*Process MeNtOR 3.o*

***Uni-SEP***

# <Warehouse Management System>

# **Design Document**

| | |
|---|---|
| Version: | 2.0 |
| Print Date: | |
| Release Date: | |
| Release State: | |
| Approval State: | |
| Approved by: | |

| | |
|---|---|
| Prepared by: | |
| Reviewed by: | |
| Path Name: | |
| File Name: | |
| Document No: | |

# Document Change Control

| Version | Date | Authors | Summary of Changes |
|---|---|---|---|
| 2.0 | 23/11/23 | HB, VV, PA, DE | |
| | | | |
| | | | |
| | | | |

# Document Sign-Off

| Name (Position) | Signature | Date |
|---|---|---|
| Hashaam Bajwa | HB | 23/11/23 |
| Vaidik Vekaria | VV | 23/11/23 |
| Peter Ayade | PA | 23/11/23 |
| Dexter Eromosle | DE | 23/11/23 |

# Contents

# 1 Introduction

## 1.1 Purpose

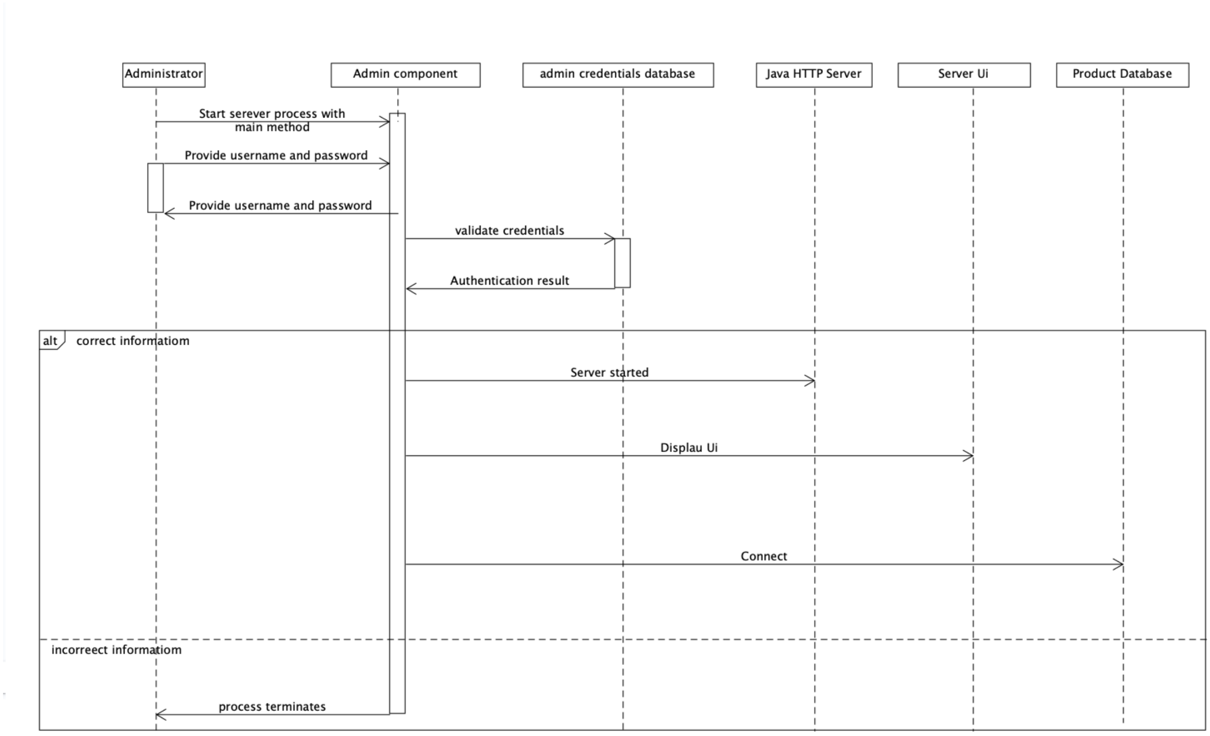
*This document details the requirements of the system <Warehouse Management System>.*

## 1.2 Overview

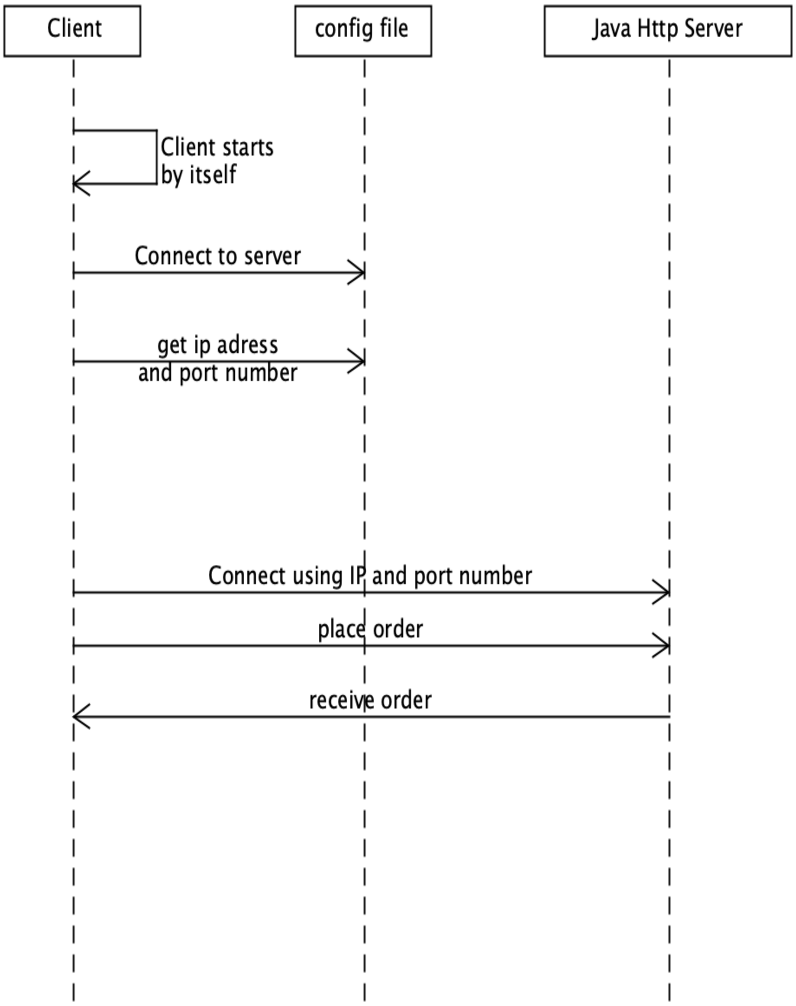
*This document showcases a systematic breakdown of how the system functions and what the system encompasses. Initially the Sequence and Activity Diagrams of the different Use cases are shown. The sequence diagram will describe the behavior of specific objects for a particular use case while the activity diagram highlights how objects collaborate. Next a modular breakdown of the system is given. After this, the project plan is highlighted in the form of a Gantt Chart and backlog. Also, the meeting minutes for past meetings are posted as well. At the end, the different test cases are highlighted to ensure the system is functional.*

# 2 Sequence Diagrams

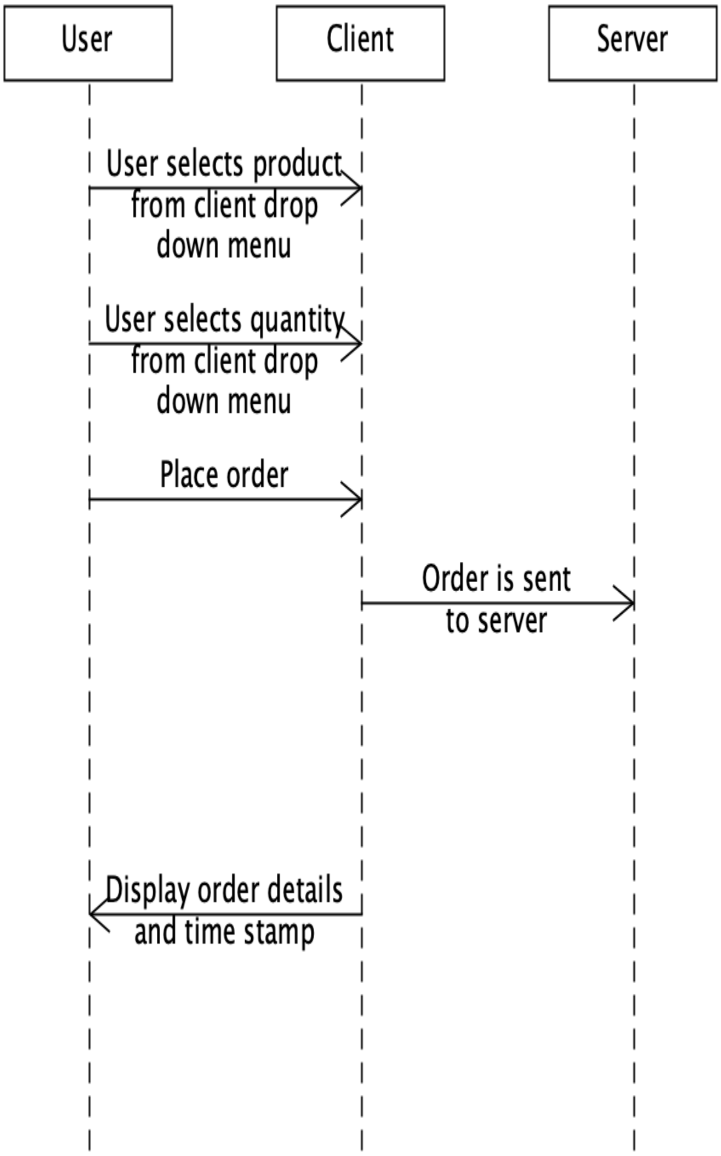
***Use Case 1***



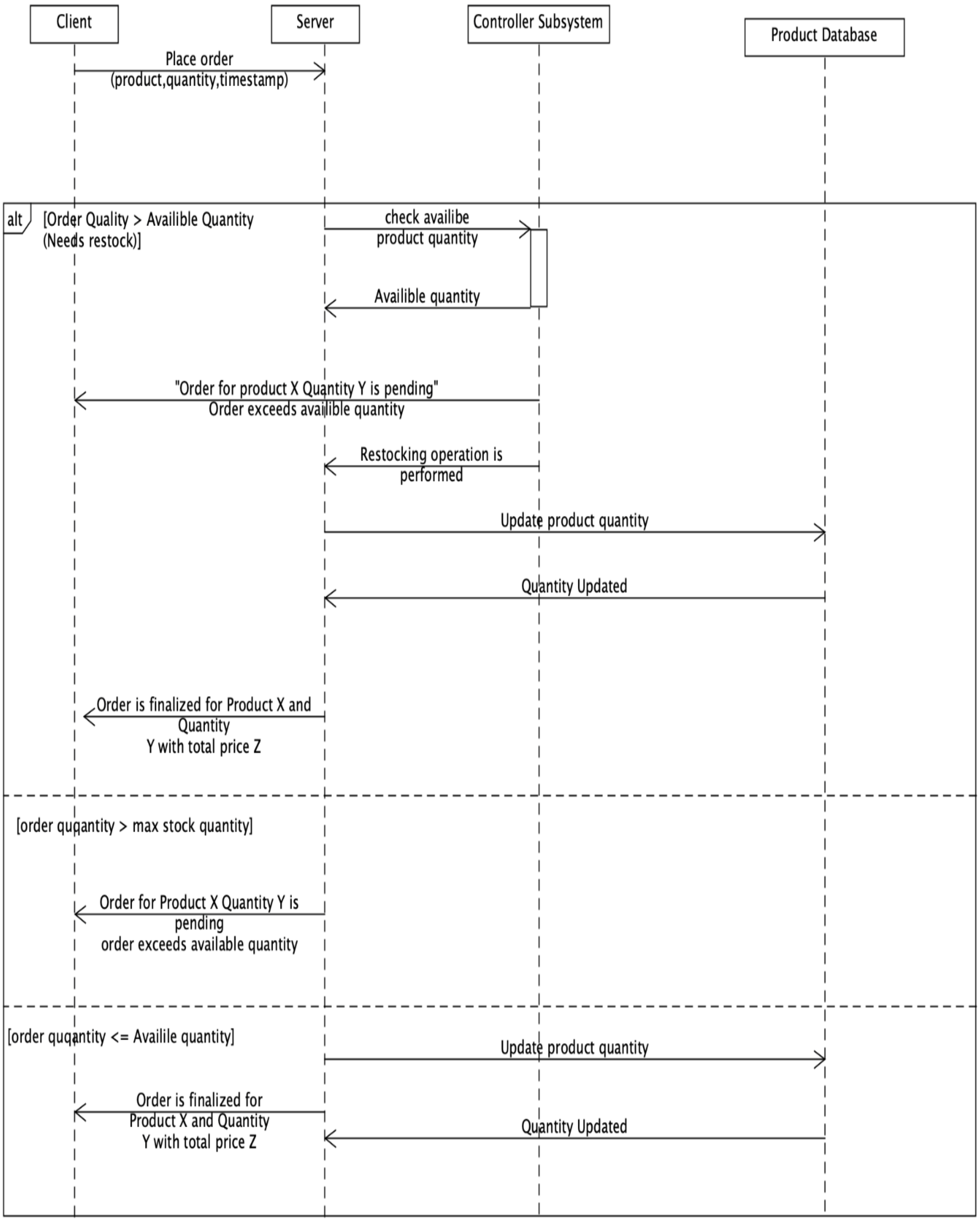
### *Use Case 2*


Client
config file
Java Http Server
Client starts by itself
Connect to server
get ip adress and port number
Connect using IP and port number
place order
receive order

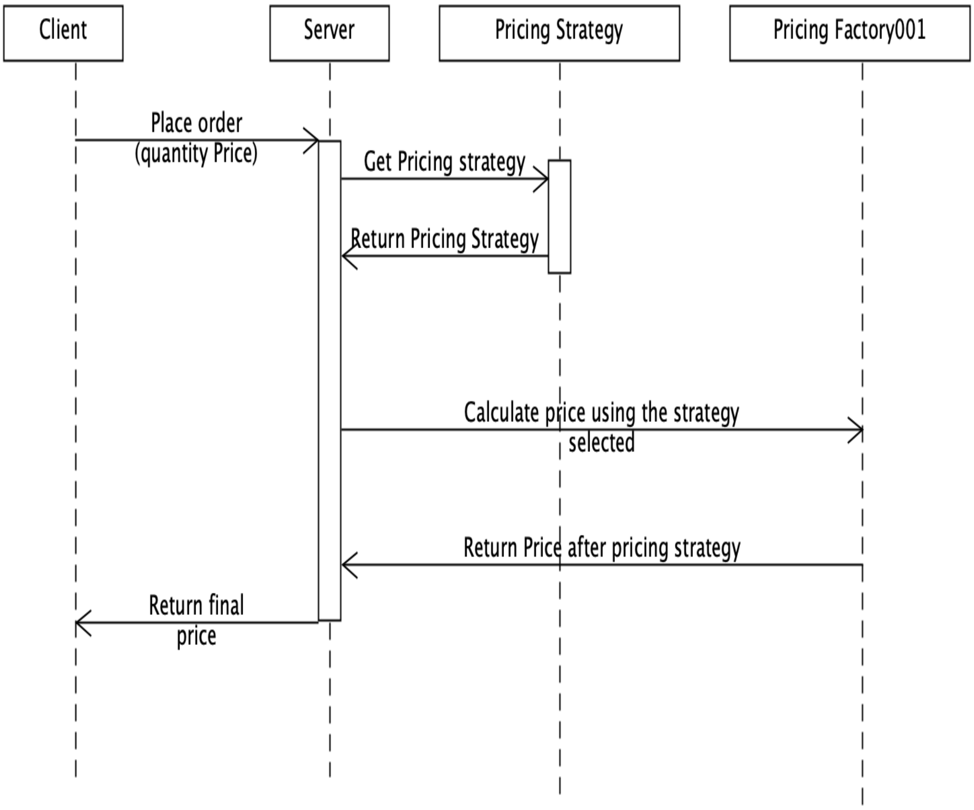
***Use Case 3***


User
Client
Server
User selects product from client drop down menu
User selects quantity from client drop down menu
Place order
Order is sent to server
Display order details and time stamp

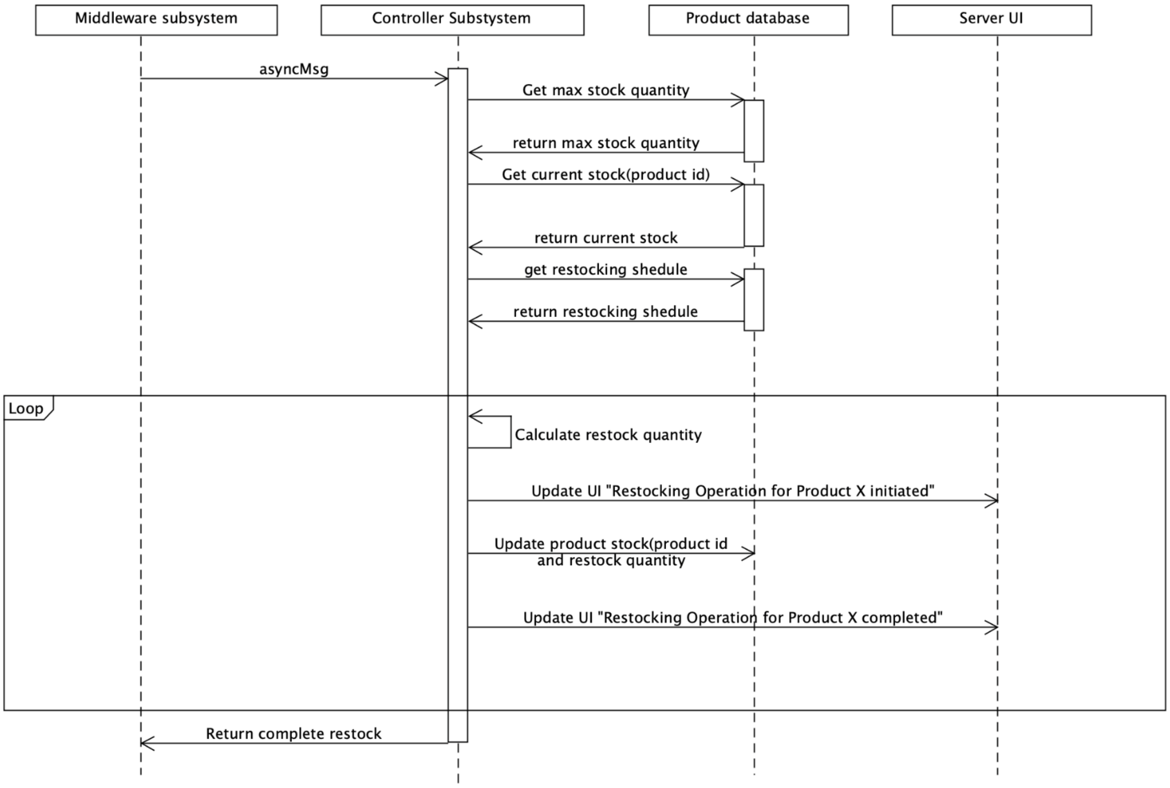
***\Use Case 4***


Client
Server
Controller Subsystem
Product Database
Place order (product,quantity,timestamp)
alt
[Order Quality > Availible Quantity (Needs restock)]
check availibe product quantity
Availible quantity
"Order for product X Quantity Y is pending" Order exceeds availible quantity
Restocking operation is performed
Update product quantity
Quantity Updated
Order is finalized for Product X and Quantity Y with total price Z
[order quqantity > max stock quantity]
Order for Product X Quantity Y is pending order exceeds available quantity
[order quqantity <= Availile quantity]
Update product quantity
Order is finalized for Product X and Quantity Y with total price Z
Quantity Updated

### *Use Case 5*

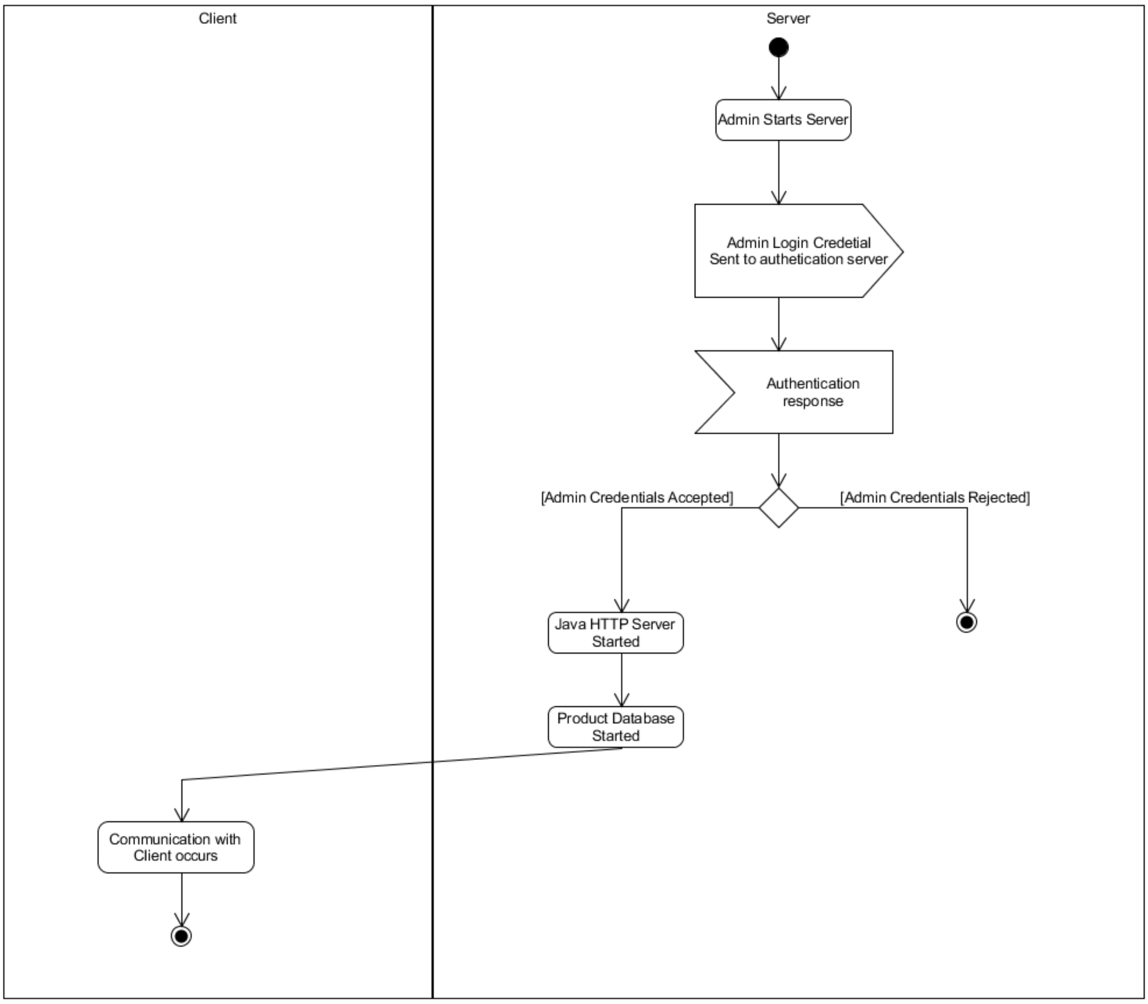

Client
Server
Pricing Strategy
Pricing Factory001
Place order (quantity Price)
Get Pricing strategy
Return Pricing Strategy
Calculate price using the strategy selected
Return Price after pricing strategy
Return final price

## *Use Case 6*

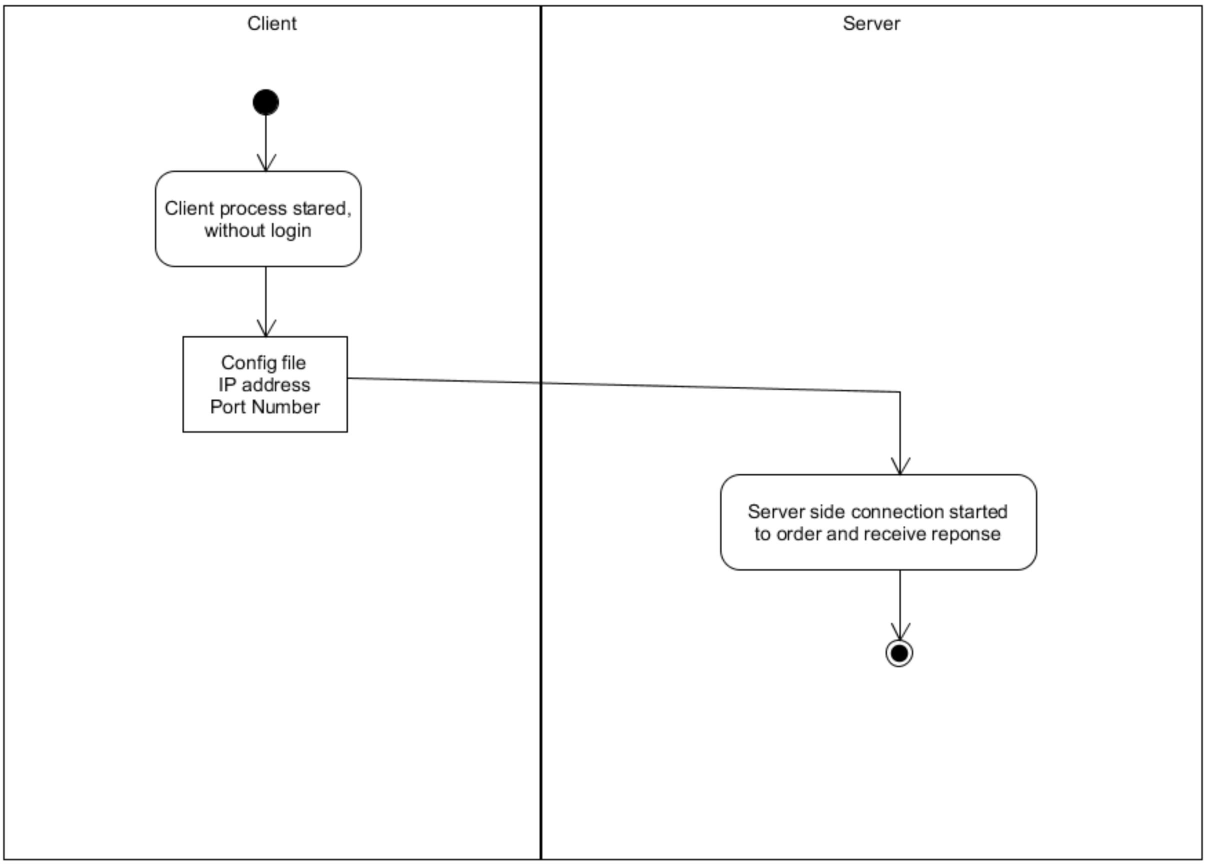

Middleware subsystem
Controller Substystem
Product database
Server UI
asyncMsg
Get max stock quantity
return max stock quantity
Get current stock(product id)
return current stock
get restocking shedule
return restocking shedule
Loop
Calculate restock quantity
Update UI "Restocking Operation for Product X initiated"
Update product stock(product id
and restock quantity
Update UI "Restocking Operation for Product X completed"
Return complete restock

# 3 Activity Diagrams

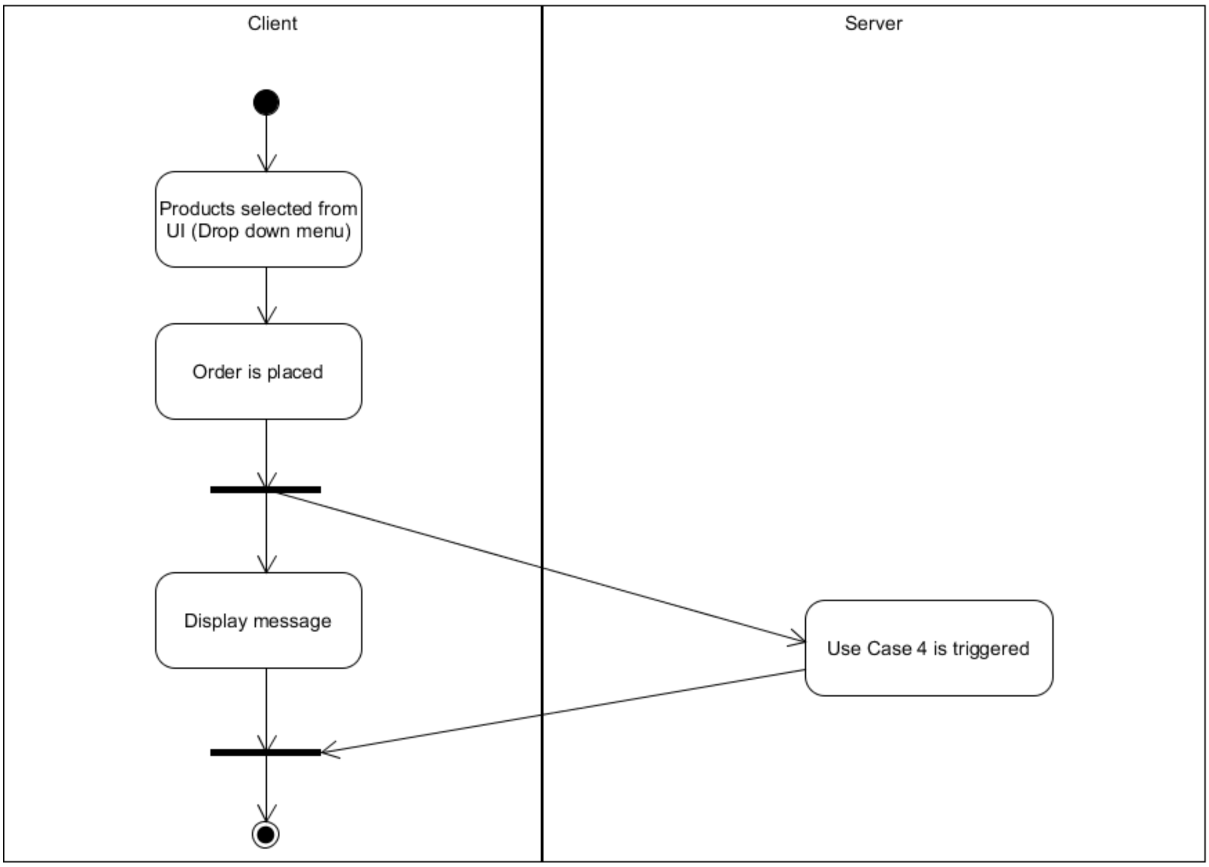
USE CASE 1

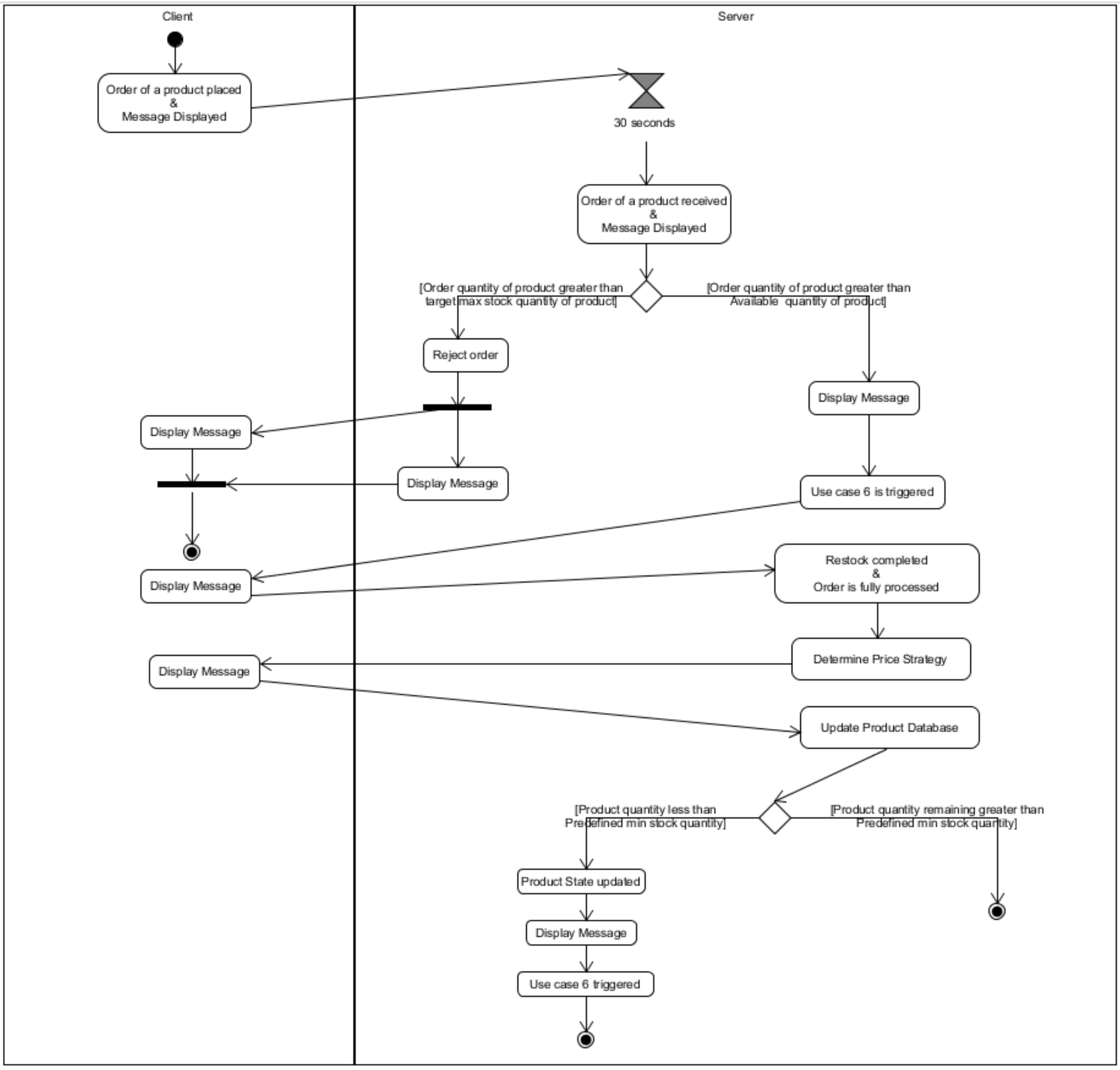

Client
Server
Admin Starts Server
Admin Login Credetial Sent to authetication server
Authentication response
[Admin Credentials Accepted]
[Admin Credentials Rejected]
Java HTTP Server Started
Product Database Started
Communication with Client occurs

# USE CASE 2


Client
Server
Client process stared, without login
Config file
IP address
Port Number
Server side connection started to order and receive reponse

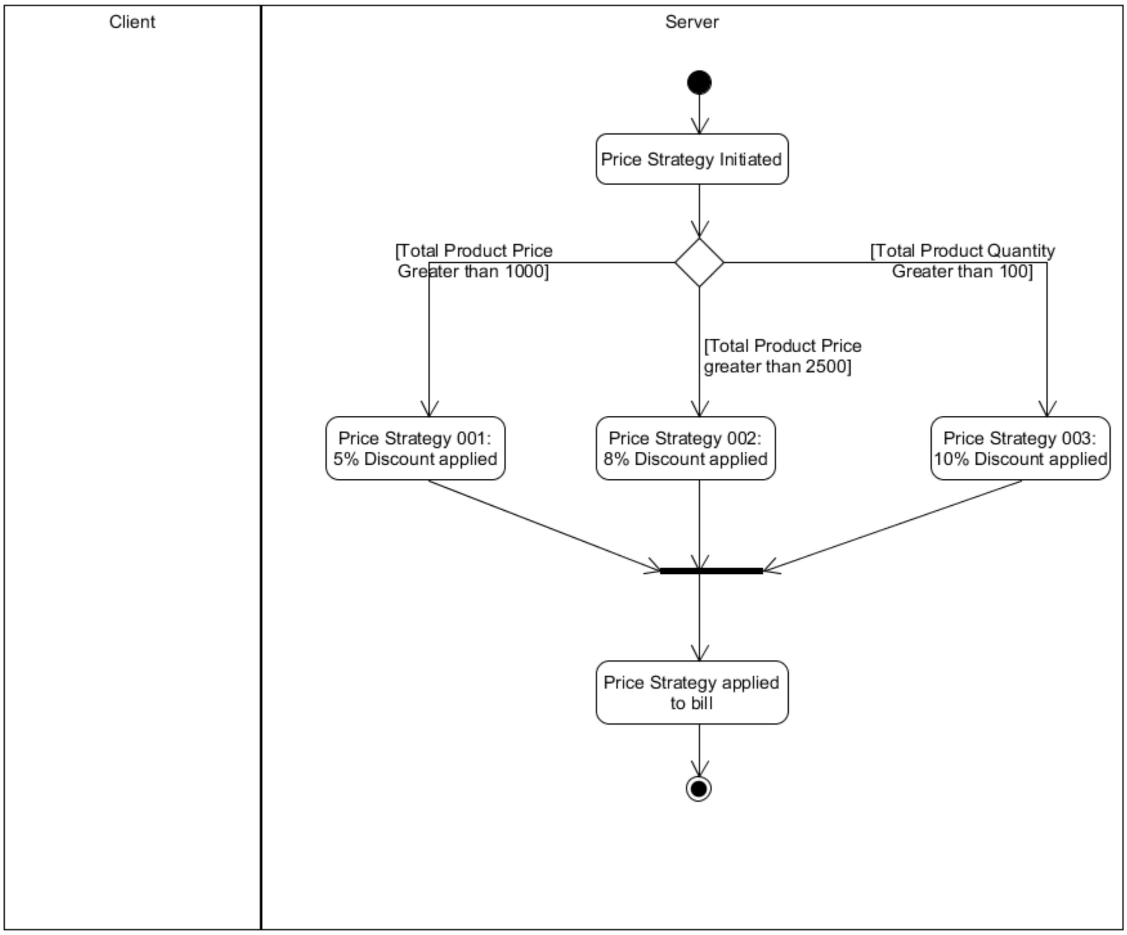
# USE CASE 3


Client
Server
Products selected from UI (Drop down menu)
Order is placed
Display message
Use Case 4 is triggered

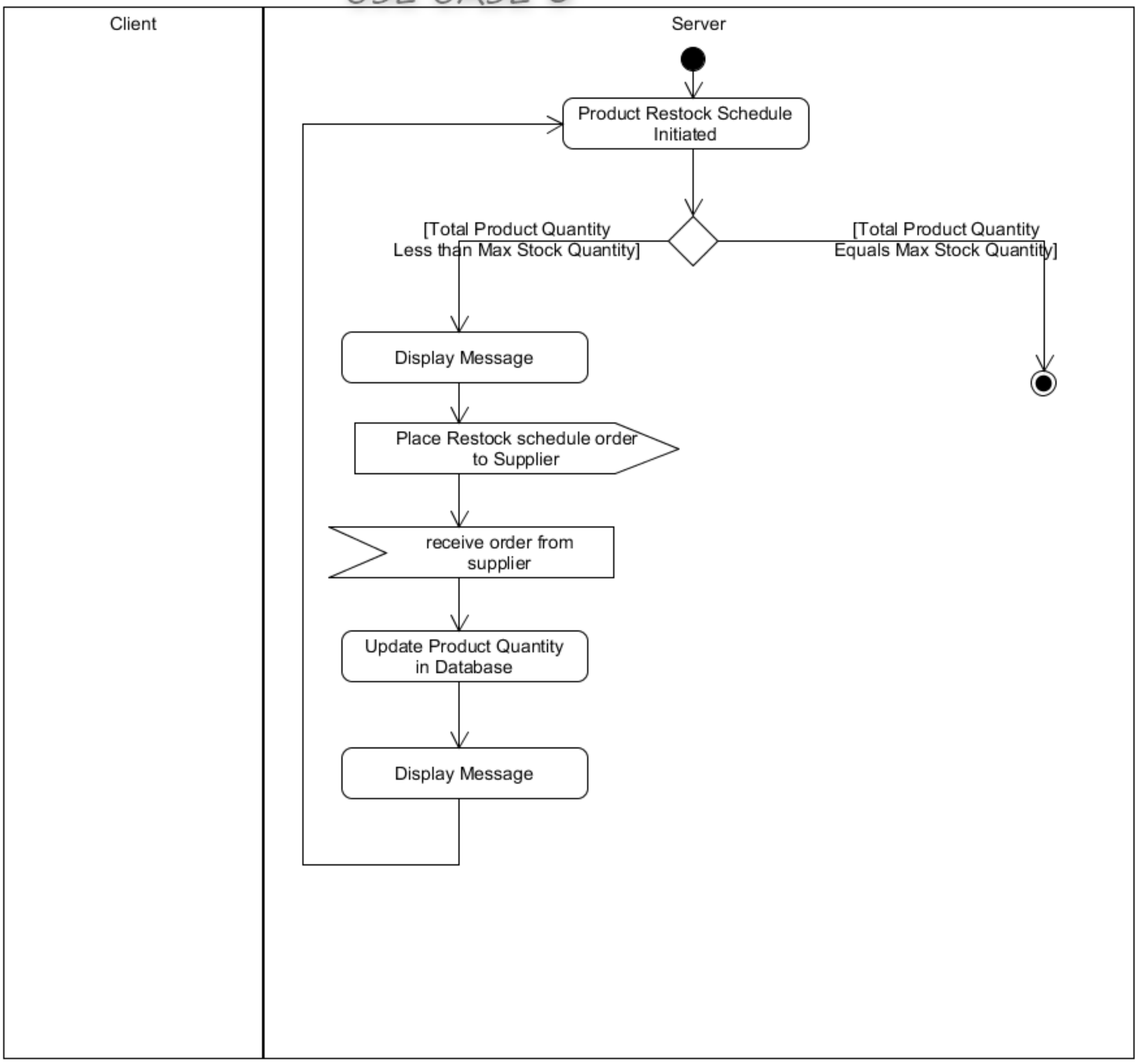

Client
Server
Order of a product placed & Message Displayed
30 seconds
Order of a product received & Message Displayed
[Order quantity of product greater than target max stock quantity of product]
[Order quantity of product greater than Available quantity of product]
Reject order
Display Message
Display Message
Display Message
Use case 6 is triggered
Display Message
Restock completed & Order is fully processed
Determine Price Strategy
Display Message
Update Product Database
[Product quantity less than Predefined min stock quantity]
[Product quantity remaining greater than Predefined min stock quantity]
Product State updated
Display Message
Use case 6 triggered

USE CASE 4

# USE CASE 5


Client
Server
Price Strategy Initiated
[Total Product Price Greater than 1000]
[Total Product Quantity Greater than 100]
[Total Product Price greater than 2500]
Price Strategy 001: 5% Discount applied
Price Strategy 002: 8% Discount applied
Price Strategy 003: 10% Discount applied
Price Strategy applied to bill

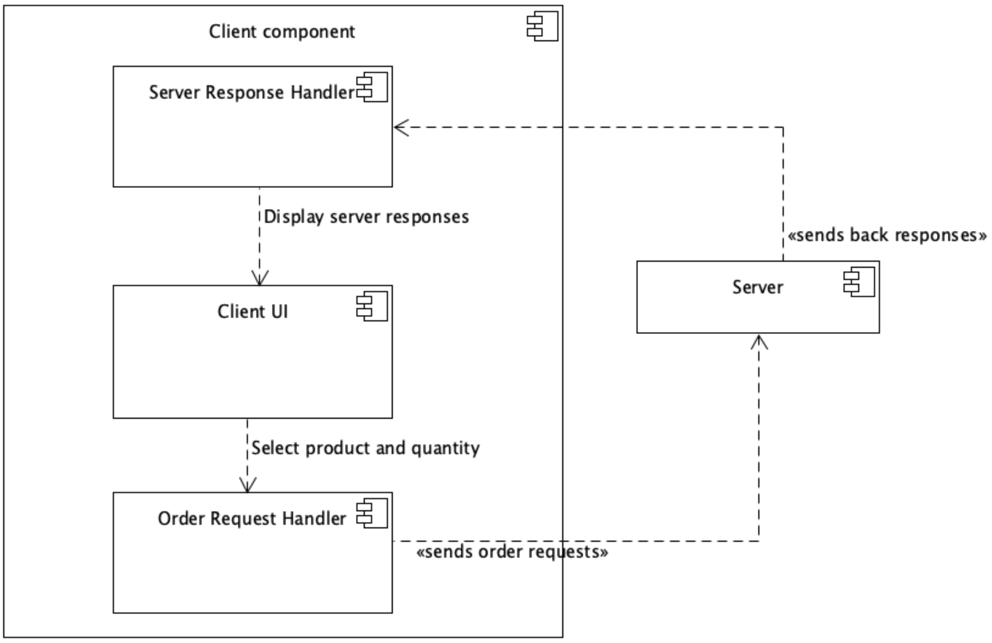
USE CASE 6


Client
Server
Product Restock Schedule Initiated
[Total Product Quantity Less than Max Stock Quantity]
[Total Product Quantity Equals Max Stock Quantity]
Display Message
Place Restock schedule order to Supplier
receive order from supplier
Update Product Quantity in Database
Display Message

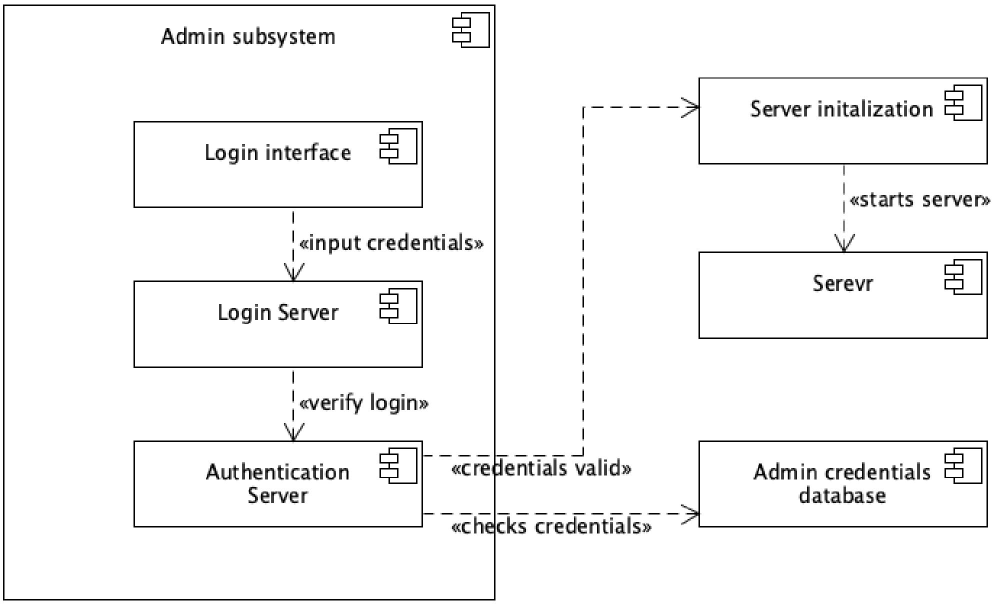
# 4 Architecture

## 4.1 component diagrams -Peter Ayade

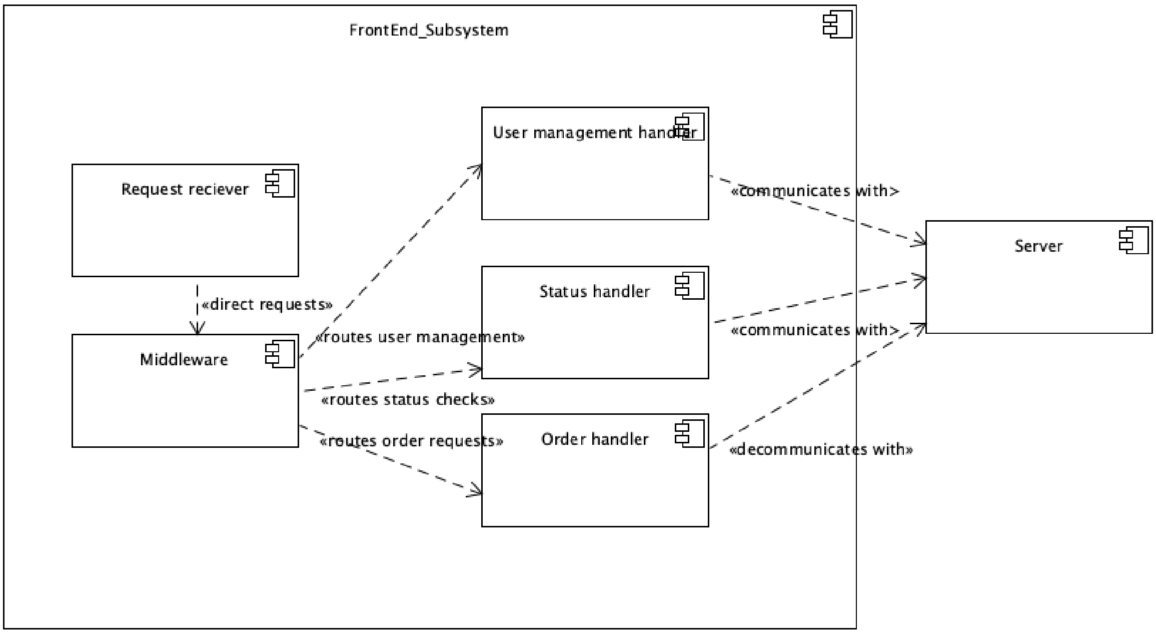
### client



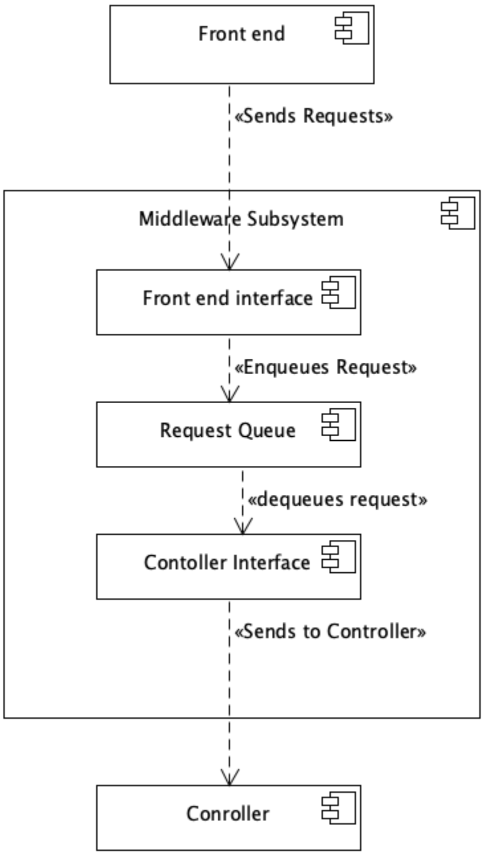
### admin



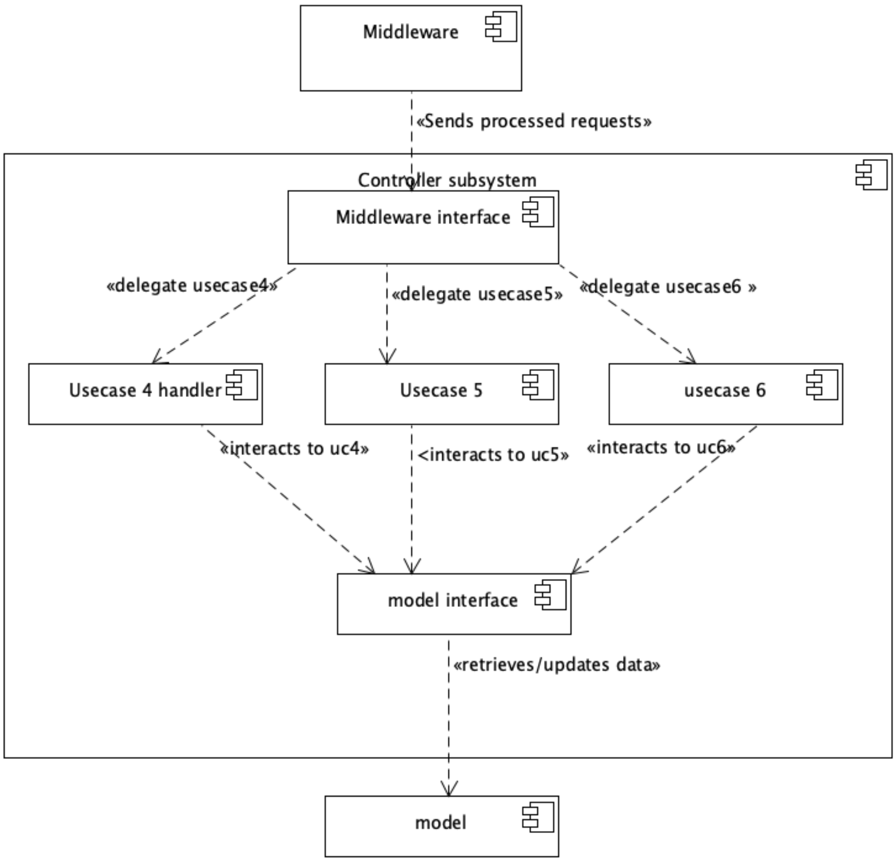
## frontend subsystem



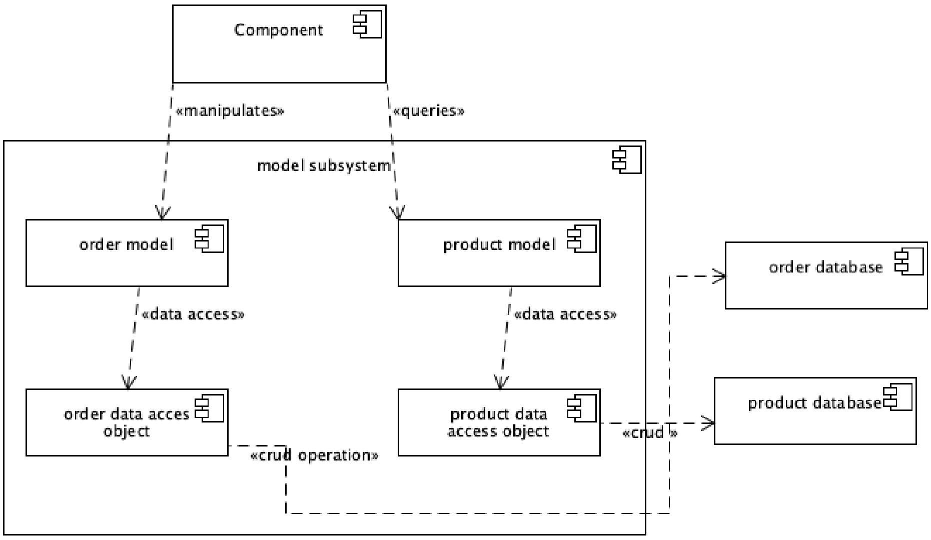
## middleware subsystem



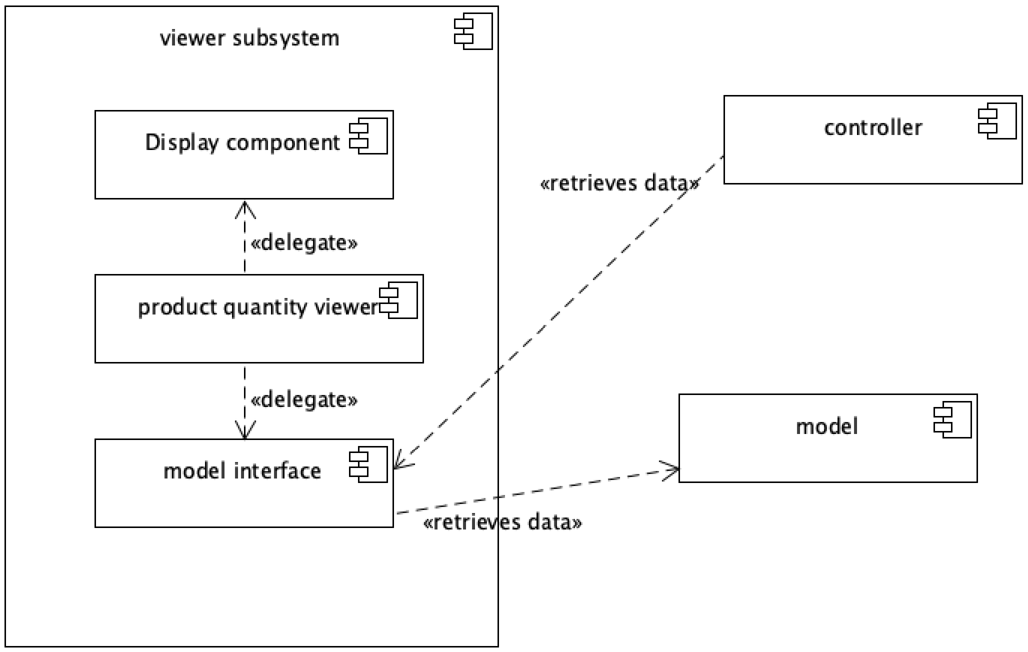
## controller subsystem



## model subsystem



## Viewer subsystem



## 4.2 Interface Definitions - Hashaam Bajwa

| Modules | | | |
|---|---|---|---|
| **Module Name** | **Description** | **Exposed Interface Names** | **Interface Description** |
| Admin Subsystem | Manages administrator authentication and system access. | Admin : Admin Authentication<br>Admin: | - This interface is responsible for handling the authentication process, ensuring that |

| | | | |
|---|---|---|---|
| | | AdminServerManagement | only authorized administrators can access the functionality of the Admin subsystem.<br>● This interface is responsible for managing server-related actions within the subsystem, ensuring that the initialization of the server is done under the supervision of the authenticated admins. |
| Front_End Subsystem | Receives incoming requests and defines handlers. | Front_End Subsystem : Front-End interface | ● This interface defines methods that handle incoming requests and manage the flow of communication. Based on certain request types (HTTP methods), the interface will determine the appropriate handler or action. |
| Middleware Subsystem | Isolates the Front_End and the Controller | Middleware : Communication Interface | ● This interface defines methods responsible for processing requests and managing the communication flow between the Front_End and Controller Subsystems within the Middleware Subsystem. |
| Controller Subsystem | Implements the business logic of the system. | Controller : Request Processing Interface<br>Controller: Business Logic Interface<br>Controller: Data Management Interface | ● This interface defines methods for processing requests within the Controller Subsystem<br>● This interface defines methods for implementing some of the business logic for the warehouse system. This includes restocking and |

| | | | |
|---|---|---|---|
| | | | fulfilling order functions which are two of the main functions.<br>● This interface is responsible for ensuring that the data is being updated as requests are being processed so that the application is up to date. |
| Model Subsystem | Represents runtime data using the MVC pattern | Model: Data Management Interface<br>Model: Observer Interface | ● This interface is responsible for data retrieval and updation.<br>● This interface will notify observers when there is an update to the data. |
| Viewer Subsystem | Displays the current warehouse product quantities. | Viewer: Display Interface<br>Viewer: Observer Interface | ● This interface holds a set of methods that are responsible for rendering warehouse information for the user interface.<br>● This interface holds a set of methods that will observe changes in the product quantities and perhaps alert messages such as restocking/quantity low messages will be rendered in the UI. |
| Product Database | Stores information about products | Product Database: Database Query Interface<br>Product Database: Update Interface | ● This interface holds a set of functions that are responsible for querying information from the product database. Information such as quantity, price, description are all retrieved through the set of functionality kept in this interface.<br>● This interface is responsible for updating information in the product database. |

| | | | |
|---|---|---|---|
| Admin Credentials Database | Stores valid username and passwords | AdminCredentialsDatabase: Verification Interface | ● This interface defines methods that are responsible for verifying admin creds. There will be a separate database accessed to retrieve admin credentials information. |

| Interfaces | | |
|---|---|---|
| **Interface Name** | **Operations** | **Operation Description** |
| Admin : Admin Authentication | Boolean authenticate(String username, String password)<br>Used by the Admin subsystem associated with logging in and starting the server, as specified in UC1. | ● This operation takes in a username and password and further processes it to ensure validation. |
| Admin : AdminServerManagement | Void startServer()<br>Used by the Admin subsystem and invoked via a component in the subsystem that is responsible for starting the server. | ● This operation is responsible for initiating the server component after successful authentication. This method is typically called when an admin with valid credentials initiates the server, allowing the system to then go on to process requests and orders. It is there for security to ensure valid credentials are being used. |
| Front_End: Front-End Interface | Void handleRequest(Request request)<br>Used by client process and middleware subsystem | ● This operation is responsible for the initial processing of an incoming HTTP request from the user. It takes a |

| | | |
|---|---|---|
| | | request and determines the appropriate request type and extracts the actual request in json format. |
| Middleware : Middleware Communication Interface | Void routeRequest(Request request)<br>Used by the Front_End subsystem | ● This operation is responsible for managing comm between front end and controller subsystems. It takes a request after initial processing and determines the appropriate handler. |
| Controller : Request Processing | Void processOrder(Order order):<br>Used by middleware subsystem<br>Void calculatePrice(Order order):<br>Used by middleware subsystem | ● Manages the processing of orders received by the middleware subsystem.<br>● Calculates the price of an order based on pricing strategies. |
| Controller: Business logic | Void fulfillOrder(Order order)<br>Void initiateRestocking(Product product)<br>Both used by the middleware subsystem | ● Handles the fulfillment of an order, updating the product database and triggering restocking if necessary<br>● This method initiates the restocking process for a specific product. |
| Controller : Data Management Interface | Void updateData(Data data):<br>Used by the middleware subsystem | ● Updates the real time data such as the orders, products and quantities. |
| Model: Model Data Management Interface | Void updateData(Data data)<br>Data retrieveData():<br>Both used by the controller | ● Updates the runtime data such as orders, products, and quantities.<br>● Retrieves the current state of the runtime |

| | | |
|---|---|---|
| | subsystem. | data from the model subsystem. |
| Viewer : Viewer Display Interface | Void displayQuantities (Quantities quantities):<br>Used by the Front_End subsystem<br>Void displayRestockingMessage( String message):<br>Used by the front_end subsystem | ● Displays the current product quantities in the warehouse UI.<br>● Displays messages related to restocking operations in the warehouse. For example, if items are low in quantity. |
| Viewer : Viewer Observer Interface | Void notifyQuantitiesUpdate(Quantities quantities):<br>Used by the controller subsystem<br>Void notifyRestockingMessage(String message)<br>Used by the controller subsystem | ● This operation notifies the viewer subsystem about changes in product quantities.<br>● This operation notifies the viewer subsystem about changes in restocking messages. |
| Product Database : Database Query Interface | ProductDetails queryProductDetails (String productID)<br>Used by the controller subsystem | ● This operation is used to retrieve details about a specified product (via product ID) and return relevant product details for the user to see. |
| Product Database: Database Update Interface | Void updateProductStock(String productID, int newStock):<br>Used by the controller subsystem | ● This operation is responsible for updating the stock quantity of a specific product after an order has been processed and there has been a change in stock. |
| Admin Credentials Database: Credentials Verification | Boolean verifyCredentials (String username, String password):<br>Void addCredentials(String username, String | ● This operation is responsible for ensuring entered admin credentials are present in the admin database |

| | | |
|---|---|---|
| | password):<br>Used by the Admin subsystem | ● This operation is responsible for adding credentials to the admin database. |

## 4.3 Discussion on the use of design patterns - Vaidik Vekaria

### Proxy Design Pattern

Use of Proxy design pattern, `LoginServer` can use the `AuthenticationServerProxy` to authenticate admin credentials, and the proxy ensures additional security checks before allowing access to the real authentication server. Please find detailed class diagrams and example in section 4.4

### Factory Design Pattern

The Factory Design Pattern can be effectively employed in a system with a factory repository, HashMap, and factory product classes to dynamically choose products at runtime. The key components include creating a set of product classes, each representing a specific product type and implementing a common interface. A factory repository manages the creation logic, allowing for the registration of product types and dynamically creating instances based on runtime product type identifiers. A HashMap facilitates the mapping between product type identifiers and their corresponding factory classes, offering a flexible way to choose products dynamically.

This approach enhances extensibility and encapsulation, enabling the system to easily adapt to new product types without modifying existing client code. The centralized creation logic in the factory repository streamlines maintenance and updates, making it a robust design pattern for managing product instantiation in a flexible and maintainable manner. Please example in section 4.4

The Factory Method Design Pattern is a valuable approach when implementing pricing strategies. We can create a family of classes, each representing a specific pricing strategy, and define a common interface or abstract base class. The Factory Method allows subclasses to alter the instantiation of these pricing strategies, providing a flexible mechanism for choosing and creating them at runtime.

To implement this, create an interface or abstract base class representing the pricing strategy. Subclasses will encapsulate the specific pricing logic for each strategy, adhering to the common interface. The Factory Method, declared in the interface or abstract class, is responsible for creating instances of the pricing strategy subclasses. This way, at runtime, the appropriate pricing strategy can be selected by invoking the Factory Method without exposing the details of the concrete classes to the client code.

This design fosters extensibility, allowing you to introduce new pricing strategies by extending the abstract class and implementing the Factory Method. The client code remains agnostic to the specific pricing strategy classes, promoting encapsulation and making it easier to incorporate and manage new pricing strategies as your system evolves.

### Singleton Design Pattern

The Singleton Design Pattern is valuable for ensuring that a class has only one instance and providing a global point of access to that instance. In the context of our Warehouse Management System project, you could apply the Singleton pattern to critical classes like the Factory Repository. This ensures that there is a single repository managing product factories, preventing unnecessary duplication and maintaining a consistent state across the system. The Singleton pattern's advantage lies in its ability to control access to a shared instance, reducing resource consumption and promoting a centralized point for managing global state, fostering efficiency, and preventing unintended conflicts in a multi-objective environment.

## Strategy Design Pattern

In the Warehouse Management System project, the Strategy Design Pattern can be applied to pricing strategies by encapsulating each pricing algorithm in separate strategy classes, such as PricingStrategy001 and PricingStrategy002. These classes implement a common interface, enabling interchangeable use within the system. By dynamically selecting a pricing strategy at runtime, the Strategy pattern allows for flexible adaptation to different business requirements or changes in pricing logic. This approach promotes an open-closed principle, facilitating the addition of new pricing strategies without modifying existing code. The Strategy Design Pattern enhances maintainability and scalability in the pricing module, providing a modular and easily extensible solution to accommodate diverse pricing scenarios. Please find class diagram in section 4.4

## State Design Pattern

In the Warehouse Management System, implementing the State Design Pattern involves creating distinct state classes that encapsulate the behavior associated with different product states. For example, the "low stock" state could be represented by a `LowStockState` class, and the "restocking to fulfill order" state could be managed by a `RestockingState` class. The `Product` class would then maintain a reference to its current state, and operations related to the product would delegate to the corresponding state class. This approach decouples the state-specific behavior from the `Product` class, promoting a modular and easily extendable system. Adding a new product state or modifying the behavior of an existing state becomes a streamlined process without requiring changes to the core product class. The State Design Pattern fosters code organization, flexibility, and maintainability, providing a robust foundation for handling diverse product states in the dynamic environment of a warehouse management system.

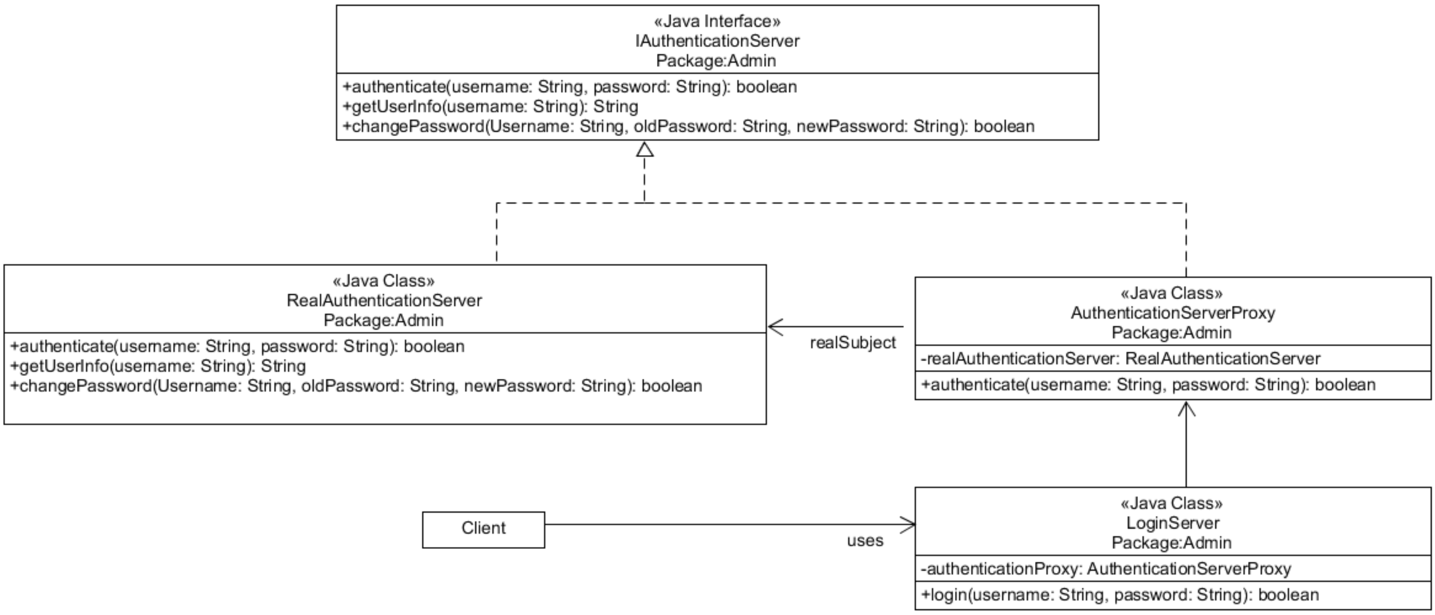
## Observer Design Pattern

The Observer Design Pattern can be effectively applied to facilitate communication and updates between components, ensuring that changes in one part of the system trigger responses in other related components. For instance, you can implement an `OrderObserver` interface with methods like `updateOrderStatus` and have classes representing various system components, such as the server and clients, implement this interface. When an order's status changes (e.g., from "pending" to "fulfilled" or "restocking initiated"), the relevant components can notify their observers (clients) about the change. Clients, acting as observers, can then update their user interfaces or perform additional actions based on the order status. This pattern enhances the system's modularity and extensibility, allowing for flexible communication between different modules while minimizing dependencies. It provides a scalable solution for handling real-time updates and maintaining consistency across the various components of our Warehouse Management System.

## 4.4 Class diagrams for three cases of use of three different design patterns - Vaidik Vekaria

### Proxy Design Pattern

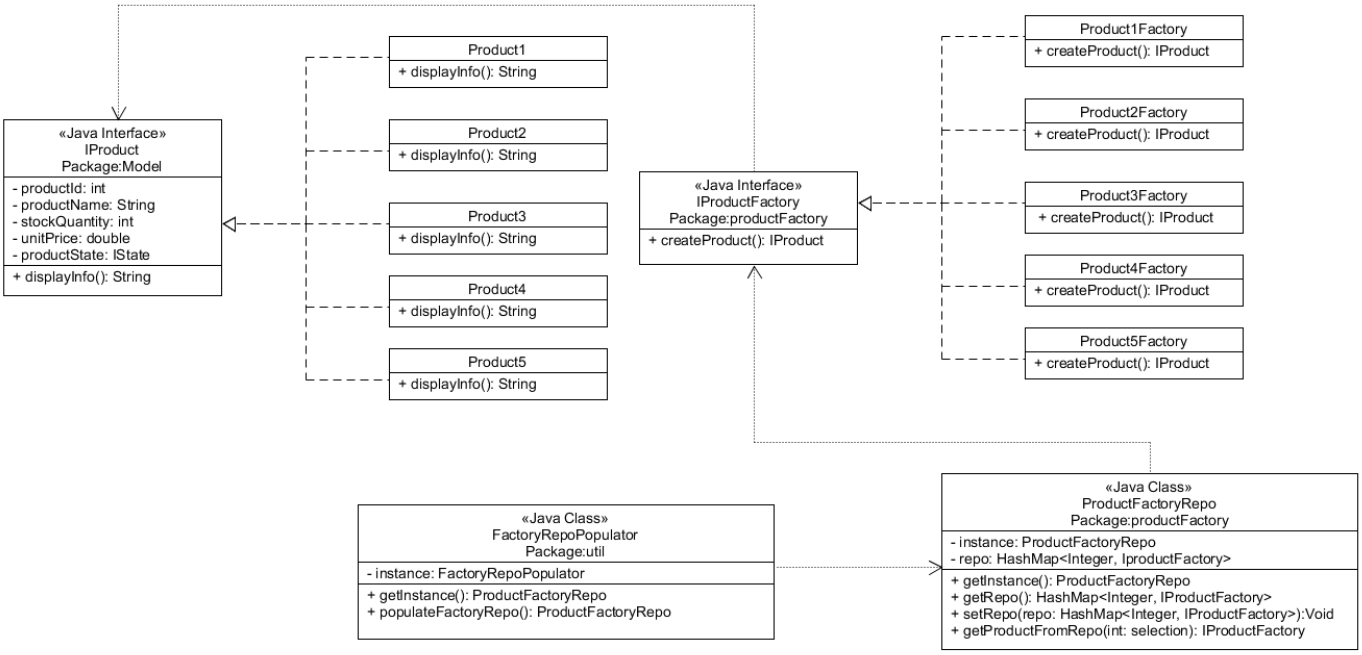
In this class diagram we are implementing the Proxy design pattern for authentication of admin credentials between the login server and authentication server in our Warehouse Management System project.


«Java Interface»
IAuthenticationServer
Package:Admin
+authenticate(username: String, password: String): boolean
+getUserInfo(username: String): String
+changePassword(Username: String, oldPassword: String, newPassword: String): boolean
«Java Class»
RealAuthenticationServer
Package:Admin
+authenticate(username: String, password: String): boolean
+getUserInfo(username: String): String
+changePassword(Username: String, oldPassword: String, newPassword: String): boolean
realSubject
«Java Class»
AuthenticationServerProxy
Package:Admin
-realAuthenticationServer: RealAuthenticationServer
+authenticate(username: String, password: String): boolean
Client
uses
«Java Class»
LoginServer
Package:Admin
-authenticationProxy: AuthenticationServerProxy
+login(username: String, password: String): boolean

- `AuthenticationServer` is the subject interface with a method for authenticating admin credentials.
- `RealAuthenticationServer` is the real subject class implementing the actual authentication logic.
- AuthenticationServerProxy is the proxy class that controls access to the `RealAuthenticationServer` and may perform additional tasks like logging, security checks, etc.
- `LoginServer` is a class that uses the Proxy. It holds an instance of `AuthenticationServerProxy` to authenticate admin credentials.

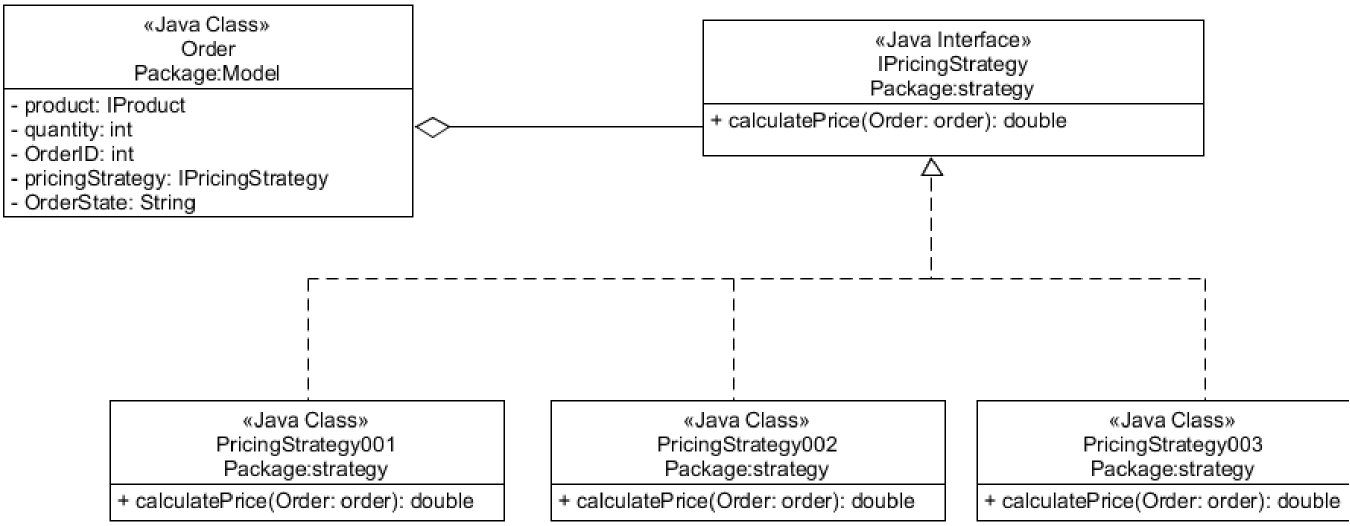
### Factory Design Pattern with Singleton Design pattern used in the Repo.

We'll introduce a `Product` interface, concrete product classes for items in our warehouse, and factories to create these products dynamically.



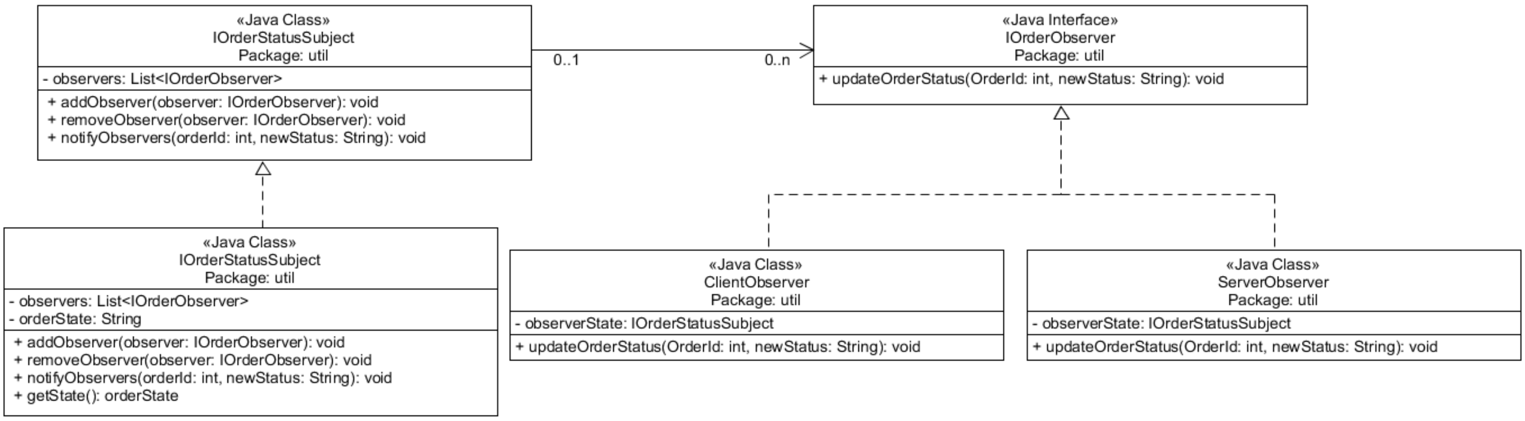
In this adapted example, the `Product` interface represents the common attributes and behaviors of items in our warehouse. The concrete product classes, such as `Product1` and `Product`2, implement this interface. The factory pattern enables dynamic product creation based on product type, facilitating flexible handling of various items within our Warehouse Management System.

## Strategy Design Pattern



In this example, the `WarehouseProduct` class has a `PricingStrategy` field, allowing dynamic switching between different pricing strategies. Each pricing strategy class implements the `PricingStrategy` interface, providing a `calculatePrice` method to determine the product's price based on the chosen strategy. The Strategy Design Pattern allows for flexibility in handling various pricing scenarios and supports the addition of new pricing strategies without modifying existing code.

## Observer Design Pattern

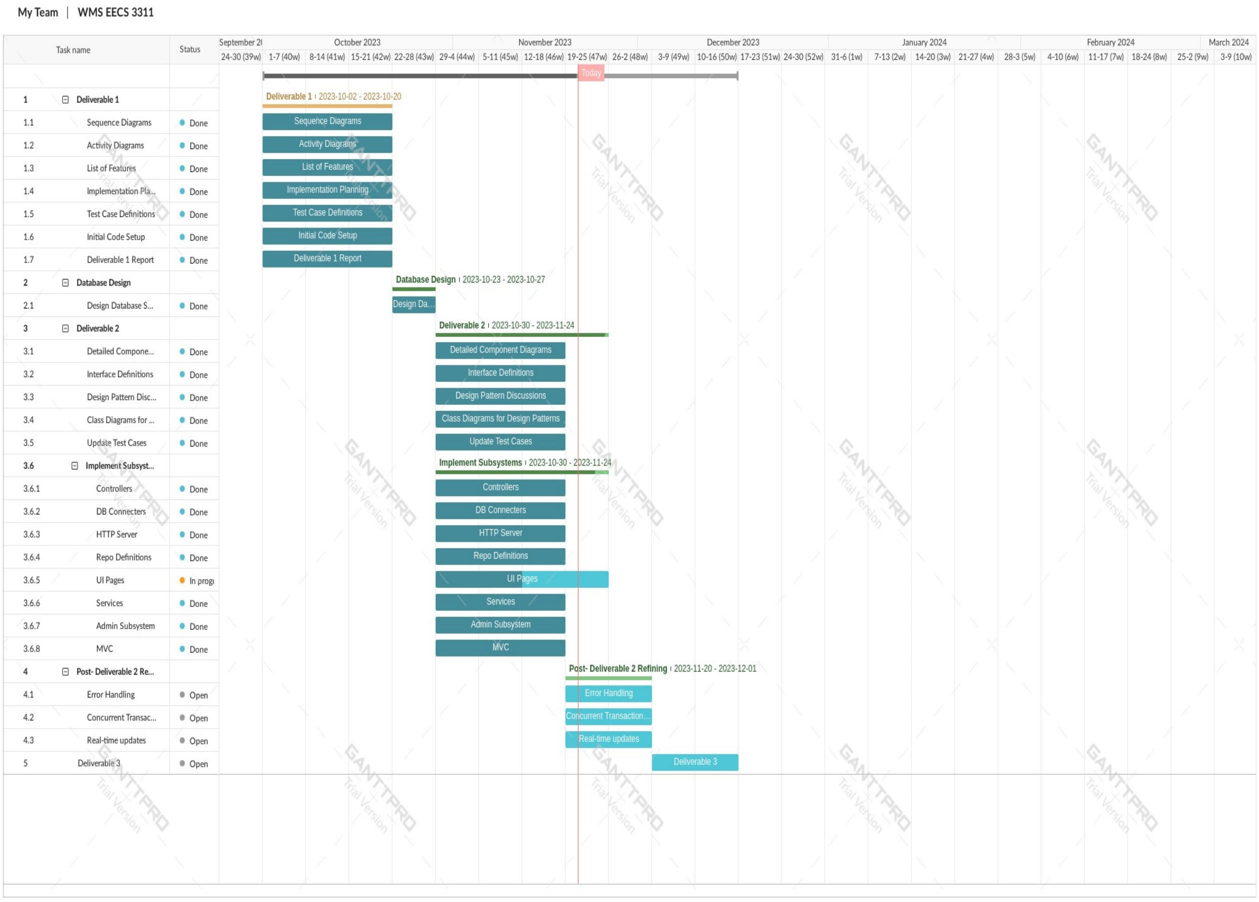

«Java Class»
IOrderStatusSubject
Package: util
- observers: List<IOrderObserver>
+ addObserver(observer: IOrderObserver): void
+ removeObserver(observer: IOrderObserver): void
+ notifyObservers(orderId: int, newStatus: String): void
0..1
0..n
«Java Interface»
IOrderObserver
Package: util
+ updateOrderStatus(OrderId: int, newStatus: String): void
«Java Class»
IOrderStatusSubject
Package: util
- observers: List<IOrderObserver>
- orderState: String
+ addObserver(observer: IOrderObserver): void
+ removeObserver(observer: IOrderObserver): void
+ notifyObservers(orderId: int, newStatus: String): void
+ getState(): orderState
«Java Class»
ClientObserver
Package: util
- observerState: IOrderStatusSubject
+ updateOrderStatus(OrderId: int, newStatus: String): void
«Java Class»
ServerObserver
Package: util
- observerState: IOrderStatusSubject
+ updateOrderStatus(OrderId: int, newStatus: String): void

In this example, the `OrderStatusSubject` class acts as the subject that maintains a list of observers (`OrderObserver`). Both the server and client classes implement the `OrderObserver` interface and can register with the `OrderStatusSubject` to receive updates. When an order's status changes, the subject notifies all registered observers, triggering the `updateOrderStatus` method in each observer. This pattern allows for efficient communication and real-time updates across different components in your Warehouse Management System.

# 5 Activities Plan

## 5.1 Project Planning

*A clearer view is attached with the report submission in the form of a Gantt chart pdf.*



My Team | WMS EECS 3311

| | Task name | Status |
|---|---|---|
| 1 | ⊟ Deliverable 1 | |
| 1.1 | Sequence Diagrams | ● Done |
| 1.2 | Activity Diagrams | ● Done |
| 1.3 | List of Features | ● Done |
| 1.4 | Implementation Pla... | ● Done |
| 1.5 | Test Case Definitions | ● Done |
| 1.6 | Initial Code Setup | ● Done |
| 1.7 | Deliverable 1 Report | ● Done |
| 2 | ⊟ Database Design | |
| 2.1 | Design Database S... | ● Done |
| 3 | ⊟ Deliverable 2 | |
| 3.1 | Detailed Compone... | ● Done |
| 3.2 | Interface Definitions | ● Done |
| 3.3 | Design Pattern Disc... | ● Done |
| 3.4 | Class Diagrams for ... | ● Done |
| 3.5 | Update Test Cases | ● Done |
| 3.6 | ⊟ Implement Subsyst... | |
| 3.6.1 | Controllers | ● Done |
| 3.6.2 | DB Connecters | ● Done |
| 3.6.3 | HTTP Server | ● Done |
| 3.6.4 | Repo Definitions | ● Done |
| 3.6.5 | UI Pages | ● In progr |
| 3.6.6 | Services | ● Done |
| 3.6.7 | Admin Subsystem | ● Done |
| 3.6.8 | MVC | ● Done |
| 4 | ⊟ Post- Deliverable 2 Re... | |
| 4.1 | Error Handling | ● Open |
| 4.2 | Concurrent Transac... | ● Open |
| 4.3 | Real-time updates | ● Open |
| 5 | Deliverable 3 | ● Open |

## 5.2 Project Backlog and Sprint Backlog

Product Backlog:

- Error Handling:

- Implement robust error handling mechanisms to gracefully manage and report errors during order placement, server processing, and other system activities.
- Concurrency and Transaction Management:
  - Address potential issues related to concurrent order processing.
  - Implement transaction management to ensure data consistency during order fulfillment and restocking operations.
- Real-time Updates:
  - Decide whether the system should provide real-time updates to clients or operate in batch mode.
  - Implement features for real-time notifications and updates.
- Testing:
  - Develop a robust testing strategy, including unit tests, integration tests, and end-to-end tests.
  - Implement automated testing wherever possible
  - Unit Testing
    - Write unit tests for each module and functionality
    - Ensure that each module passes its respective test cases
  - Integration Testing
    - Test the integration of different system components
    - Ensure that the entire system functions as expected
- Finalize Code/ Documentation/ Project Review

## 5.3 Group Meeting Logs

| **Present Group Members** | **Meeting Date** | **Issues Discussed / Resolved** |
|---|---|---|
| All Members Present | October 10th | -Each Member was allocated a specific section to work on<br>-Each section was clarified to ensure every group member adequately understood the requirements.<br>-Team also set an internal deadline to meet before the due date. |
| All Members Present | Nov 13th | -There was discussion on what is expected of us for deliverable 2<br>-After discussion, portions were allocated to each member to work on.<br>-Internal deadlines were set before the due date so work could be reviewed. |

# 6 Test Driven Development

*Note: Base Initial condition for all test cases is the server is running properly and accessible through HTTP requests. Initial condition being none means the only initial condition that needs to be satisfied is the base initial condition.*

| | |
|---|---|
| TestID | 1 |
| Category | Evaluation of successful administrator registration |
| Requirements Coverage | UC1-Successful-Admin-Registration |
| Initial Condition | None |
| Procedure | 1. The user selects sign up.<br>2. The user provides all the details required by the sign up form.<br>3. The user clicks the register button. |
| Expected Outcome | Registration is successful and the administrator is redirected to the login page. |
| Notes | The password in form must be at least 8 characters. There can only be one registered administrator. If an administrator already exists, admin registration will fail. |

| | |
|---|---|
| TestID | 2 |
| Category | Evaluation of unsuccessful administrator registration |
| Requirements Coverage | UC2-Unsuccessful-Admin-Registration-Admin-Exists |
| Initial Condition | Admin already exists in the database. |
| Procedure | 1. The user selects sign up.<br>2. The user provides all the details required by the sign up form.<br>3. The user clicks the register button. |
| Expected Outcome | Registration fails and a failed to register alert is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 3 |
| Category | Evaluation of unsuccessful administrator registration |

| | |
|---|---|
| Requirements Coverage | UC3-Unsuccessful-Admin-Registration-Invalid-Password |
| Initial Condition | None |
| Procedure | 1. The user selects sign up.<br>2. The user provides all the details required by the sign up form.<br>3. The user clicks the register button. |
| Expected Outcome | Registration fails and a failed to register alert is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 4 |
| Category | Evaluation of unsuccessful administrator registration |
| Requirements Coverage | UC4-Unsuccessful-Admin-Registration-Missing-Form-Info |
| Initial Condition | None |
| Procedure | 1. The user selects sign up.<br>2. The user provides some/none of the details required by the sign up form.<br>3. The user clicks the register button. |
| Expected Outcome | Registration fails and a failed to register alert is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 5 |
| Category | Evaluation of successful administrator login |
| Requirements Coverage | UC5-Successful-Admin-Login |
| Initial Condition | Admin exists in the database. |
| Procedure | 1. The user selects login.<br>2. The user provides a valid username and a valid password.<br>3. The user clicks the login button |
| Expected Outcome | Login is successful and the user is redirected to the home admin page. |
| Notes | None |

| | |
|---|---|
| TestID | 6 |
| Category | Evaluation of unsuccessful administrator login |
| Requirements Coverage | UC6-Unsuccessful-Admin-Login-Admin-Not-Exists |
| Initial Condition | Admin does not exist in the database. |
| Procedure | 1. The user selects login.<br>2. The user provides a username and a password.<br>3. The user clicks on the login button. |
| Expected Outcome | Login is unsuccessful and a failed to login alert is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 7 |
| Category | Evaluation of unsuccessful administrator login |
| Requirements Coverage | UC7-Unsuccessful-Admin-Login-Invalid-Username |
| Initial Condition | Admin exists in the database. |
| Procedure | 1. The user selects login.<br>2. The user provides an invalid username and a password.<br>3. The user clicks on the login button. |
| Expected Outcome | Login is unsuccessful and a failed login alert is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 8 |
| Category | Evaluation of unsuccessful administrator login |
| Requirements Coverage | UC8-Unsuccessful-Admin-Login-Invalid-Password |
| Initial Condition | Admin exists in the database. |
| Procedure | 1. The user selects login.<br>2. The user provides a username and an invalid password.<br>3. The user clicks on the login button. |
| Expected Outcome | Login is unsuccessful and a failed login alert is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 9 |
| Category | Evaluation of successful administrator logout |
| Requirements Coverage | UC9-Successful-Admin-Logout |
| Initial Condition | Admin is already logged in. |
| Procedure | 1. The admin clicks on logout. |
| Expected Outcome | Logout is successful and the admin is redirected to the login page. |
| Notes | None |

| | |
|---|---|
| TestID | 10 |
| Category | Evaluation of successful addition of products |
| Requirements Coverage | UC10-Successful-Product-Quantity-Addition |
| Initial Condition | The admin is logged in. The admin is on the admin homepage. |
| Procedure | 1. The admin clicks “add product”.<br>2. The admin correctly fills the ‘add product form’ including the initial quantity.<br>3. The admin clicks “submit”. |
| Expected Outcome | A product is successfully added to the database and the product should be able to be viewed on the client side. |
| Notes | None |

| | |
|---|---|
| TestID | 11 |
| Category | Evaluation of unsuccessful addition of products |
| Requirements Coverage | UC11-Unsuccessful-Product-Addition-Quantity-Zero |
| Initial Condition | The admin is logged in. The admin is on the admin homepage. |
| Procedure | 1. The admin clicks “add product”.<br>2. The admin fills the ‘add product form’ but puts the initial quantity as zero.<br>3. The admin clicks “submit”. |

| | |
|---|---|
| Expected Outcome | Addition of product fails. Error message with “cannot add product with zero quantity” is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 12 |
| Category | Evaluation of unsuccessful addition of products |
| Requirements Coverage | UC12-Unsuccessful-Product-Quantity-Addition-Missing-Product-Info |
| Initial Condition | The admin is logged in. The admin is on the admin homepage. |
| Procedure | 1. The admin clicks “add product”.<br>2. The admin provides some/none of the details required by the add product form.<br>3. The admin clicks “submit”. |
| Expected Outcome | Addition of product fails. Error message with “cannot add product with missing product details” is displayed. |
| Notes | None |

| | |
|---|---|
| TestID | 13 |
| Category | Evaluation of successful order placement |
| Requirements Coverage | UC13-Successful-Order-Placement |
| Initial Condition | The user is on the client homepage. |
| Procedure | 1. The user selects a product from a list of products.<br>2. The user inputs a valid quantity desired.<br>3. The user clicks “submit order”. |
| Expected Outcome | The order is placed successfully. The server UI displays a message with the details of the last order received and the bar graph with current available quantities. The client UI displays “Order for Product X Quantity Y is sent” |
| Notes | None |

| | |
|---|---|
| TestID | 14 |

| | |
|---|---|
| Category | Evaluation of unsuccessful order placement |
| Requirements Coverage | UC14-Unsuccessful-Order-Placement-Max-Stock-Quantity-Exceeded |
| Initial Condition | The user is on the client homepage. |
| Procedure | 1. The user selects a product from a list of products.<br>2. The user inputs an invalid quantity more than the max stock quantity.<br>3. The user clicks “submit order”. |
| Expected Outcome | The placement of order fails. An error alert “Order exceeds the max quantity set for this product and cannot be processed” is displayed on both Client and Server UI |
| Notes | None |

| | |
|---|---|
| TestID | 15 |
| Category | Evaluation of successful order placement |
| Requirements Coverage | UC15-Successful-Order-Placement-Low-Stock |
| Initial Condition | The client is on the homepage.<br>The quantity for products is low. |
| Procedure | 1. The user selects a product from a list of products.<br>2. The user inputs a quantity more than the available quantity in the warehouse for this product at the time the order is processed by the server.<br>3. The user clicks “submit order”. |
| Expected Outcome | Order is placed successfully. An alert is displayed on the server UI “Order for Product X Quantity Y is pending – order exceeds available quantity”. Restock is initiated. |
| Notes | None |

| | |
|---|---|
| TestID | 16 |
| Category | Evaluation of successful restock for low stock after fulfilling an order |
| Requirements Coverage | UC16-Successful-Restock-After-Order-Fulfillment |
| Initial Condition | Product X exists in the database. An order for product X has been fulfilled and the quantity of the product X  in the |

| | |
|---|---|
| | warehouse drops below the predefined min stock quantity set product X. |
| Procedure | 1. Fulfill user order. |
| Expected Outcome | Product X's state is set as "low stock". Server UI displays "Restocking Operation for Product X initiated". Restock is initiated. |
| Notes | None |

| | |
|---|---|
| TestID | 17 |
| Category | Evaluation of order placement sequence |
| Requirements Coverage | UC17-Order-Sequence-Successfully-Preserved |
| Initial Condition | Products exist in the database. |
| Procedure | 1. User 1 places multiple valid orders.<br>2. User 2 places multiple valid orders. |
| Expected Outcome | Orders are processed in valid sequence on a first-come-first-serve basis on the server |
| Notes | None |

| | |
|---|---|
| TestID | 18 |
| Category | Evaluation of pricing strategy |
| Requirements Coverage | UC18-Pricing-Strategy-Valid |
| Initial Condition | Order for Product X has been placed. |
| Procedure | 1. Server processes order.<br>2. Server uses a valid pricing strategy. |
| Expected Outcome | Final Order Price is valid based on pricing strategy used. |
| Notes | None |

| | |
|---|---|
| TestID | 19 |
| Category | Evaluation of pricing strategy |

| | |
|---|---|
| Requirements Coverage | UC19-Pricing-Strategy-Invalid-Strategy-ID |
| Initial Condition | Order for Product X has been placed. |
| Procedure | 1. Server processes order.<br>2. Server uses a pricing strategy with an id that has been deleted/not in the database. |
| Expected Outcome | This should not happen. |
| Notes | None. |

| | |
|---|---|
| TestID | 20 |
| Category | Evaluation of pricing strategy |
| Requirements Coverage | UC20-Pricing-Strategy-Multiple-Strategies |
| Initial Condition | Order for multiple products has been placed. |
| Procedure | 1. Server processes order.<br>2. Server uses different valid strategies on each product depending on the strategy and product criteria. |
| Expected Outcome | Final Order Price of all products are valid based on the different pricing strategy used. |
| Notes | None |

| | |
|---|---|
| TestID | 21 |
| Category | Evaluation of Product Restocking |
| Requirements Coverage | UC21-Product-Restocking |
| Initial Condition | Product X is in low stock status. |
| Procedure | 1. Admin initiates the restocking operation for Product X.<br>2. The restocking process is allowed to complete. |
| Expected Outcome | ● The server UI displays “Restocking Operation for Product X initiated” and later “Restocking Operation for Product X completed.” |

| | |
|---|---|
| | ● Product X’s stock quantity is restored to its maximum level. |
| Notes | Verify that the restocking operation effectively replenishes the stock of the product. |

| | |
|---|---|
| TestID | 22 |
| Category | Evaluation of Order Processing Time |
| Requirements Coverage | UC22-Order-Processing-Time |
| Initial Condition | Multiple orders are in the queue. |
| Procedure | 1. Admin places multiple orders with varying quantities.<br>2. Wait for the orders to be processed. |
| Expected Outcome | ● Orders are processed in a reasonable time frame, considering the simulated delay for fulfilling each order.<br>● The server UI reflects the correct order processing sequence. |
| Notes | Assess the system’s performance in handling and processing multiple orders concurrently. |

| | |
|---|---|
| TestID | 23 |
| Category | Evaluation of Dynamic Pricing Strategy Update |
| Requirements Coverage | UC23-Dynamic-Pricing-Strategy |

| | |
|---|---|
| Initial Condition | Pricing strategy for Product Y is initially set to Strategy A. |
| Procedure | 1. Admin updates the pricing strategy for Product Y to Strategy B.<br>2. Places an order for Product Y. |
| Expected Outcome | ● The pricing strategy for Product Y is dynamically updated to Strategy B.<br>● The final order price reflects the changes made to the pricing strategy. |
| Notes | Ensure that dynamic updates to pricing strategies are accurately applied to orders. |

| | |
|---|---|
| TestID | 24 |
| Category | Stock level display |
| Requirements Coverage | UC24-stock-level-display |
| Initial Condition | order for a product x is made |
| Procedure | 1.check the viewer subsystem for X’s stock<br>2. Place an order for product X |
| Expected Outcome | viewer subsystem is updated and shows reduced stock for product x |
| Notes | stock levels should be confirmed before and after the order |

| | |
|---|---|
| TestID | 25 |
| Category | Restocking trigger |
| Requirements Coverage | UC25- Restock -notification |
| Initial Condition | product stock below minimum required level |
| Procedure | 1.monitor stock level<br>2. verify restocking notification |
| Expected Outcome | Restocking operation is initiated for the product |
| Notes | Restocking schedule in the product database should match the notification |

| | |
|---|---|
| Test ID | 26 |
| Category | Network Failure Handling |
| Requirements Coverage | UC26 |
| Initial Condition | Server is initialized and connected to product database |
| Procedure | 1. Start a client process<br>2. Simulate a network failure by disconnecting the client from the server during order placement |
| Expected Outcome | The system handles the failure and the server/UI provides a relevant error message. The order is not processed. |
| Notes | Unprocessed order makes for data consistency |

| | |
|---|---|
| Test ID | 27 |
| Category | Order Processing with Random Delay |
| Requirements Coverage | UC27 |
| Initial Condition | Server is initialized and connected to product database |

| | |
|---|---|
| Procedure | 1. Start a client process<br>2. Connect to server<br>3. Place order/introduce delay |
| Expected Outcome | The system introduces a random delay before processing order, simulating a real world delay. The order is eventually fulfilled. |
| Notes | This test case mimics a real world scenario where the server might be processing multiple requests and is required to load balance. |

| | |
|---|---|
| Test ID | 28 |
| Category | Invalid Restocking Schedule |
| Requirements Coverage | UC6 |
| Initial Condition | Server is initialized and connected to product database |
| Procedure | 1. Manually set an invalid restocking schedule for a product<br>2. Initiate a restocking operation for the affected product |
| Expected Outcome | The system will detect and handle the invalid restocking schedule, providing an error message and preventing this operation. |
| Notes | Invalid input handling |

| | |
|---|---|
| Test ID | 29 |
| Category | Concurrent Restocking and Order Placement |
| Requirements Coverage | UC4, UC6 |
| Initial Condition | Server is initialized and connected to product database with low stock for a particular product |
| Procedure | 1. Simultaneously initiate restocking for a product and place an order for the same product. |
| Expected Outcome | The system will handle the concurrent requests and if the restocking is processed before the order processing, than the order will go through. |

| | |
|---|---|
| Notes | Concurrent Processes |

| | |
|---|---|
| Test ID | 30 |
| Category | Database Update Race Condition |
| Requirements Coverage | UC27 |
| Initial Condition | Server is initialized and connected to product database |
| Procedure | 1. Simulate a race condition by initiating concurrent database update operations from multiple sources (different admins). |
| Expected Outcome | The system will handle the race condition, processing both updates. |
| Notes | Concurrent requests |