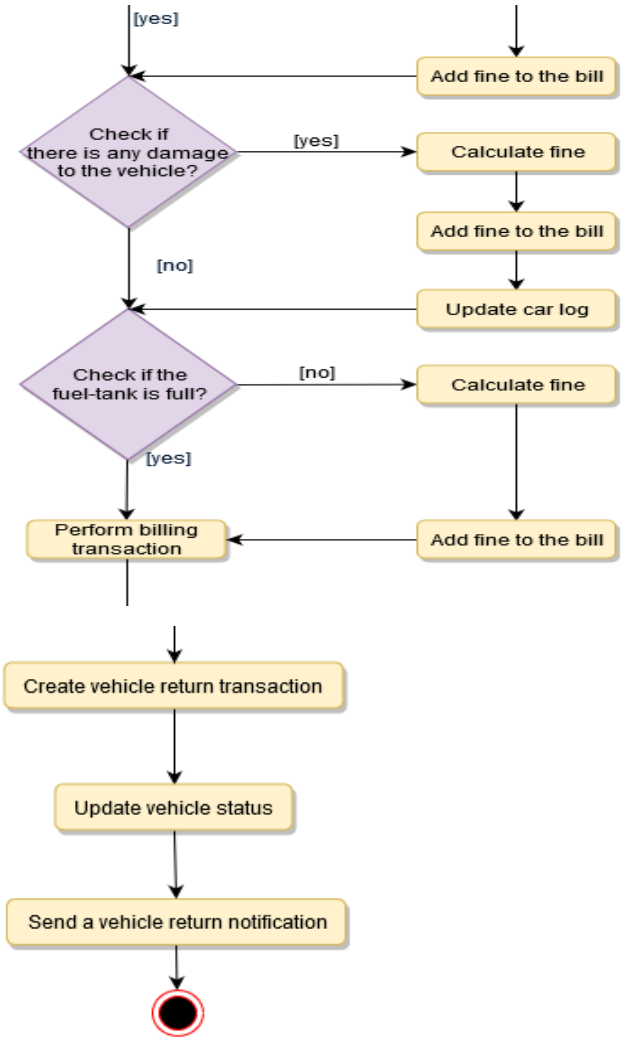
Activity diagrams

Pick up a vehicle: Any member can perform this activity. Here are the set of steps to pick up a vehicle:



Return a vehicle: Any worker can perform this activity. While returning a vehicle, the system has to collect the late fee from the member if the return date is after the due date. Here are the different steps for returning a vehicle:





Code

Here is the high-level definition for the classes described above.

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

```
public enum BillItemType{
  BASE_CHARGE,
  ADDTIONAL_SERVICE,
  FINE,
  OTHER
}

public enum VehicleLogType{
  ACCIDENT,
  FUELING,
  CLEANING_SERVICE,
  OIL_CHANGE,
  REPAIR,
  OTHER
```

```
}

public enum VanType{
  PASSENGER,
  CARGO
}

public enum CarType{
  ECONOMY,
  COMPACT,
  INTERMEDIATE,
  STANDARD,
  FULL_SIZE,
  PREMIUM,
  LUXURY
}

public enum VehicleStatus{
  AVAILABLE,
  RESERVED,
  LOANED,
  LOST,
  BEING_SERVICED,
  OTHER
}

public enum ReservationStatus{
  ACTIVE,
  PENDING,
  CONFIRMED,
  COMPLETED,
  CANCELLED,
  NONE
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  BLOCKED
}

public enum PaymentStatus{
  UNPAID,
  PENDING,
  COMPLETED,
  FILLED,
  DECLINED,
  CANCELLED,
```

```
  ABONDENED,
  SETTLING,
  SETTLED,
  REFUNDED
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}

public class Person {
  private String name;
  private Address address;
  private String email;
  private String phone;
}
```

Account, Member, Receptionist, and Additional Driver: These classes represent different people that interact with our system

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

public class Account {
  private String id;
  private String password;
  private AccountStatus status;
  private Person person;

  public boolean resetPassword();
}

public class Member extends Account {
  private int totalVehiclesReserved;

  public List<VehicleReservation> getReservations();
}

public class Receptionist extends Account {
  private Date dateJoined;

  public List<Member> searchMember(String name);
}

public class AdditionalDriver {
  private String driverID;
  private Person person;
```

```
}
```

CarRentalSystem and CarRentalLocation: These classes represent the top level classes:

```
public class CarRentalLocation {
  private String name;
  private Address location;

  public Address getLocation();
}

public class CarRentalSystem {
  private String name;
  private List<CarRentalLocation> locations;

  public boolean addNewLocation(CarRentalLocation location);
}
```

Vehicle, VehicleLog, and VehicleReservation: To encapsulate a vehicle, log, and reservation. The VehicleReservation class will be responsible for processing reservation and return of a vehicle:

```
public abstract class Vehicle {
  private String licenseNumber;
  private String stockNumber;
  private int passengerCapacity;
  private String barcode;
  private boolean hasSunroof;
  private VehicleStatus status;
  private String model;
  private String make;
  private int manufecturingYear;
  private int mileage;

  private List<VehicleLog> log;

  public boolean reserveVehicle();
  public boolean returnVehicle();
}

public class Car extends Vehicle {
  private CarType type;
}

public class Van extends Vehicle {
  private VanType type;
}

public class Truck extends Vehicle {
  private String type;
}

// We can have similar defination for other vehicle types
```

```
public class VehicleLog {
  private String id;
  private VehicleLogType type;
  private String description;
  private Date creationDate;

  public bool update();
  public List<VehicleLogType> searchByLogType(VehicleLogType type);
}

public class VehicleReservation {
  private String reservationNumber;
  private Date creationDate;
  private ReservationStatus status;
  private Date dueDate;
  private Date returnDate;
  private String pickupLocationName;
  private String returnLocationName;

  private int customerID;
  private Vehicle vehicle;
  private Bill bill;
  private List<AdditionalDriver> additionalDrivers;
  private List<Notification> notifications;
  private List<RentalInsurance> insurances;
  private List<Equipment> equipments;
  private List<Service> services;

  public static VehicleReservation fectchReservationDetails(String reservationNumber);
  public List<Passenger> getAdditionalDrivers();
}
```

VehicleInventory and Search: VehicleInventory will implement an interface 'Search' to facilitate searching of vehicles:

```
public interface Search {
  public List<BookItem> searchByType(String title);
  public List<BookItem> searchByModel(String author);
}

public class VehicleInventory implements Search {
  private HashMap<String, List<Vehicle>> vehicleTypes;
  private HashMap<String, List<Vehicle>> vehicleModels;

  public List<Vehicle> searchByType(String query) {
    // return all vehicles of the given type.
    return vehicleTypes.get(query);
  }

  public List<Vehicle> searchByModel(String query) {
    // return all vehicles of the given model.
    return vehicleModels.get(query);
```

```
  }
}
```

Design Linkedin

Let's design Linkedin.

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 some 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 other members or companies by their name.
3. Members should be able to send or accept connection requests to 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 or connection invitation or a comment on their post.
10. Members will be able to create a page for a Company and can 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.

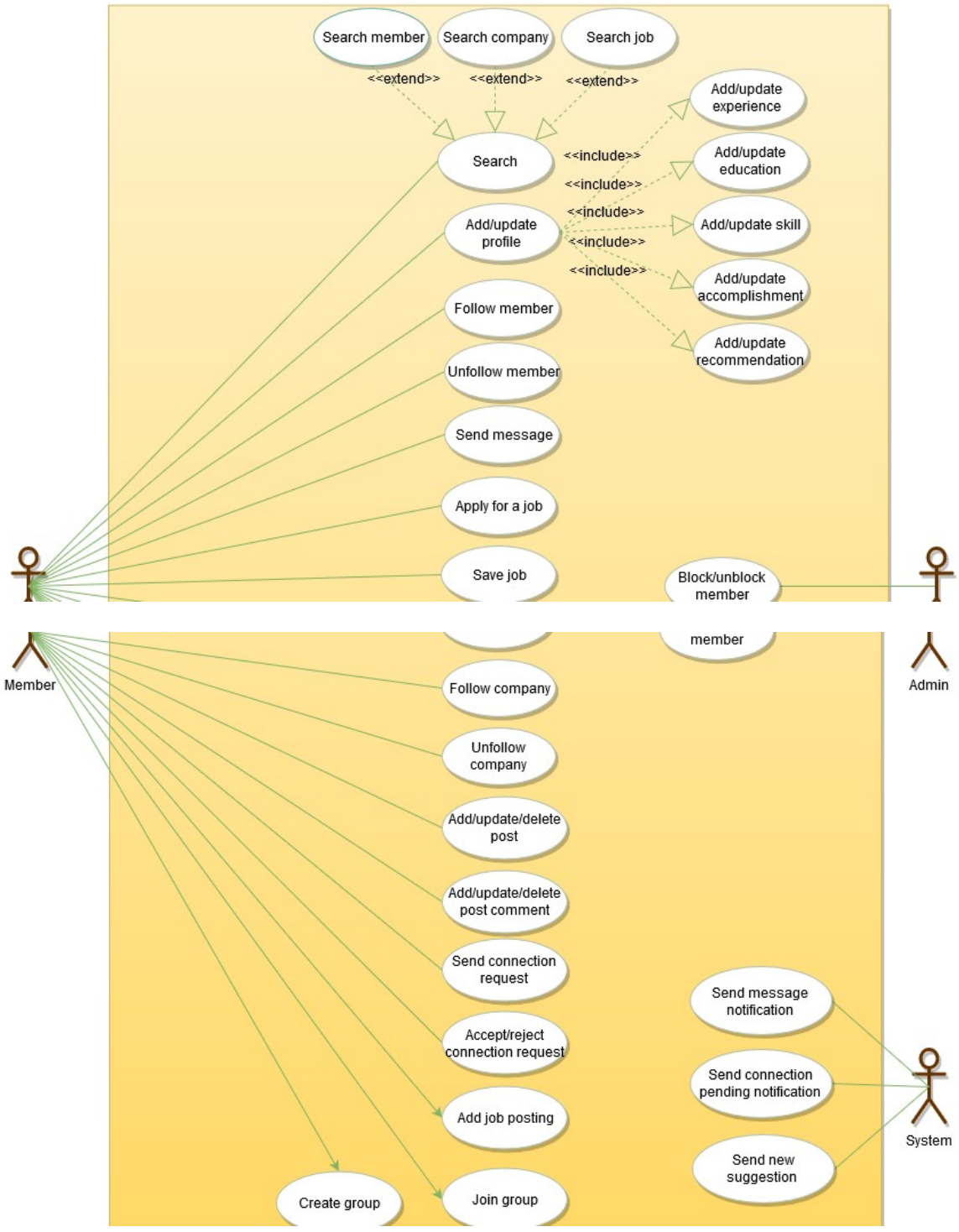
Usecase diagram

We have three main Actors in our system:

- **Member:** All members can search other members, companies or jobs, as well as send requests for connection, 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, 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 connection.
- **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 notification** The system will be able to send notifications for new messages, connection invites, etc.



Use case diagram

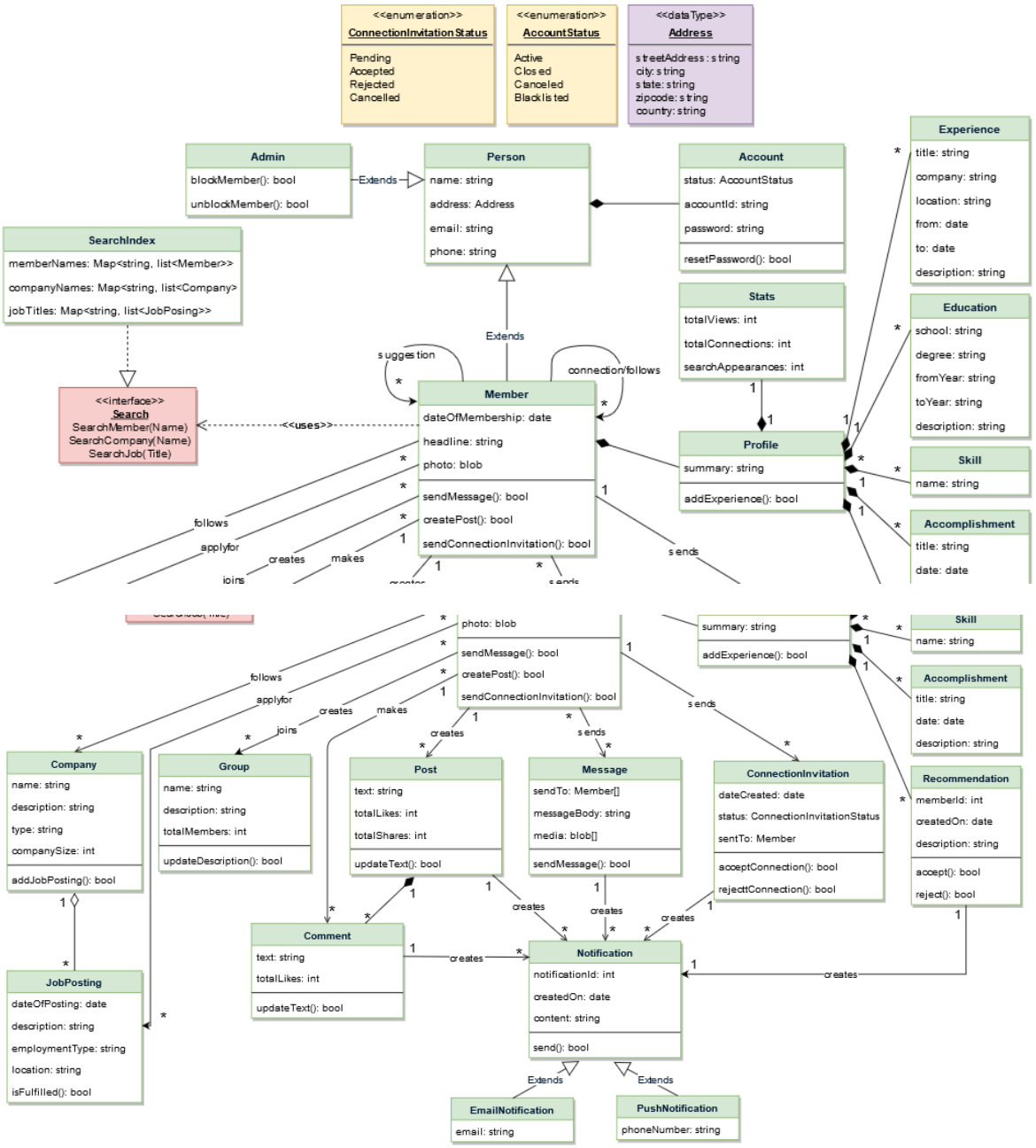
Class diagram

Here are the main classes of the Linkedin system:

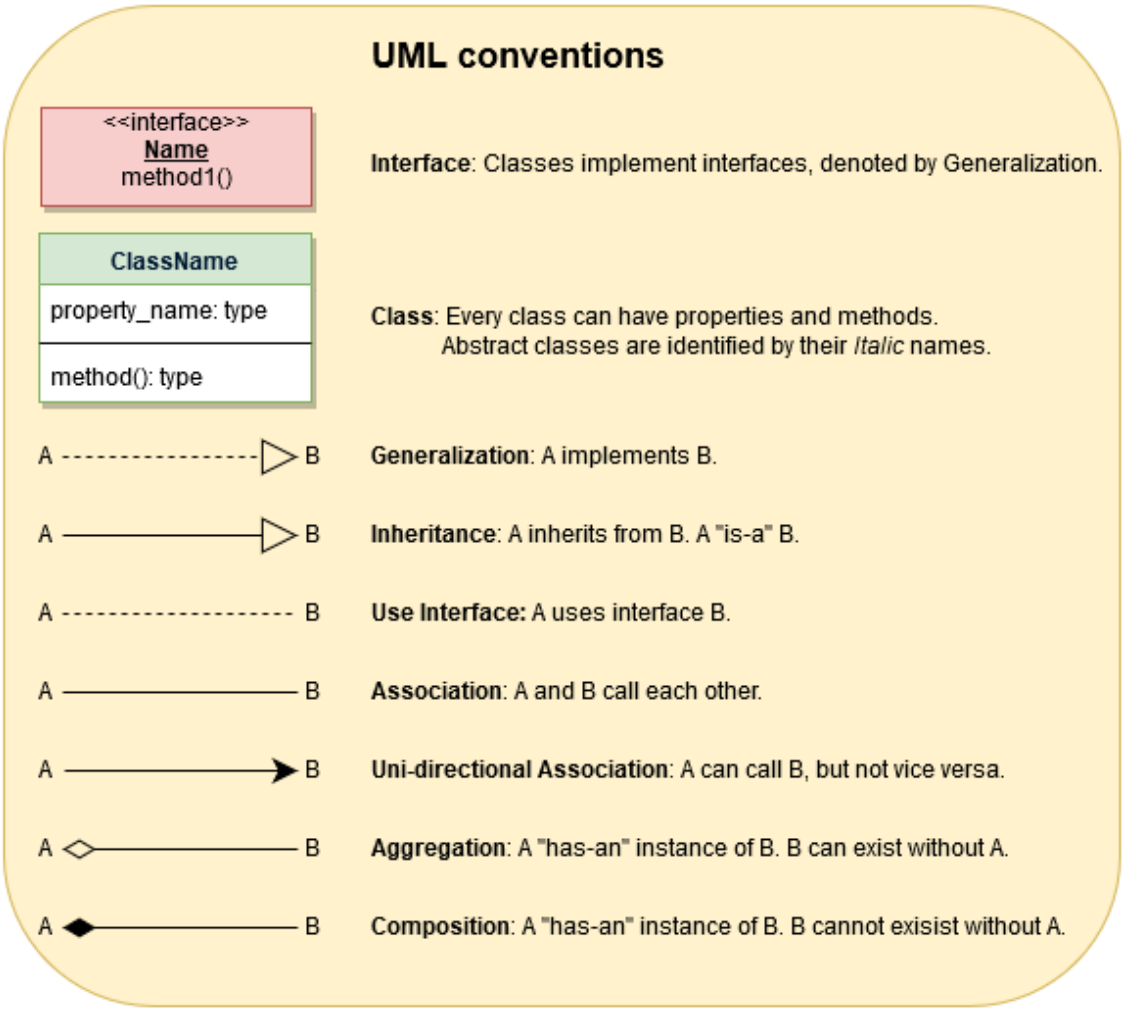
- **Member:** This will be the main component of our system. Each member will have a profile which includes their Experiences, Education, Skills, Accomplishments, and Recommendations. Members will be connected to other members and they can follow

companies and members. Members will also have suggestions to make connections with other members.

- **Search:** Our system will support searching 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.

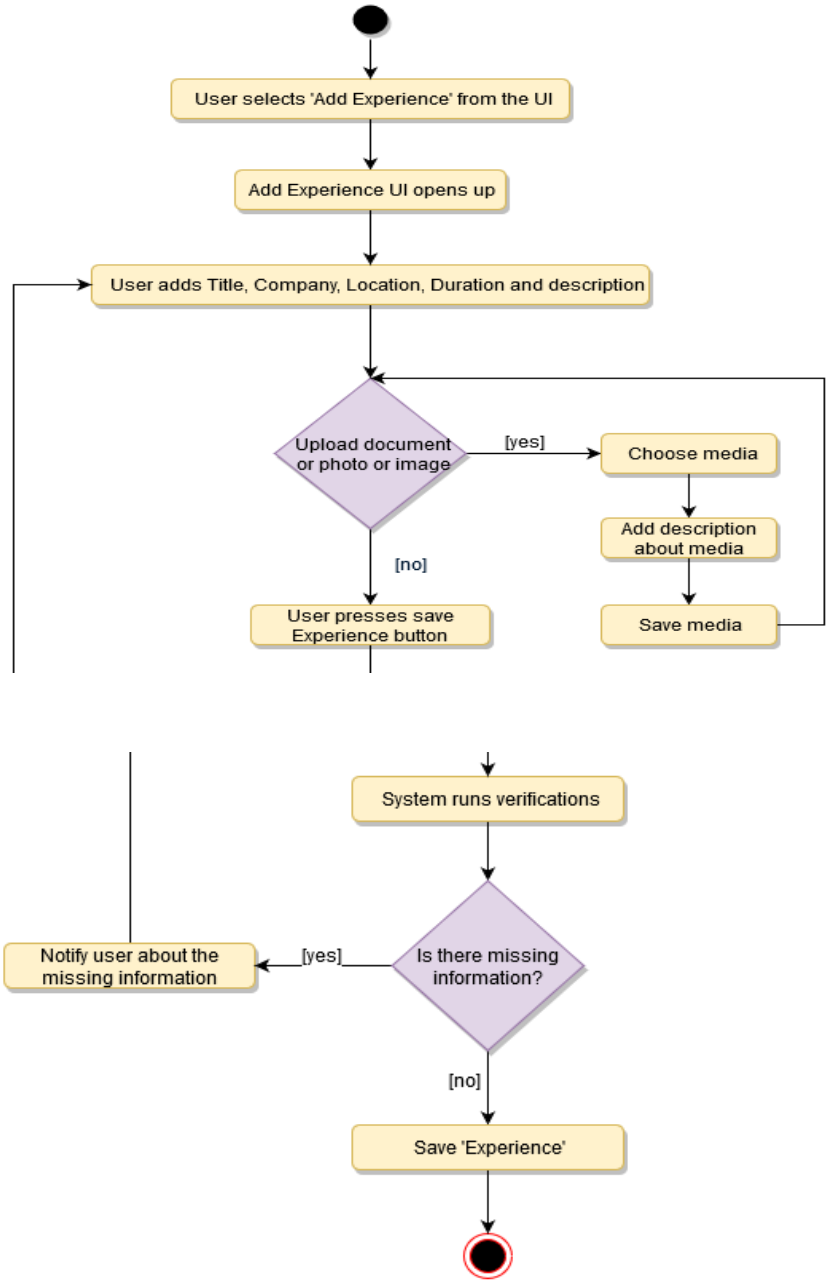


Class diagram

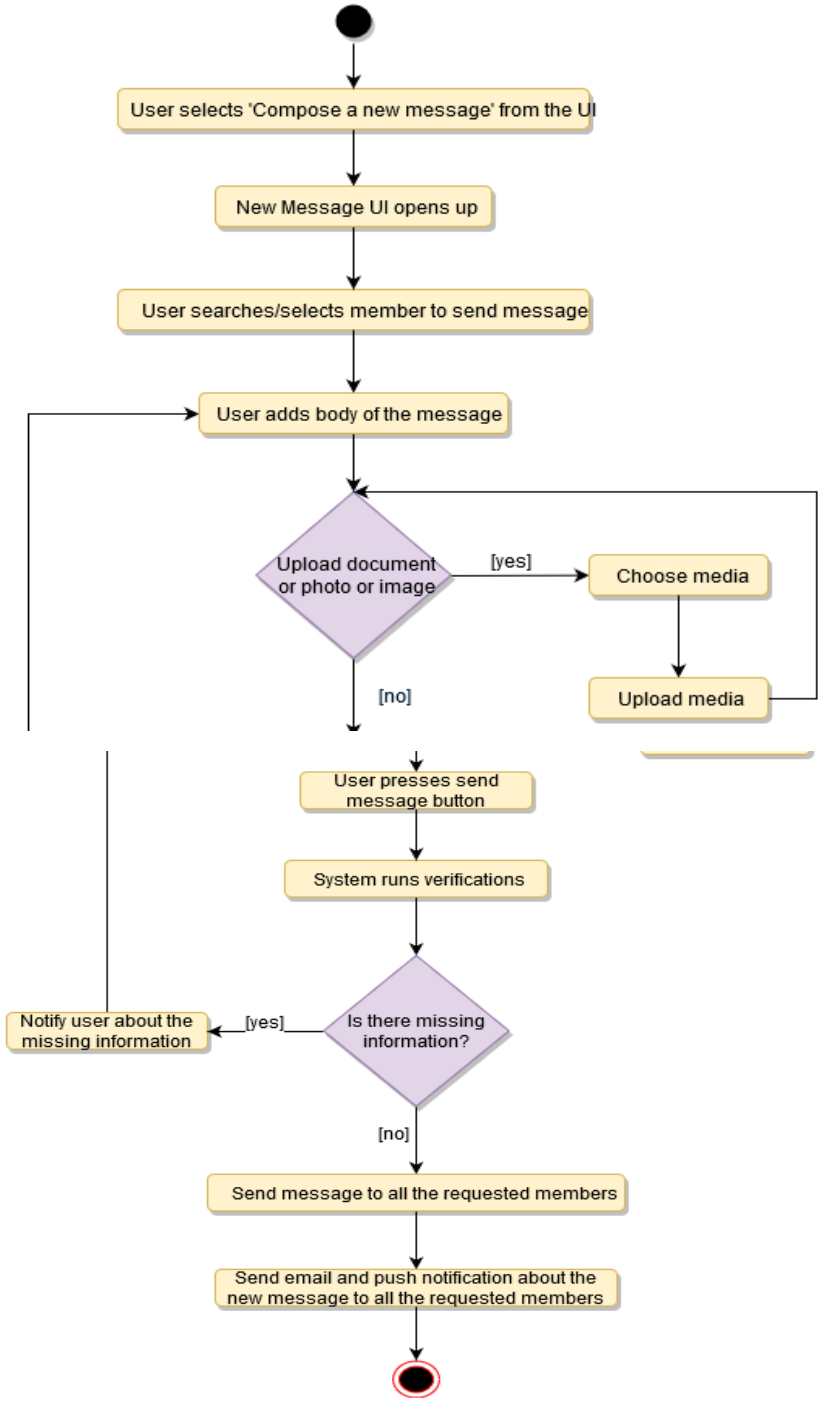


Activity diagrams

Add experience to profile: Any Linkedin member can perform this activity. Here are the set of steps to add experience to a member profile:



Send message: Any Member can perform this activity. After sending a message, the system needs to send notification to all the requested members. Here are the different steps for sending a message:



Code

Here is the high-level definition for the classes described above.

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

```
public enum ConnectionInvitationStatus{
  PENDING,
  ACCEPTED,
  CONFIRMED,
```

```
  REJECTED,
  CANCELED
}

public enum AccountStatus{
  ACTIVE,
  BLOCKED,
  BANNED,
  COMPROMIZED,
  ARCHIVED,
  UNKNOWN
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}
```

Account, Person, Member, and Admin: These classes represent different people that interact with our system:

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

public abstract class Account {
  private String id;
  private String password;
  private AccountStatus status;

  public boolean resetPassword();
}

public abstract class Person extends Account{
  private String name;
  private Address address;
  private String email;
  private String phone;

  private Account account;
}

public class Member extends Person {
  private Date dateOfMembership;
  private String headline;
  private byte[] photo;
  private List<Member> memberSuggestions;
  private List<Member> memberFollows;
```

```
  private List<Member> memberConnections;
  private List<Company> companyFollows;
  private List<Group> groupFollows;
  private Profile profile;

  public boolean sendMessage(Message message);
  public boolean createPost(Post post);
  public boolean sendConnectionInvitation(ConnectionInvitation invitation);
}

public class Admin extends Person {
  public boolean blockUser(Customer customer);
  public boolean unblockUser(Customer customer);
}
```

Profile, Experience, etc: A member's profile will have their job experiences, educations, skills, etc:

```
public class Profile {
  private String summary;
  private List<Experience> experiences;
  private List<Education> educations;
  private List<Skill> skills;
  private List<Accomplishment> accomplishments;
  private List<Recommendation> recommendations;
  private List<Stat> stats;

  public boolean addExperience(Experience experience);
  public boolean addEducation(Education education);
  public boolean addSkill(Skill skill);
  public boolean addAccomplishment(Accomplishment accomplishment);
  public boolean addRecommendation(Recommendation recommendation);
}

public class Experience {
  private String title;
  private String company;
  private String location;
  private Date from;
  private Date to;
  private String description;
}
```

Company and JobPosting: Companies can have multiple job postings:

```
public class Company {
  private String name;
  private String description;
  private String type;
  private int companySize;

  private List<JobPosting> activeJobPostings;
```

```
}

public class JobPosting {
  private Date dateOfPosting;
  private String description;
  private String employmentType;
  private String location;
  private boolean isFulfilled;

  private List<JobPosting> activeJobPostings;
}
```

Group, Post, and Message: Members can create posts, send messages and join groups:

```
public class Group {
  private String name;
  private String description;
  private int totalMembers;
  private List<Member> members;

  public boolean addMember(Member member);
  public boolean updateDescription(String description);
}

public class Post {
  private String text;
  private int totalLikes;
  private int totalShares;
  private Member owner;
}

public class Message {
  private Memberp[] sentTo;
  private String messageBody;
  private byte[] media;
}
```

Search interface and SearchIndex: SearchIndex will implement Search to facilitate searching of members, companies and job postings:

```
public interface Search {
  public List<Mamber> searchMamber(String name);
  public List<Company> searchCompany(String name);
  public List<JobPosting> searchJob(String title);
}

public class SearchIndex implements Search {
   HashMap<String, List<Mamber>> memberNames;
   HashMap<String, List<Company>> companyNames;
   HashMap<String, List<JobPosting>> jobTitles;

   public boolean addMember(Member member) {
```

```
    if(memberNames.containsKey(member.getName())) {
      memberNames.get(member.getName()).add(member);
    } else {
      memberNames.put(member.getName(), member);
    }
  }

  public boolean addCompany(Company company);
  public boolean addJobPosting(JobPosting jobPosting);

 public List<Member> searchMamber(String name) {
  return memberNames.get(name);
 }

 public List<Company> searchByLanguage(String name) {
  return companyNames.get(name);
 }

 public List<JobPosting> searchByCity(String title) {
  return jobTitles.get(title);
 }
}
```

Design Cricinfo

Let's 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 the cricket playing teams and their matches.
2. The system should show live ball-by-ball commentary of cricket matches.
3. All international cricket's 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.

Usecase diagram

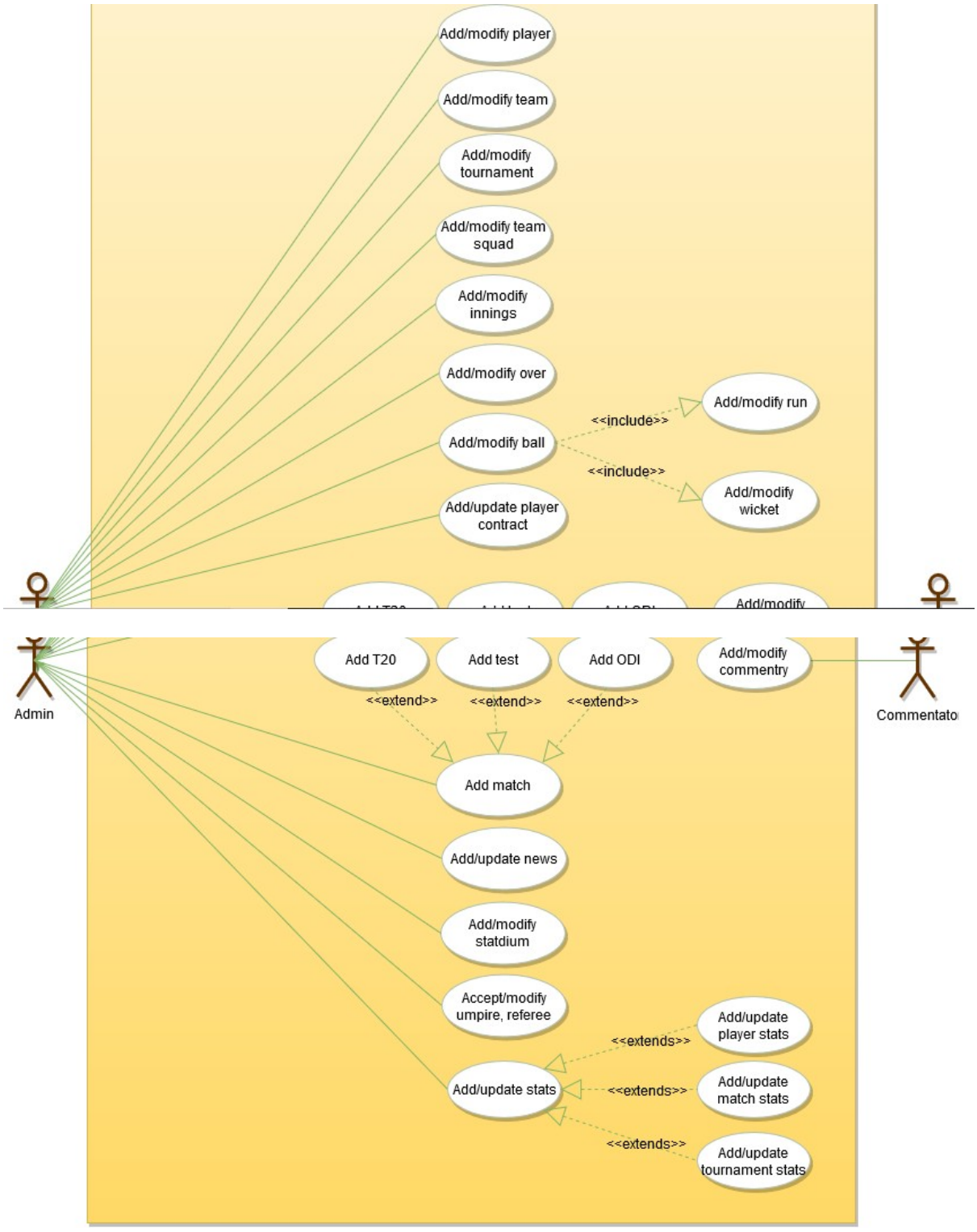
We have two main Actors in our system:

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

Here are the top use cases of our system:

- **Add/modify teams and players:** Admin will add players to teams and keeps an update-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 detail of a match.
- **Add stadium, umpire, and referee** The system will also keep track of stadiums as well as umpires and referees managing 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.

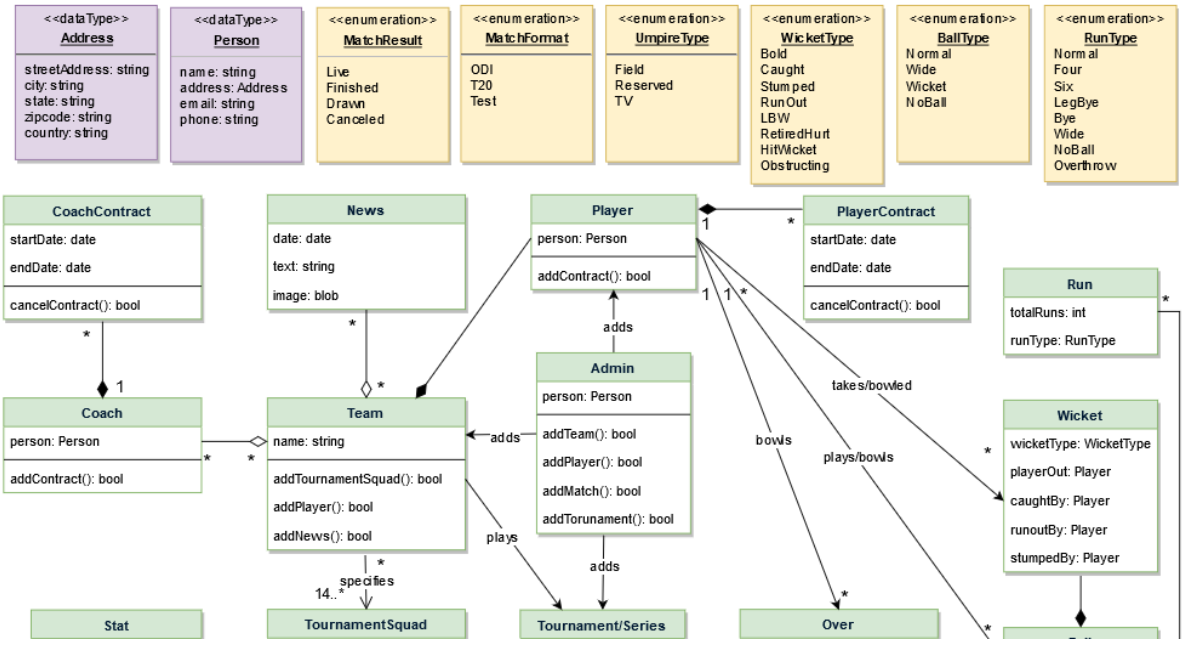


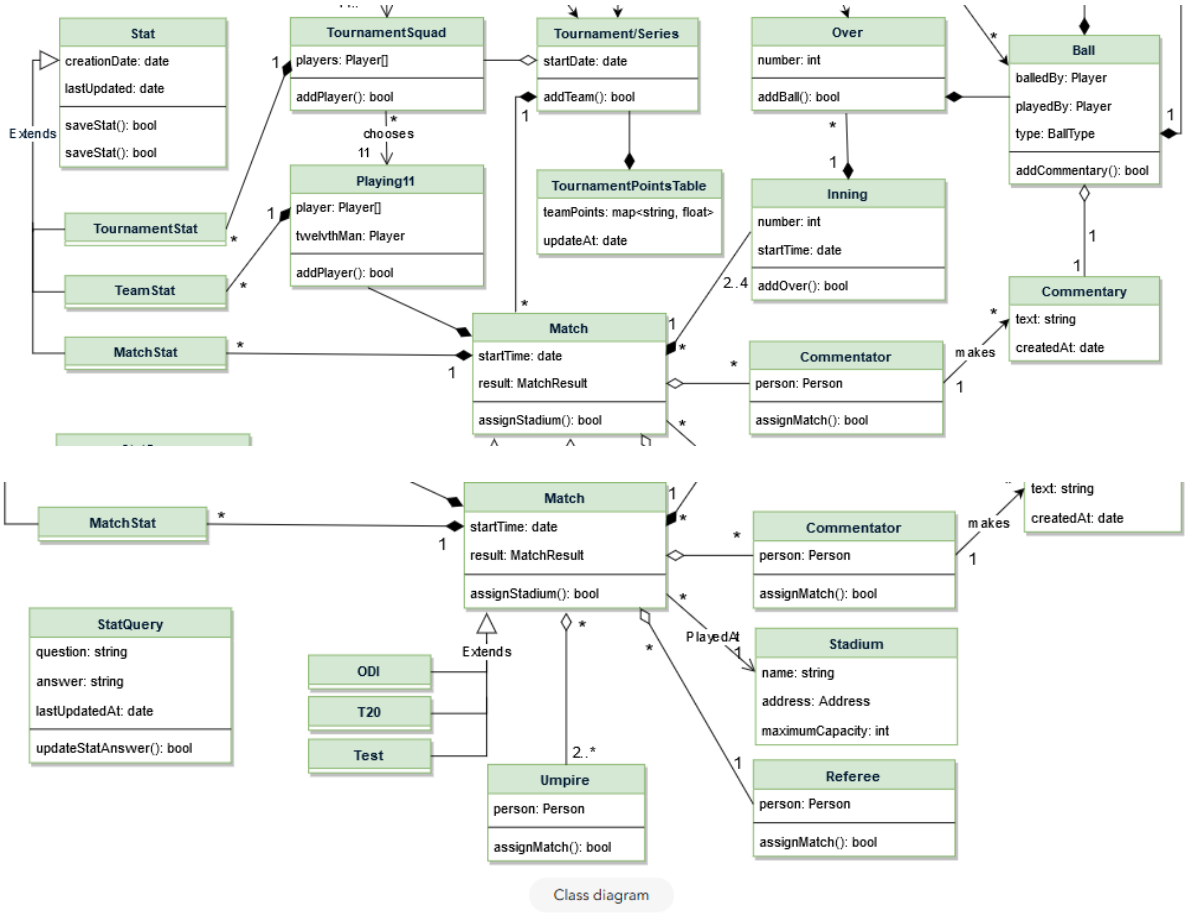
Use case diagram

Class diagram

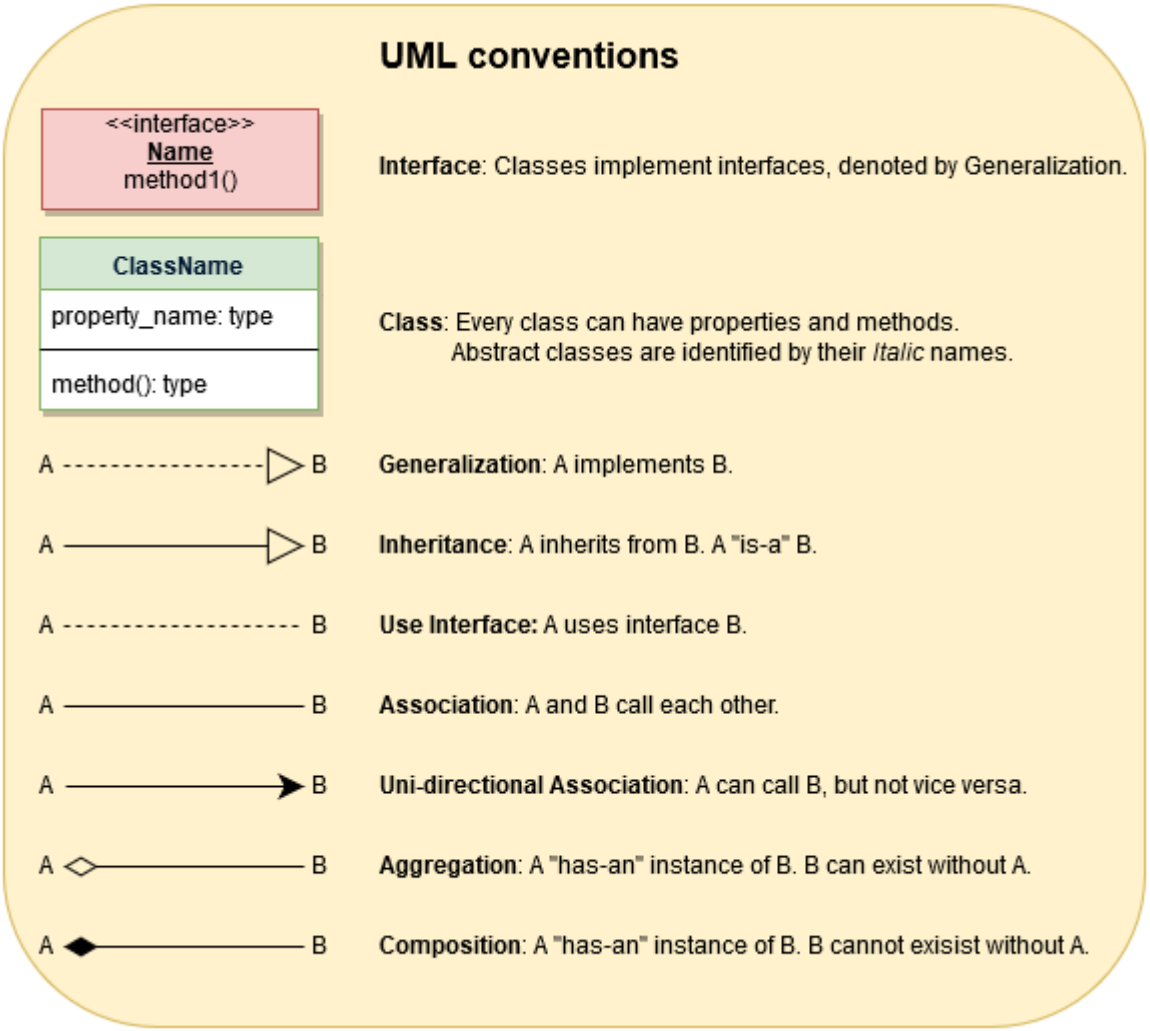
Here are the main classes of the Cricinfo system:

- **Player:** Keeps a record of a cricket player, their basic profile and contracts.
- **Team:** This class manages cricket teams.
- **Tournament:** Manages cricket tournaments and keep track of a points table for all the 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, like, how many runs scored, if it is a wicket-taking ball, etc.
- **Run:** Records how many and what type of runs were scored on a ball. Different run types are Wide, LegBy, Four, Six, etc.
- **Commentator and Commentary:** 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 stats for every player, match and tournament.
- **StatQuery:** This class will encapsulate general stat queries and their answer, like "who has scored maximum 100s in ODIs?" or "Which bowler has taken maximum wickets in test matches?"



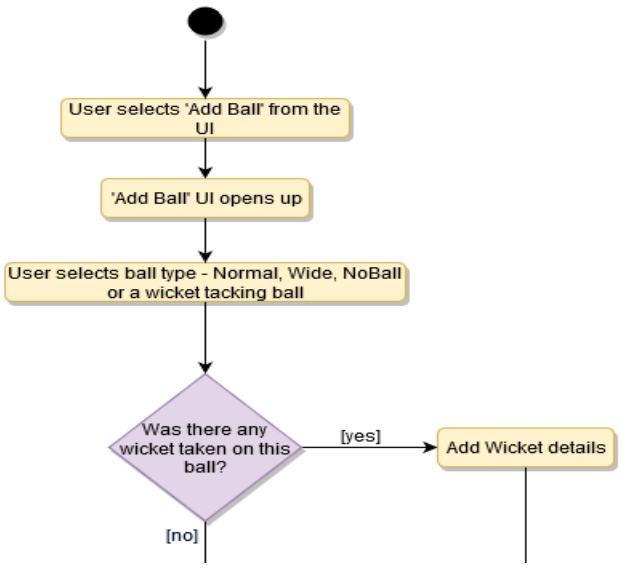


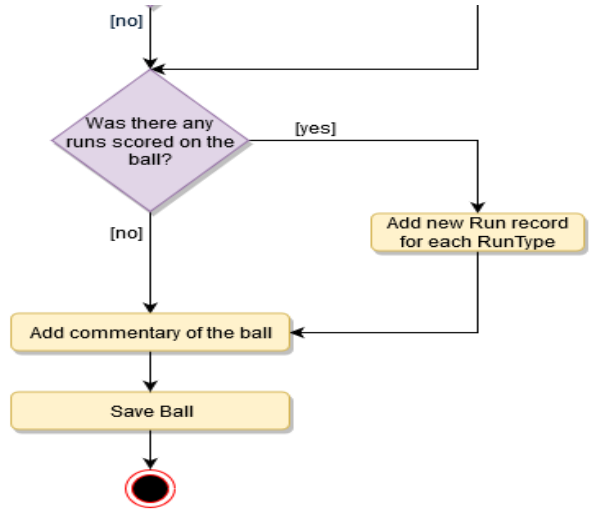
Class diagram



Activity diagrams

Record a Ball of an Over: Here are the set of steps to record a ball of an over in the system:





Code

Here is the high-level definition for the classes described above.

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

```
public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}

public class Person {
  private String name;
  private Address address;
  private String email;
  private String phone;
}

public enum MatchFormat {
  ODI,
  T20,
  TEST
}

public enum MatchResult {
  LIVE,
  FINISHED,
  DRAWN,
  CANCELED
}

public enum UmpireType {
```

```
  FIELD,
  RESERVED,
  TV
}

public enum WicketType {
  BOLD,
  CAUGHT,
  STUMPED,
  RUN_OUT,
  LBW,
  RETIRED_HURT,
  HIT_WICKET,
  OBSTRUCTING
}

public enum BallType {
  NORMAL,
  WIDE,
  WICKET,
  NO_BALL
}

public enum RunType {
  NORMAL,
  FOUR,
  SIX,
  LEG_BYE,
  BYE,
  NO_BALL,
  OVERTHROW
}
```

Admin, Player, Umpire, Referee, and Commentator These classes represent different people that interact with our system:

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

public class Player {
  private Person person;
  private ArryList<PlayerContract> contracts;

  public boolean addContract();
}

public class Admin {
  private Person person;
  public boolean addMatch(Match match);
  public boolean addTeam(Team team);
```

```
  public boolean addTournament(Tournament tournament);
}

public class Umpire {
  private Person person;
  public boolean assignMatch(Match match);
}

public class Referee {
  private Person person;
  public boolean assignMatch(Match match);
}

public class Commentator {
  private Person person;
  public boolean assignMatch(Match match);
}
```

Team, TournamentSquad, and Playing11: Team will announce a squad for a tournament, out of this squad, playing eleven will be chosen:

```
public class Team {
  private String name;
  private List<Player> players;
  private List<News> news;
  private Coach coach;

  public boolean addTournamentSquad(TournamentSquad tournamentSquad);
  public boolean addPlayer(Player player);
  public boolean addNews(News news);
}

public class TournamentSquad {
  private List<Player> players;
  private List<TournamentStat> tournamentStats;

  public boolean addPlayer(Player player);
}

public class Playing11 {
  private List<Player> players;
  private Player twelvthMan;

  public boolean addPlayer(Player player);
}
```

Over, Ball, Wicket, Commentary, Inning and Match: Match will be an abstract class, extended by ODI, Test and T20:

```
public class Over {
  private int number;
  private List<Ball> balls;

  public boolean addBall(Ball ball);
}

public class Ball {
  private Player balledBy;
  private Player playedBy;
  private BallType type;

  private Wicket wicket;
  private List<Run> runs;
  private Commentary commentary;

}

public class Wicket {
  private WicketType wicketType;
  private Player playerOut;
  private Player caughtBy;
  private Player runoutBy;
  private Player stimpedBy;
}

public class Commentary {
  private String text;
  private Date createdAt;
  private Commentator createdBy;
}

public class Inning {
  private int number;
  private Date startTime;
  private List<Over> overs;

  public boolean addOver(Over over);
}

public abstract class Match {
  private int number;
  private Date startTime;
  private MatchResult result;

  private Playing11[] teams;
  private List<Inning> innings;
  private List<Umpire> umpires;
  private Refree refree;
  private List<Commentator> commentators;
```

```
  private List<MatchStat> matchStats;

  public boolean assignStatdium(Stadium stadium);
  public boolean assignRefree(Refree refree);
}

public class ODI extends Match {
  //...
}

public class Test extends Match {
  //...
}
```

Design Facebook - a social network

Facebook is an online social networking service where users can connect with other users to post and read messages. Users access Facebook through their website interface or mobile app.

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 connection requests from other members.
4. Members should be able to follow other members without becoming their connection.
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 connections.
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 connections. Members can link any post with a privacy list, to make the post visible to the members of that privacy list only.
9. Any member should be able to send messages to other members.
10. Any member will 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 connection invitation or a comment on their post.
12. Members should be able to search through posts for a word.

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

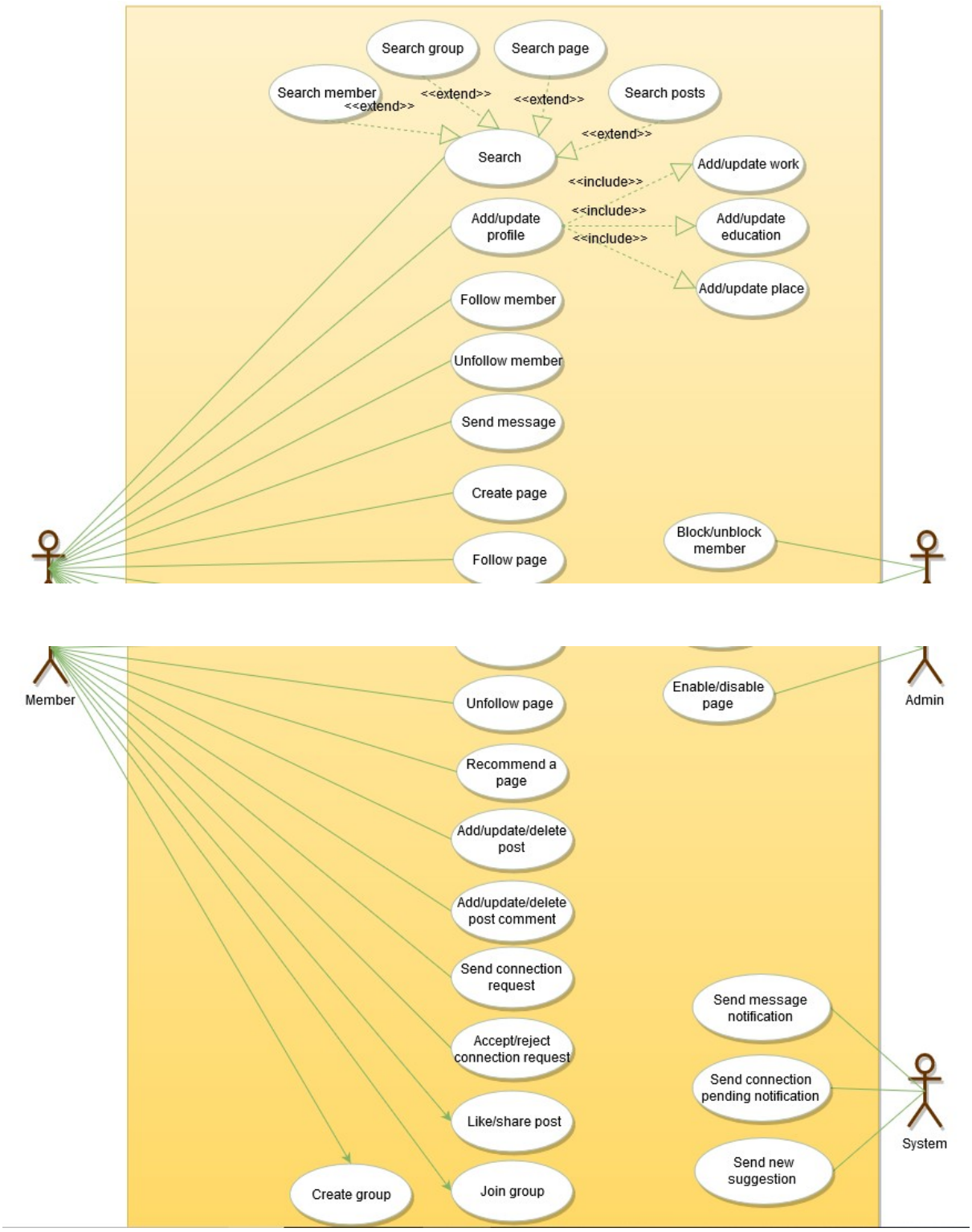
Use-case diagram

We have three main Actors in our system:

- **Member:** All members can search other members, groups, pages, or post, as well as send requests for connection, 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, 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 work experiences, education, etc.
- **Search:** Members can search other members, groups or pages. Members can send a connection request to other members.
- **Follow or Unfollow a member or a page:** Any member can follow or unfollow any other member or a page.
- **Send message** Any member can send a message to any of their connection.
- **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 notification** The system will be able to send notifications for new messages, connection invites, etc.



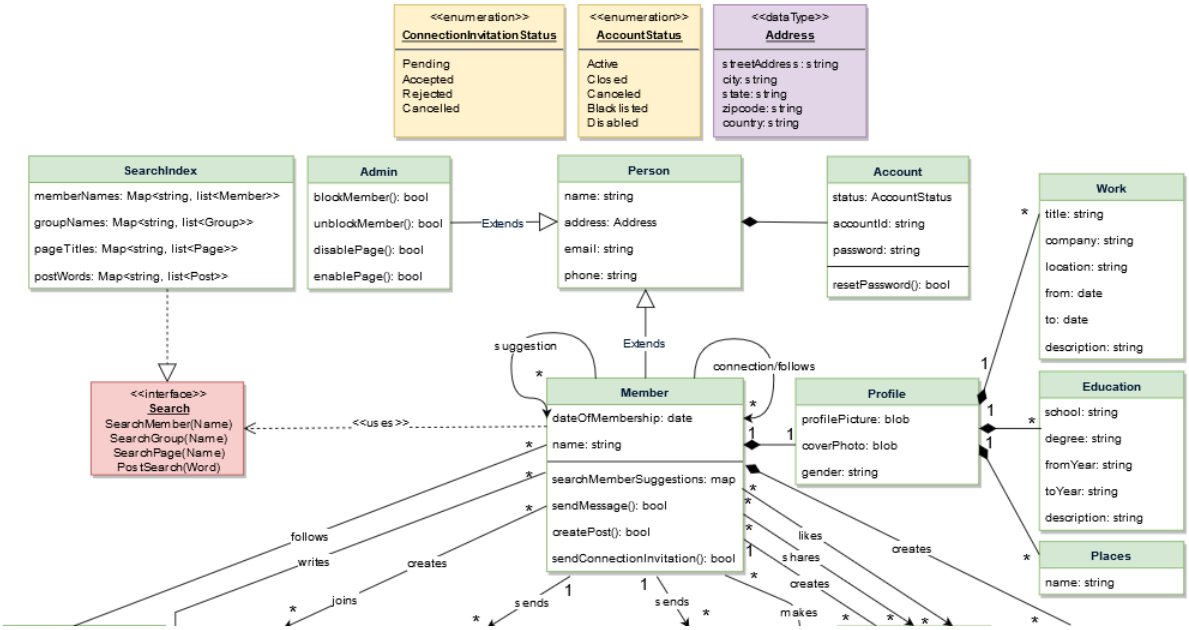
Class diagram

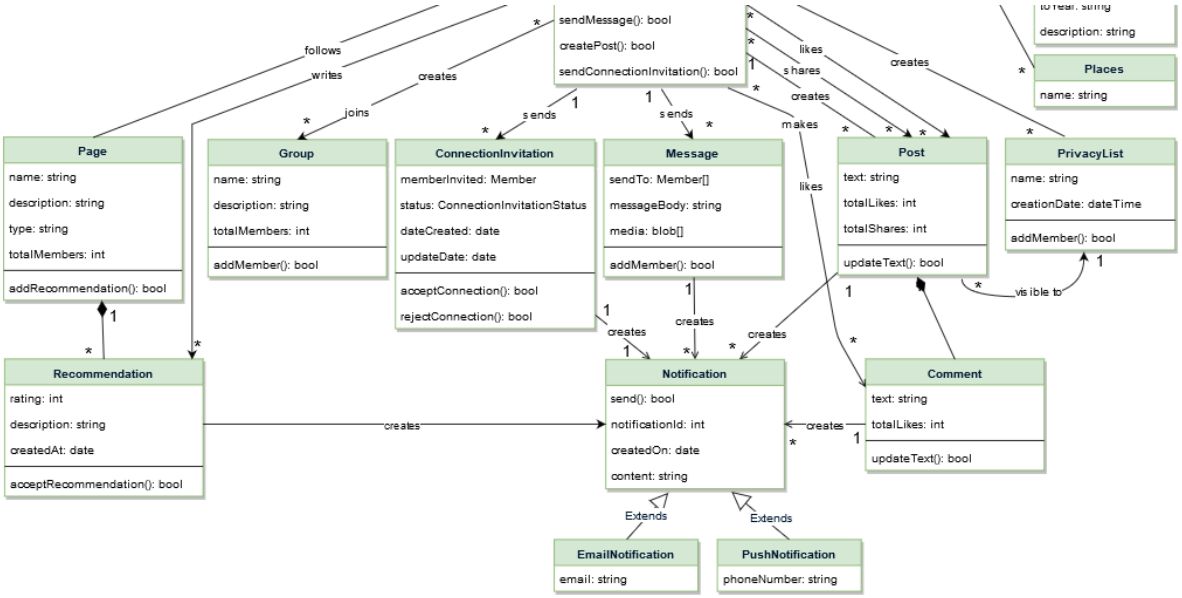
Here are the main classes of the Facebook system:

- **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

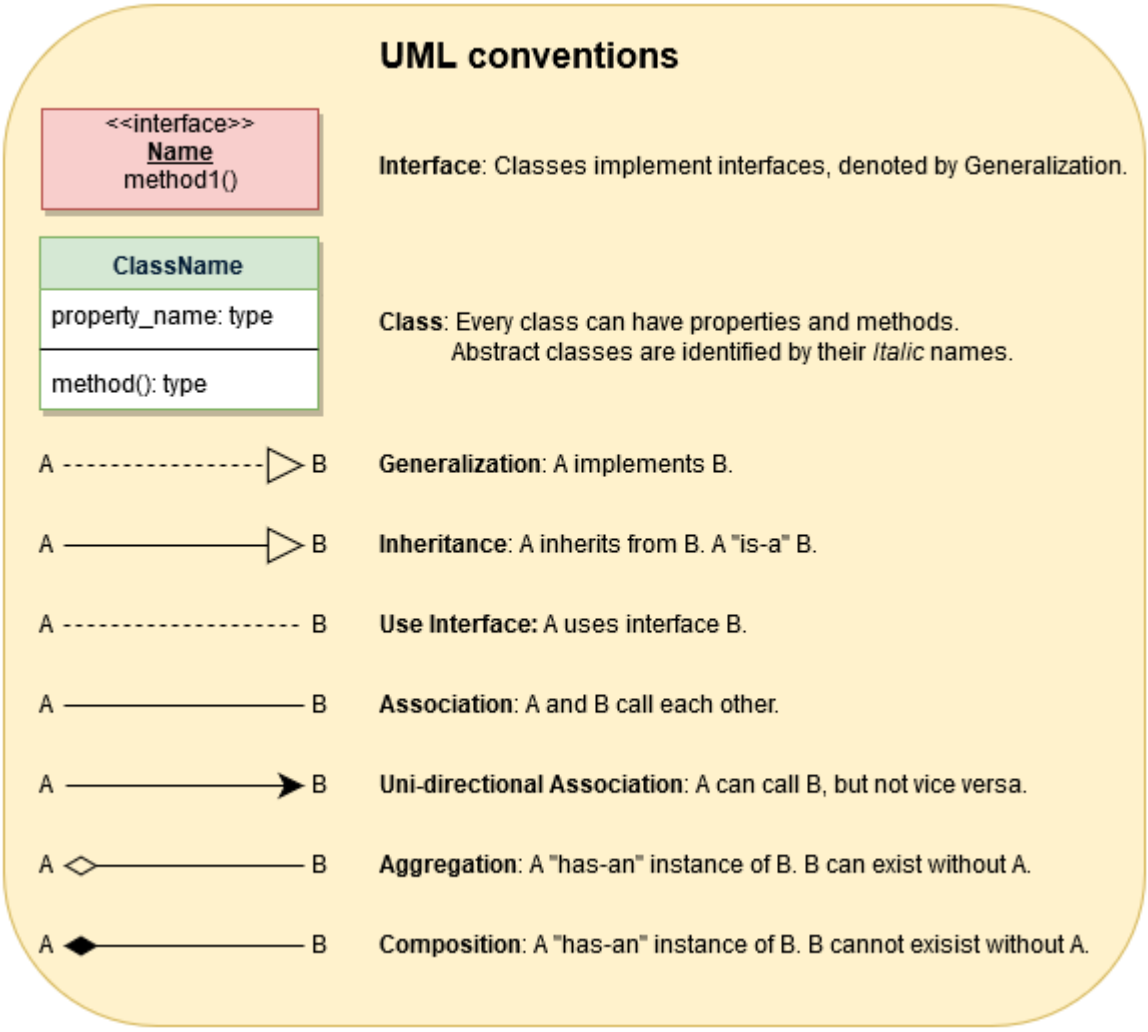
connected to other members and they can follow other members and pages. Members will also have suggestions to make connections with other members.

- **Search:** Our system will support searching other members, groups and pages by their names, and 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 connections. Members can link any post with a privacy list, to make the post visible only to the members of that privacy 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 send an email.



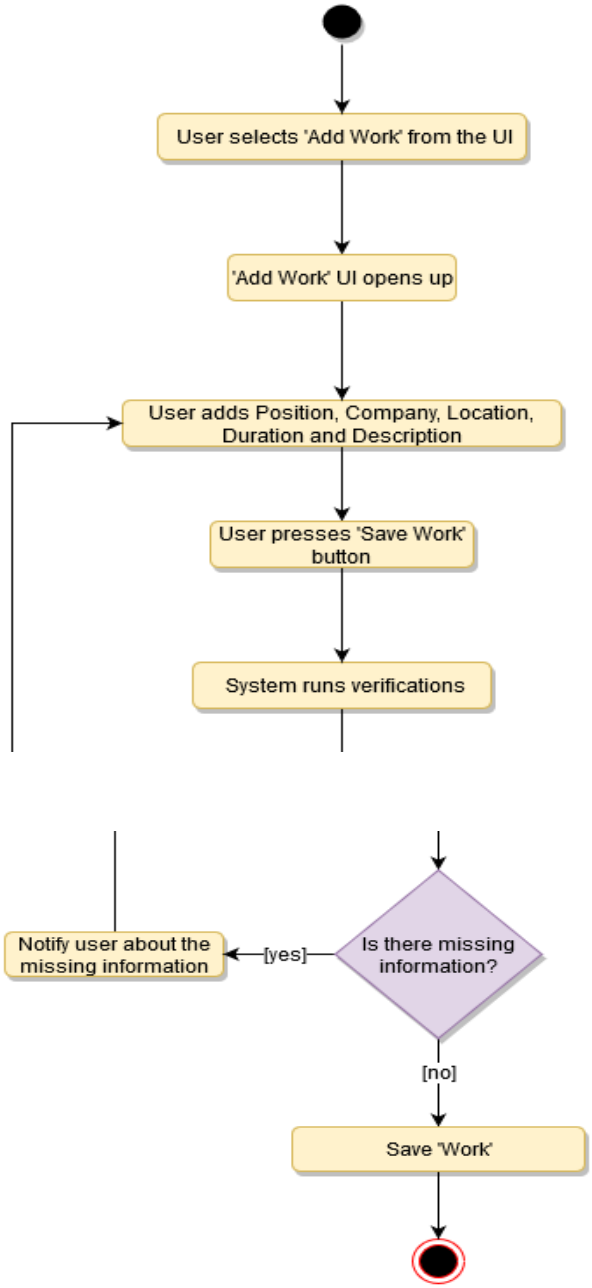


Class diagram

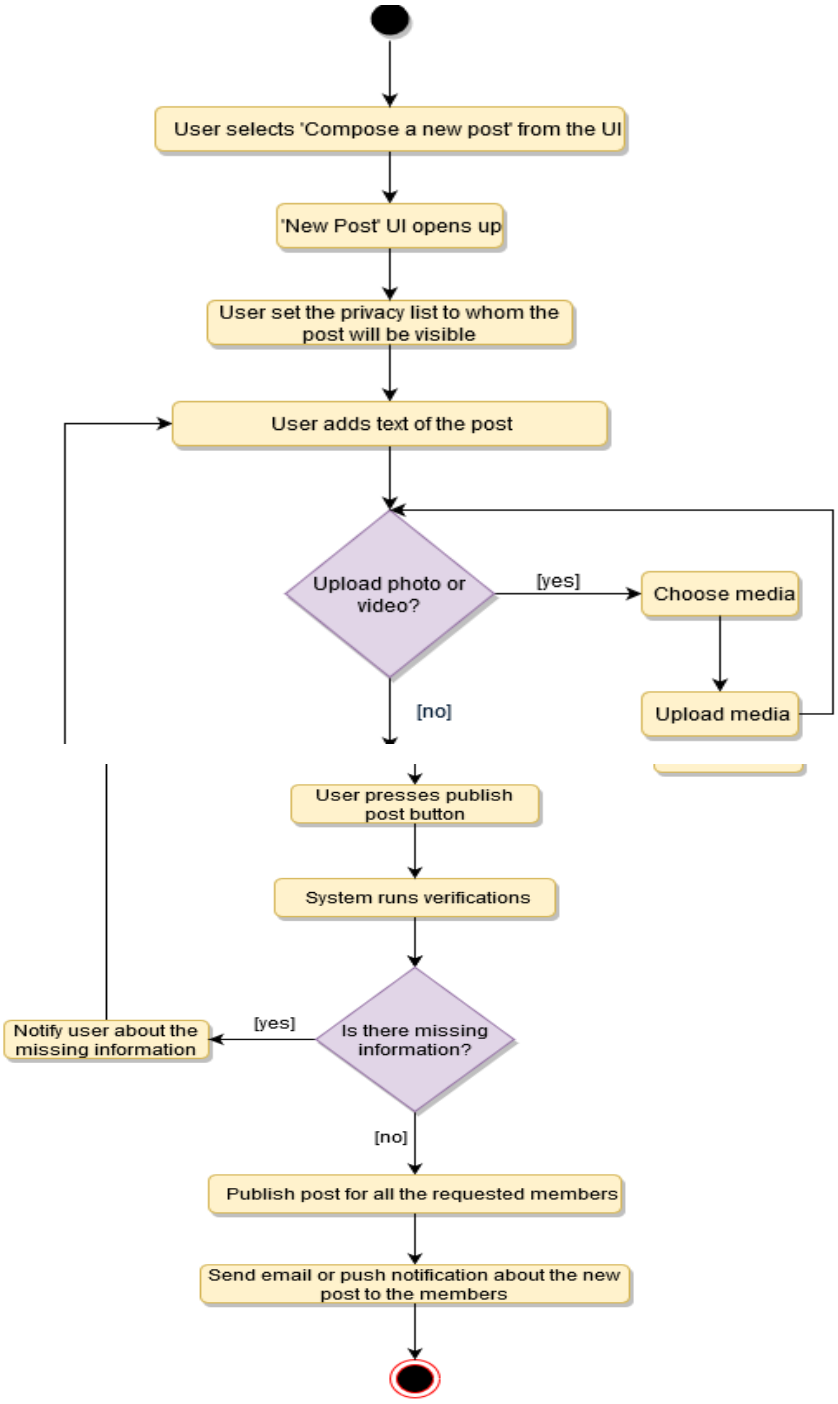


Activity diagrams

Add work experience to profile: Any Facebook member can perform this activity. Here are the set of steps to add work experience to a member's profile:



Create a new post: Any Member can perform this activity. Here are the different steps for creating a post:



Code

Here is the high-level definition for the classes described above.

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

```
public enum ConnectionInvitationStatus{
  PENDING,
  ACCEPTED,
  REJECTED,
  CANCELED
```

```
}

public enum AccountStatus{
  ACTIVE,
  CLOSED,
  CANCELED,
  BLACKLISTED,
  DISABLED
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}
```

Account, Person, Member, and Admin: These classes represent different people that interact with our system:

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

public abstract class Account {
  private String id;
  private String password;
  private AccountStatus status;

  public boolean resetPassword();
}

public abstract class Person extends Account{
  private String name;
  private Address address;
  private String email;
  private String phone;

  private Account account;
}

public class Member extends Person {
  private Integer memberId;
  private Date dateOfMembership;
  private String name;

  private Profile profile;
  private HashSet<Integer> memberFollows;
  private HashSet<Integer> memberConnections;
  private HashSet<Integer> pageFollows;
```

```
  private HashSet<Integer> memberSuggestions;
  private HashSet<ConnectionInvitation> connectionInvitations;
  private HashSet<Integer> groupFollows;

  public boolean sendMessage(Message message);
  public boolean createPost(Post post);
  public boolean sendConnectionInvitation(ConnectionInvitation invitation);
  private Map<Integer, Integer> searchMemberSuggestions();
}

public class Admin extends Person {
  public boolean blockUser(Customer customer);
  public boolean unblockUser(Customer customer);
  public boolean enablePage(Page page);
  public boolean disablePage(Page page);
}

public class ConnectionInvitation {
  private Member meberInvited;
  private ConnectionInvitationStatus status;
  private Date dateCreated;
  private Date dateUpdated;

  public bool acceptConnection();
  public bool rejectConnection();
}
```

Profile, Work, etc: A member's profile will have their work experiences, educations, places, etc:

```
public class Profile {
  private byte[] profilePicture;
  private byte[] coverPhoto;
  private byte[] coverPhoto;
  private String gender;


  private List<Work> workExperiences;
  private List<Education> educations;
  private List<Place> places;
  private List<Stat> stats;

  public boolean addWorkExperience(Work work);
  public boolean addEducation(Education education);
  public boolean addPlace(Place place);
}

public class Work {
  private String title;
  private String company;
  private String location;
  private Date from;
```

```
  private Date to;
  private String description;
}

//...
```

Page and Recommendation: Each page can have multiple recommendations, members will follow/like pages:

```
public class Page {
  private Integer pageId;
  private String name;
  private String description;
  private String type;
  private int totalMembers;
  private List<Recommendation> recommendation;

  private List<Recommendation> getRecommendation();
}

public class Recommendation {
  private Integer recommendationId;
  private int rating;
  private String description;
  private Date createdAt;

  private List<JobPosting> activeJobPostings;
}
```

Group, Post, Message, and Comment: Members can create posts, comment on posts, send messages and join groups:

```
public class Group {
  private Integer groupId;
  private String name;
  private String description;
  private int totalMembers;
  private List<Member> members;

  public boolean addMember(Member member);
  public boolean updateDescription(String description);
}

public class Post {
  private Integer postId;
  private String text;
  private int totalLikes;
  private int totalShares;
  private Member owner;
}

public class Message {
```

```
  private Integer messageId;
  private Memberp[] sentTo;
  private String messageBody;
  private byte[] media;

  public boolean addMember(Member member);
}

public class Comment {
  private Integer commentId;
  private String text;
  private int totalLikes;
  private Member owner;
}
```

Search interface and SearchIndex: SearchIndex will implement Search to facilitate searching of members, companies and job postings:

```
public interface Search {
  public List<Mamber> searchMamber(String name);
  public List<Group> searchGroup(String name);
  public List<Page> searchPage(String name);
  public List<Post> searchPost(String word);
}

public class SearchIndex implements Search {
   HashMap<String, List<Mamber>> memberNames;
   HashMap<String, List<Group>> groupNames;
   HashMap<String, List<Page>> pageTitles;
   HashMap<String, List<Post>> posts;

   public boolean addMember(Member member) {
     if(memberNames.containsKey(member.getName())) {
       memberNames.get(member.getName()).add(member);
     } else {
       memberNames.put(member.getName(), member);
     }
   }

   public boolean addGroup(Group group);
   public boolean addPage(Page page);
   public boolean addPost(Post post);

  public List<Member> searchMamber(String name) {
    return memberNames.get(name);
  }

  public List<Group> searchGroup(String name) {
    return groupNames.get(name);
  }
```

```
  public List<Page> searchPage(String title) {
    return pageTitles.get(title);
  }

  public List<Post> searchPost(String word) {
    return posts.get(word);
  }
}
```

Extended requirement

Here is the code for finding connection suggestions for a member.

There can be many strategies to search for connection suggestions; we will do a two-level deep breadth-first search to find people who have most connections with each other. These people could be good candidates for connection suggestion:

```
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.stream.Collectors;
import static java.util.Collections.reverseOrder;

public class Member extends Person {
  private Integer memberId;
  private Date dateOfMembership;
  private String name;

  private Profile profile;
  private HashSet<Integer> memberFollows;
  private HashSet<Integer> memberConnections;
  private HashSet<Integer> pageFollows;
  private HashSet<Integer> memberSuggestions;
  private HashSet<ConnectionInvitation> connectionInvitations;
  private HashSet<Integer> groupFollows;

  public boolean sendMessage(Message message);
  public boolean createPost(Post post);
  public boolean sendConnectionInvitation(ConnectionInvitation invitation);

  private Map<Integer, Integer> searchMemberSuggestions() {
    Map<Integer, Integer> suggestions = new HashMap<>();
    for(Integer memberId : this.memberConnections) {
      HashSet<Integer> firstLevelConnections = new
Member(memberId).getMemberConnections());
      for(Integer firstLevelConnectionId : firstLevelConnections) {
        this.findMemberSuggestion(suggestions, firstLevelConnectionId);
        HashSet<Integer> secondLevelConnections = new
Member(firstLevelConnectionId).getMemberConnections());
```

```
      for(Integer secondLevelConnectionId : secondLevelConnections) {
        this.findMemberSuggestion(suggestions, secondLevelConnectionId);
      }
    }
          }

  // sort by value (increasing count), i.e., by highest number of mutual connection count
  Map<Integer, Integer> result = new LinkedHashMap<>();
  suggestions.entrySet().stream()
     .sorted(reverseOrder(Map.Entry.comparingByValue()))
     .forEachOrdered(x -> result.put(x.getKey(), x.getValue()));

  return result;
 }

 private void findMemberSuggestion(Map<Integer, Integer> suggestions, Integer
connectionId) {
  // return if the proposed suggestion is already a connection or if there is a
  // pending connection invitation
  if(this.memberConnections.contains(connectionId) ||
     this.connectionInvitations.contains(connectionId)) {
    return;
  }

  int count = suggestions.containsKey(connectionId) ? suggestions.get(connectionId) : 0;
  suggestions.put(connectionId, count + 1);
 }
}
```

Feedback

We can be reached at hello@designgurus.org

For feedback, comments, and suggestions, please contact us at ood@designgurus.org