

***UE21CS352B - Object Oriented Analysis & Design using Java***

Mini Project Report “Enhanced E-Commerce Bidding Platform”

***Submitted by:***

# Lekhan Saathvik Lekkala

# Managli Bhavya

# Meenal Bagare

# Meghana Anand

**PES2UG21CS255PES2UG21CS277 PES2UG21CS289 PES2UG21CS291**

*6th Semester E Section*

# Prof. Shilpa

Associate Professor

**January - May 2024**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

FACULTY OF ENGINEERING

**PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013) 100ft Ring Road, Bengaluru – 560 085, Karnataka, India

# ABSTRACT

This project involves implementing a client-server model for a stock auction system, as well as a publisher-subscriber model for sharing stock profit information. The client-server model allows traders to bid on various stocks, with the server handling bid updates and tracking bid changes. The publisher-subscriber model enables companies to publish monthly profit information, which traders can then subscribe to and receive updates on. The project requires understanding of socket programming, multithreading, and message passing between different components of the system.

# SYNOPSIS:

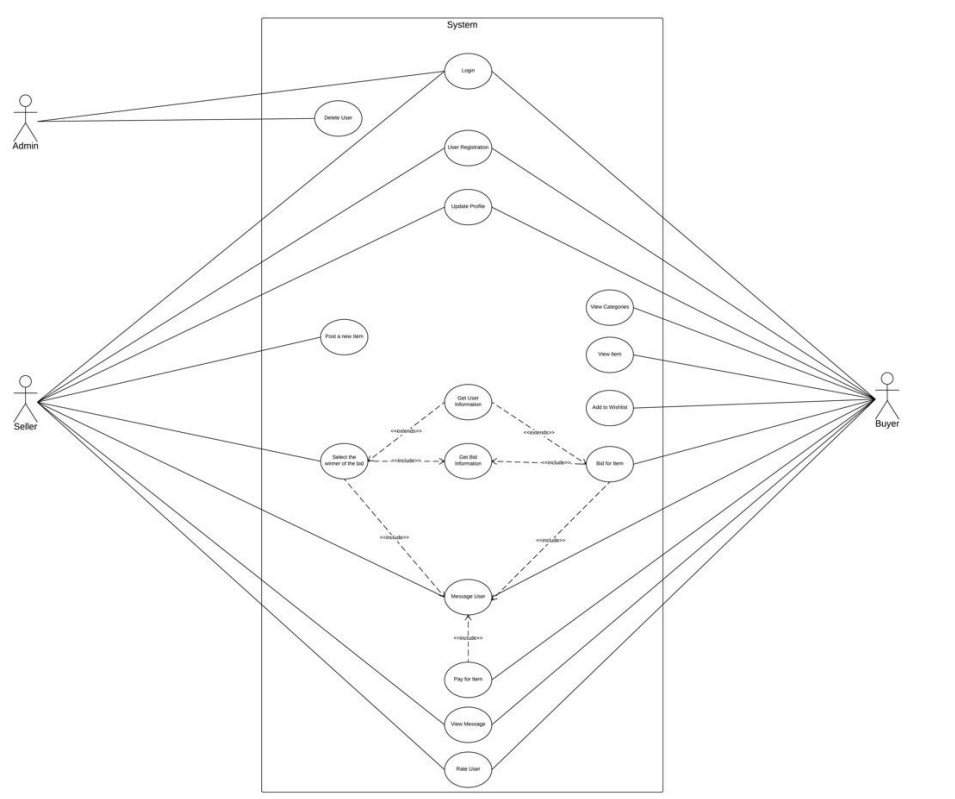
This project involves implementing a client-server model to replicate the functionality of a stock auction, as well as a publisher-subscriber model to enable the sharing of stock profit information.In the client-server model, the server acts as the auctioneer, maintaining information about the different stocks being auctioned. This information includes the stock symbol, base price, security number, and profit. The server listens for incoming connections from clients (traders) on port 2021 and handles multiple concurrent connections.

Clients can interact with the server through various functionalities. They can query the server to get the current highest bid for a specific stock. They can also place new bids, as long as the bid is higher than the current highest bid. The server tracks all bid changes and updates the highest bid accordingly.The publisher-subscriber model is an extension to the client-server model, where the server listens for incoming connections on port 2022. In this model, publishers (companies) can publish monthly profit information for the different stocks, and subscribers (traders) can subscribe to receive these updates.

When a publisher updates the profit information for a stock, the server validates the input and updates the stored data. Subscribers can then receive these updates in real-time, enabling them to stay informed about the financial status of the stocks they are interested in.The project requires a good understanding of socket programming, multithreading, and message passing between the different components of the system. The server must be able to efficiently handle the various functionalities and maintain the necessary data structures to support the stock auction and profit information sharing

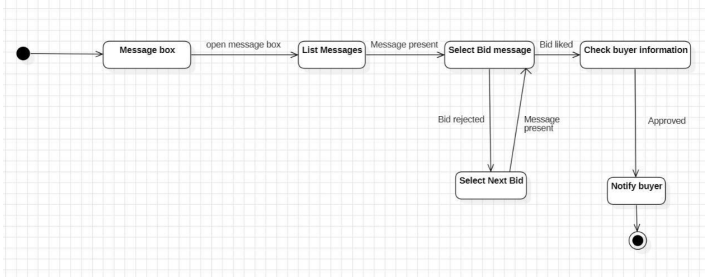
# UML DIAGRAMS:

# USE CASE DIAGRAM

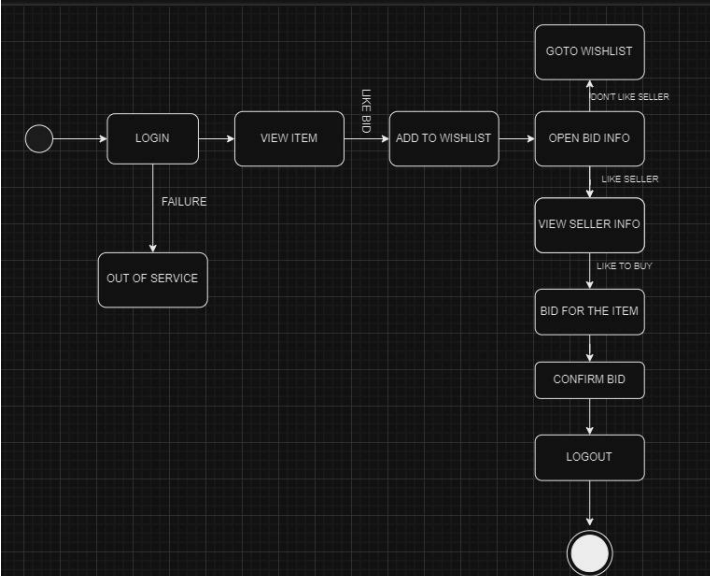
****

**State diagram:**

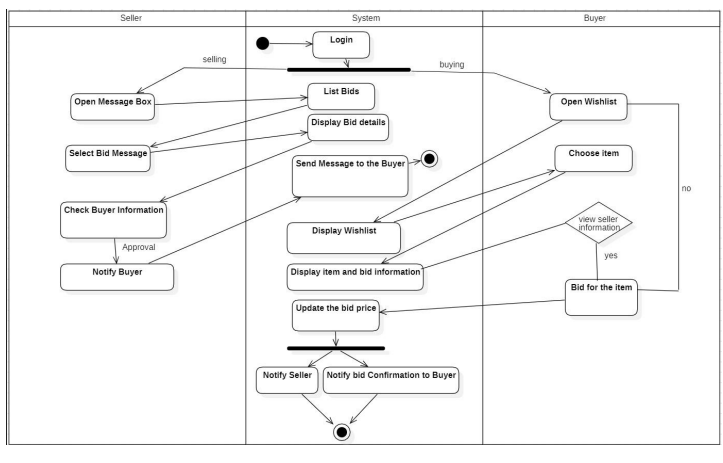
Seller:

****

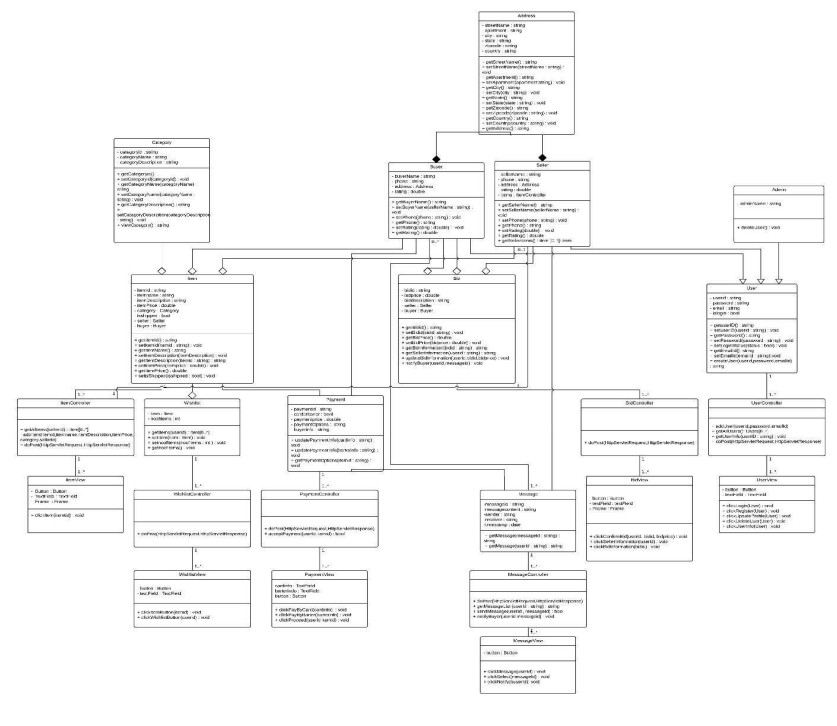
Buyer:



Activity Diagram:



**CLASS DIAGRAM :**

****

1. **ARCHITECTURE PATTERNS**:

**The Model-View-Controller (MVC):**

1. Model : The "Model" component in this project is responsible for managing the data and the business logic of the stock auction system.

- The stock.py file likely contains the classes and data structures that represent the different stocks, their properties (e.g., symbol, base price, security number, profit), and the bid history for each stock.

- The auction.py file probably encapsulates the core functionality of the stock auction, such as managing the bidding process, updating the highest bids, and handling the auction timeline (e.g., extending the bidding time if a bid is placed within the last minute).

- The publish\_subscribe.py file likely handles the publishing and subscribing of stock profit information, maintaining the necessary data structures and managing the communication between publishers and subscribers.

1. View : The "View" component in this project is responsible for the user interface and the presentation of information to the users (clients and publishers).

- The client.py file probably defines the client-side interface, allowing traders to interact with the auction system, such as displaying the current highest bids, placing new bids, and receiving notifications about bid updates.

- The publisher.py file likely provides the interface for companies to publish their monthly profit information, enabling them to update the server with the latest financial data.

1. Controller : The "Controller" component in this project acts as the intermediary between the "Model" and the "View", handling user interactions and coordinating the flow of data between the two.

- The server.py file is likely the main entry point of the application, serving as the controller. It sets up the server, listens for incoming connections from clients and publishers, and dispatches the appropriate actions based on the user requests.

- The server.py file probably interacts with the "Model" components (e.g., stock.py , auction.py , publish\_subscribe.py ) to perform the necessary operations, such as retrieving stock information, processing bids, and managing the publishing and subscribing of profit data.

- The server.py file may also be responsible for sending the appropriate responses back to the "View" components (e.g., client.py , publisher.py ) to update the user interface and provide the necessary information.

1. **DESIGN PRINCIPLES:**

## Single Responsibility Principle (SRP)

* + A class should have only one reason to change, meaning that a class should have a single responsibility or job.

## Used in the project:

## - ServerTask class focuses on handling server-side tasks like user authentication and registration.

## - MessageHandler interface and its implementations (Msg) are responsible for message handling.

## - UserDataAccess and CompanyDataAccess interfaces along with their implementations (TraderDataTxt, CompanyDataTxt) are responsible for data access operations.

## Dependency Inversion Principle (DIP)

* + High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

## Used in the project:

* + - High-level modules (e.g., controllers) depend on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations.
    - ServerTask class depends on abstractions (MessageHandler, UserDataAccess, CompanyDataAccess) rather than concrete classes.
    - Concrete implementations are injected into ServerTask through constructor injection, allowing flexibility in substituting implementations.
    - Company directory with conconn and connection files also adheres to DIP.

## Open/Closed Principle (OCP)

* + Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

## Used in the project:

-The ServerTask class is designed to be extended with new message handling or data access strategies by injecting different implementations of interfaces (MessageHandler, UserDataAccess, CompanyDataAccess).

## Interface Segregation Principle (ISP)

* + Clients should not be forced to depend on interfaces they do not use.

## Used in the project:

* + The presence of separate interfaces for different concerns in the packages suggests that the project may be adhering to the ISP.
  + This principle can lead to better modularity and maintainability by ensuring that classes only depend on the interfaces they actually need.

1. **DESIGN PATTERNS**

Here are three design patterns used in the project, with their definitions, descriptions, and where they are used, in the requested format:

## Observer Pattern

* + The Observer pattern has an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods..

## Description:

* + - The StoreSubConn class represents a publisher-subscriber system, where ServerSubHandler objects act as subscribers that are added to or removed from the list maintained by StoreSubConn. This pattern facilitates a one-to-many dependency where subscribers (ServerSubHandler) receive updates from the publisher (StoreSubConn).
    - The ActionHandler class, we register it as an observer to the SubscribeList. This allows the ActionHandler to receive updates from the SubscribeList and react accordingly. The observer pattern decouples the subject (in this case, the SubscribeList) from its observers (ActionHandler), allowing for easier maintenance and extension..

## Command Pattern

* + The pattern intends to encapsulate in an object all the data required for performing a given action (command), including what method to call, the method’s arguments, and the object to which the method belongs.
  + This model allows us to decouple objects that produce the commands from their consumers, so that’s why the pattern is commonly known as the producer-consumer pattern..

## Description:

* + - The handleAction method in the ActionHandler class is seen as an application of the command pattern. Each action (command) received by the ActionHandler is encapsulated as an object, allowing for parameterization and decoupling of the sender (source of the command) from the receiver (ActionHandler)..

## Model-View-Presenter (MVP) Pattern

* + The MVP pattern is a derivative of the Model-View-Controller (MVC) pattern,

where the Presenter acts as an intermediary between the View and the Model, handling user input and updating the View accordingly.

## Description:

* The structure of the code, with separate packages for model, view, and controller components (client, clientCore, clientGui, clientModel, clientSupMethod), suggests adherence to the MVC pattern. This pattern helps in separating concerns, improving maintainability, and facilitating code reuse.

## Facade Pattern

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about

In our project, the implementation of the Facade pattern is :

* ActionHandler.java file serve as a Facade by encapsulating interactions with various subsystems or components within the handleAction method. By providing a simplified interface for handling actions, it hides the complexity of interactions with underlying subsystems from the calling code, which aligns with the goals of the Facade pattern.

1. **INDIVIDUAL CONTRIBUTION:**

**Person 1 (Model):**

- Implement the `stock.py` file, which includes the classes and data structures for stocks, their properties, and bid history.

- Develop the `auction.py` file, which encapsulates the core functionality of the stock auction, such as managing the bidding process, updating the highest bids, and handling the auction timeline.

- Understand the requirements and design of the stock auction system.

- Collaborate with the team to ensure the Model component aligns with the overall system architecture.

- Optimize the data structures and algorithms used in the auction and stock management logic.

**Person 2 (Model):**

- Implement the `publish\_subscribe.py` file, which handles the publishing and subscribing of stock profit information, maintaining the necessary data structures and managing the communication between publishers and subscribers.

- Analyze the requirements for the publisher-subscriber model and design the necessary data structures and communication protocols.

- Ensure the publish-subscribe functionality is well-integrated with the overall stock auction system.

- Optimize the performance and scalability of the publish-subscribe component.

**Person 3 (View):**

- Implement the `client.py` file, which defines the client-side interface, allowing traders to interact with the auction system, such as displaying the current highest bids, placing new bids, and receiving notifications about bid updates.

- Implement the `publisher.py` file, which provides the interface for companies to publish their monthly profit information, enabling them to update the server with the latest financial data.

- Design the user interfaces and user experience for both the client and publisher components.

- Ensure the user interfaces are intuitive, responsive, and align with the overall system requirements.

**Person 4 (Controller):**

- Implement the `server.py` file, which serves as the main entry point of the application, setting up the server, listening for incoming connections from clients and publishers, and dispatching the appropriate actions based on the user requests.

- Integrate the "Model" and "View" components, ensuring the Controller properly handles the interactions and coordinates the flow of data between them.

- Understand the overall system architecture and the interactions between the different components.

- Design the communication protocols and message formats between the server, clients, and publishers.

**GITHUB LINK TO REPO:**

**OUTPUT SCREENSHOTS:**