CPEN 333

AMAZOOM: AUTOMATED WAREHOUSE

DESIGN DOCUMENT

Name	Student ID
Metehan Sahin	26883363
Kevin Jiang	22715874
Lucy Hua	72659469
Nic Ricci	32936999

Executive Summary:

This proposal describes warehouse automation implementation to improve flow of operation and reduce costs for Amazoom warehouses. Automation system includes a central computer, robots and delivery/restocking trucks.

The central computer keeps track of all products and inventory, the locations of inventory, a list of orders (received, ready to deliver, and out for delivery), and arrival/departure of delivery/restocking trucks.

Robots carry items from restocking trucks to racks or from racks to delivery trucks. They have a carrying capacity that is not exceeded. Robots can handle all items in an order if total item weight is less than or equal to carrying capacity. If not, the central computer assigns another robot to process order. Restocking happens in a similar pattern, where robots take as many items as they can from restocking trucks and put them to available locations.

Each robot and delivery/restocking truck has their own thread to handle item carrying. These threads are created and handled by the warehouse central computer process. The remote webserver is responsible for keeping track of different warehouses by handling multiple warehouse central computer processes.

The system has 2 user interfaces: client and manager. Client UI allows clients to search for items, add items to cart and order items that are in the cart. Manager UI lets the manager query the status of an order, check the number in stock of an item, and get alerts about low-stock items

This system design is extremely practical, since the depth of automation makes it easier to handle warehouse coordination. There is a severe reduction in the tasks that employees are required to do within the warehouse. Clients are interacting with a very user-friendly and simple web application that makes it easy to order items. Overall, this warehouse automation proposal provides benefits that will help Amazoom reduce cost and increase efficiency instantly and be more swift with warehouse management and logistics.

Function Specifications:

Classes:

- **Order:**
    - **public void addItemtoOrder(Item NewItem, int Quantity):** Adds the item to list of items, itemList. Also updates total order weight.
    - **public void removeItemfromOrder(Item NewItem, int Quantity):** Adds item back to quantity list. Updates list of items and total order weight accordingly.
- **Pathfinding:**
    - **public int CalculateDistanceCost(TileNode a, TileNode b):** Calculates distance cost to move between TileNode a and TileNode b.
- **Robot:**
    - **public void move(int endX, int endY, WarehouseGrid<TileNode> grid, char symbol):** Moves the robot to X and Y coordinates specified as endX and endY
    - **public void addItemToItemsCarried(Item item):** Adds the item to the list of items carried and updates total weight of items carried by the robot. Also removes the item from the shelf and adds the rack of the item to the empty rack list.
    - **public bool goToLocation(TileNode location):** Robot goes to the specific location.
    - **public void goToTruck(TileNode truckDockedLocation):** Robot goes to truck specified as truckDockedLocation
    - **public void processOrder(Order order):** Robot carries items from order to delivery truck. Updates items carried and weight of items carried sequentially.
    - **public void processRestock(RestockingTruck truck):** Robot gets items from the restocking truck up until weight capacity is reached or there are no items left in the restocking truck, goes to an available shelf and places items. Repeatedly checks the shelf capacity and if exceeded, find a new shelf to place items. Updates items carried and weight of items carried sequentially.
- **DeliveryTruck:**
    - **public bool loadItem(Item newItem):** Checks that adding new item does not exceed, then loads item from robot to the truck. Returns true if successful and false if exceeded weight capacity.
- **RestockingTruck:**
    - **public Item unloadItem():** Removes the next item in the list of items and returns it or null if there are no items left in the restocking truck.
- **Warehouse:**

- **public void Start():** This is the main method that gets called when the warehouse is initiated. It handles all the scheduling in the warehouse. If there is a pending order, finds an available robot and assigns it to the order.
- **public void AddItem(int itemID, string itemName):** Adds item to list of items and sets the quantity of it to zero. Each item has an ID that is used to keep track of the amount currently stored and for location finding.
- **public void truckMovement(int numTrucks, List<TileNode> listOfLoadingDocks):** Coordinates truck movement into and out of loading dock. Trucks wait in a queue while the loading dock is at full capacity and when a spot opens, the truck in front of the queue is placed there.

Signaling and Communication Protocols:

Between separate Warehouse Computers:

Memory mapped files are used to share resources and communicate between warehouse computers. This works for our demo since warehouse computers are simulated on different processes. In real applications, memory mapped files would not be accessible across separate computers. Instead we would use a database such as SQL.

Between Warehouse Computer and Robot:

Each Robot object is constructed by passing in the Warehouse computer it belongs to. Consequently, the robot is able to write to and read properties of the warehouse computer.

The warehouse computer stores data like the following:

Dictionary<int, List<TileNode>> itemLocation	A dictionary that stores the item ID as its key and a list of locations containing the item as the value. When a robot restocks an item, it will update the dictionary by adding the location of the item. Similarly, a robot will search the dictionary by key using the item ID to locate an item needed for a customer's order.

The robot also has the following flags to signal the warehouse computer if it is free for a new task. When there is an order available or restocking truck at the loading dock, the warehouse computer iterates through a list of robots to find a free robot and then assign it a respective task.

bool isLoading	If true, the robot is in the process of locating items in its assigned order and loading the order to its assigned truck. When the robot is finished, it will set isLoading to false to indicate it is free.
bool isStocking	If true, the robot is in the process of unloading items from its assigned truck and restocking the items to empty shelves. When the robot is finished, it will set isStocking to false to indicate it is free.

Between Warehouse Computer and Truck

Trucks arrive at the loading dock and enter a queue. This queue is for docking. Truck that is at the front of the queue is signaled to move to the docking station once there is an available spot. The available spot is determined by a warehouse computer, where it keeps track of every truck currently docked and docking stations occupied. The truck at the front of the queue gets notified by the warehouse computer with the information of location to go and dock.

Between Webserver Database and Client Interface

In order to place orders, the client is able to interact with the webserver database through a client interface. The interface developed for this project was an ASP.NET MVC (Model-View-Controller) due to its high customizability. A series of views were created to provide ease of navigation as well as a professional appearance: a home page, product list of the database inventory as well as a place orders were created. Upon viewing the product list, a client fills out the input fields of the order page, which will ask for the item ID, quantity and order ID. This information gets stored into a SQL datatable in the database, and transmitted to the warehouse computer for processing.

Between Warehouse Computer and Webserver Database

The data from the warehouse and client-interface are stored in a SQL database project; one data table listed the warehouse inventory while another stored client orders made through the interface. To allow the warehouse computer to write and query data from the database (and vice versa), the Visual Studio extension Dapper was used, a convenient tool that helped map object oriented models to databases (and vice versa).

Using Dapper, the database connects to the warehouse computer using a common connection string. Then, within the warehouse computer, several query functions were written in order to perform operations on the database data e.g insert an item (INSERT), updating item stock (UPDATE). Using these functions, the warehouse computer is able to interact with the database by simply passing in item parameters. This ensured that the front-end and back-end of the project were securely connected.

In the case of multiple warehouses, the items in the order can be assigned to different warehouses by the database. The warehouses will then send robots to reserve the items from their locations and the stocks for each respective warehouse will be updated. For the case of multiple orders, the warehouse computer will check the order ID fields of each item purchase to regulate orders; if the order IDs matched, they were part of the same order and will be processed together, otherwise they will not be.

Between Robot and Robots (collision avoidance)

All robots within a warehouse access the warehouse computer's grid for path finding. Each isle tileNode within the grid has a status flag "isFree". When a robot occupies a tileNode, it will set isFree to false and back to true when the robot moves to another tileNode. When isFree is false, the path finding algorithm will consider the tileNode as unwalkable and find a path around it. As such, robots are able to communicate to each other and avoid collisions.

User Stories and Use Case Descriptions:

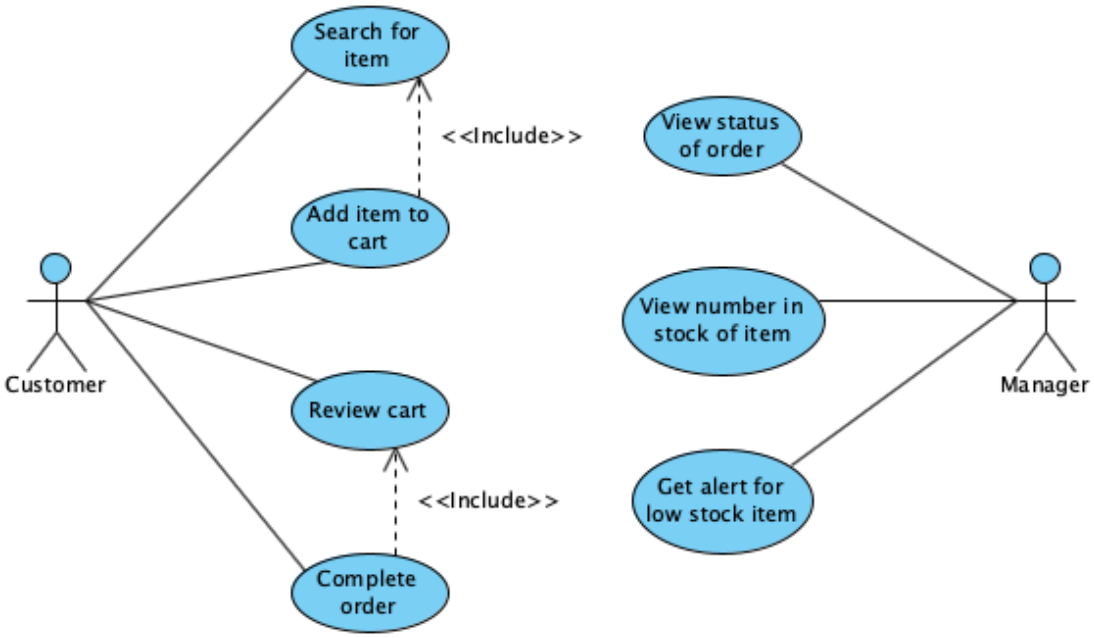
- As a customer, I want to be able to search for items.
    - **Use Case: Search item**
        - Benefits: Customer can view desired items and relevant information
        - Interactions:
            - Customer writes down the name of the item to the search bar.
            - System checks warehouses for the item.
            - Information about the item is displayed with an option to add to cart.
        - Effects: Webserver page is updated with item information.
        - **Use Case Scenarios:**
            - If the system cannot find the item, a message is displayed to indicate that item is not available.
            - If the item is not in stock, a message is displayed to indicate it and the button to add item to order becomes non-clickable.
- As a customer, I want to be able to add items to my cart.
    - **Use Case: Add item to cart**
        - Benefits: Customer can add item to cart to order
        - Interactions:
            - Customer clicks the button to add item to cart.
            - System adds item to item list of the order.
        - Effects: Items in cart are updated.
        - **Use Case Scenarios:** No specific use case scenarios, but this use case includes searching for an item.
- As a customer, I want to be able to review my cart.
    - **Use Case: Review cart**
        - Benefits: Customer can view all items in the cart before completing the order, change quantity and/or remove items.
        - Interactions:
            - Customer clicks the button to review cart.
            - System displays the cart with all the items within the cart, including quantities.
        - Effects: Webserver page is updated with cart displayed.
        - **Use Case Scenarios:** If there are no items in cart, a message is displayed to indicate this situation.

- As a customer, I want to be able to complete my order.
    - **Use Case: Complete order**
        - Benefits: Customer completes the order and the order is ready for processing.
        - Interactions:
            - While on the cart page, customer clicks checkout button.
            - A message to confirm order asks customer to confirm order.
            - Customer clicks confirm order.
            - A message indicating that order is confirmed is displayed.
        - Effects: Order is added to queue of orders, with all
        - **Use Case Scenarios:** If customer does not confirm order, webserver returns to cart page.
- As a manager, I want to be able to query the status of an order.
    - **Use Case: View status of order**
        - Benefits: Manager can address customer problems/concerns if necessary.
        - Interactions:
            - While on the warehouse computer, order status are displayed
        - Effects: Order status is displayed in the manager desktop application.
        - **Use Case Scenarios:** Manger is able to monitor the orders and be sure that orders are being processed.
- As a manager, I want to be able to check the number in stock of an item.
    - **Use Case: View number in stock of item**
        - Benefits: Manager can take precautionary actions to avoid scarcity.
        - Interactions:
            - While on the warehouse computer, item information including stock, ID, and name are all displayed
        - Effects: Quantity of an item is displayed in the manager desktop application.
        - **Use Case Scenarios:** When restocking the warehouse, the manager wants to know what items are near being out of stock to order.
- As a manager, I want to be able to get alerts about low stock items.
    - **Use Case: Get alert for low stock item**
        - Benefits: Manager can order more of the item for restocking.
        - Interactions:

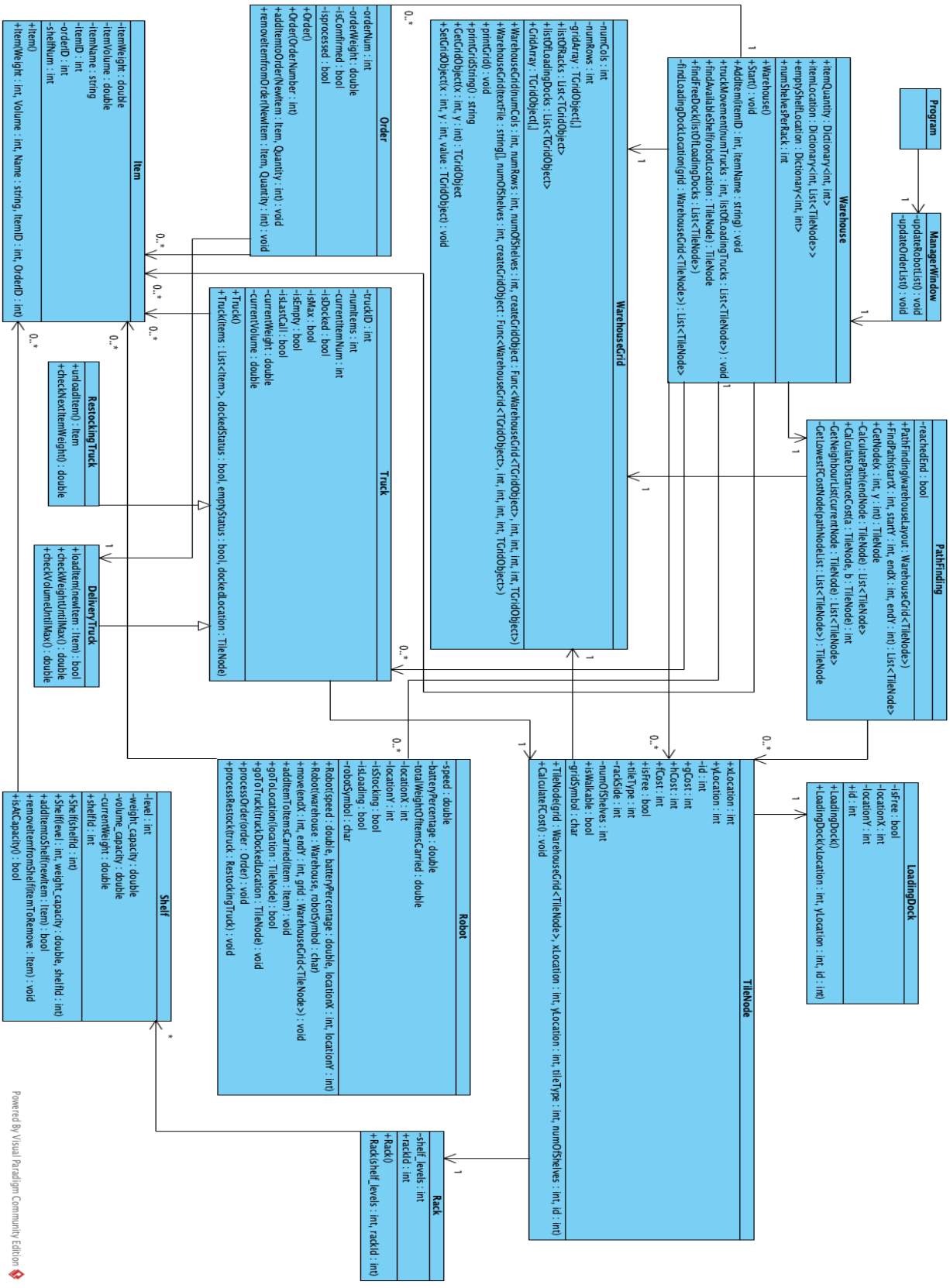
- While on the warehouse computer, the manager is able to enter the item ID along with the desired quantity and click submit. The items are then placed on order

- Effects: Alert message is displayed for manager to notice.

Use-Case Diagram:

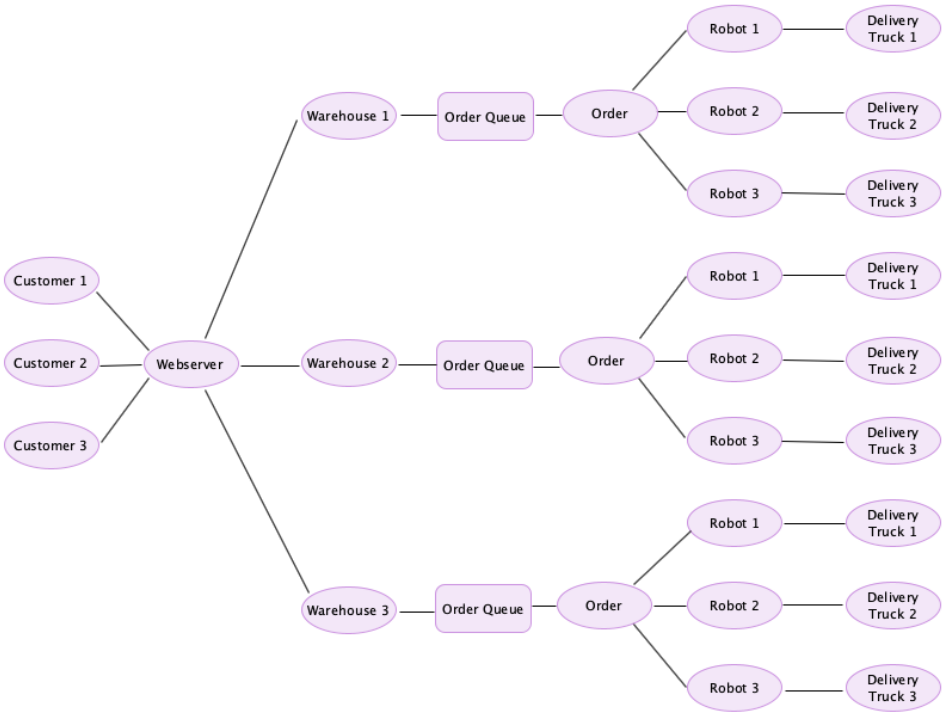


Class Diagram:

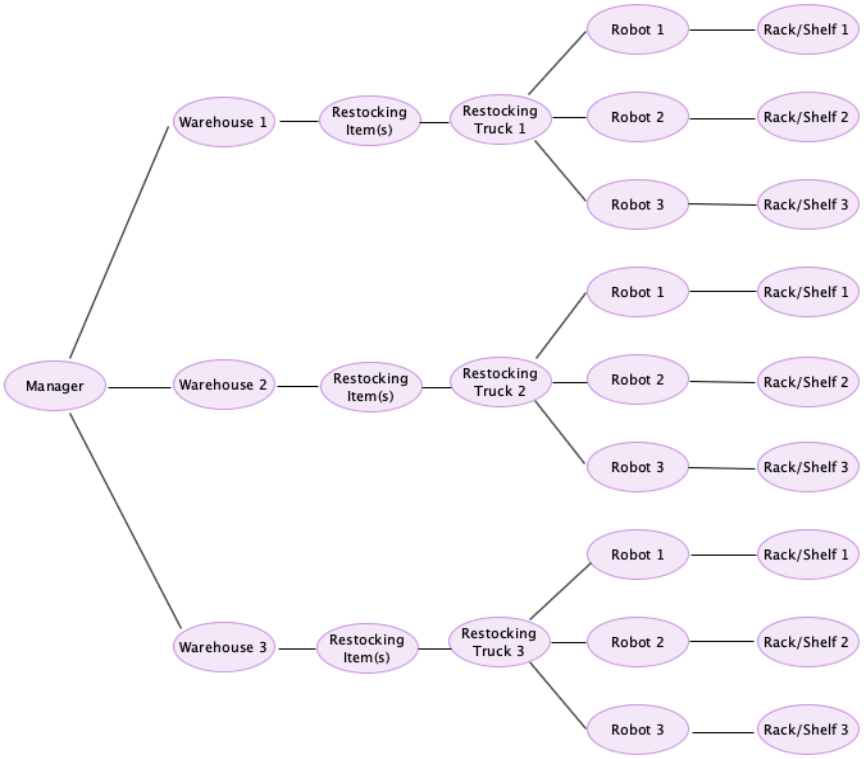


Object Interaction Diagram:

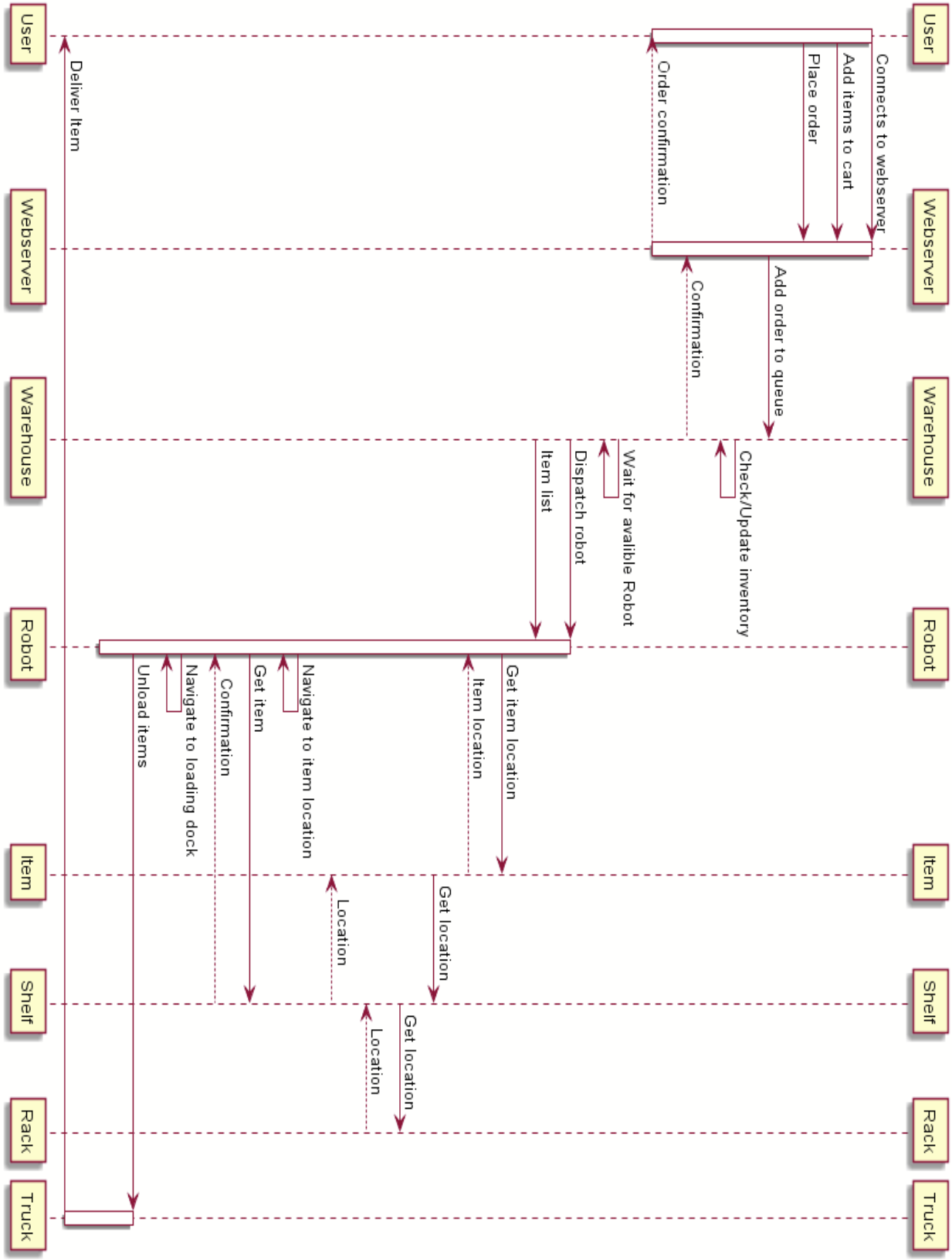
Order Processing (Only used 3 customers/warehouses/robots/delivery trucks for convenience):



Restocking (Only used 3 customers/warehouses/restocking trucks/robots/racks/shelves for convenience):



Sequence Diagram:



Testing:

Testing was a crucial element in the design process for this project. Given the magnitude of the project and all the different use-cases, testing helped ensure that our algorithms were functioning as planned, and that integrating our final system will be as smooth as possible. The two main types of tests we used were Regression and Integration testing; unit testing was performed on isolated methods to ensure they were functional, but not used as extensively as the other types of tests. This is because unit testing was not as time efficient in a large project like this that had numerous methods/algorithms and spanned many different files/branches.

Regression Tests:

Regression tests involved ensuring the code worked as it did after making changes e.g adding new features or optimization code. Given the vast size of the project, slight mishaps such as a mistyped character could be extremely detrimental to the whole system and hard to trace. Therefore, we frequently re-built our solution files after changes and back-tracked if errors occurred. Regression tests were used extensively in the warehouse computer and the database, elements that had many functionalities. Everytime a functionality was added, the code would be rebuilt and rerun to ensure that it, and any previous functionalities, worked flawlessly.

Integration Tests:

Integration tests addressed testing the system as a whole to ensure that each part functioned and interacted as expected. There were numerous parts to this project that needed to be connected together in the final product: The warehouse computer, database, client interface as well as numerous algorithms and methods. Interfacing all these parts proved quite the challenge, however integration tests were very effective during the debugging process eventually aided in getting the project up and running. Through integration tests, we were also able to test project functionalities such as the dynamic addition of robots, multiple warehouses, querying data from the database to the warehouse computer etc. Lastly, these tests helped us prepare for the demo by ensuring it would go as seamlessly as possible given the short time window.