26. Online stock brokerage System

An Online Stock Brokerage System facilitates its users the trade (i.e. buying and selling) of stocks online. It allows clients to keep track of and execute their transactions, and shows performance charts of the different stocks in their portfolios. It also provides security for their transactions and alerts them to pre-defined levels of changes in stocks, without the use of any middlemen.

The online stock brokerage system automates traditional stock trading using computers and the internet, making the transaction faster and cheaper. This system also gives speedier access to stock reports, current market trends, and real-time stock prices.

System Requirements

1. We will focus on the following set of requirements while designing the online stock brokerage system:
2. Any user of our system should be able to buy and sell stocks.
3. Any user can have multiple watchlists containing multiple stock quotes.
4. Users should be able to place stock trade orders of the following types: 1) market, 2) limit, 3) stop loss and, 4) stop limit.

5. Users can have multiple ‘lots’ of a stock. This means that if a user has bought a stock multiple times, the system should be able to differentiate between different lots of the same stock.
6. The system should be able to generate reports for quarterly updates and yearly tax statements.
7. Users should be able to deposit and withdraw money either via check, wire, or electronic bank transfer.
8. The system should be able to send notifications whenever trade orders are executed.

Use Case Diagram

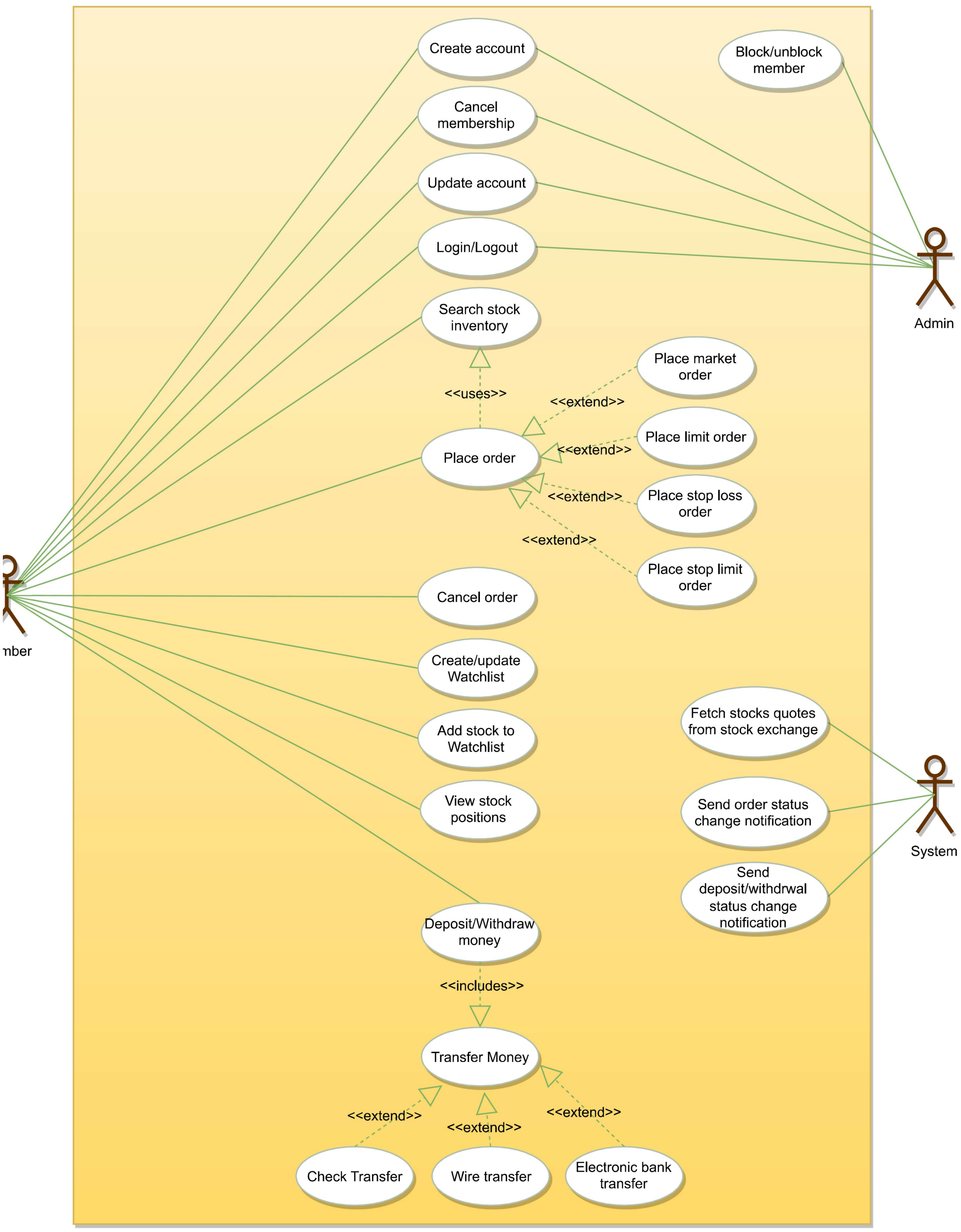
We have three main Actors in our system:

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

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

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

Here is the use case diagram of an Online Stock Brokerage System:



Online Stock Brokerage System Use Case Diagram

Use Case Diagram for Online Stock Brokerage System

Class Diagram

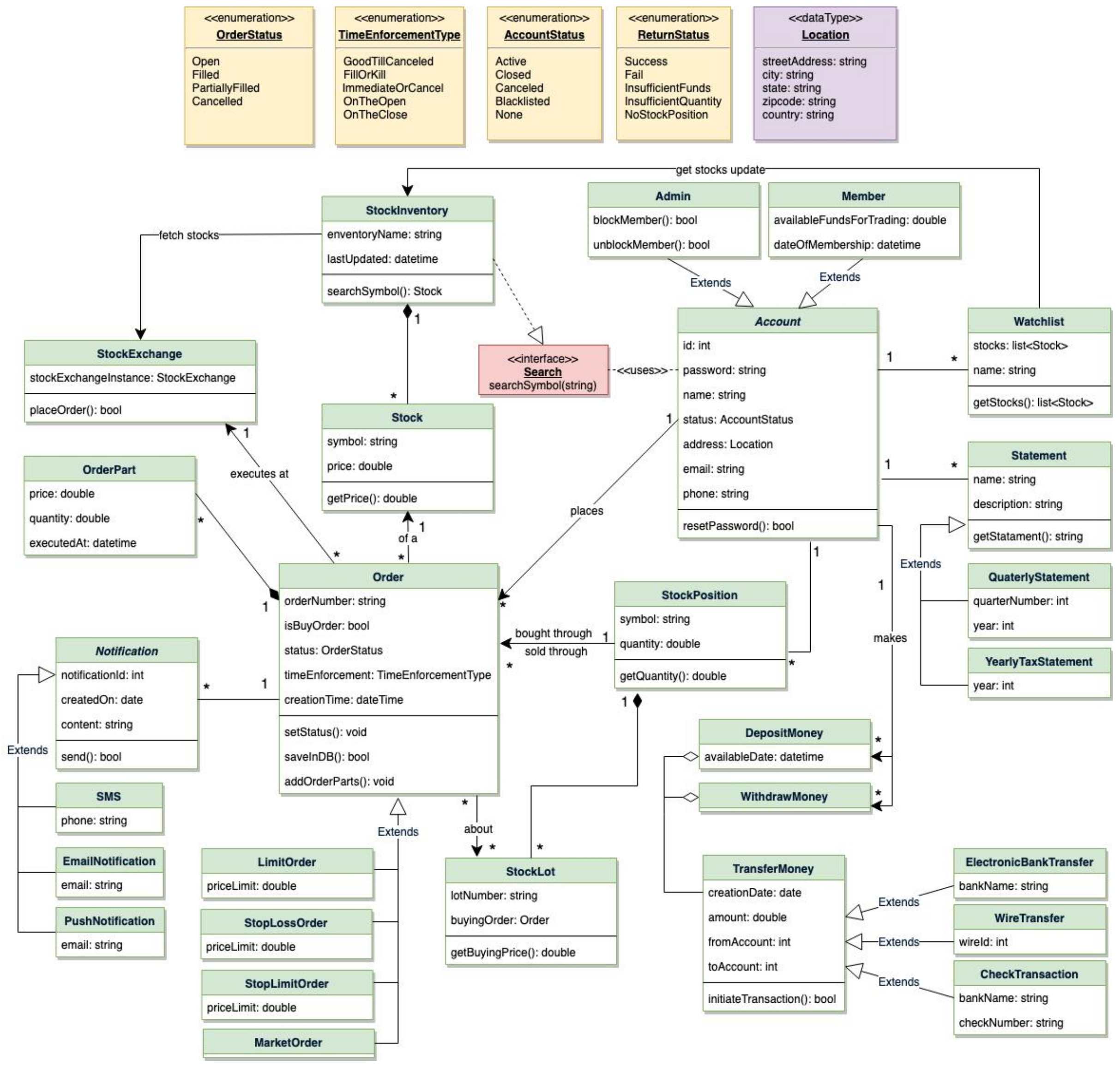
Here are the main classes of our Online Stock Brokerage System:

- Account: Consists of the member’s name, address, e-mail, phone, total funds, funds that are available for trading, etc. We’ll have two types of accounts in the system: one will be a general member, and the other will be an Admin. The Account class will also contain all the stocks the member is holding.
- StockExchange: The stockbroker system will fetch all stocks and their current prices from the stock exchange. StockExchange will be a singleton class encapsulating all interactions with the stock exchange. This class will also be used to place stock trading orders on the stock exchange.
- Stock: The basic building block of the system. Every stock will have a symbol, current trading price, etc.
- StockInventory: This class will fetch and maintain the latest stock prices from the StockExchange. All system components will read the most recent stock prices from this class.
- Watchlist: A watchlist will contain a list of stocks that the member wants to follow.
- Order: Members can place stock trading orders whenever they would like to sell or buy stock positions. The system would support multiple types of orders:
- Market Order: Market order will enable users to buy or sell stocks immediately at the current market price.
- Limit Order: Limit orders will allow a user to set a price at which they want to buy or sell a stock.
- Stop Loss Order: An order to buy or sell once the stock reaches a certain price.
- Stop Limit Order: The stop-limit order will be executed at a specified price or better after a given stop price has been reached. Once the stop price is

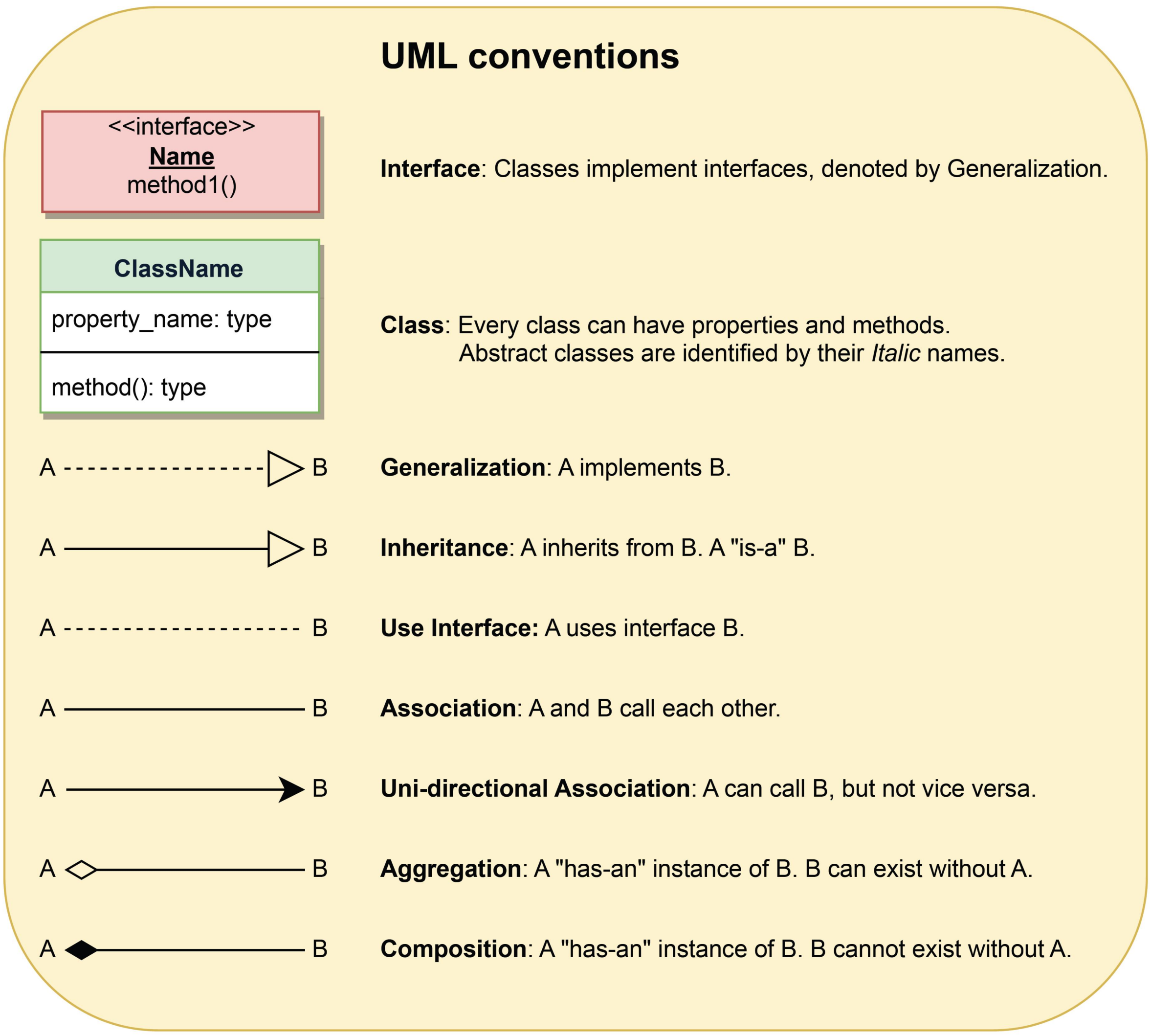
reached, the stop-limit order becomes a limit order to buy or sell at the limit price or better.

- OrderPart: An order could be fulfilled in multiple parts. For example, a market order to buy 100 stocks could have one part containing 70 stocks at $10 and another part with 30 stocks at $10.05.
- StockLot: Any member can buy multiple lots of the same stock at different times. This class will represent these individual lots. For example, the user could have purchased 100 shares of AAPL yesterday and 50 more stocks of AAPL today. While selling, users will be able to select which lot they want to sell first.
- StockPosition: This class will contain all the stocks that the user holds.
- Statement: All members will have reports for quarterly updates and yearly tax statements.
- DepositMoney & WithdrawMoney: Members will be able to move money through check, wire or electronic bank transfers.
- Notification: Will take care of sending notifications to members.

Online Stock Brokerage System Class Diagram



Online Stock Brokerage System UML



Interface: Classes implement interfaces, denoted by Generalization.

Class: Every class can have properties and methods.
Abstract classes are identified by their *Italic* names.

Generalization: A implements B.

Inheritance: A inherits from B. A "is-a" B.

Use Interface: A uses interface B.

Association: A and B call each other.

Uni-directional Association: A can call B, but not vice versa.

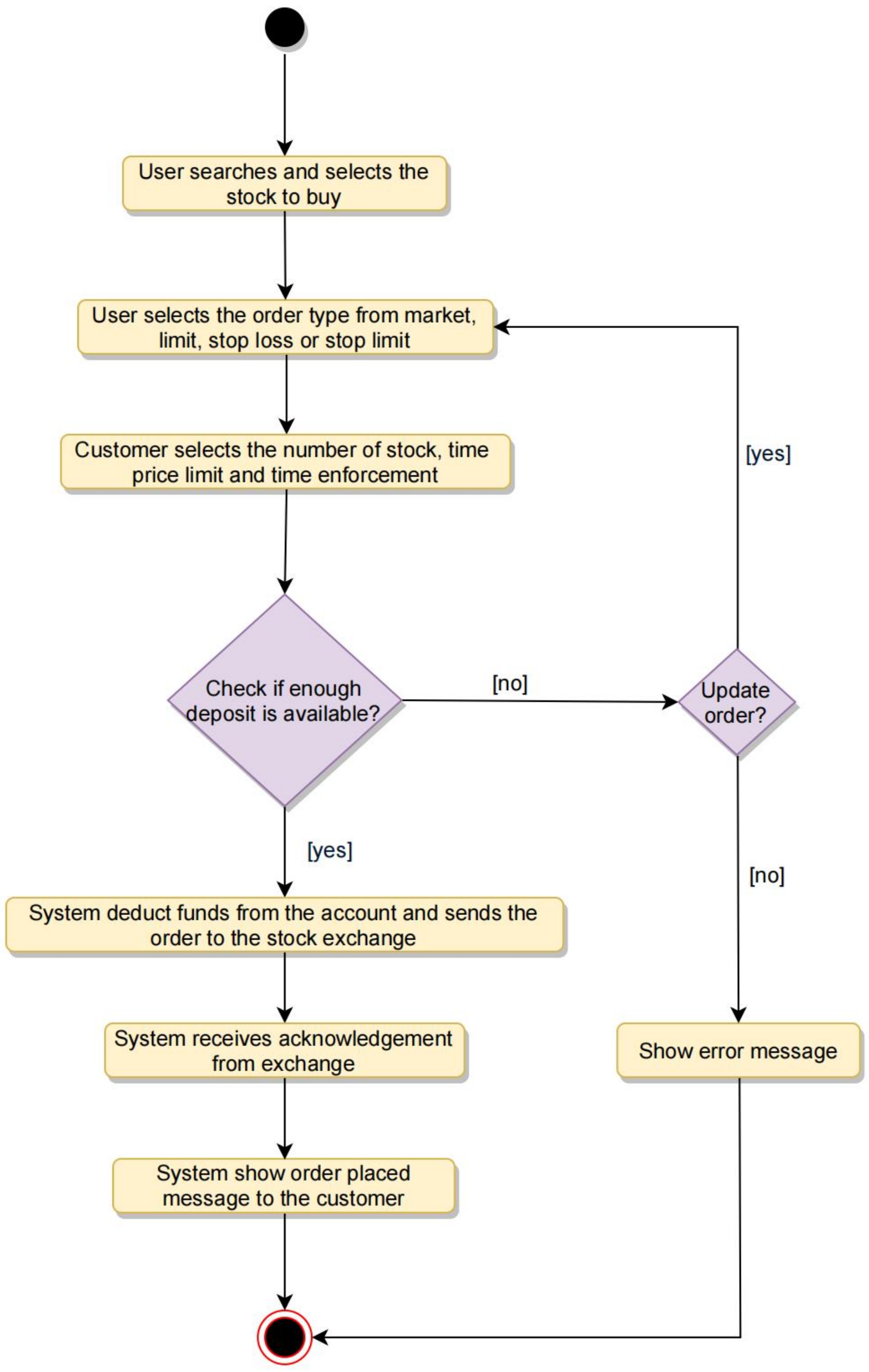
Aggregation: A "has-an" instance of B. B can exist without A.

Composition: A "has-an" instance of B. B cannot exist without A.

UML for Online Stock Brokerage System

Activity Diagrams

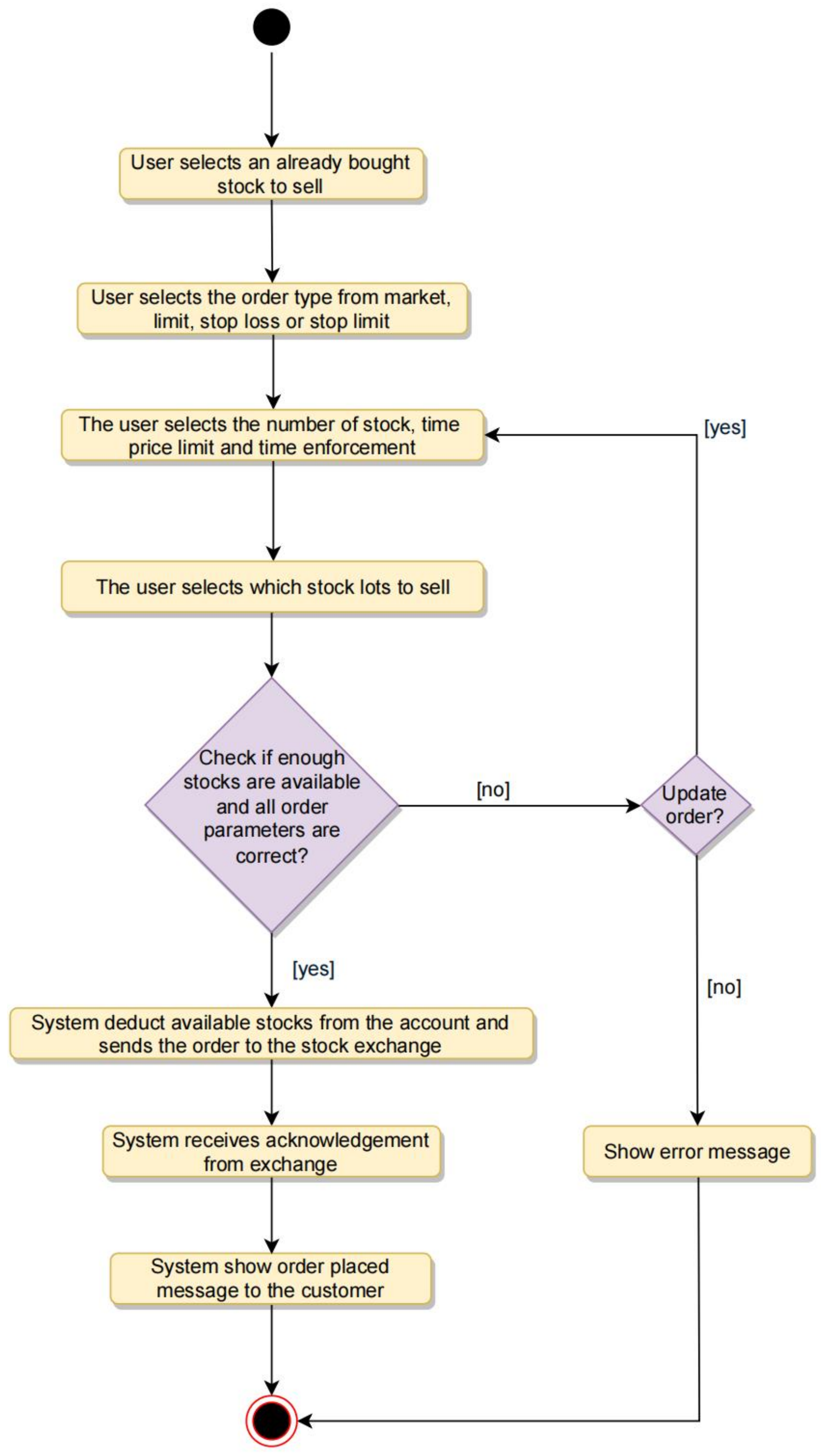
Place a buy order: Any system user can perform this activity. Here are the steps to place a buy order:



User searches and selects the stock to buy
User selects the order type from market, limit, stop loss or stop limit
Customer selects the number of stock, time price limit and time enforcement
Check if enough deposit is available?
[no]
Update order?
[yes]
[yes]
[no]
System deduct funds from the account and sends the order to the stock exchange
System receives acknowledgement from exchange
System show order placed message to the customer
Show error message

Online Stock Brokerage System Buy Order

Activity Diagram for Online Stock Brokerage System Buy Order


User selects an already bought stock to sell
User selects the order type from market, limit, stop loss or stop limit
The user selects the number of stock, time price limit and time enforcement
[yes]
The user selects which stock lots to sell
Check if enough stocks are available and all order parameters are correct?
[no]
Update order?
[yes]
[no]
System deduct available stocks from the account and sends the order to the stock exchange
System receives acknowledgement from exchange
System show order placed message to the customer
Show error message

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

Online Stock Brokerage System Sell Order

Activity Diagram for Online Stock Brokerage System Sell Order

Code

Here is the code for the top use cases.

Enums and Constants:

Here are the required enums and constants:

Enum Definitions:

```
public enum ReturnStatus {
   SUCCESS(1), FAIL(2), INSUFFICIENT_FUNDS(3), INSUFFICIENT_QUANTITY(4),
NO_STOCK_POSITION(5);

   private final int value;
   ReturnStatus(int value) { this.value = value; }
   public int getValue() { return this.value; }
}

public enum OrderStatus {
   OPEN(1), FILLED(2), PARTIALLY_FILLED(3), CANCELLED(4);

   private final int value;
   OrderStatus(int value) { this.value = value; }
   public int getValue() { return this.value; }
}
```

```
public enum TimeEnforcementType {
   GOOD_TILL_CANCELLED(1), FILL_OR_KILL(2), IMMEDIATE_OR_CANCEL(3), ON_THE_OPEN(4),
ON_THE_CLOSE(5);


   private final int value;
   TimeEnforcementType(int value) { this.value = value; }
   public int getValue() { return this.value; }
}


public enum AccountStatus {
   ACTIVE(1), CLOSED(2), CANCELED(3), BLACKLISTED(4), NONE(5);


   private final int value;
   AccountStatus(int value) { this.value = value; }
   public int getValue() { return this.value; }
}
```

Location Class:

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

```
    public Location(String street, String city, String state, String zipCode, String country) {

        this.streetAddress = street;

        this.city = city;

        this.state = state;

        this.zipCode = zipCode;

        this.country = country;

    }

}
```

Constants Class:

```
public class Constants {

    public static final double MONEY_TRANSFER_LIMIT = 100000.0;

}
```

StockExchange Singleton:

```
public class StockExchange {

    private static StockExchange instance;



    private StockExchange() { }



    public static StockExchange getInstance() {

        if (instance == null) {

            instance = new StockExchange();

        }

        return instance;
    }



    public ReturnStatus placeOrder(Order order) {

        // Submit the order

        return ReturnStatus.SUCCESS; // Placeholder for actual implementation

    }

}
```

Order Class (Abstract Class):

```
import java.util.*;



public abstract class Order {

    private String orderId;

    private boolean isBuyOrder;

    private OrderStatus status;

    private TimeEnforcementType timeEnforcement;

    private Date creationTime;

    private Map<String, Order> parts;



    public Order(String id) {

        this.orderId = id;

        this.isBuyOrder = false;

        this.status = OrderStatus.OPEN;

        this.timeEnforcement = TimeEnforcementType.ON_THE_OPEN;

        this.creationTime = new Date();
        this.parts = new HashMap<>();

    }



    public void setStatus(OrderStatus status) {

        this.status = status;

    }



    public void saveInDB() {

        // Save to DB

    }



    public void addOrderParts(List<Order> parts) {

        for (Order part : parts) {

            this.parts.put(part.getOrderId(), part);

        }

    }



    public String getOrderId() {

        return orderId;

    }

}
```

LimitOrder Class:

```
public class LimitOrder extends Order {

    private double priceLimit;
    public LimitOrder(String id, double priceLimit) {
        super(id);
        this.priceLimit = priceLimit;
    }

    public double getPriceLimit() {
        return priceLimit;
    }
}
```

Account Class (Abstract Class):

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

    public Account(String id, String password, String name, Location address, String email, String
phone, AccountStatus status) {
        this.id = id;
        this.password = password;
        this.name = name;
        this.address = address;

        this.email = email;

        this.phone = phone;

        this.status = status;

    }



    public void resetPassword() {

        // Reset password logic

    }

}
```

Member Class:

```
import java.util.*;



public class Member extends Account {

    private double availableFundsForTrading;

    private Date dateOfMembership;

    private Map<String, Integer> stockPositions; // stockId -> quantity

    private Map<String, Order> activeOrders; // orderId -> Order



    public Member(String id, String password, String name, Location address, String email, String
phone, AccountStatus status) {

        super(id, password, name, address, email, phone, status);

        this.availableFundsForTrading = 0.0;

        this.dateOfMembership = new Date();
        this.stockPositions = new HashMap<>();

        this.activeOrders = new HashMap<>();

    }


    public ReturnStatus placeSellLimitOrder(String stockId, int quantity, double limitPrice,
TimeEnforcementType enforcementType) {

        if (!stockPositions.containsKey(stockId)) {

            return ReturnStatus.NO_STOCK_POSITION;

        }


        int stockQuantity = stockPositions.get(stockId);

        if (stockQuantity < quantity) {

            return ReturnStatus.INSUFFICIENT_QUANTITY;

        }


        LimitOrder order = new LimitOrder(stockId, limitPrice);

        order.setStatus(OrderStatus.OPEN);

        order.saveInDB();

        ReturnStatus result = StockExchange.getInstance().placeOrder(order);


        if (result != ReturnStatus.SUCCESS) {

            order.setStatus(OrderStatus.CANCELLED);

            order.saveInDB();

        } else {

            activeOrders.put(order.getOrderId(), order);

        }
        return result;
    }


    public ReturnStatus placeBuyLimitOrder(String stockId, int quantity, double limitPrice,
TimeEnforcementType enforcementType) {
        double totalCost = quantity * limitPrice;
        if (availableFundsForTrading < totalCost) {
            return ReturnStatus.INSUFFICIENT_FUNDS;
        }


        LimitOrder order = new LimitOrder(stockId, limitPrice);
        order.setStatus(OrderStatus.OPEN);
        order.saveInDB();
        ReturnStatus result = StockExchange.getInstance().placeOrder(order);


        if (result != ReturnStatus.SUCCESS) {
            order.setStatus(OrderStatus.CANCELLED);
            order.saveInDB();
        } else {
            activeOrders.put(order.getOrderId(), order);
        }


        return result;
    }
    public void callbackStockExchange(String orderId, List<Order> orderParts, OrderStatus status) {

        Order order = activeOrders.get(orderId);

        order.addOrderParts(orderParts);

        order.setStatus(status);

        order.saveInDB();


        if (status == OrderStatus.FILLED || status == OrderStatus.CANCELLED) {

            activeOrders.remove(orderId);

        }

    }

}
```